Николай Прохоренок

Python 3 и PyQt
Разработка приложений

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.06
ББК 32.973.26-018.2
П84

Прохоренок Н. А.

П84 Python 3 и PyQt. Разработка приложений. — СПб.: БХВ-Петербург, 2012. — 704 с.: ил.

ISBN 978-5-9775-0797-4

Описан базовый синтаксис языка Python: типы данных, операторы, условия, циклы, регулярные выражения, встроенные функции, объектно-ориентированное программирование, работа с файлами и каталогами, часто используемые модули стандартной библиотеки. Приведены основы базы данных SQLite, интерфейс доступа к базе и способы получения данных из Интернета. Особое внимание уделено библиотеке PyQt, позволяющей создавать приложения с графическим интерфейсом на языке Python. Рассмотрены способы обработки сигналов и событий, управление свойствами окна, создание формы с помощью программы Qt Designer, работа многопоточных приложений, а также все основные компоненты (кнопки, текстовые поля, списки, таблицы, меню, панели инструментов и др.) и варианты их размещения внутри окна. На сайте издательства приведены все примеры из книги.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Марины Дамбиевой*
Зав. производством	*Николай Тверских*

Подписано в печать 01.12.11.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 56,76.
Тираж 1200 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0797-4

© Прохоренок Н. А., 2012
© Оформление, издательство "БХВ-Петербург", 2012

Оглавление

Введение ... **13**

ЧАСТЬ I. ОСНОВЫ PYTHON 3 ... **15**

Глава 1. Первые шаги ... **17**
1.1. Установка Python ... 17
1.2. Первая программа на Python ... 23
1.3. Структура программы ... 24
1.4. Комментарии ... 27
1.5. Скрытые возможности IDLE ... 28
1.6. Вывод результатов работы программы ... 29
1.7. Ввод данных ... 31
1.8. Доступ к документации ... 33

Глава 2. Переменные ... **36**
2.1. Именование переменных ... 36
2.2. Типы данных ... 38
2.3. Присваивание значения переменным ... 41
2.4. Проверка типа данных ... 43
2.5. Преобразование типов данных ... 44
2.6. Удаление переменной ... 47

Глава 3. Операторы ... **48**
3.1. Математические операторы ... 48
3.2. Двоичные операторы ... 50
3.3. Операторы для работы с последовательностями ... 51
3.4. Операторы присваивания ... 52
3.5. Приоритет выполнения операторов ... 53

Глава 4. Условные операторы и циклы ... **55**
4.1. Операторы сравнения ... 56
4.2. Оператор ветвления *if...else* ... 58
4.3. Цикл *for* ... 61

4.4. Функции *range()* и *enumerate()* ... 63
4.5. Цикл *while* ... 66
4.6. Оператор *continue*. Переход на следующую итерацию цикла ... 67
4.7. Оператор *break*. Прерывание цикла ... 67

Глава 5. Числа ... 69
5.1. Встроенные функции для работы с числами ... 70
5.2. Модуль *math*. Математические функции ... 72
5.3. Модуль *random*. Генерация случайных чисел ... 73

Глава 6. Строки ... 76
6.1. Создание строки ... 77
6.2. Специальные символы ... 81
6.3. Операции над строками ... 81
6.4. Форматирование строк ... 84
6.5. Метод *format()* ... 90
6.6. Функции и методы для работы со строками ... 94
6.7. Настройка локали ... 97
6.8. Изменение регистра символов ... 98
6.9. Функции для работы с символами ... 98
6.10. Поиск и замена в строке ... 99
6.11. Проверка типа содержимого строки ... 102
6.12. Тип данных *bytes* ... 105
6.13. Тип данных *bytearray* ... 108
6.14. Преобразование объекта в последовательность байтов ... 112
6.15. Шифрование строк ... 112

Глава 7. Регулярные выражения ... 114
7.1. Синтаксис регулярных выражений ... 114
7.2. Поиск первого совпадения с шаблоном ... 123
7.3. Поиск всех совпадений с шаблоном ... 127
7.4. Замена в строке ... 128
7.5. Прочие функции и методы ... 130

Глава 8. Списки и кортежи ... 132
8.1. Создание списка ... 132
8.2. Операции над списками ... 136
8.3. Многомерные списки ... 139
8.4. Перебор элементов списка ... 139
8.5. Генераторы списков и выражения-генераторы ... 140
8.6. Функции *map()*, *zip()*, *filter()* и *reduce()* ... 141
8.7. Добавление и удаление элементов списка ... 144
8.8. Поиск элемента в списке ... 147
8.9. Переворачивание и перемешивание списка ... 148
8.10. Выбор элементов случайным образом ... 148
8.11. Сортировка списка ... 149
8.12. Заполнение списка числами ... 150
8.13. Преобразование списка в строку ... 151
8.14. Кортежи ... 151

8.15. Модуль *itertools* ... 153
8.15.1. Генерация неопределенного количества значений ... 153
8.15.2. Генерация комбинаций значений ... 154
8.15.3. Фильтрация элементов последовательности ... 156
8.15.4. Прочие функции ... 157

Глава 9. Словари и множества ... 159
9.1. Создание словаря ... 159
9.2. Операции над словарями ... 161
9.3. Перебор элементов словаря ... 163
9.4. Методы для работы со словарями ... 164
9.5. Генераторы словарей ... 166
9.6. Множества ... 167
9.7. Генераторы множеств ... 172

Глава 10. Работа с датой и временем ... 173
10.1. Получение текущей даты и времени ... 173
10.2. Форматирование даты и времени ... 175
10.3. "Засыпание" скрипта ... 177
10.4. Модуль *datetime*. Манипуляции датой и временем ... 178
10.4.1. Класс *timedelta* ... 178
10.4.2. Класс *date* ... 180
10.4.3. Класс *time* ... 183
10.4.4. Класс *datetime* ... 184
10.5. Модуль *calendar*. Вывод календаря ... 189
10.5.1. Методы классов *TextCalendar* и *LocaleTextCalendar* ... 190
10.5.2. Методы классов *HTMLCalendar* и *LocaleHTMLCalendar* ... 191
10.5.3. Другие полезные функции ... 192
10.6. Измерение времени выполнения фрагментов кода ... 195

Глава 11. Пользовательские функции ... 197
11.1. Создание функции и ее вызов ... 197
11.2. Расположение определений функций ... 200
11.3. Необязательные параметры и сопоставление по ключам ... 201
11.4. Переменное число параметров в функции ... 203
11.5. Анонимные функции ... 205
11.6. Функции-генераторы ... 206
11.7. Декораторы функций ... 207
11.8. Рекурсия. Вычисление факториала ... 209
11.9. Глобальные и локальные переменные ... 210
11.10. Вложенные функции ... 213
11.11. Аннотации функций ... 215

Глава 12. Модули и пакеты ... 216
12.1. Инструкция *import* ... 216
12.2. Инструкция *from* ... 219
12.3. Пути поиска модулей ... 222
12.4. Повторная загрузка модулей ... 223
12.5. Пакеты ... 224

Глава 13. Объектно-ориентированное программирование ... 228
13.1. Определение класса и создание экземпляра класса ... 228
13.2. Методы *__init__()* и *__del__()* ... 231
13.3. Наследование ... 232
13.4. Множественное наследование ... 234
13.5. Специальные методы ... 236
13.6. Перегрузка операторов ... 239
13.7. Статические методы и методы класса ... 242
13.8. Абстрактные методы ... 243
13.9. Ограничение доступа к идентификаторам внутри класса ... 244
13.10. Свойства класса ... 245
13.11. Декораторы классов ... 246

Глава 14. Обработка исключений ... 247
14.1. Инструкция *try...except...else...finally* ... 248
14.2. Инструкция *with...as* ... 252
14.3. Классы встроенных исключений ... 254
14.4. Пользовательские исключения ... 256

Глава 15. Работа с файлами и каталогами ... 259
15.1. Открытие файла ... 259
15.2. Методы для работы с файлами ... 265
15.3. Доступ к файлам с помощью модуля *os* ... 271
15.4. Классы *StringIO* и *BytesIO* ... 273
15.5. Права доступа к файлам и каталогам ... 277
15.6. Функции для манипулирования файлами ... 278
15.7. Преобразование пути к файлу или каталогу ... 281
15.8. Перенаправление ввода/вывода ... 283
15.9. Сохранение объектов в файл ... 286
15.10. Функции для работы с каталогами ... 289

Глава 16. Основы SQLite ... 293
16.1. Создание базы данных ... 293
16.2. Создание таблицы ... 295
16.3. Вставка записей ... 301
16.4. Обновление и удаление записей ... 303
16.5. Изменение свойств таблицы ... 304
16.6. Выбор записей ... 305
16.7. Выбор записей из нескольких таблиц ... 308
16.8. Условия в инструкции *WHERE* ... 309
16.9. Индексы ... 312
16.10. Вложенные запросы ... 314
16.11. Транзакции ... 315
16.12. Удаление таблицы и базы данных ... 317

Глава 17. Доступ к базе данных SQLite из Python ... 318
17.1. Создание и открытие базы данных ... 319
17.2. Выполнение запроса ... 319
17.3. Обработка результата запроса ... 324

17.4. Управление транзакциями ... 327
17.5. Создание пользовательской сортировки ... 328
17.6. Поиск без учета регистра символов ... 330
17.7. Создание агрегатных функций ... 331
17.8. Преобразование типов данных ... 332
17.9. Сохранение в таблице даты и времени ... 336
17.10. Обработка исключений ... 337

Глава 18. Взаимодействие с Интернетом ... 340
18.1. Разбор URL-адреса ... 340
18.2. Кодирование и декодирование строки запроса ... 343
18.3. Преобразование относительной ссылки в абсолютную ... 347
18.4. Разбор HTML-эквивалентов ... 347
18.5. Обмен данными по протоколу HTTP ... 348
18.6. Обмен данными с помощью модуля *urllib.request* ... 354
18.7. Определение кодировки ... 357

ЧАСТЬ II. СОЗДАНИЕ ОКОННЫХ ПРИЛОЖЕНИЙ ... 359

Глава 19. Знакомство с PyQt ... 361
19.1. Установка PyQt ... 361
19.2. Первая программа ... 364
19.3. Структура программы ... 365
19.4. ООП-стиль создания окна ... 367
19.5. Создание окна с помощью программы Qt Designer ... 371
19.5.1. Создание формы ... 371
19.5.2. Загрузка ui-файла в программе ... 373
19.5.3. Преобразование ui-файла в py-файл ... 375
19.6. Модули PyQt ... 377
19.7. Типы данных в PyQt ... 377
19.8. Управление основным циклом приложения ... 379
19.9. Многопоточные приложения ... 381
19.9.1. Класс *QThread*. Создание потока ... 381
19.9.2. Управление циклом внутри потока ... 384
19.9.3. Модуль *queue*. Создание очереди заданий ... 388
19.9.4. Классы *QMutex* и *QMutexLocker* ... 391
19.10. Вывод заставки ... 395
19.11. Доступ к документации ... 397

Глава 20. Управление окном приложения ... 398
20.1. Создание и отображение окна ... 398
20.2. Указание типа окна ... 399
20.3. Изменение и получение размеров окна ... 401
20.4. Местоположение окна на экране ... 403
20.5. Указание координат и размеров ... 406
20.5.1. Класс *QPoint*. Координаты точки ... 407
20.5.2. Класс *QSize*. Размеры прямоугольной области ... 408
20.5.3. Класс *QRect*. Координаты и размеры прямоугольной области ... 410
20.6. Разворачивание и сворачивание окна ... 415
20.7. Управление прозрачностью окна ... 416

20.8. Модальные окна ... 417
20.9. Смена иконки в заголовке окна ... 419
20.10. Изменение цвета фона окна ... 420
20.11. Использование изображения в качестве фона ... 421
20.12. Создание окна произвольной формы ... 422
20.13. Всплывающие подсказки ... 423
20.14. Закрытие окна из программы ... 424

Глава 21. Обработка сигналов и событий ... 426
21.1. Назначение обработчиков сигналов ... 426
21.2. Блокировка и удаление обработчика ... 430
21.3. Генерация сигнала из программы ... 433
21.4. Новый стиль назначения и удаления обработчиков ... 435
21.5. Передача данных в обработчик ... 438
21.6. Использование таймеров ... 439
21.7. Перехват всех событий ... 442
21.8. События окна ... 445
21.8.1. Изменение состояния окна ... 445
21.8.2. Изменение положения окна и его размеров ... 446
21.8.3. Перерисовка окна или его части ... 447
21.8.4. Предотвращение закрытия окна ... 448
21.9. События клавиатуры ... 449
21.9.1. Установка фокуса ввода ... 449
21.9.2. Назначение клавиш быстрого доступа ... 451
21.9.3. Нажатие и отпускание клавиши на клавиатуре ... 454
21.10. События мыши ... 455
21.10.1. Нажатие и отпускание кнопки мыши ... 455
21.10.2. Перемещение указателя ... 456
21.10.3. Наведение и выведение указателя ... 457
21.10.4. Прокрутка колесика мыши ... 457
21.10.5. Изменение внешнего вида указателя мыши ... 458
21.11. Технология drag & drop ... 460
21.11.1. Запуск перетаскивания ... 460
21.11.2. Класс *QMimeData* ... 462
21.11.3. Обработка сброса ... 463
21.12. Работа с буфером обмена ... 465
21.13. Фильтрация событий ... 465
21.14. Искусственные события ... 466

Глава 22. Размещение нескольких компонентов в окне ... 468
22.1. Абсолютное позиционирование ... 468
22.2. Горизонтальное и вертикальное выравнивание ... 469
22.3. Выравнивание по сетке ... 472
22.4. Выравнивание компонентов формы ... 473
22.5. Классы *QStackedLayout* и *QStackedWidget* ... 475
22.6. Класс *QSizePolicy* ... 477
22.7. Объединение компонентов в группу ... 478
22.8. Панель с рамкой ... 479
22.9. Панель с вкладками ... 480

22.10. Компонент "аккордеон" ... 484
22.11. Панели с изменяемым размером ... 485
22.12. Область с полосами прокрутки ... 487

Глава 23. Основные компоненты ... 489
23.1. Надпись ... 489
23.2. Командная кнопка ... 492
23.3. Переключатель ... 494
23.4. Флажок ... 494
23.5. Однострочное текстовое поле ... 495
23.5.1. Основные методы и сигналы ... 495
23.5.2. Ввод данных по маске ... 498
23.5.3. Контроль ввода ... 499
23.6. Многострочное текстовое поле ... 500
23.6.1. Основные методы и сигналы ... 500
23.6.2. Изменение настроек поля ... 502
23.6.3. Изменение характеристик текста и фона ... 504
23.6.4. Класс *QTextDocument* ... 505
23.6.5. Класс *QTextCursor* ... 508
23.7. Текстовый браузер ... 511
23.8. Поля для ввода целых и вещественных чисел ... 512
23.9. Поля для ввода даты и времени ... 514
23.10. Календарь ... 516
23.11. Электронный индикатор ... 517
23.12. Индикатор хода процесса ... 518
23.13. Шкала с ползунком ... 519
23.14. Класс *QDial* ... 521
23.15. Полоса прокрутки ... 522

Глава 24. Списки и таблицы ... 523
24.1. Раскрывающийся список ... 523
24.1.1. Добавление, изменение и удаление элементов ... 523
24.1.2. Изменение настроек ... 524
24.1.3. Поиск элемента внутри списка ... 526
24.1.4. Сигналы ... 526
24.2. Список для выбора шрифта ... 526
24.3. Роли элементов ... 527
24.4. Модели ... 528
24.4.1. Доступ к данным внутри модели ... 528
24.4.2. Класс *QStringListModel* ... 529
24.4.3. Класс *QStandardItemModel* ... 530
24.4.4. Класс *QStandardItem* ... 533
24.5. Представления ... 536
24.5.1. Класс *QAbstractItemView* ... 536
24.5.2. Простой список ... 539
24.5.3. Таблица ... 540
24.5.4. Иерархический список ... 543
24.5.5. Управление заголовками строк и столбцов ... 545
24.6. Управление выделением элементов ... 547
24.7. Промежуточные модели ... 549

Глава 25. Работа с графикой ... 551
25.1. Вспомогательные классы ... 551
25.1.1. Класс *QColor*. Цвет ... 551
25.1.2. Класс *QPen*. Перо ... 555
25.1.3. Класс *QBrush*. Кисть ... 556
25.1.4. Класс *QLine*. Линия ... 557
25.1.5. Класс *QPolygon*. Многоугольник ... 558
25.1.6. Класс *QFont*. Шрифт ... 560
25.2. Класс *QPainter* ... 562
25.2.1. Рисование линий и фигур ... 562
25.2.2. Вывод текста ... 565
25.2.3. Вывод изображения ... 566
25.2.4. Преобразование систем координат ... 567
25.2.5. Сохранение команд рисования в файл ... 568
25.3. Работа с изображениями ... 569
25.3.1. Класс *QPixmap* ... 570
25.3.2. Класс *QBitmap* ... 572
25.3.3. Класс *QImage* ... 573
25.3.4. Класс *QIcon* ... 576

Глава 26. Графическая сцена ... 578
26.1. Класс *QGraphicsScene*. Сцена ... 578
26.1.1. Настройка параметров сцены ... 579
26.1.2. Добавление и удаление графических объектов ... 579
26.1.3. Добавление компонентов на сцену ... 580
26.1.4. Поиск объектов ... 581
26.1.5. Управление фокусом ввода ... 582
26.1.6. Управление выделением объектов ... 582
26.1.7. Прочие методы и сигналы ... 583
26.2. Класс *QGraphicsView*. Представление ... 584
26.2.1. Настройка параметров представления ... 584
26.2.2. Преобразования между координатами представления и сцены ... 586
26.2.3. Поиск объектов ... 586
26.2.4. Трансформация систем координат ... 587
26.2.5. Прочие методы ... 587
26.3. Класс *QGraphicsItem*. Базовый класс для графических объектов ... 588
26.3.1. Настройка параметров объекта ... 588
26.3.2. Трансформация объекта ... 590
26.3.3. Прочие методы ... 591
26.4. Графические объекты ... 592
26.4.1. Линия ... 592
26.4.2. Класс *QAbstractGraphicsShapeItem* ... 593
26.4.3. Прямоугольник ... 593
26.4.4. Многоугольник ... 593
26.4.5. Эллипс ... 594
26.4.6. Изображение ... 594
26.4.7. Простой текст ... 595
26.4.8. Форматированный текст ... 595

26.5. Группировка объектов ... 597
26.6. Эффекты ... 597
26.6.1. Класс *QGraphicsEffect* ... 597
26.6.2. Тень ... 598
26.6.3. Размытие ... 599
26.6.4. Изменение цвета ... 599
26.6.5. Изменение прозрачности ... 600
26.7. Обработка событий ... 600
26.7.1. События клавиатуры ... 600
26.7.2. События мыши ... 601
26.7.3. Обработка перетаскивания и сброса ... 603
26.7.4. Фильтрация событий ... 605
26.7.5. Обработка изменения состояния объекта ... 605

Глава 27. Диалоговые окна ... 607
27.1. Пользовательские диалоговые окна ... 607
27.2. Класс *QDialogButtonBox* ... 610
27.3. Класс *QMessageBox* ... 612
27.3.1. Основные методы и сигналы ... 614
27.3.2. Окно для вывода обычного сообщения ... 616
27.3.3. Окно запроса подтверждения ... 616
27.3.4. Окно для вывода предупреждающего сообщения ... 617
27.3.5. Окно для вывода критического сообщения ... 617
27.3.6. Окно "О программе" ... 618
27.3.7. Окно "About Qt" ... 618
27.4. Класс *QInputDialog* ... 618
27.4.1. Основные методы и сигналы ... 619
27.4.2. Окно для ввода строки ... 621
27.4.3. Окно для ввода целого числа ... 621
27.4.4. Окно для ввода вещественного числа ... 622
27.4.5. Окно для выбора пункта из списка ... 623
27.5. Класс *QFileDialog* ... 623
27.5.1. Основные методы и сигналы ... 624
27.5.2. Окно для выбора каталога ... 626
27.5.3. Окна для открытия файла ... 627
27.5.4. Окна для сохранения файла ... 628
27.6. Окно для выбора цвета ... 629
27.7. Окно для выбора шрифта ... 630
27.8. Окно для вывода сообщения об ошибке ... 631
27.9. Окно с индикатором хода процесса ... 632
27.10. Создание многостраничного мастера ... 633
27.10.1. Класс *QWizard* ... 633
27.10.2. Класс *QWizardPage* ... 637

Глава 28. Создание SDI- и MDI-приложений ... 640
28.1. Создание главного окна приложения ... 640
28.2. Меню ... 644
28.2.1. Класс *QMenuBar* ... 645
28.2.2. Класс *QMenu* ... 646

28.2.3. Контекстное меню ... 648
28.2.4. Класс *QAction* ... 649
28.2.5. Объединение переключателей в группу ... 652
28.3. Панели инструментов ... 653
28.3.1. Класс *QToolBar* ... 653
28.3.2. Класс *QToolButton* ... 655
28.4. Прикрепляемые панели ... 656
28.5. Управление строкой состояния ... 658
28.6. MDI-приложения ... 659
28.6.1. Класс *QMdiArea* ... 659
28.6.2. Класс *QMdiSubWindow* ... 662
28.7. Добавление иконки приложения в область уведомлений ... 663

Заключение ... 665

Приложение. Описание электронного архива ... 666

Предметный указатель ... 667

Введение

Добро пожаловать в мир Python и PyQt!

Python — это интерпретируемый объектно-ориентированный язык программирования высокого уровня, предназначенный для самого широкого круга задач. С его помощью можно обрабатывать различные данные, создавать изображения, работать с базами данных, разрабатывать Web-сайты и приложения с графическим интерфейсом. Python является кроссплатформенным языком, позволяющим создавать программы, которые будут работать во всех операционных системах. В этой книге мы рассмотрим базовые возможности Python 3.2 применительно к операционной системе Windows.

Согласно официальной версии название языка произошло вовсе не от змеи. Создатель языка Гвидо ван Россум (Guido van Rossum) назвал свое творение в честь британского комедийного телешоу BBC "Monty Python's Flying Circus". Поэтому более правильно будет "Пайтон". Тем не менее многие считают, что для русского человека более привычно называть язык "Питон". Так или иначе, в этой книге мы будем придерживаться традиционного написания слова Python на английском языке.

Программа на языке Python представляет собой обычный текстовый файл с расширением ру (консольная программа) или pyw (программа с графическим интерфейсом). Все инструкции из этого файла выполняются интерпретатором построчно. Для ускорения работы при первом импорте модуля создается промежуточный байт-код и сохраняется в файле с расширением рус. При последующих запусках, если модуль не был изменен, исполняется именно байт-код. Для выполнения низкоуровневых операций и задач, требующих высокой скорости работы, можно написать модуль на языке С, скомпилировать его, а затем подключить к основной программе.

Python является объектно-ориентированным языком. Это означает, что практически все данные являются объектами, даже сами типы данных. В переменной всегда сохраняется только ссылка на объект, а не сам объект. Например, можно создать функцию, сохранить ссылку на нее в переменной, а затем вызвать функцию через эту переменную. Данное обстоятельство делает язык Python идеальным инструментом для создания программ, использующих функции обратного вызова, например, при разработке графического интерфейса. Тот факт, что язык является объектно-ориентированным, отнюдь не означает, что ООП-стиль программирования является обязательным. На языке Python можно писать программы как в ООП-стиле, так и в процедурном стиле.

Python — самый стильный язык программирования в мире, не допускающий двоякого написания кода. Например, в языке Perl существует зависимость от контекста и множественность синтаксиса. Часто два программиста, пишущих на Perl, просто не понимают код друг

друга. В Python код можно написать только одним способом, т. к. в нем отсутствуют лишние конструкции. Все программисты должны придерживаться стандарта, описанного в документе **http://python.org/dev/peps/pep-0008/**. Более читаемого кода нет ни в одном другом языке программирования.

Синтаксис языка Python вызывает много нареканий у программистов, знакомых с другими языками. На первый взгляд может показаться, что отсутствие ограничительных символов (фигурных скобок или конструкции `begin...end`) для выделения блоков и обязательная вставка пробелов впереди инструкций может приводить к ошибкам. Однако это только первое и неправильное впечатление. Хороший стиль программирования в любом языке обязывает выделять инструкции внутри блока одинаковым количеством пробелов. В этой ситуации ограничительные символы просто являются лишними. Бытует мнение, что программа будет по-разному смотреться в разных редакторах. Это неверно. Согласно стандарту для выделения блоков необходимо использовать *четыре пробела*. Четыре пробела в любом редакторе будут смотреться одинаково. Если в другом языке вас не приучили к хорошему стилю программирования, то язык Python быстро это исправит. Если количество пробелов внутри блока будет разным, то интерпретатор выведет сообщение о фатальной ошибке, и программа будет остановлена. Таким образом, язык Python приучает программистов писать красивый и понятный код.

Так как программа на языке Python представляет собой обычный текстовый файл, его можно редактировать с помощью любого текстового редактора, например с помощью Notepad++. Однако лучше воспользоваться специализированными редакторами, которые не только подсвечивают код, но и выводят различные подсказки и позволяют отладить программу. Таких редакторов очень много, например PyScripter, PythonWin, UliPad, Eclipse + PyDev, Netbeans и др. Полный список редакторов расположен на странице **http://wiki.python.org/moin/PythonEditors**. В этой книге мы будем пользоваться редактором IDLE, который входит в состав стандартной библиотеки Python в Windows.

Во второй части книги мы рассмотрим библиотеку PyQt, позволяющую создавать кроссплатформенные приложения с графическим интерфейсом. Библиотека очень проста в использовании и идеально подходит для разработки оконных приложений практически любой сложности. В состав библиотеки входит программа Qt Designer, с помощью которой можно размещать компоненты на форме путем перетаскивания мышью. При сохранении формы Qt Designer создает XML-файл, который можно загрузить внутри программы или автоматически преобразовать в код на языке Python.

Желаю приятного прочтения и надеюсь, что эта книга станет верным спутником в вашей повседневной жизни.

ЧАСТЬ I

Основы Python 3

Глава 1. Первые шаги

Глава 2. Переменные

Глава 3. Операторы

Глава 4. Условные операторы и циклы

Глава 5. Числа

Глава 6. Строки

Глава 7. Регулярные выражения

Глава 8. Списки и кортежи

Глава 9. Словари и множества

Глава 10. Работа с датой и временем

Глава 11. Пользовательские функции

Глава 12. Модули и пакеты

Глава 13. Объектно-ориентированное программирование

Глава 14. Обработка исключений

Глава 15. Работа с файлами и каталогами

Глава 16. Основы SQLite

Глава 17. Доступ к базе данных SQLite из Python

Глава 18. Взаимодействие с Интернетом

ГЛАВА 1

Первые шаги

Прежде чем мы начнем рассматривать синтаксис языка, необходимо сделать два замечания. Во-первых, не забывайте, что книги по программированию нужно не только читать, но и выполнять все примеры, а также экспериментировать, изменяя что-нибудь в примерах. Поэтому, если вы удобно устроились на диване и настроились просто читать, у вас практически нет шансов изучить язык. Во-вторых, помните, что прочитать эту книгу один раз недостаточно. Первую часть книги вы должны знать наизусть! Сколько на это уйдет времени, зависит от ваших способностей и желания. Чем больше вы будете делать самостоятельно, тем большему научитесь. Ну что, приступим к изучению языка? Python достоин того, чтобы его знал каждый программист!

1.1. Установка Python

Вначале необходимо установить на компьютер интерпретатор Python.

1. Для загрузки дистрибутива переходим на страницу **http://python.org/download/** и скачиваем файл python-3.2.msi. Затем запускаем программу установки с помощью двойного щелчка на значке файла.
2. В открывшемся окне (рис. 1.1) устанавливаем переключатель **Install for all users** (Установить для всех пользователей) и нажимаем кнопку **Next.**
3. На следующем шаге (рис. 1.2) предлагается выбрать каталог для установки. Оставляем каталог по умолчанию (C:\Python32\) и нажимаем кнопку **Next.**
4. В следующем диалоговом окне (рис. 1.3) выбираем компоненты, которые необходимо установить. По умолчанию устанавливаются все компоненты и прописывается ассоциация с файловыми расширениями py, pyw и др. В этом случае запускать программы можно с помощью двойного щелчка мышью на значке файла. Оставляем выбранными все компоненты и нажимаем кнопку **Next.**
5. После завершения установки будет выведено окно, изображенное на рис. 1.4. Нажимаем кнопку **Finish** для выхода из программы установки.

В результате установки исходные файлы интерпретатора будут скопированы в папку C:\Python32. В этой папке расположены два исполняемых файла: python.exe и pythonw.exe. Файл python.exe предназначен для выполнения консольных приложений. Именно эта программа запускается при двойном щелчке на значке файла с расширением py. Файл pythonw.exe используется для запуска оконных приложений. В этом случае окно консоли выводиться не будет. Эта программа запускается при двойном щелчке на значке файла с расширением pyw.

Рис. 1.1. Установка Python. Шаг 1

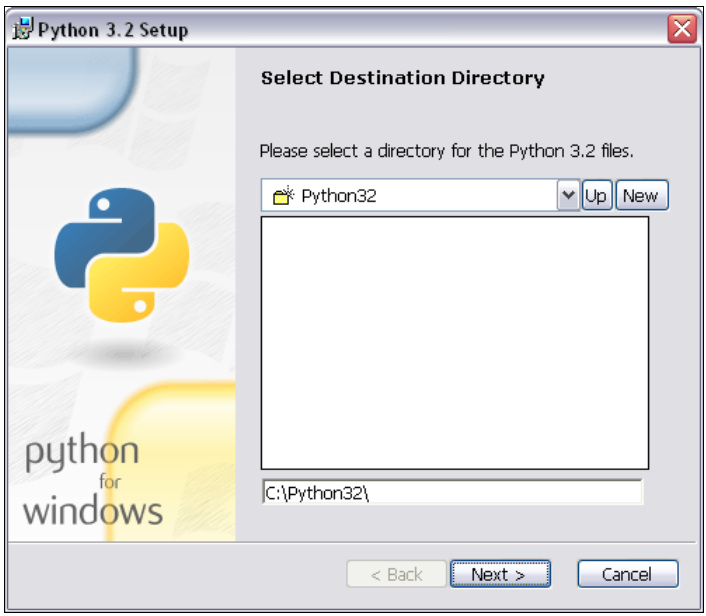

Рис. 1.2. Установка Python. Шаг 2

Если сделать двойной щелчок на файле python.exe, то запустится интерактивная оболочка в окне консоли (рис. 1.5). Символы >>> в этом окне означают приглашение для ввода инструкций на языке Python. Если после этих символов ввести, например, 2 + 2 и нажать клавишу <Enter>, то на следующей строке сразу будет выведен результат выполнения, а затем

опять приглашение для ввода новой инструкции. Таким образом, это окно можно использовать в качестве калькулятора, а также для изучения языка. Открыть такое же окно можно с помощью пункта **Python (command line)** в меню **Пуск | Программы | Python 3.2**.

Рис. 1.3. Установка Python. Шаг 3

Рис. 1.4. Установка Python. Шаг 4

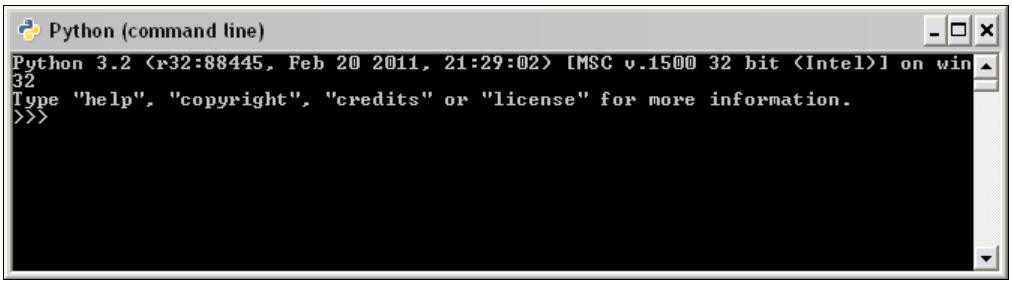

Рис. 1.5. Интерактивная оболочка

Вместо интерактивной оболочки для изучения языка, а также создания и редактирования файлов с программой, лучше воспользоваться редактором IDLE, который входит в состав установленных компонентов. Для запуска редактора в меню **Пуск | Программы | Python 3.2** выбираем пункт **IDLE (Python GUI)**. В результате откроется окно **Python Shell** (рис. 1.6), которое выполняет все функции интерактивной оболочки, но дополнительно производит подсветку синтаксиса, выводит подсказки и др. Именно этим редактором мы будем пользоваться на протяжении всей книги. Более подробно редактор IDLE мы рассмотрим немного позже.

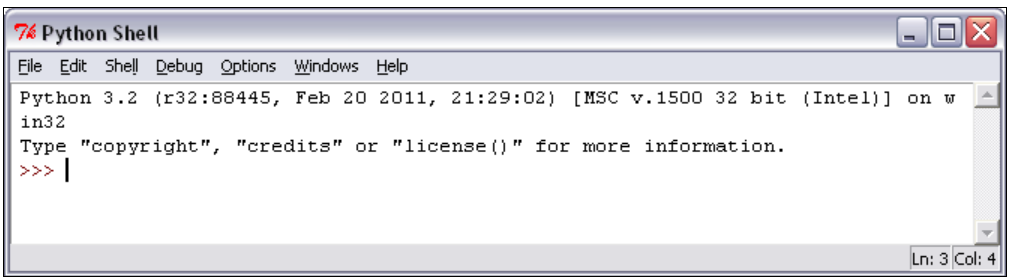

Рис. 1.6. Окно **Python Shell** редактора IDLE

Версии языка Python выпускаются с завидной регулярностью, но, к сожалению, сторонние разработчики не успевают за такой скоростью и не так часто обновляют свои модули. Поэтому приходится при наличии версии Python 3 использовать на практике версию Python 2. Как же быть, если установлена версия 3.2, а необходимо запустить модуль для версии 2.7 или 2.5? В этом случае удалять версию 3.2 с компьютера не нужно. Все программы установки позволяют выбрать устанавливаемые компоненты. Так, например, существует возможность задать ассоциацию с файловым расширением. И вот этот компонент необходимо просто отключить при установке.

В качестве примера мы дополнительно установим на компьютер версию 3.1, но вместо программы установки с сайта **http://python.org/** выберем альтернативный дистрибутив от компании ActiveState. Переходим на страницу **http://www.activestate.com/activepython/downloads/** и скачиваем дистрибутив. Последовательность запуска нескольких программ установки от компании ActiveState имеет значение, т. к. в контекстное меню добавляется пункт **Edit with Pythonwin**. С помощью этого пункта запускается редактор PythonWin, который можно использовать вместо IDLE. Так вот из контекстного меню будет открываться версия PythonWin, которая была установлена последней. Установку программы производим в каталог по умолчанию (C:\Python31\). Обратите особое внимание: при установке в окне **Custom Setup** (рис. 1.7) необходимо отключить компонент **Register as Default Python** (рис. 1.8). Не забудьте это сделать, иначе текущей версией будет не Python 3.2.

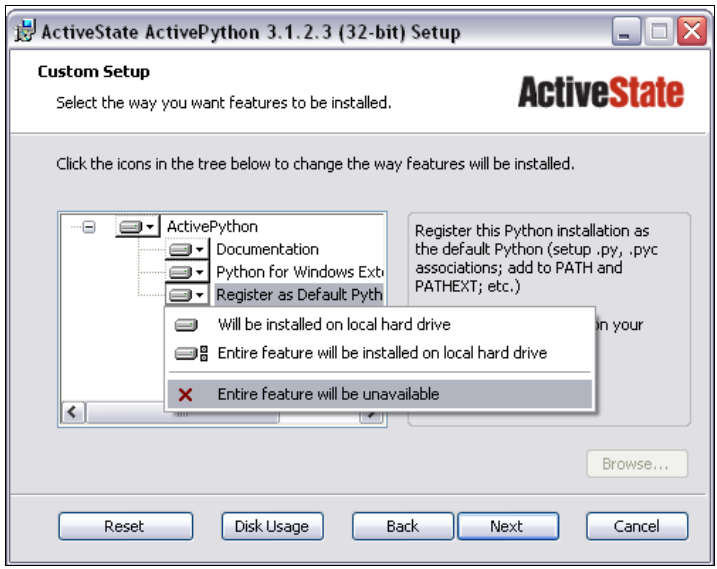

Рис. 1.7. Окно **Custom Setup**

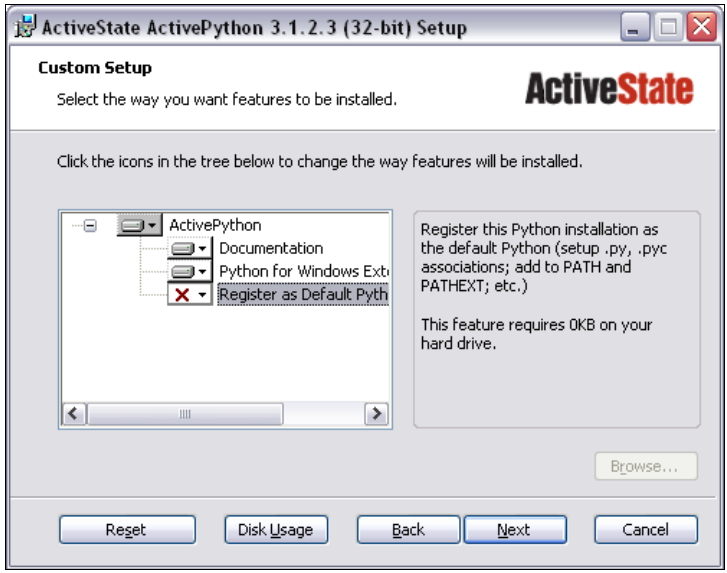

Рис. 1.8. Компонент **Register as Default Python** отключен

В состав ActivePython кроме редактора PythonWin входит также редактор IDLE. Однако ни в одном меню нет пункта, с помощью которого можно запустить редактор IDLE. Чтобы это исправить, создадим файл IDLE31.bat. Содержимое файла:

```
@echo off
start C:\Python31\pythonw.exe C:\Python31\Lib\idlelib\idle.pyw
```

С помощью двойного щелчка на этом файле можно запускать IDLE для версии Python 3.1. Чтобы запустить редактор для версии Python 3.2, в меню **Пуск | Программы | Python 3.2** выбираем пункт **IDLE (Python GUI)**.

Теперь рассмотрим запуск программы с помощью разных версий Python. По умолчанию при двойном щелчке на значке файла запускается Python 3.2. Чтобы запустить с помощью другой версии, щелкаем правой кнопкой мыши на значке файла с программой и в контекстном меню выбираем пункт **Открыть с помощью**. При первом запуске в списке будет только программа python.exe. Чтобы добавить другую версию, щелкаем на пункте **Выбрать программу**, в открывшемся окне нажимаем кнопку **Обзор** и выбираем программу python31.exe из папки C:\Python31. В результате список в контекстном меню будет выглядеть так, как показано на рис. 1.9. Выбирая нужную версию из списка, можно производить запуск программы с помощью разных версий Python.

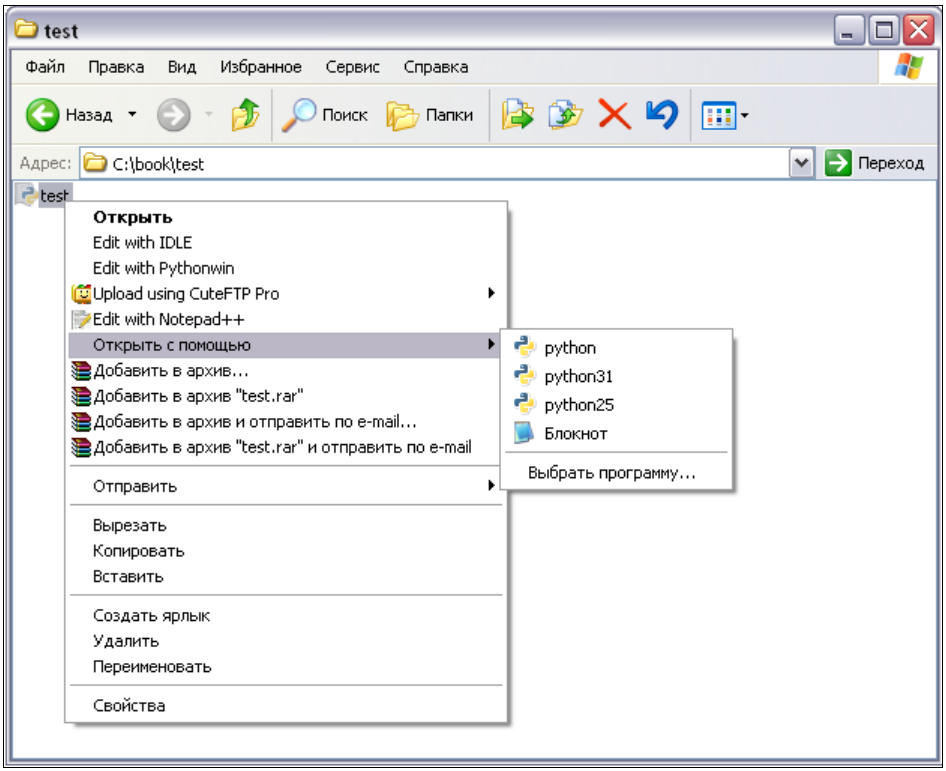

Рис. 1.9. Варианты запуска программы разными версиями Python

Для проверки установки создайте файл test.py с помощью любого текстового редактора, например Блокнота. Содержимое файла приведено в листинге 1.1.

Листинг 1.1. Проверка установки

```
import sys
print (tuple(sys.version_info))
try:
    raw_input()         # Python 2
except NameError:
    input()             # Python 3
```

Затем запустите программу с помощью двойного щелчка на значке файла. Если результат выполнения — `(3, 2, 0, 'final', 0)`, то установка прошла нормально, а если `(3, 1, 2, 'final', 0)`, то вы не отключили компонент **Register as Default Python**.

Для изучения материала этой книги по умолчанию должна запускаться версия Python 3.2.

1.2. Первая программа на Python

Изучение языков программирования принято начинать с программы, выводящей надпись "Привет, мир!". Не будем нарушать традицию и продемонстрируем, как это будет выглядеть на Python (листинг 1.2).

Листинг 1.2. Первая программа на Python

```
# Выводим надпись с помощью функции print()
print("Привет, мир!")
```

Для запуска программы в меню **Пуск** выбираем пункт **Программы | Python 3.2 | IDLE (Python GUI)**. В результате откроется окно **Python Shell**, в котором символы >>> означают приглашение ввести команду. Вводим сначала первую строку из листинга 1.2, а затем вторую. После ввода строки нажимаем клавишу <Enter>. На следующей строке сразу отобразится результат, а далее приглашение для ввода новой команды. Последовательность выполнения нашей программы показана в листинге 1.3.

Листинг 1.3. Последовательность выполнения программы в окне *Python Shell*

```
>>> # Выводим надпись с помощью функции print()
>>> print("Привет, мир!")
Привет, мир!
>>>
```

> ***Примечание***
>
> Символы >>> вводить не нужно, они вставляются автоматически.

Для создания файла с программой в меню **File** выбираем пункт **New Window**. В открывшемся окне набираем код из листинга 1.2, а затем сохраняем его под именем test.py, выбрав пункт меню **File | Save As**. При этом редактор сохранит файл в кодировке UTF-8 без BOM (*Byte Order Mark*, метка порядка байтов). Именно кодировка UTF-8 является кодировкой по умолчанию в Python 3. Если файл содержит инструкции в другой кодировке, то необходимо в первой или второй строке указать кодировку с помощью инструкции:

```
# -*- coding: <Кодировка> -*-
```

Например, для кодировки Windows-1251 инструкция будет выглядеть так:

```
# -*- coding: cp1251 -*-
```

Редактор IDLE учитывает указанную кодировку и автоматически производит перекодирование при сохранении файла. При использовании других редакторов следует проконтролировать соответствие указанной кодировки и реальной кодировки файла. Если кодировки не совпадают, то данные будут преобразованы некорректно или во время преобразования произойдет ошибка.

Запустить программу на выполнение можно, выбрав пункт меню **Run | Run Module** или нажав клавишу <F5>. Результат выполнения программы будет отображен в окне **Python Shell**.

Запустить программу можно также с помощью двойного щелчка мыши на значке файла. В этом случае результат выполнения будет отображен в консоли Windows. Следует учитывать, что после вывода результата окно консоли сразу закрывается. Чтобы предотвратить закрытие окна, необходимо добавить вызов функции `input()`, которая будет ожидать нажатия клавиши <Enter> и не позволит окну сразу закрыться. С учетом сказанного ранее наша программа будет выглядеть так, как показано в листинге 1.4.

Листинг 1.4. Программа для запуска с помощью двойного щелчка мыши

```
# -*- coding: utf-8 -*-
print("Привет, мир!")          # Выводим строку
input()                        # Ожидаем нажатия клавиши <Enter>
```

> ***Примечание***
>
> Если до функции `input()` возникнет ошибка, то сообщение о ней будет выведено в консоль, но сама консоль после этого сразу будет закрыта, и вы не сможете прочитать сообщение об ошибке. Если подобная ситуация возникла, то запустите программу из командной строки или с помощью редактора IDLE и вы сможете прочитать сообщение об ошибке.

В языке Python 3 строки по умолчанию хранятся в кодировке Unicode. При выводе кодировка Unicode автоматически преобразуется в кодировку терминала. Поэтому русские буквы отображаются корректно, хотя в окне консоли в Windows по умолчанию используется кодировка cp866, а файл с программой у нас в кодировке UTF-8.

Чтобы отредактировать уже созданный файл, щелкаем правой кнопкой мыши на значке файла и из контекстного меню выбираем пункт **Edit with IDLE**. Так как программа на языке Python представляет собой обычный текстовый файл, сохраненный с расширением py или pyw, его можно редактировать с помощью других программ, например Notepad++. Можно также воспользоваться специализированными редакторами, скажем, PyScripter.

1.3. Структура программы

Как вы уже знаете, программа на языке Python представляет собой обычный текстовый файл с инструкциями. Каждая инструкция располагается на отдельной строке. Если инструкция не является вложенной, то она должна начинаться с начала строки, иначе будет выведено сообщение об ошибке (листинг 1.5).

Листинг 1.5. Ошибка `SyntaxError`

```
>>>  import sys

SyntaxError: unexpected indent
>>>
```

В этом случае перед инструкцией `import` расположен один лишний пробел, который привел к выводу сообщения об ошибке.

Если программа предназначена для исполнения в операционной системе UNIX, то в первой строке необходимо дополнительно указать путь к интерпретатору Python:

```
#!/usr/bin/python
```

В некоторых операционных системах путь к интерпретатору выглядит по-другому:

```
#!/usr/local/bin/python
```

Иногда можно не указывать точный путь к интерпретатору, а передать название языка программе `env`:

```
#!/usr/bin/env python
```

В этом случае программа `env` произведет поиск интерпретатора Python в соответствии с настройками путей поиска.

Помимо указания пути к интерпретатору Python необходимо, чтобы был установлен бит на выполнение в правах доступа к файлу. Кроме того, следует помнить, что перевод строки в операционной системе Windows состоит из последовательности двух символов — `\r` (перевод каретки) и `\n` (перевод строки). В операционной системе UNIX перевод строки осуществляется только одним символом `\n`. Если загрузить файл программы по протоколу FTP в бинарном режиме, то символ `\r` вызовет фатальную ошибку. По этой причине файлы по протоколу FTP следует загружать только в текстовом режиме (режим `ASCII`). В этом режиме символ `\r` будет удален автоматически. После загрузки файла следует установить права на выполнение. Для исполнения скриптов на Python устанавливаем права в 755 (`-rwxr-xr-x`).

Во второй строке (для ОС Windows в первой строке) следует указать кодировку. Если кодировка не указана, то предполагается, что файл сохранен в кодировке UTF-8. Для кодировки Windows-1251 строка будет выглядеть так:

```
# -*- coding: cp1251 -*-
```

Редактор IDLE учитывает указанную кодировку и автоматически производит перекодирование при сохранении файла. Получить полный список поддерживаемых кодировок и их псевдонимы позволяет код, приведенный в листинге 1.6.

Листинг 1.6. Вывод списка поддерживаемых кодировок

```
# -*- coding: utf-8 -*-
import encodings.aliases
arr = encodings.aliases.aliases
keys = list( arr.keys() )
keys.sort()
for key in keys:
    print("%s => %s" % (key, arr[key]))
```

Во многих языках программирования (например, в PHP, Perl и др.) каждая инструкция должна завершаться точкой с запятой. В языке Python в конце инструкции также можно поставить точку с запятой, но это не обязательно. В отличие от языка JavaScript, где рекомендуется завершать инструкции точкой с запятой, в языке Python точку с запятой ставить *не рекомендуется*. Концом инструкции является конец строки. Тем не менее, если необходимо разместить несколько инструкций на одной строке, то точку с запятой *следует указать* (листинг 1.7).

Листинг 1.7. Несколько инструкций на одной строке

```
>>> x = 5; y = 10; z = x + y # Три инструкции на одной строке
>>> print(z)
15
```

Еще одной отличительной особенностью языка Python является отсутствие ограничительных символов для выделения инструкций внутри блока. Например, в языке PHP инструкции внутри цикла `while` выделяются фигурными скобками:

```
$i = 1;
while ($i<11) {
   echo $i . "\n";
   $i++;
}
echo "Конец программы";
```

В языке Python тот же код будет выглядеть по-другому (листинг 1.8).

Листинг 1.8. Выделение инструкций внутри блока

```
i = 1
while i<11:
    print(i)
    i += 1
print("Конец программы")
```

Обратите внимание, что перед всеми инструкциями внутри блока расположено одинаковое количество пробелов. Таким образом в языке Python выделяются блоки. Инструкции, перед которыми расположено одинаковое количество пробелов, являются телом блока. В нашем примере две инструкции выполняются десять раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция `print()`, которая выводит строку "Конец программы". Если количество пробелов внутри блока будет разным, то интерпретатор выведет сообщение о фатальной ошибке и программа будет остановлена. Так язык Python приучает программистов писать красивый и понятный код.

> ***Примечание***
>
> В языке Python принято использовать четыре пробела для выделения инструкций внутри блока.

Если блок состоит из одной инструкции, то допустимо разместить ее на одной строке с основной инструкцией. Например, код

```
for i in range(1, 11):
    print(i)
print("Конец программы")
```

можно записать так:

```
for i in range(1, 11): print(i)
print("Конец программы")
```

Если инструкция является слишком длинной, то ее можно перенести на следующую строку нижеприведенными способами:

- в конце строки разместить символ \. После этого символа должен следовать символ перевода строки. Другие символы (в том числе и комментарии) недопустимы. Пример:

```
x = 15 + 20 \
    + 30
print(x)
```

- поместить выражение внутри круглых скобок. Это лучший способ, т. к. любое выражение можно разместить внутри круглых скобок. Пример:

```
x = (15 + 20                # Это комментарий
    + 30)
print(x)
```

- определение списка и словаря можно разместить на нескольких строках, т. к. используются квадратные и фигурные скобки соответственно. Пример определения списка:

```
arr = [15, 20,              # Это комментарий
       30]
print(arr)
```

Пример определения словаря:

```
arr = {"x": 15, "y": 20,    # Это комментарий
       "z": 30}
print(arr)
```

1.4. Комментарии

Комментарии предназначены для вставки пояснений в текст скрипта, и интерпретатор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая инструкции, которые выполнять не следует. Помните, комментарии нужны программисту, а не интерпретатору Python. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке Python присутствует только *однострочный комментарий*. Он начинается с символа #:

```
# Это комментарий
```

Однострочный комментарий может начинаться не только с начала строки, но и располагаться после инструкции. Например, после инструкции вывести надпись "Привет, мир!":

```
print("Привет, мир!") # Выводим надпись с помощью функции print()
```

Если символ комментария разместить перед инструкцией, то она не будет выполнена:

```
# print("Привет, мир!") Эта инструкция выполнена не будет
```

Если символ # расположен внутри кавычек или апострофов, то он не является символом комментария:

```
print("# Это НЕ комментарий")
```

Так как в языке Python нет многострочного комментария, то часто комментируемый фрагмент размещают внутри утроенных кавычек (или утроенных апострофов):

```
"""
Эта инструкция выполнена не будет
print("Привет, мир!")
"""
```

Следует заметить, что этот фрагмент кода не игнорируется интерпретатором, т. к. он не является комментарием. В результате выполнения фрагмента будет создан объект строкового типа. Тем не менее инструкции внутри утроенных кавычек выполнены не будут, т. к. все инструкции будут считаться простым текстом. Такие строки являются строками документирования, а не комментариями.

1.5. Скрытые возможности IDLE

На всем протяжении этой книги в качестве редактора мы будем использовать IDLE. По этой причине рассмотрим некоторые возможности этой среды разработки.

Как вы уже знаете, в окне **Python Shell** символы >>> означают приглашение ввести команду. После ввода команды нажимаем клавишу <Enter>. На следующей строке сразу отобразится результат (при условии, что инструкция возвращает значение), а далее — приглашение для ввода новой команды. При вводе многострочной команды после нажатия клавиши <Enter> редактор автоматически вставит отступ и будет ожидать дальнейшего ввода. Чтобы сообщить редактору о конце ввода команды, необходимо дважды нажать клавишу <Enter>. Пример:

```
>>> for n in range(1, 3):
        print(n)


1
2
>>>
```

В предыдущем разделе мы выводили строку "Привет, мир!" с помощью функции `print()`. В окне **Python Shell** это делать не обязательно. Например, мы можем просто ввести строку и нажать клавишу <Enter> для получения результата:

```
>>> "Привет, мир!"
'Привет, мир!'
>>>
```

Обратите внимание на то, что строки выводятся в апострофах. Этого не произойдет, если выводить строку с помощью функции `print()`:

```
>>> print("Привет, мир!")
Привет, мир!
>>>
```

Учитывая возможность получить результат сразу после ввода команды, окно **Python Shell** можно использовать для изучения команд, а также в качестве многофункционального калькулятора. Пример:

```
>>> 12 * 32 + 54
438
>>>
```

Результат вычисления последней инструкции сохраняется в переменной _ (одно подчеркивание). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата. Вместо него достаточно ввести символ подчеркивания. Пример:

```
>>> 125 * 3            # Умножение
375
>>> _ + 50             # Сложение. Эквивалентно 375 + 50
425
>>> _ / 5              # Деление. Эквивалентно 425 / 5
85.0
>>>
```

При вводе команды можно воспользоваться комбинацией клавиш <Ctrl>+<Пробел>. В результате будет отображен список, из которого можно выбрать нужный идентификатор. Если при открытом списке вводить буквы, то показываться будут идентификаторы, начинающиеся с этих букв. Выбирать идентификатор необходимо с помощью клавиш <↑> и <↓>. После выбора не следует нажимать клавишу <Enter>, иначе это приведет к выполнению инструкции. Просто вводите инструкцию дальше, а список закроется. Такой же список будет автоматически появляться (с некоторой задержкой) при обращении к атрибутам объекта или модуля после ввода точки. Для автоматического завершения идентификатора после ввода первых букв можно воспользоваться комбинацией клавиш <Alt>+</>. При каждом последующем нажатии этой комбинации будет вставляться следующий идентификатор. Эти две комбинации клавиш очень удобны, если вы забыли, как пишется слово, или хотите, чтобы редактор закончил его за вас.

При необходимости повторно выполнить ранее введенную инструкцию ее приходится набирать заново. Можно, конечно, скопировать инструкцию, а затем вставить, но как вы можете сами убедиться, в контекстном меню нет пунктов **Copy** (Копировать) и **Paste** (Вставить). Они расположены в меню **Edit**. Постоянно выбирать пункты из этого меню очень неудобно. Одним из решений проблемы является использование комбинации клавиш быстрого доступа — <Ctrl>+<C> (копировать) и <Ctrl>+<V> (вставить). Комбинации стандартны для Windows, и вы наверняка их уже использовали ранее. Но опять-таки, прежде чем скопировать инструкцию, ее предварительно необходимо выделить. Редактор IDLE избавляет нас от лишних действий и предоставляет комбинацию клавиш <Alt>+<N> для вставки первой введенной инструкции, а также комбинацию <Alt>+<P> для вставки последней инструкции. Каждое последующее нажатие этих клавиш будет вставлять следующую (или предыдущую) инструкцию. Для еще более быстрого повторного ввода инструкции следует предварительно ввести ее первые буквы. В этом случае перебираться будут только инструкции, начинающиеся с этих букв.

1.6. Вывод результатов работы программы

Вывести результаты работы программы можно с помощью функции `print()`. Функция имеет следующий формат:

```
print([<Объекты>][, sep=' '][, end='\n'][, file=sys.stdout])
```

Функция `print()` преобразует объект в строку и посылает ее в стандартный вывод `stdout`. С помощью параметра `file` можно перенаправить вывод в другое место, например в файл. Перенаправление вывода мы подробно рассмотрим при изучении файлов.

После вывода строки автоматически добавляется символ перевода строки:

```
print("Строка 1")
print("Строка 2")
```

Результат:

```
Строка 1
Строка 2
```

Если необходимо вывести результат на той же строке, то в функции `print()` данные указываются через запятую в первом параметре:

```
print("Строка 1", "Строка 2")
```

Результат:

```
Строка 1 Строка 2
```

Как видно из примера, между выводимыми строками автоматически вставляется пробел. С помощью параметра `sep` можно указать другой символ. Выведем строки без пробела между ними:

```
print("Строка1", "Строка2", sep="")
```

Результат:

```
Строка 1Строка 2
```

После вывода объектов в конце добавляется символ перевода строки. Если необходимо произвести дальнейший вывод на той же строке, то в параметре `end` следует указать другой символ:

```
print("Строка 1", "Строка 2", end=" ")
print("Строка 3")
# Выведет: Строка 1 Строка 2 Строка 3
```

Если наоборот необходимо вставить символ перевода строки, то функция `print()` указывается без параметров. Пример:

```
for n in range(1, 5):
    print(n, end=" ")
print()
print("Это текст на новой строке")
```

Результат выполнения:

```
1 2 3 4
Это текст на новой строке
```

В этом примере мы использовали цикл `for`, который позволяет последовательно перебирать элементы. На каждой итерации цикла переменной `n` присваивается новое число, которое мы выводим с помощью функции `print()`, расположенной на следующей строке. Обратите внимание, что перед функцией мы добавили четыре пробела. Таким образом в языке Python выделяются *блоки*. Инструкции, перед которыми расположено одинаковое количество пробелов, являются *телом цикла*. Все эти инструкции выполняются определенное количество раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция `print()` без параметров, которая вставляет символ перевода строки.

Если необходимо вывести большой блок текста, то его следует разместить между утроенными кавычками или утроенными апострофами. В этом случае текст сохраняет свое форматирование. Пример:

```
print("""Строка 1
Строка 2
Строка 3""")
```

В результате выполнения этого примера мы получим три строки:

```
Строка 1
Строка 2
Строка 3
```

Для вывода результатов работы программы вместо функции `print()` можно использовать метод `write()` объекта `sys.stdout`:

```
import sys                      # Подключаем модуль sys
sys.stdout.write("Строка")      # Выводим строку
```

В первой строке, с помощью оператора `import`, мы подключаем модуль `sys`, в котором объявлен объект. Далее с помощью метода `write()` выводим строку. Следует заметить, что метод не вставляет символ перевода строки. Поэтому при необходимости следует добавить его самим с помощью символа `\n`:

```
import sys
sys.stdout.write("Строка 1\n")
sys.stdout.write("Строка 2")
```

1.7. Ввод данных

Для ввода данных в Python 3 предназначена функция `input()`, которая получает данные со стандартного ввода `stdin`. Функция имеет следующий формат:

```
[<Значение> = ] input([<Сообщение>])
```

Для примера переделаем нашу первую программу так, чтобы она здоровалась не со всем миром, а только с нами (листинг 1.9).

Листинг 1.9. Пример использования функции `input()`

```
# -*- coding: utf-8 -*-
name = input("Введите ваше имя: ")
name = name.rstrip("\r")                    # Для версии 3.2.0
print("Привет,", name)
input("Нажмите <Enter> для закрытия окна")
```

> ***Примечание***
>
> В версии 3.2.0 функция `input()` при вводе данных в консоле Windows помимо самих данных дополнительно возвращает символ возврата каретки `\r`. Чтобы избавиться от символа, следует удалить его с помощью метода `rstrip()`. В версии 3.2.1 ошибка была исправлена.

Вводим код и сохраняем файл, например, под названием test.py, а затем запускаем программу на выполнение с помощью двойного щелчка на значке файла. Откроется черное

окно, в котором будет надпись "Введите ваше имя: ". Вводим свое имя, например "Николай", и нажимаем клавишу <Enter>. В результате будет выведено приветствие "Привет, Николай". Чтобы окно сразу не закрылось, повторно вызываем функцию `input()`. В этом случае окно не закроется, пока не будет нажата клавиша <Enter>.

При использовании функции `input()` следует учитывать, что при достижении конца файла или при нажатии комбинации клавиш <Ctrl>+<Z>, а затем <Enter> генерируется исключение `EOFError`. Если не предусмотреть обработку исключения, то программа аварийно завершится. Обработать исключение можно следующим образом:

```
try:
    s = input("Введите данные: ")
    print(s)
except EOFError:
    print("Обработали исключение EOFError")
```

Если внутри блока `try` возникнет исключение `EOFError`, то управление будет передано в блок `except`. После исполнения инструкций в блоке `except` программа нормально продолжит работу.

В Python 2 для ввода данных применялись две функции: `raw_input()` и `input()`. Функция `raw_input()` просто возвращала введенные данные, а функция `input()` предварительно обрабатывала данные с помощью функции `eval()` и затем возвращала результат ее выполнения. В Python 3 функция `raw_input()` была переименована в `input()`, а прежняя функция `input()` была удалена. Чтобы вернуться к поведению функции `input()` в Python 2, необходимо передать значение в функцию `eval()` явным образом:

```
# -*- coding: utf-8 -*-
result = eval(input("Введите инструкцию: ")) # Вводим:  2 + 2
print("Результат:", result)                  # Выведет: 4
input()
```

> ***Внимание!***
>
> Функция `eval()` выполнит любую введенную инструкцию. Никогда не используйте этот код, если не доверяете пользователю.

Передать данные можно в командной строке после названия файла. Такие данные доступны через список `argv` модуля `sys`. Первый элемент списка `argv` будет содержать название файла, а последующие элементы — переданные данные. В качестве примера создадим файл test.py в папке C:\book. Содержимое файла приведено в листинге 1.10.

Листинг 1.10. Получение данных из командной строки

```
# -*- coding: utf-8 -*-
import sys
arr = sys.argv[:]
for n in arr:
    print(n)
```

Теперь запустим программу на выполнение из командной строки и передадим данные. Запускаем командную строку. Для этого в меню **Пуск** выбираем пункт **Выполнить**. В открывшемся окне набираем команду `cmd` и нажимаем кнопку **OK**. Откроется черное окно,

в котором будет приглашение для ввода команд. Переходим в папку C:\book. Для этого набираем команду:

```
cd C:\book
```

В командной строке должно быть приглашение:

```
C:\book>
```

Для запуска нашей программы вводим команду:

```
C:\Python32\python.exe test.py -uNik -p123
```

В этой команде мы передаем название файла (test.py) и некоторые данные (-uNik -p123). Результат выполнения программы будет выглядеть так:

```
test.py
-uNik
-p123
```

1.8. Доступ к документации

Вместе с установкой интерпретатора Python на компьютер автоматически копируется документация в формате CHM. Чтобы отобразить документацию, в меню **Пуск** выбираем пункт **Программы | Python 3.2 | Python Manuals**.

Если в меню **Пуск** выбрать пункт **Программы | Python 3.2 | Module Docs**, то откроется окно **pydoc** (рис. 1.10). С помощью этого инструмента можно получить дополнительную информацию, которая расположена внутри модулей. Чтобы получить список всех модулей, установленных на компьютере, оставляем текстовое поле пустым и нажимаем кнопку **open browser**. В результате в окне Web-браузера, используемого в системе по умолчанию, отобразится список всех модулей. Каждое название модуля является ссылкой, при переходе по которой доступна документация по конкретному модулю. Если в окне **pydoc** в текстовом поле ввести какое-либо название и нажать клавишу <Enter>, то будет отображен список совпадений. Выделяем пункт в списке и нажимаем кнопку **go to selected**. Результат будет отображен в окне Web-браузера.

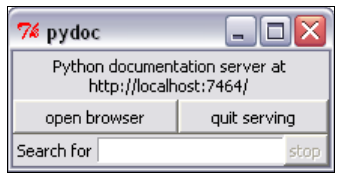

Рис. 1.10. Окно **pydoc**

В окне **Python Shell** редактора IDLE также можно отобразить документацию. Для этого предназначена функция `help()`. В качестве примера отобразим документацию по встроенной функции `input()`:

```
>>> help(input)
```

Результат выполнения:

```
Help on built-in function input in module builtins:

input(...)
    input([prompt]) -> string
```

```
    Read a string from standard input.  The trailing newline is stripped.
    If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
    On Unix, GNU readline is used if enabled.  The prompt string, if given,
    is printed without a trailing newline before reading.
```

С помощью функции `help()` можно получить документацию не только по конкретной функции, но и по всему модулю сразу. Для этого предварительно необходимо подключить модуль. Например, подключим модуль `builtins`, содержащий определения всех встроенных функций и классов, а затем выведем документацию по модулю:

```
>>> import builtins
>>> help(builtins)
```

При рассмотрении комментариев, мы говорили, что часто для комментирования большого фрагмента кода используются утроенные кавычки или утроенные апострофы. Такие строки не являются комментариями в полном смысле этого слова. Вместо комментирования фрагмента создается объект строкового типа, который сохраняется в атрибуте `__doc__`. Функция `help()` при составлении документации получает информацию из этого атрибута. Такие строки называются *строками документирования*.

В качестве примера создадим два файла в одной папке. Содержимое файла test.py:

```
# -*- coding: utf-8 -*-
""" Это описание нашего модуля """
def func():
    """ Это описание функции"""
    pass
```

Теперь подключим этот модуль и выведем содержимое строк документирования:

```
# -*- coding: utf-8 -*-
import test                         # Подключаем файл test.py
help(test)
```

Результат выполнения:

```
Help on module test:

NAME
    test — Это описание нашего модуля

FUNCTIONS
    func()
        Это описание функции

FILE
    c:\book\test.py
```

Теперь получим содержимое строк документирования с помощью атрибута `__doc__`:

```
# -*- coding: utf-8 -*-
import test                         # Подключаем файл test.py
print(test.__doc__)
print(test.func.__doc__)
```

Результат выполнения:

```
Это описание нашего модуля
Это описание функции
```

Атрибут `__doc__` можно использовать вместо функции `help()`. В качестве примера получим документацию по функции `input()`:

```
>>> print(input.__doc__)
```

Результат выполнения:

```
input([prompt]) -> string

Read a string from standard input.  The trailing newline is stripped.
If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
On Unix, GNU readline is used if enabled.  The prompt string, if given,
is printed without a trailing newline before reading.
```

Получить список всех идентификаторов внутри модуля позволяет функция `dir()`:

```
# -*- coding: utf-8 -*-
import test                         # Подключаем файл test.py
print( dir(test) )
```

Результат выполнения:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
'__package__', 'func']
```

Теперь получим список всех встроенных идентификаторов:

```
>>> import builtins
>>> dir(builtins)
```

Функция `dir()` может не принимать параметров вообще. В этом случае возвращается список идентификаторов текущего модуля:

```
# -*- coding: utf-8 -*-
import test                         # Подключаем файл test.py
print( dir() )
```

Результат выполнения:

```
['__builtins__', '__doc__', '__name__', '__package__', 'test']
```

ГЛАВА 2

Переменные

Все данные в языке Python представлены объектами. Каждый объект имеет тип данных и значение. Для доступа к объекту предназначены *переменные*. При инициализации в переменной сохраняется ссылка (адрес объекта в памяти компьютера) на объект. Благодаря этой ссылке можно в дальнейшем изменять объект из программы.

2.1. Именование переменных

Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знаков подчеркивания, причем имя переменной не может начинаться с цифры. Кроме того, следует избегать указания символа подчеркивания в начале имени, т. к. идентификаторы с таким символом имеют специальное значение. Например, имена, начинающиеся с символа подчеркивания, не импортируются из модуля с помощью инструкции `from module import *`, а имена, имеющие по два символа подчеркивания в начале и конце, для интерпретатора имеют особый смысл.

В качестве имени переменной нельзя использовать ключевые слова. Получить список всех ключевых слов позволяет код, приведенный в листинге 2.1.

Листинг 2.1. Список всех ключевых слов

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами. В отличие от ключевых слов, встроенные идентификаторы можно переопределять, но дальнейший результат может стать для вас неожиданным (листинг 2.2).

Листинг 2.2. Ошибочное переопределение встроенных идентификаторов

```
>>> help(abs)
Help on built-in function abs in module builtins:
```

```
abs(...)
    abs(number) -> number

    Return the absolute value of the argument.

>>> help = 10
>>> help
10
>>> help(abs)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    help(abs)
TypeError: 'int' object is not callable
```

В этом примере мы с помощью встроенной функции `help()` получаем справку по функции `abs()`. Далее переменной `help` присваиваем число 10. После переопределения идентификатора мы больше не можем пользоваться функцией `help()`, т. к. это приведет к выводу сообщения об ошибке. По этой причине лучше избегать имен, совпадающих со встроенными идентификаторами. Очень часто подобная ошибка возникает при попытке назвать переменную, в которой предполагается хранение строки, именем `str`. Вроде бы логично, но `str` является часто используемым встроенным идентификатором и после такого переопределения поведение программы становится непредсказуемым. В редакторе IDLE встроенные идентификаторы подсвечиваются фиолетовым цветом. Обращайте внимание на цвет переменной, он должен быть черным. Если вы заметили, что переменная подсвечена, то название переменной следует обязательно изменить. Получить полный список встроенных идентификаторов позволяет код, приведенный в листинге 2.3.

Листинг 2.3. Получение списка встроенных идентификаторов

```
>>> import builtins
>>> dir(builtins)
```

Правильные имена переменных: `x`, `y1`, `strName`, `str_name`.

Неправильные имена переменных: `1y`, `ИмяПеременной`.

Последнее имя неправильное, т. к. в нем используются русские буквы. Хотя на самом деле такой вариант также будет работать, но лучше русские буквы все же не применять:

```
>>> ИмяПеременной = 10              # Лучше так не делать!!!
>>> ИмяПеременной
10
```

При указании имени переменной важно учитывать регистр букв: `x` и `X` — разные переменные:

```
>>> x = 10; X = 20
>>> x, X
(10, 20)
```

2.2. Типы данных

В Python 3 объекты могут иметь следующие типы данных:

- `bool` — логический тип данных. Может содержать значения `True` или `False`, которые ведут себя как числа `1` и `0` соответственно:

```
>>> type(True), type(False)
(<class 'bool'>, <class 'bool'>)
>>> int(True), int(False)
(1, 0)
```

- `NoneType` — объект со значением `None` (обозначает отсутствие значения):

```
>>> type(None)
<class 'NoneType'>
```

  В логическом контексте значение `None` интерпретируется как `False`:

```
>>> bool(None)
False
```

- `int` — целые числа. Размер числа ограничен лишь объемом оперативной памяти:

```
>>> type(2147483647), type(99999999999999999999999)
(<class 'int'>, <class 'int'>)
```

- `float` — вещественные числа:

```
>>> type(5.1), type(8.5e-3)
(<class 'float'>, <class 'float'>)
```

- `complex` — комплексные числа:

```
>>> type(2+2j)
<class 'complex'>
```

- `str` — Unicode-строки:

```
>>> type("Строка")
<class 'str'>
```

- `bytes` — неизменяемая последовательность байтов:

```
>>> type(bytes("Строка", "utf-8"))
<class 'bytes'>
```

- `bytearray` — изменяемая последовательность байтов:

```
>>> type(bytearray("Строка", "utf-8"))
<class 'bytearray'>
```

- `list` — списки. Тип данных `list` аналогичен массивам в других языках программирования:

```
>>> type( [1, 2, 3] )
<class 'list'>
```

- `tuple` — кортежи:

```
>>> type( (1, 2, 3) )
<class 'tuple'>
```

- `dict` — словари. Тип данных `dict` аналогичен ассоциативным массивам в других языках программирования:

```
>>> type( {"x": 5, "y": 20} )
<class 'dict'>
```

- `set` — множества (коллекции уникальных объектов):

```
>>> type( {"a", "b", "c"} )
<class 'set'>
```

- `frozenset` — неизменяемые множества:

```
>>> type(frozenset(["a", "b", "c"]))
<class 'frozenset'>
```

- `ellipsis` — обозначается в виде трех точек или слова `Ellipsis`. Тип `ellipsis` используется в расширенном синтаксисе получения среза:

```
>>> type(...), ..., ... is Ellipsis
(<class 'ellipsis'>, Ellipsis, True)
>>> class C():
        def __getitem__(self, obj): return obj

>>> c = C()
>>> c[..., 1:5, 0:9:1, 0]
(Ellipsis, slice(1, 5, None), slice(0, 9, 1), 0)
```

- `function` — функции:

```
>>> def func(): pass

>>> type(func)
<class 'function'>
```

- `module` — модули:

```
>>> import sys
>>> type(sys)
<class 'module'>
```

- `type` — классы и типы данных. Не удивляйтесь! Все данные в языке Python являются объектами, даже сами типы данных!

```
>>> class C: pass

>>> type(C)
<class 'type'>
>>> type(type(""))
<class 'type'>
```

Основные типы данных делятся на *изменяемые* и *неизменяемые*. К изменяемым типам относятся списки, словари и тип `bytearray`. Пример изменения элемента списка:

```
>>> arr = [1, 2, 3]
>>> arr[0] = 0                  # Изменяем первый элемент списка
>>> arr
[0, 2, 3]
```

К неизменяемым типам относятся числа, строки, кортежи и тип `bytes`. Например, чтобы получить строку из двух других строк, необходимо использовать операцию *конкатенации*, а ссылку на новый объект присвоить переменной:

```
>>> str1 = "авто"
>>> str2 = "транспорт"
>>> str3 = str1 + str2          # Конкатенация
>>> print(str3)
автотранспорт
```

Кроме того, типы данных делятся на *последовательности* и *отображения*. К последовательностям относятся строки, списки, кортежи, типы `bytes` и `bytearray`, а к отображениям — словари.

Последовательности и отображения поддерживают механизм итераторов, позволяющий произвести обход всех элементов с помощью метода `__next__()` или функции `next()`. Например, вывести элементы списка можно так:

```
>>> arr = [1, 2]
>>> i = iter(arr)
>>> i.__next__()              # Метод __next__()
1
>>> next(i)                   # Функция next()
2
```

Если используется словарь, то на каждой итерации возвращается ключ:

```
>>> d = {"x": 1, "y": 2}
>>> i = iter(d)
>>> i.__next__()              # Возвращается ключ
'y'
>>> d[i.__next__()]           # Получаем значение по ключу
1
```

На практике подобным способом не пользуются. Вместо него применяется цикл `for`, который использует механизм итераторов незаметно для нас. Например, вывести элементы списка можно так:

```
>>> for i in [1, 2]:
        print(i)
```

Перебрать слово по буквам можно точно так же. Для примера вставим тире после каждой буквы:

```
>>> for i in "Строка":
        print(i + " -", end=" ")
```

Результат:

```
С - т - р - о - к - а -
```

Пример перебора элементов словаря:

```
>>> d = {"x": 1, "y": 2}
>>> for key in d:
        print( d[key] )
```

Последовательности поддерживают также обращение к элементу по индексу, получение среза, конкатенацию (оператор +), повторение (оператор *) и проверку на вхождение (оператор in). Все эти операции мы будем подробно рассматривать по мере изучения языка.

2.3. Присваивание значения переменным

В языке Python используется *динамическая типизация*. Это означает, что при присваивании переменной значения интерпретатор автоматически относит переменную к одному из типов данных. Значение переменной присваивается с помощью оператора = таким образом:

```
>>> x = 7           # Тип int
>>> y = 7.8         # Тип float
>>> s1 = "Строка"   # Переменной s1 присвоено значение Строка
>>> s2 = 'Строка'   # Переменной s2 также присвоено значение Строка
>>> b = True        # Переменной b присвоено логическое значение True
```

В одной строке можно присвоить значение сразу нескольким переменным:

```
>>> x = y = 10
>>> x, y
(10, 10)
```

После присваивания значения в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел, строк и кортежей, но для изменяемых объектов этого делать нельзя. Пример:

```
>>> x = y = [1, 2]          # Якобы создали два объекта
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов и присвоили значение переменным x и y. Теперь попробуем изменить значение в переменной y:

```
>>> y[1] = 100              # Изменяем второй элемент
>>> x, y
([1, 100], [1, 100])
```

Как видно из примера, изменение значения в переменной y привело также к изменению значения в переменной x. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> y[1] = 100              # Изменяем второй элемент
>>> x, y
([1, 2], [1, 100])
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор is. Если переменные ссылаются на один и тот же объект, то оператор is возвращает значение True:

```
>>> x = y = [1, 2]          # Один объект
>>> x is y
```

```
True
>>> x = [1, 2]              # Разные объекты
>>> y = [1, 2]              # Разные объекты
>>> x is y
False
```

Следует заметить, что в целях эффективности кода интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то в этих переменных будет сохранена ссылка на один и тот же объект. Пример:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
```

Посмотреть количество ссылок на объект позволяет метод `getrefcount()` из модуля `sys`:

```
>>> import sys              # Подключаем модуль sys
>>> sys.getrefcount(2)
304
```

Когда число ссылок на объект становится равно нулю, объект автоматически удаляется из оперативной памяти. Исключением являются объекты, которые подлежат кэшированию.

Помимо группового присваивания язык Python поддерживает позиционное присваивание. В этом случае переменные указываются через запятую слева от оператора =, а значения — через запятую справа. Пример позиционного присваивания:

```
>>> x, y, z = 1, 2, 3
>>> x, y, z
(1, 2, 3)
```

С помощью позиционного присваивания можно поменять значения переменных местами. Пример:

```
>>> x, y = 1, 2; x, y
(1, 2)
>>> x, y = y, x; x, y
(2, 1)
```

По обе стороны оператора = могут быть указаны последовательности. Напомню, что к последовательностям относятся строки, списки, кортежи, типы `bytes` и `bytearray`. Пример:

```
>>> x, y, z = "123"              # Строка
>>> x, y, z
('1', '2', '3')
>>> x, y, z = [1, 2, 3]          # Список
>>> x, y, z
(1, 2, 3)
>>> x, y, z = (1, 2, 3)          # Кортеж
>>> x, y, z
(1, 2, 3)
>>> [x, y, z] = (1, 2, 3)        # Список слева, кортеж справа
>>> x, y, z
(1, 2, 3)
```

Обратите внимание на то, что количество элементов справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = (1, 2, 3, 4)
Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    x, y, z = (1, 2, 3, 4)
ValueError: too many values to unpack (expected 3)
```

В Python 3 существует возможность сохранения в переменной списка, состоящего из лишних элементов, при несоответствии количества элементов справа и слева от оператора =. Для этого перед именем переменной указывается звездочка (*). Пример:

```
>>> x, y, *z = (1, 2, 3, 4)
>>> x, y, z
(1, 2, [3, 4])
>>> x, *y, z = (1, 2, 3, 4)
>>> x, y, z
(1, [2, 3], 4)
>>> *x, y, z = (1, 2, 3, 4)
>>> x, y, z
([1, 2], 3, 4)
>>> x, y, *z = (1, 2, 3)
>>> x, y, z
(1, 2, [3])
>>> x, y, *z = (1, 2)
>>> x, y, z
(1, 2, [])
```

Как видно из примера, переменная, перед которой указана звездочка, всегда содержит список. Если для этой переменной не хватило значений, то ей присваивается пустой список. Следует помнить, что звездочку можно указать только перед одной переменной. В противном случае возникнет неоднозначность и интерпретатор выведет сообщение об ошибке:

```
>>> *x, y, *z = (1, 2, 3, 4)
SyntaxError: two starred expressions in assignment
```

2.4. Проверка типа данных

Python в любой момент времени изменяет тип переменной в соответствии с данными, хранящимися в ней. Пример:

```
>>> a = "Строка"            # Тип str
>>> a = 7                   # Теперь переменная имеет тип int
```

Определить, на какой тип данных ссылается переменная, позволяет функция `type(<Имя переменной>)`:

```
>>> type(a)
<class 'int'>
```

Проверить тип данных переменной можно следующими способами:

- сравнить значение, возвращаемое функцией `type()`, с названием типа данных:

```
>>> x = 10
>>> if type(x) == int:
        print("Это тип int")
```

- проверить тип с помощью функции `isinstance()`:

```
>>> s = "Строка"
>>> if isinstance(s, str):
        print("Это тип str")
```

2.5. Преобразование типов данных

Как вы уже знаете, в языке Python используется *динамическая типизация*. После присваивания значения в переменной сохраняется ссылка на объект определенного типа, а не сам объект. Если затем переменной присвоить значение другого типа, то переменная будет ссылаться на другой объект, и тип данных соответственно изменится. Таким образом, тип данных в языке Python — это характеристика объекта, а не переменной. Переменная всегда содержит только ссылку на объект.

После присваивания переменной значения над объектом можно производить операции, предназначенные для этого типа данных. Например, строку нельзя сложить с числом, т. к. это приведет к выводу сообщения об ошибке:

```
>>> 2 + "25"
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    2 + "25"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для преобразования типов данных предназначены следующие функции:

- `bool([<Объект>])` — преобразует объект в логический тип данных. Примеры:

```
>>> bool(0), bool(1), bool(""), bool("Строка")
(False, True, False, True)
```

- `int([<Объект>[, <Система счисления>]])` — преобразует объект в число. Во втором параметре можно указать систему счисления (значение по умолчанию — 10). Примеры:

```
>>> int(7.5), int("71")
(7, 71)
>>> int("71", 10), int("71", 8), int("0o71", 8), int("A", 16)
(71, 57, 57, 10)
```

  Если преобразование невозможно, то возбуждается исключение:

```
>>> int("71s")
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    int("71s")
ValueError: invalid literal for int() with base 10: '71s'
```

- `float([<Число или строка>])` — преобразует целое число или строку в вещественное число. Примеры:

```
>>> float(7), float("7.1")
(7.0, 7.1)
>>> float("Infinity"), float("-inf")
(inf, -inf)
>>> float("Infinity") + float("-inf")
nan
```

- `str([<Объект>])` — преобразует объект в строку. Примеры:

```
>>> str(125), str([1, 2, 3])
('125', '[1, 2, 3]')
>>> str((1, 2, 3)), str({"x": 5, "y": 10})
('(1, 2, 3)', "{'y': 10, 'x': 5}")
>>> str( bytes("строка", "utf-8") )
"b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0\\xba\\xd0
\\xb0'"
>>> str( bytearray("строка", "utf-8") )
"bytearray(b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0
\\xba\\xd0\\xb0')"
```

- `str(<Объект>[, <Кодировка>[, <Обработка ошибок>]])` — преобразует объект типа `bytes` или `bytearray` в строку. В третьем параметре могут быть указаны значения "strict" (при ошибке возбуждается исключение `UnicodeDecodeError`; значение по умолчанию), "replace" (неизвестный символ заменяется символом, имеющим код `\uFFFD`) или "ignore" (неизвестные символы игнорируются). Примеры:

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    str(obj1, "ascii", "strict")
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
'1'
```

- `bytes(<Строка>, <Кодировка>[, <Обработка ошибок>])` — преобразует строку в объект типа `bytes`. В третьем параметре могут быть указаны значения "strict" (значение по умолчанию), "replace" или "ignore". Примеры:

```
>>> bytes("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка123", "ascii", "ignore")
b'123'
```

- `bytes(<Последовательность>)` — преобразует последовательность целых чисел от 0 до 255 в объект типа `bytes`. Если число не попадает в диапазон, то возбуждается исключение `ValueError`. Пример:

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

- `bytearray(<Строка>, <Кодировка>[, <Обработка ошибок>])` — преобразует строку в объект типа `bytearray`. В третьем параметре могут быть указаны значения "strict" (значение по умолчанию), "replace" или "ignore".

Пример:

```
>>> bytearray("строка", "cp1251")
bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
```

- `bytearray(<Последовательность>)` — преобразует последовательность целых чисел от 0 до 255 в объект типа `bytearray`. Если число не попадает в диапазон, то возбуждается исключение `ValueError`. Пример:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

- `list(<Последовательность>)` — преобразует элементы последовательности в список. Примеры:

```
>>> list("12345")              # Преобразование строки
['1', '2', '3', '4', '5']
>>> list((1, 2, 3, 4, 5))      # Преобразование кортежа
[1, 2, 3, 4, 5]
```

- `tuple(<Последовательность>)` — преобразует элементы последовательности в кортеж:

```
>>> tuple("123456")            # Преобразование строки
('1', '2', '3', '4', '5', '6')
>>> tuple([1, 2, 3, 4, 5])     # Преобразование списка
(1, 2, 3, 4, 5)
```

В качестве примера рассмотрим возможность сложения двух чисел, введенных пользователем. Как вы уже знаете, вводить данные позволяет функция `input()`. Воспользуемся этой функцией для получения чисел от пользователя (листинг 2.4).

Листинг 2.4. Получение данных от пользователя

```
# -*- coding: utf-8 -*-
x = input("x = ")            # Вводим 5
y = input("y = ")            # Вводим 12
x = x.rstrip("\r")           # Для версии 3.2.0 (см. разд. 1.7)
y = y.rstrip("\r")           # Для версии 3.2.0 (см. разд. 1.7)
print(x + y)
input()
```

Результатом выполнения этого скрипта будет не число, а строка "512". Таким образом, следует запомнить, что функция `input()` возвращает результат в виде строки. Чтобы просуммировать два числа, необходимо преобразовать строку в число (листинг 2.5).

Листинг 2.5. Преобразование строки в число

```
# -*- coding: utf-8 -*-
x = int(input("x = "))       # Вводим 5
y = int(input("y = "))       # Вводим 12
print(x + y)
input()
```

В этом случае мы получим число 17, как и должно быть. Однако если пользователь вместо числа введет строку, то программа завершится с фатальной ошибкой. Как обработать ошибку, мы будем рассматривать по мере изучения языка.

2.6. Удаление переменной

Удалить переменную можно с помощью инструкции `del`:

```
del <Переменная1>[, ..., <ПеременнаяN>]
```

Пример удаления одной переменной:

```
>>> x = 10; x
10
>>> del x; x
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    del x; x
NameError: name 'x' is not defined
```

Пример удаления нескольких переменных:

```
>>> x, y = 10, 20
>>> del x, y
```

ГЛАВА 3

Операторы

Операторы позволяют произвести определенные действия с данными. Например, операторы присваивания служат для сохранения данных в переменной, математические операторы позволяют выполнить арифметические вычисления, а оператор конкатенации строк используется для соединения двух строк в одну. Рассмотрим операторы, доступные в Python 3, подробно.

3.1. Математические операторы

Производить операции над числами позволяют следующие операторы:

- `+` — сложение:

```
>>> 10 + 5              # Целые числа
15
>>> 12.4 + 5.2          # Вещественные числа
17.6
>>> 10 + 12.4           # Целые и вещественные числа
22.4
```

- `-` — вычитание:

```
>>> 10 - 5              # Целые числа
5
>>> 12.4 - 5.2          # Вещественные числа
7.2
>>> 12 - 5.2            # Целые и вещественные числа
6.8
```

- `*` — умножение:

```
>>> 10 * 5              # Целые числа
50
>>> 12.4 * 5.2          # Вещественные числа
64.48
>>> 10 * 5.2            # Целые и вещественные числа
52.0
```

- `/` — деление. Результатом деления всегда является вещественное число, даже если производится деление целых чисел. Обратите внимание на эту особенность, если вы раньше

программировали на Python 2. В Python 2 при делении целых чисел остаток отбрасывался и возвращалось целое число, в Python 3 поведение оператора изменилось. Примеры:

```
>>> 10 / 5               # Деление целых чисел без остатка
2.0
>>> 10 / 3               # Деление целых чисел с остатком
3.3333333333333335
>>> 10.0 / 5.0           # Деление вещественных чисел
2.0
>>> 10.0 / 3.0           # Деление вещественных чисел
3.3333333333333335
>>> 10 / 5.0             # Деление целого числа на вещественное
2.0
>>> 10.0 / 5             # Деление вещественного числа на целое
2.0
```

- `//` — деление с округлением вниз. Вне зависимости от типа чисел остаток отбрасывается. Примеры:

```
>>> 10 // 5           # Деление целых чисел без остатка
2
>>> 10 // 3           # Деление целых чисел с остатком
3
>>> 10.0 // 5.0       # Деление вещественных чисел
2.0
>>> 10.0 // 3.0       # Деление вещественных чисел
3.0
>>> 10 // 5.0         # Деление целого числа на вещественное
2.0
>>> 10 // 3.0         # Деление целого числа на вещественное
3.0
>>> 10.0 // 5         # Деление вещественного числа на целое
2.0
>>> 10.0 // 3         # Деление вещественного числа на целое
3.0
```

- `%` — остаток от деления:

```
>>> 10 % 5            # Деление целых чисел без остатка
0
>>> 10 % 3            # Деление целых чисел с остатком
1
>>> 10.0 % 5.0        # Операция над вещественными числами
0.0
>>> 10.0 % 3.0        # Операция над вещественными числами
1.0
>>> 10 % 5.0  # Операция над целыми и вещественными числами
0.0
>>> 10 % 3.0  # Операция над целыми и вещественными числами
1.0
>>> 10.0 % 5  # Операция над целыми и вещественными числами
0.0
>>> 10.0 % 3  # Операция над целыми и вещественными числами
1.0
```

- `**` — возведение в степень:

```
>>> 10 ** 2, 10.0 ** 2
(100, 100.0)
```

- унарный – (минус) и унарный + (плюс):

```
>>> +10, +10.0, -10, -10.0, -(-10), -(-10.0)
(10, 10.0, -10, -10.0, 10, 10.0)
```

Как видно из примеров, операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое и вещественное числа, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции будет вещественное число.

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
-2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат. Если необходимо производить операции с фиксированной точностью, то следует использовать модуль `decimal`:

```
>>> from decimal import Decimal
>>> Decimal("0.3") - Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
Decimal('0.0')
```

3.2. Двоичные операторы

Побитовые операторы предназначены для манипуляции отдельными битами. Язык Python поддерживает следующие побитовые операторы:

- `~` — двоичная инверсия. Значение каждого бита заменяется на противоположное:

```
>>> x = 100                 # 01100100
>>> x = ~x                  # 10011011
```

- `&` — двоичное И:

```
>>> x = 100                 # 01100100
>>> y = 75                  # 01001011
>>> z = x & y               # 01000000
>>> "{0:b} & {1:b} = {2:b}".format(x, y, z)
'1100100 & 1001011 = 1000000'
```

- `|` — двоичное ИЛИ:

```
>>> x = 100                 # 01100100
>>> y = 75                  # 01001011
>>> z = x | y               # 01101111
>>> "{0:b} | {1:b} = {2:b}".format(x, y, z)
'1100100 | 1001011 = 1101111'
```

- ^ — двоичное исключающее ИЛИ:

```
>>> x = 100                  # 01100100
>>> y = 250                  # 11111010
>>> z = x ^ y                # 10011110
>>> "{0:b} ^ {1:b} = {2:b}".format(x, y, z)
'1100100 ^ 11111010 = 10011110'
```

- << — сдвиг влево — сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:

```
>>> x = 100                  # 01100100
>>> y = x << 1               # 11001000
>>> z = y << 1               # 10010000
>>> k = z << 2               # 01000000
```

- >> — сдвиг вправо — сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

```
>>> x = 100                  # 01100100
>>> y = x >> 1               # 00110010
>>> z = y >> 1               # 00011001
>>> k = z >> 2               # 00000110
```

Если число отрицательное, то разряды слева заполняются единицами:

```
>>> x = -127                 # 10000001
>>> y = x >> 1               # 11000000
>>> z = y >> 2               # 11110000
>>> k = z << 1               # 11100000
>>> m = k >> 1               # 11110000
```

3.3. Операторы для работы с последовательностями

Для работы с последовательностями предназначены следующие операторы:

- + — конкатенация:

```
>>> print("Строка1" + "Строка2")   # Конкатенация строк
Строка1Строка2
>>> [1, 2, 3] + [4, 5, 6]         # Списки
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)         # Кортежи
(1, 2, 3, 4, 5, 6)
```

- * — повторение:

```
>>> "s" * 20                      # Строки
'ssssssssssssssssssss'
>>> [1, 2] * 3                    # Списки
[1, 2, 1, 2, 1, 2]
>>> (1, 2) * 3                    # Кортежи
(1, 2, 1, 2, 1, 2)
```

- `in` — проверка на вхождение. Если элемент входит в последовательность, то возвращается логическое значение `True`:

```
>>> "Строка" in "Строка для поиска"   # Строки
True
>>> "Строка2" in "Строка для поиска"  # Строки
False
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]    # Списки
(True, False)
>>> 2 in (1, 2, 3), 6 in (1, 2, 3)    # Кортежи
(True, False)
```

3.4. Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной. Перечислим операторы присваивания, доступные в языке Python:

- `=` — присваивает переменной значение:

```
>>> x = 5; x
5
```

- `+=` — увеличивает значение переменной на указанную величину:

```
>>> x = 5; x += 10            # Эквивалентно x = x + 10
>>> x
15
```

  Для последовательностей оператор `+=` производит конкатенацию:

```
>>> s = "Стр"; s += "ока"
>>> print(s)
Строка
```

- `-=` — уменьшает значение переменной на указанную величину:

```
>>> x = 10; x -= 5            # Эквивалентно x = x - 5
>>> x
5
```

- `*=` — умножает значение переменной на указанную величину:

```
>>> x = 10; x *= 5            # Эквивалентно x = x * 5
>>> x
50
```

  Для последовательностей оператор `*=` производит повторение:

```
>>> s = "*"; s *= 20
>>> s
'********************'
```

- `/=` — делит значение переменной на указанную величину:

```
>>> x = 10; x /= 3            # Эквивалентно x = x / 3
>>> x
3.3333333333333335
```

```
>>> y = 10.0; y /= 3.0        # Эквивалентно y = y / 3.0
>>> y
3.3333333333333335
```

- //= — деление с округлением вниз и присваиванием:

```
>>> x = 10; x //= 3           # Эквивалентно x = x // 3
>>> x
3
>>> y = 10.0; y //= 3.0       # Эквивалентно y = y // 3.0
>>> y
3.0
```

- %= — деление по модулю и присваивание:

```
>>> x = 10; x %= 2            # Эквивалентно x = x % 2
>>> x
0
>>> y = 10; y %= 3            # Эквивалентно y = y % 3
>>> y
1
```

- **= — возведение в степень и присваивание:

```
>>> x = 10; x **= 2           # Эквивалентно x = x ** 2
>>> x
100
```

3.5. Приоритет выполнения операторов

В какой последовательности будет вычисляться приведенное ниже выражение?

```
x = 5 + 10 * 3 / 2
```

Это зависит от приоритета выполнения операторов. В данном случае последовательность вычисления выражения будет такой:

1. Число 10 будет умножено на 3, т. к. приоритет оператора умножения выше приоритета оператора сложения.
2. Полученное значение будет поделено на 2, т. к. приоритет оператора деления равен приоритету оператора умножения (а операторы с равными приоритетами выполняются слева направо), но выше чем у оператора сложения.
3. К полученному значению будет прибавлено число 5, т. к. оператор присваивания = имеет наименьший приоритет.
4. Значение будет присвоено переменной x.

```
>>> x = 5 + 10 * 3 / 2
>>> x
20.0
```

С помощью скобок можно изменить последовательность вычисления выражения:

```
x = (5 + 10) * 3 / 2
```

Теперь порядок вычислений будет другим:

1. К числу 5 будет прибавлено 10.
2. Полученное значение будет умножено на 3.
3. Полученное значение будет поделено на 2.
4. Значение будет присвоено переменной `x`.

```
>>> x = (5 + 10) * 3 / 2
>>> x
22.5
```

Перечислим операторы в порядке убывания приоритета:

1. `-x`, `+x`, `~x`, `**` — унарный минус, унарный плюс, двоичная инверсия, возведение в степень. Если унарные операторы расположены слева от оператора `**`, то возведение в степень имеет больший приоритет, а если справа — то меньший. Например, выражение

   ```
   -10 ** -2
   ```

   эквивалентно следующей расстановке скобок:

   ```
   -(10 ** (-2))
   ```

2. `*`, `%`, `/`, `//` — умножение (повторение), остаток от деления, деление, деление с округлением вниз.
3. `+`, `–` — сложение (конкатенация), вычитание.
4. `<<`, `>>` — двоичные сдвиги.
5. `&` — двоичное И.
6. `^` — двоичное исключающее ИЛИ.
7. `|` — двоичное ИЛИ.
8. `=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=` — присваивание.

ГЛАВА 4

Условные операторы и циклы

Условные операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его. Логические выражения возвращают только два значения `True` (истина) или `False` (ложь), которые ведут себя как целые числа `1` и `0` соответственно:

```
>>> True + 2                    # Эквивалентно 1 + 2
3
>>> False + 2                   # Эквивалентно 0 + 2
2
```

Логическое значение можно сохранить в переменной:

```
>>> x = True; y = False
>>> x, y
(True, False)
```

Любой объект в логическом контексте может интерпретироваться как истина (`True`) или как ложь (`False`). Для определения логического значения можно использовать функцию `bool()`.

Значение `True` возвращают следующие объекты:

- любое число, не равное нулю:

```
>>> bool(1), bool(20), bool(-20)
(True, True, True)
>>> bool(1.0), bool(0.1), bool(-20.0)
(True, True, True)
```

- не пустой объект:

```
>>> bool("0"), bool([0, None]), bool((None,)), bool({"x": 5})
(True, True, True, True)
```

Следующие объекты интерпретируются как `False`:

- число, равное нулю:

```
>>> bool(0), bool(0.0)
(False, False)
```

- пустой объект:

```
>>> bool(""), bool([]), bool(())
(False, False, False)
```

- значение None:

```
>>> bool(None)
False
```

4.1. Операторы сравнения

Операторы сравнения используются в логических выражениях. Перечислим их:

- == — равно:

```
>>> 1 == 1, 1 == 5
(True, False)
```

- != — не равно:

```
>>> 1 != 5, 1 != 1
(True, False)
```

- < — меньше:

```
>>> 1 < 5, 1 < 0
(True, False)
```

- > — больше:

```
>>> 1 > 0, 1 > 5
(True, False)
```

- <= — меньше или равно:

```
>>> 1 <= 5, 1 <= 0, 1 <= 1
(True, False, True)
```

- >= — больше или равно:

```
>>> 1 >= 0, 1 >= 5, 1 >= 1
(True, False, True)
```

- in — проверка на вхождение в последовательность:

```
>>> "Строка" in "Строка для поиска" # Строки
True
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]  # Списки
(True, False)
>>> 2 in (1, 2, 3), 4 in (1, 2, 3)  # Кортежи
(True, False)
```

  Оператор in можно также использовать для проверки существования ключа словаря:

```
>>> "x" in {"x": 1, "y": 2}, "z" in {"x": 1, "y": 2}
(True, False)
```

- is — проверяет, ссылаются ли две переменные на один и тот же объект. Если переменные ссылаются на один и тот же объект, то оператор is возвращает значение True:

```
>>> x = y = [1, 2]
>>> x is y
True
```

```
>>> x = [1, 2]; y = [1, 2]
>>> x is y
False
```

Следует заметить, что в целях эффективности интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то в этих переменных будет сохранена ссылка на один и тот же объект. Пример:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
```

Значение логического выражения можно инвертировать с помощью оператора `not`:

```
>>> x = 1; y = 1
>>> x == y
True
>>> not (x == y), not x == y
(False, False)
```

Если переменные `x` и `y` равны, то возвращается значение `True`, но т. к. перед выражением стоит оператор `not`, выражение вернет `False`. Круглые скобки можно не указывать, т. к. оператор `not` имеет более низкий приоритет выполнения, чем операторы сравнения.

При необходимости инвертировать значение оператора `in`, оператор `not` указывается непосредственно перед этим оператором:

```
>>> 2 in [1, 2, 3], 2 not in [1, 2, 3]
(True, False)
```

Чтобы инвертировать значение оператора `is`, оператор `not` указывается непосредственно после этого оператора:

```
>>> x = y = [1, 2]
>>> x is y, x is not y
(True, False)
```

В логическом выражении можно указывать сразу несколько условий:

```
>>> x = 10
>>> 1 < x < 20, 11 < x < 20
(True, False)
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

- `and` — `x and y` — логическое И. Если `x` интерпретируется как `False`, то возвращается `x`, в противном случае — `y`:

```
>>> 1 < 5 and 2 < 5              # True and True == True
True
>>> 1 < 5 and 2 > 5              # True and False == False
False
>>> 1 > 5 and 2 < 5              # False and True == False
False
>>> 10 and 20, 0 and 20, 10 and 0
(20, 0, 0)
```

- `or` — `x or y` — логическое ИЛИ. Если `x` интерпретируется как `False`, то возвращается `y`, в противном случае — `x`:

```
>>> 1 < 5 or 2 < 5              # True or True == True
True
>>> 1 < 5 or 2 > 5              # True or False == True
True
>>> 1 > 5 or 2 < 5              # False or True == True
True
>>> 1 > 5 or 2 > 5              # False or False == False
False
>>> 10 or 20, 0 or 20, 10 or 0
(10, 20, 10)
>>> 0 or "" or None or [] or "s"
's'
```

Это выражение вернет `True` только в случае, если оба выражения вернут `True`:

```
x1 == x2 and x2 != x3
```

А это выражение вернет `True`, если хотя бы одно из выражений вернет `True`:

```
x1 == x2 or x3 == x4
```

Перечислим операторы сравнения в порядке убывания приоритета:

1. `<`, `>`, `<=`, `>=`, `==`, `!=`, `<>`, `is`, `is not`, `in`, `not in`.
2. `not` — логическое отрицание.
3. `and` — логическое И.
4. `or` — логическое ИЛИ.

4.2. Оператор ветвления *if...else*

Оператор ветвления `if...else` позволяет в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его. Оператор имеет следующий формат:

```
if <Логическое выражение>:
    <Блок, выполняемый, если условие истинно>
[elif <Логическое выражение>:
    <Блок, выполняемый, если условие истинно>
]
[else:
    <Блок, выполняемый, если все условия ложны>
]
```

Как вы уже знаете, блоки внутри составной инструкции выделяются одинаковым количеством пробелов (обычно четыре пробела). Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В некоторых языках программирования логическое выражение заключается в круглые скобки. В языке Python это делать необязательно, но можно, т. к. любое выражение может быть расположено внутри круглых скобок. Круг-

лые скобки следует использовать только при необходимости разместить условие на нескольких строках.

Для примера напишем программу, которая проверяет, является ли введенное пользователем число четным или нет (листинг 4.1). После проверки выводится соответствующее сообщение.

Листинг 4.1. Проверка числа на четность

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, "- Четное число")
else:
    print(x, "- Нечетное число")
input()
```

Если блок состоит из одной инструкции, то эту инструкцию можно разместить на одной строке с заголовком:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0: print(x, "- Четное число")
else: print(x, "- Нечетное число")
input()
```

В этом случае концом блока является конец строки. Это означает, что можно разместить сразу несколько инструкций на одной строке, разделяя их точкой с запятой:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0: print(x, end=" "); print("- Четное число")
else: print(x, end=" "); print("- Нечетное число")
input()
```

Знайте, что так можно сделать, но никогда на практике не пользуйтесь этим способом, т. к. подобная конструкция нарушает стройность кода и ухудшает его сопровождение в дальнейшем. Всегда размещайте инструкцию на отдельной строке, даже если блок содержит только одну инструкцию. Согласитесь, что этот код читается намного проще, чем предыдущий:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, end=" ")
    print("- Четное число")
else:
    print(x, end=" ")
    print("- Нечетное число")
input()
```

Оператор `if...else` позволяет проверить сразу несколько условий. Рассмотрим это на примере (листинг 4.2).

Листинг 4.2. Проверка нескольких условий

```
# -*- coding: utf-8 -*-
print("""Какой операционной системой вы пользуетесь?
1 — Windows 7
2 — Windows XP
3 — Windows Vista
4 — Другая""")
os = input("Введите число, соответствующее ответу: ")
os = os.rstrip("\r")           # Для версии 3.2.0 (см. разд. 1.7)
if   os == "1":
    print("Вы выбрали — Windows 7")
elif os == "2":
    print("Вы выбрали — Windows XP")
elif os == "3":
    print("Вы выбрали — Windows Vista")
elif os == "4":
    print("Вы выбрали — Другая")
elif not os:
    print("Вы не ввели число")
else:
    print("Мы не смогли определить вашу операционную систему")
input()
```

С помощью инструкции `elif` мы можем определить выбранное значение и вывести соответствующее сообщение. Обратите внимание на то, что логическое выражение не содержит операторов сравнения:

```
elif not os:
```

Такая запись эквивалентна следующей:

```
elif os == "":
```

Проверка на равенство выражения значению `True` выполняется по умолчанию. Так как пустая строка интерпретируется как `False`, мы инвертируем возвращаемое значение с помощью оператора `not`.

Один условный оператор можно вложить в другой. В этом случае отступ вложенной инструкции должен быть в два раза больше (листинг 4.3).

Листинг 4.3. Вложенные инструкции

```
# -*- coding: utf-8 -*-
print("""Какой операционной системой вы пользуетесь?
1 — Windows 7
2 — Windows XP
3 — Windows Vista
4 — Другая""")
os = input("Введите число, соответствующее ответу: ")
os = os.rstrip("\r")           # Для версии 3.2.0 (см. разд. 1.7)
```

```
if os != "":
    if   os == "1":
        print("Вы выбрали — Windows 7")
    elif os == "2":
        print("Вы выбрали — Windows XP")
    elif os == "3":
        print("Вы выбрали — Windows Vista")
    elif os == "4":
        print("Вы выбрали — Другая")
    else:
        print("Мы не смогли определить вашу операционную систему")
else:
    print("Вы не ввели число")
input()
```

Оператор `if...else` имеет еще один формат:

```
<Переменная> = <Если истина> if <Условие> else <Если ложь>
```

Пример:

```
>>> print("Yes" if 10 % 2 == 0 else "No")
Yes
>>> s = "Yes" if 10 % 2 == 0 else "No"
>>> s
'Yes'
>>> s = "Yes" if 11 % 2 == 0 else "No"
>>> s
'No'
```

4.3. Цикл *for*

Предположим, нужно вывести все числа от 1 до 100 по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```
print(1)
print(2)
...
print(100)
```

При помощи циклов то же действие можно выполнить одной строкой кода:

```
for x in range(1, 101): print(x)
```

Иными словами, циклы позволяют выполнить одни и те же инструкции многократно.

В языке Python используются два цикла: `for` и `while`. Цикл `for` применяется для перебора элементов последовательности. Имеет следующий формат:

```
for <Текущий элемент> in <Последовательность>:
    <Инструкции внутри цикла>
[else:
    <Блок, выполняемый, если не использовался оператор break>
]
```

Здесь присутствуют следующие конструкции:

- <Последовательность> — объект, поддерживающий механизм итерации. Например, строка, список, кортеж, словарь и др.;
- <Текущий элемент> — на каждой итерации через этот параметр доступен текущий элемент последовательности или ключ словаря;
- <Инструкции внутри цикла> — блок, который будет многократно выполняться;
- если внутри цикла не использовался оператор `break`, то после завершения выполнения цикла будет выполнен блок в инструкции `else`. Данный блок не является обязательным.

Пример перебора букв в слове приведен в листинге 4.4.

Листинг 4.4. Перебор букв в слове

```
for s in "str":
    print(s, end=" ")
else:
    print("\nЦикл выполнен")
```

Результат выполнения:

```
s t r
Цикл выполнен
```

Теперь выведем каждый элемент списка и кортежа на отдельной строке (листинг 4.5).

Листинг 4.5. Перебор списка и кортежа

```
for x in [1, 2, 3]:
    print(x)
for y in (1, 2, 3):
    print(y)
```

Цикл `for` позволяет также перебрать элементы словарей, хотя словари и не являются последовательностями. В качестве примера выведем элементы словаря двумя способами. Первый способ использует метод `keys()`, возвращающий объект `dict_keys`, содержащий все ключи словаря. Во втором способе мы просто указываем словарь в качестве параметра. На каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу (листинг 4.6).

Листинг 4.6. Перебор элементов словаря

```
>>> arr = {"x": 1, "y": 2, "z": 3}
>>> arr.keys()
dict_keys(['y', 'x', 'z'])
>>> for key in arr.keys():     # Использование метода keys()
        print(key, arr[key])

y 2
x 1
z 3
```

```
>>> for key in arr:            # Словари также поддерживают итерации
        print(key, arr[key])

y 2
x 1
z 3
```

Обратите внимание на то, что элементы словаря выводятся в произвольном порядке, а не в порядке, в котором они были указаны при создании объекта. Чтобы вывести элементы в алфавитном порядке, следует отсортировать ключи с помощью функции `sorted()`:

```
>>> arr = {"x": 1, "y": 2, "z": 3}
>>> for key in sorted(arr):
        print(key, arr[key])

x 1
y 2
z 3
```

С помощью цикла `for` можно перебирать сложные структуры данных. В качестве примера выведем элементы списка кортежей (листинг 4.7).

Листинг 4.7. Перебор элементов списка кортежей

```
>>> arr = [(1, 2), (3, 4)]          # Список кортежей
>>> for a, b in arr:
        print(a, b)

1 2
3 4
```

4.4. Функции *range()* и *enumerate()*

До сих пор мы только выводили элементы последовательностей. Теперь попробуем умножить каждый элемент списка на 2:

```
arr = [1, 2, 3]
for i in arr:
    i = i * 2
print(arr)          # Результат выполнения: [1, 2, 3]
```

Как видно из примера, список не изменился. Переменная `i` на каждой итерации цикла содержит лишь копию значения текущего элемента списка. Изменить таким способом элементы списка нельзя. Чтобы получить доступ к каждому элементу, можно, например, воспользоваться функцией `range()` для генерации индексов. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемые значения. Если параметр

<Шаг> не указан, то используется значение 1. Функция возвращает объект, поддерживающий итерационный протокол. С помощью этого объекта внутри цикла for можно получить значение текущего элемента. В качестве примера умножим каждый элемент списка на 2 (листинг 4.8).

Листинг 4.8. Пример использования функции range()

```
arr = [1, 2, 3]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)          # Результат выполнения: [2, 4, 6]
```

В этом примере мы получаем количество элементов списка с помощью функции len() и передаем результат в функцию range(). В итоге функция range() возвращает диапазон значений от 0 до len(arr) − 1. На каждой итерации цикла через переменную i доступен текущий элемент из диапазона индексов. Чтобы получить доступ к элементу списка, указываем индекс внутри квадратных скобок. Умножаем каждый элемент списка на 2, а затем выводим результат с помощью функции print().

Рассмотрим несколько примеров использования функции range(). Выведем числа от 1 до 100:

```
for i in range(1, 101): print(i)
```

Можно не только увеличивать значение, но и уменьшать его. Выведем все числа от 100 до 1:

```
for i in range(100, 0, -1): print(i)
```

Можно также изменять значение не только на единицу. Выведем все четные числа от 1 до 100:

```
for i in range(2, 101, 2): print(i)
```

В Python 2 функция range() возвращала список чисел. В Python 3 поведение функции изменилось. Теперь функция возвращает объект, поддерживающий итерационный протокол и позволяющий получить значение по индексу. Чтобы получить список чисел, следует передать объект, возвращаемый функцией range(), в функцию list() (листинг 4.9).

Листинг 4.9. Объект range

```
>>> obj = range(len([1, 2, 3]))
>>> obj
range(0, 3)
>>> obj[0], obj[1], obj[2]      # Доступ по индексу
(0, 1, 2)
>>> obj[0:2]                    # Получение среза
range(0, 2)
>>> i = iter(obj)
>>> next(i), next(i), next(i)   # Доступ с помощью итераторов
(0, 1, 2)
>>> list(obj)                   # Преобразование объекта в список
[0, 1, 2]
>>> 1 in obj, 7 in obj          # Проверка вхождения значения
(True, False)
```

Начиная с версии Python 3.2, объект `range` поддерживает два метода:

- `index(<Значение>)` — возвращает индекс элемента, имеющего указанное значение. Если значение не входит в объект, возбуждается исключение `ValueError`. Пример:

```
>>> obj = range(1, 5)
>>> obj.index(1), obj.index(4)
(0, 3)
>>> obj.index(5)
... Фрагмент опущен ...
ValueError: 5 is not in range
```

- `count(<Значение>)` — возвращает количество элементов с указанным значением. Если элемент не входит в объект, то возвращается значение `0`. Пример:

```
>>> obj = range(1, 5)
>>> obj.count(1), obj.count(10)
(1, 0)
```

Функция `enumerate(<Объект>[, start=0])` на каждой итерации цикла `for` возвращает кортеж из индекса и значения текущего элемента. С помощью необязательного параметра `start` можно задать начальное значение индекса. В качестве примера умножим на 2 каждый элемент списка, который содержит четное число (листинг 4.10).

Листинг 4.10. Пример использования функции `enumerate()`

```
arr = [1, 2, 3, 4, 5, 6]
for i, elem in enumerate(arr):
    if elem % 2 == 0:
        arr[i] *= 2
print(arr)                # Результат выполнения: [1, 4, 3, 8, 5, 12]
```

Функция `enumerate()` не создает список, а возвращает итератор. С помощью функции `next()` можно обойти всю последовательность. Когда перебор будет закончен, возбуждается исключение `StopIteration`:

```
>>> arr = [1, 2]
>>> obj = enumerate(arr, start=2)
>>> next(obj)
(2, 1)
>>> next(obj)
(3, 2)
>>> next(obj)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(obj)
StopIteration
```

Весь этот процесс цикл `for` выполняет незаметно для нас.

4.5. Цикл *while*

Выполнение инструкций в цикле `while` продолжается до тех пор, пока логическое выражение истинно. Цикл `while` имеет следующий формат:

```
<Начальное значение>
while <Условие>:
    <Инструкции>
    <Приращение>
[else:
    <Блок, выполняемый если не использовался оператор break>
]
```

Последовательность работы цикла `while`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, если оно истинно, выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Переход к пункту 2.
5. Если внутри цикла не использовался оператор `break`, то после завершения выполнения цикла будет выполнен блок в инструкции `else`. Данный блок не является обязательным.

Выведем все числа от 1 до 100, используя цикл `while` (листинг 4.11).

Листинг 4.11. Вывод чисел от 1 до 100

```
i = 1                    # <Начальное значение>
while i < 101:           # <Условие>
    print(i)             # <Инструкции>
    i += 1               # <Приращение>
```

> ***Внимание!***
>
> Если `<Приращение>` не указано, то цикл будет бесконечным. Чтобы прервать бесконечный цикл, следует нажать комбинацию клавиш <Ctrl>+<C>. В результате генерируется исключение `KeyboardInterrupt`, и выполнение программы будет остановлено. Следует учитывать, что прервать таким образом можно только цикл, который выводит данные.

Выведем все числа от 100 до 1 (листинг 4.12).

Листинг 4.12. Вывод чисел от 100 до 1

```
i = 100
while i:
    print(i)
    i -= 1
```

Обратите внимание на условие, оно не содержит операторов сравнения. На каждой итерации цикла мы вычитаем единицу из значения переменной-счетчика. Как только значение

будет равно 0, цикл остановится. Как вы уже знаете, число 0 в логическом контексте эквивалентно значению False, а проверка на равенство выражения значению True выполняется по умолчанию.

С помощью цикла while можно перебирать и элементы различных структур. Но в этом случае следует помнить, что цикл while работает медленнее цикла for. В качестве примера умножим каждый элемент списка на 2 (листинг 4.13).

Листинг 4.13. Перебор элементов списка

```
arr = [1, 2, 3]
i, count = 0, len(arr)
while i < count:
    arr[i] *= 2
    i += 1
print(arr)              # Результат выполнения: [2, 4, 6]
```

4.6. Оператор *continue*. Переход на следующую итерацию цикла

Оператор continue позволяет перейти к следующей итерации цикла до завершения выполнения всех инструкций внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно (листинг 4.14).

Листинг 4.14. Оператор `continue`

```
for i in range(1, 101):
    if 4 < i < 11:
        continue          # Переходим на следующую итерацию цикла
    print(i)
```

4.7. Оператор *break*. Прерывание цикла

Оператор break позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом (листинг 4.15).

Листинг 4.15. Оператор `break`

```
i = 1
while True:
    if i > 100: break      # Прерываем цикл
    print(i)
    i += 1
```

Здесь мы в условии указали значение True. В этом случае выражения внутри цикла будут выполняться бесконечно. Однако использование оператора break прерывает его выполнение, как только 100 строк уже напечатано.

Внимание!

Оператор `break` прерывает выполнение цикла, а не программы, т. е. далее будет выполнена инструкция, следующая сразу за циклом.

Цикл `while` совместно с оператором `break` удобно использовать для получения неопределенного заранее количества данных от пользователя. В качестве примера просуммируем неопределенное количество чисел (листинг 4.16).

Листинг 4.16. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    x = x.rstrip("\r")              # Для версии 3.2.0 (см. разд. 1.7)
    if x == "stop":
        break                       # Выход из цикла
    x = int(x)                      # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода трех чисел и получения суммы выглядит так:

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: 20
Введите число: 30
Введите число: stop
Сумма чисел равна: 60
```

Значения, введенные пользователем, выделены полужирным шрифтом.

ГЛАВА 5

Числа

Язык Python 3 поддерживает следующие числовые типы:

- `int` — целые числа. Размер числа ограничен лишь объемом оперативной памяти;
- `float` — вещественные числа;
- `complex` — комплексные числа.

Операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое и вещественное числа, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции будет вещественное число.

Создать объект целочисленного типа можно обычным способом:

```
>>> x = 0; y = 10; z = -80
>>> x, y, z
(0, 10, -80)
```

Кроме того, можно указать число в двоичной, восьмеричной или шестнадцатеричной форме. Такие числа будут автоматически преобразованы в десятичные целые числа. Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и содержат цифры от `0` или `1`:

```
>>> 0b11111111, 0b101101
(255, 45)
```

Восьмеричные числа начинаются с нуля и следующей за ним латинской буквы `o` (регистр не имеет значения) и содержат цифры от `0` до `7`:

```
>>> 0o7, 0o12, 0o777, 0O7, 0O12, 0O777
(7, 10, 511, 7, 10, 511)
```

Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать цифры от `0` до `9` и буквы от `A` до `F` (регистр букв не имеет значения):

```
>>> 0X9,0xA, 0x10, 0xFFF, 0xfff
(9, 10, 16, 4095, 4095)
```

Вещественное число может содержать точку и (или) быть представлено в экспоненциальной форме с буквой `E` (регистр не имеет значения):

```
>>> 10., .14, 3.14, 11E20, 2.5e-12
(10.0, 0.14, 3.14, 1.1e+21, 2.5e-12)
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 — 0.1 — 0.1 — 0.1
-2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат. Если необходимо производить операции с фиксированной точностью, то следует использовать модуль `decimal`:

```
>>> from decimal import Decimal
>>> Decimal("0.3") — Decimal("0.1") — Decimal("0.1") — Decimal("0.1")
Decimal('0.0')
```

Кроме того, можно использовать рациональные числа, поддержка которых содержится в модуле `fractions`:

```
>>> from fractions import Fraction
>>> Fraction("0.3") — Fraction("0.1") — Fraction("0.1") — Fraction("0.1")
Fraction(0, 1)
>>> float(Fraction(0, 1))
0.0
>>> Fraction(0.5), Fraction("2.3"), Fraction(3, 6)
(Fraction(1, 2), Fraction(23, 10), Fraction(1, 2))
```

Комплексные числа записываются в формате:

```
<Вещественная часть>+<Мнимая часть>J
```

Примеры комплексных чисел:

```
>>> 2+5J, 8j
((2+5j), 8j)
```

Рассмотрение модулей `decimal` и `fractions`, а также комплексных чисел выходит за рамки книги. За подробной информацией обращайтесь к документации.

5.1. Встроенные функции для работы с числами

Для работы с числами предназначены следующие встроенные функции:

- `int([<Объект>[, <Система счисления>]])` — преобразует объект в целое число. Во втором параметре можно указать систему счисления (значение по умолчанию `10`). Пример:

  ```
  >>> int(7.5), int("71", 10), int("0o71", 8), int("0xA", 16)
  (7, 71, 57, 10)
  >>> int(), int("0b11111111", 2)
  (0, 255)
  ```

- `float([<Число или строка>])` — преобразует целое число или строку в вещественное число:

  ```
  >>> float(7), float("7.1"), float("12.")
  (7.0, 7.1, 12.0)
  >>> float("inf"), float("-Infinity"), float("nan")
  (inf, -inf, nan)
  ```

```
>>> float()
0.0
```

- `bin(<Число>)` — преобразует десятичное число в двоичное. Возвращает строковое представление числа. Пример:

```
>>> bin(255), bin(1), bin(-45)
('0b11111111', '0b1', '-0b101101')
```

- `oct(<Число>)` — преобразует десятичное число в восьмеричное. Возвращает строковое представление числа. Пример:

```
>>> oct(7), oct(8), oct(64)
('0o7', '0o10', '0o100')
```

- `hex(<Число>)` — преобразует десятичное число в шестнадцатеричное. Возвращает строковое представление числа. Пример:

```
>>> hex(10), hex(16), hex(255)
('0xa', '0x10', '0xff')
```

- `round(<Число>[, <Количество знаков после точки>])` — возвращает число, округленное до ближайшего меньшего целого для чисел с дробной частью меньше `0.5`, или значение, округленное до ближайшего большего целого для чисел с дробной частью больше `0.5`. Если дробная часть равна `0.5`, то округление производится до ближайшего четного числа. Пример:

```
>>> round(0.49), round(0.50), round(0.51)
(0, 0, 1)
>>> round(1.49), round(1.50), round(1.51)
(1, 2, 2)
>>> round(2.49), round(2.50), round(2.51)
(2, 2, 3)
>>> round(3.49), round(3.50), round(3.51)
(3, 4, 4)
```

  Во втором параметре можно указать количество знаков в дробной части. Если параметр не указан, то используется значение `0`:

```
>>> round(1.524, 2), round(1.525, 2), round(1.5555, 3)
(1.52, 1.52, 1.556)
```

- `abs(<Число>)` — возвращает абсолютное значение:

```
>>> abs(-10), abs(10), abs(-12.5)
(10, 10, 12.5)
```

- `pow(<Число>, <Степень>[, <Остаток от деления>])` — возводит `<Число>` в `<Степень>`:

```
>>> pow(10, 2), 10 ** 2, pow(3, 3), 3 ** 3
(100, 100, 27, 27)
```

  Если указан третий параметр, то возвращается остаток от деления:

```
>>> pow(10, 2, 2), (10 ** 2) % 2, pow(3, 3, 2), (3 ** 3) % 2
(0, 0, 1, 1)
```

- `max(<Список чисел через запятую>)` — максимальное значение из списка:

```
>>> max(1, 2, 3), max(3, 2, 3, 1), max(1, 1.0), max(1.0, 1)
(3, 3, 1, 1.0)
```

- `min(<Список чисел через запятую>)` — минимальное значение из списка:

```
>>> min(1, 2, 3), min(3, 2, 3, 1), min(1, 1.0), min(1.0, 1)
(1, 1, 1, 1.0)
```

- `sum(<Последовательность>[, <Начальное значение>])` — возвращает сумму значений элементов последовательности (например, списка, кортежа) плюс `<Начальное значение>`. Если второй параметр не указан, то значение параметра равно `0`. Если последовательность пустая, то возвращается значение второго параметра. Примеры:

```
>>> sum((10, 20, 30, 40)), sum([10, 20, 30, 40])
(100, 100)
>>> sum([10, 20, 30, 40], 2), sum([], 2)
(102, 2)
```

- `divmod(x, y)` — возвращает кортеж из двух значений `(x // y, x % y)`:

```
>>> divmod(13, 2)              # 13 == 6 * 2 + 1
(6, 1)
>>> 13 // 2, 13 % 2
(6, 1)
>>> divmod(13.5, 2.0)          # 13.5 == 6.0 * 2.0 + 1.5
(6.0, 1.5)
>>> 13.5 // 2.0, 13.5 % 2.0
(6.0, 1.5)
```

5.2. Модуль *math*. Математические функции

Модуль `math` предоставляет дополнительные функции для работы с числами, а также стандартные константы. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import math
```

> ***Примечание***
>
> Для работы с комплексными числами необходимо использовать модуль `cmath`.

Модуль `math` предоставляет следующие стандартные константы:

- `pi` — возвращает число π:

```
>>> import math
>>> math.pi
3.141592653589793
```

- `e` — возвращает значение константы *e*:

```
>>> math.e
2.718281828459045
```

Перечислим основные функции для работы с числами:

- `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Значение указывается в радианах;
- `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах;

- `degrees()` — преобразует радианы в градусы:

```
>>> math.degrees(math.pi)
180.0
```

- `radians()` — преобразует градусы в радианы:

```
>>> math.radians(180.0)
3.141592653589793
```

- `exp()` — экспонента;
- `log()` — логарифм;
- `sqrt()` — квадратный корень:

```
>>> math.sqrt(100), math.sqrt(25)
(10.0, 5.0)
```

- `ceil()` — значение, округленное до ближайшего большего целого:

```
>>> math.ceil(5.49), math.ceil(5.50), math.ceil(5.51)
(6, 6, 6)
```

- `floor()` — значение, округленное до ближайшего меньшего целого:

```
>>> math.floor(5.49), math.floor(5.50), math.floor(5.51)
(5, 5, 5)
```

- `pow(<Число>, <Степень>)` — возводит `<Число>` в `<Степень>`:

```
>>> math.pow(10, 2), 10 ** 2, math.pow(3, 3), 3 ** 3
(100.0, 100, 27.0, 27)
```

- `fabs()` — абсолютное значение:

```
>>> math.fabs(10), math.fabs(-10), math.fabs(-12.5)
(10.0, 10.0, 12.5)
```

- `fmod()` — остаток от деления:

```
>>> math.fmod(10, 5), 10 % 5, math.fmod(10, 3), 10 % 3
(0.0, 0, 1.0, 1)
```

- `factorial()` — факториал числа:

```
>>> math.factorial(5), math.factorial(6)
(120, 720)
```

> ***Примечание***
>
> В этом разделе мы рассмотрели только основные функции. Чтобы получить полный список функций, обращайтесь к документации по модулю `math`.

5.3. Модуль *random*. Генерация случайных чисел

Модуль `random` позволяет генерировать случайные числа. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import random
```

Перечислим основные функции:

- `random()` — возвращает псевдослучайное число от `0.0` до `1.0`:

```
>>> import random
>>> random.random()
0.9753144027290991
>>> random.random()
0.5468390487484339
>>> random.random()
0.13015058054767736
```

- `seed([<Параметр>][, version=2])` — настраивает генератор случайных чисел на новую последовательность. По умолчанию используется системное время. Если значение параметра будет одинаковым, то генерируется одинаковое число:

```
>>> random.seed(10)
>>> random.random()
0.5714025946899135
>>> random.seed(10)
>>> random.random()
0.5714025946899135
```

- `uniform(<Начало>, <Конец>)` — возвращает псевдослучайное вещественное число в диапазоне от `<Начало>` до `<Конец>`:

```
>>> random.uniform(0, 10)
9.965569925394552
>>> random.uniform(0, 10)
0.4455638245043303
```

- `randint(<Начало>, <Конец>)` — возвращает псевдослучайное целое число в диапазоне от `<Начало>` до `<Конец>`:

```
>>> random.randint(0, 10)
4
>>> random.randint(0, 10)
10
```

- `randrange([<Начало>, ]<Конец>[, <Шаг>])` — возвращает случайный элемент из числовой последовательности. Параметры аналогичны параметрам функции `range()`:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

  Именно из списка, возвращаемого функцией `range()`, и выбирается случайный элемент:

```
>>> random.randrange(10)
5
>>> random.randrange(0, 10)
2
>>> random.randrange(0, 10, 2)
6
```

- `choice(<Последовательность>)` — возвращает случайный элемент из любой последовательности (строки, списка, кортежа):

```
>>> random.choice("string")          # Строка
'i'
```

```
>>> random.choice(["s", "t", "r"])   # Список
'r'
>>> random.choice(("s", "t", "r"))   # Кортеж
't'
```

- `shuffle(<Список>[, <Число от 0.0 до 1.0>])` — перемешивает элементы списка случайным образом. Функция перемешивает сам список и ничего не возвращает. Если второй параметр не указан, то используется значение, возвращаемое функцией `random()`. Пример:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)
>>> arr
[8, 6, 9, 5, 3, 7, 2, 4, 10, 1]
```

- `sample(<Последовательность>, <Количество элементов>)` — возвращает список из указанного количества элементов. В этот список попадут элементы из последовательности, выбранные случайным образом. В качестве последовательности можно указать любые объекты, поддерживающие итерации. Примеры:

```
>>> random.sample("string", 2)
['i', 'r']
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
[7, 10]
>>> arr # Сам список не изменяется
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample((1, 2, 3, 4, 5, 6, 7), 3)
[6, 3, 5]
>>> random.sample(range(300), 5)
[126, 194, 272, 46, 71]
```

Для примера создадим генератор паролей произвольной длины (листинг 5.1). Для этого добавляем в список `arr` все разрешенные символы, а далее в цикле получаем случайный элемент с помощью функции `choice()`. По умолчанию будет выдаваться пароль из 8 символов.

Листинг 5.1. Генератор паролей

```
# -*- coding: utf-8 -*-
import random # Подключаем модуль random
def passw_generator(count_char=8):
    arr = ['a','b','c','d','e','f','g','h','i','j','k','l','m',
    'n','o','p','q','r','s','t','u','v','w','x','y','z',
    'A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','O','P','Q','R','S','T','U','V', 'W',
    'X','Y','Z','1','2','3','4','5','6','7','8','9','0']
    passw = []
    for i in range(count_char):
        passw.append(random.choice(arr))
    return "".join(passw)

# Вызываем функцию
print( passw_generator(10) )  # Выведет что-то вроде ngODHE8J8x
print( passw_generator() )    # Выведет что-то вроде ZxcpkF5O
input()
```

ГЛАВА 6

Строки

Строки являются упорядоченными последовательностями символов. Длина строки ограничена лишь объемом оперативной памяти компьютера. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`).

Кроме того, строки относятся к неизменяемым типам данных. Поэтому практически все строковые методы в качестве значения возвращают новую строку. При использовании небольших строк это не приводит к каким-либо проблемам, но при работе с большими строками можно столкнуться с проблемой нехватки памяти. Иными словами, можно получить символ по индексу, но изменить его нельзя (листинг 6.1).

Листинг 6.1. Попытка изменить символ по индексу

```
>>> s = "Python"
>>> s[0]                    # Можно получить символ по индексу
'P'
>>> s[0] = "J"              # Изменить строку нельзя
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    s[0] = "J"             # Изменить строку нельзя
TypeError: 'str' object does not support item assignment
```

В некоторых языках программирования концом строки является нулевой символ. В языке Python нулевой символ может быть расположен внутри строки:

```
>>> "string\x00string" # Нулевой символ — это НЕ конец строки
'string\x00string'
```

Язык Python 3 поддерживает следующие строковые типы:

- `str` — Unicode-строка. Обратите внимание, что я не указал в какой конкретно кодировке (UTF-8, UTF-16 или UTF-32) хранятся символы. Рассматривайте такие строки, как строки в некой абстрактной кодировке, позволяющие хранить символы Unicode и производить манипуляции с ними. При выводе Unicode-строку необходимо преобразовать в последовательность байтов в какой-либо кодировке:

  ```
  >>> type("строка")
  <class 'str'>
  ```

```
>>> "строка".encode(encoding="cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> "строка".encode(encoding="utf-8")
b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
```

- `bytes` — неизменяемая последовательность байтов. Каждый элемент последовательности может хранить целое число от `0` до `255`, которое обозначает код символа. Объект типа `bytes` поддерживает большинство строковых методов и, если это возможно, выводится как последовательность символов. Однако доступ по индексу возвращает целое число, а не символ. Пример:

```
>>> s = bytes("стр str", "cp1251")
>>> s[0], s[5], s[0:3], s[4:7]
(241, 116, b'\xf1\xf2\xf0', b'str')
>>> s
b'\xf1\xf2\xf0 str'
```

  Объект типа `bytes` может содержать как однобайтовые символы, так и многобайтовые. Обратите внимание на то, что функции и методы строк некорректно работают с многобайтовыми кодировками, например, функция `len()` вернет количество байтов, а не символов:

```
>>> len("строка")
6
>>> len(bytes("строка", "cp1251"))
6
>>> len(bytes("строка", "utf-8"))
12
```

- `bytearray` — изменяемая последовательность байтов. Тип `bytearray` аналогичен типу `bytes`, но позволяет изменять элементы по индексу и содержит дополнительные методы, позволяющие добавлять и удалять элементы. Пример:

```
>>> s = bytearray("str", "cp1251")
>>> s[0] = 49; s                # Можно изменить символ
bytearray(b'1tr')
>>> s.append(55); s             # Можно добавить символ
bytearray(b'1tr7')
```

Во всех случаях, когда речь идет о текстовых данных, следует использовать тип `str`. Именно этот тип мы будем называть словом "строка". Типы `bytes` и `bytearray` следует использовать для хранения бинарных данных, например, изображений, а также для промежуточного хранения текстовых данных. Более подробно мы рассмотрим типы `bytes` и `bytearray` в конце этой главы.

6.1. Создание строки

Создать строку можно следующими способами:

- с помощью функции `str([<Объект>[, <Кодировка>[, <Обработка ошибок>]]])`. Если указан только первый параметр, то функция возвращает строковое представление любого объекта. Если параметры не указаны вообще, то возвращается пустая строка.

Пример:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0")
"b'\\xf1\\xf2\\xf0\\xee\\xea\\xe0'"
```

Обратите внимание на преобразование объекта типа `bytes`. Мы получили строковое представление объекта, а не нормальную строку. Чтобы получить именно строку из объектов типа `bytes` и `bytearray`, следует указать кодировку во втором параметре:

```
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0", "cp1251")
'строка'
```

В третьем параметре могут быть указаны значения "strict" (при ошибке возбуждается исключение `UnicodeDecodeError`; значение по умолчанию), "replace" (неизвестный символ заменяется символом, имеющим код `\uFFFD`) или "ignore" (неизвестные символы игнорируются):

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    str(obj1, "ascii", "strict")
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
'1'
```

- указав строку между апострофами или двойными кавычками:

```
>>> 'строка', "строка", '"x": 5', "'x': 5"
('строка', 'строка', '"x": 5', "'x': 5")
>>> print('Строка1\nСтрока2')
Строка1
Строка2
>>> print("Строка1\nСтрока2")
Строка1
Строка2
```

В некоторых языках программирования (например, в PHP) строка в апострофах отличается от строки в кавычках тем, что внутри апострофов специальные символы выводятся как есть, а внутри кавычек спецсимволы интерпретируются. В языке Python никакого отличия между строкой в апострофах и строкой в кавычках нет. Это одно и то же. Если строка содержит кавычки, то ее лучше заключить в апострофы и наоборот. Все специальные символы в таких строках интерпретируются. Например, последовательность символов `\n` преобразуется в символ новой строки. Чтобы специальный символ выводился как есть, его необходимо экранировать с помощью слэша:

```
>>> print("Строка1\\nСтрока2")
Строка1\nСтрока2
>>> print('Строка1\\nСтрока2')
Строка1\nСтрока2
```

Кавычку, внутри строки в кавычках, и апостроф, внутри строки в апострофах, также необходимо экранировать с помощью защитного слэша:

```
>>> "\"x\": 5", '\'x\': 5'
('"x": 5', "'x': 5")
```

Следует также заметить, что заключить объект в одинарные кавычки (или апострофы) на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку:

```
>>> "string
SyntaxError: EOL while scanning string literal
```

Чтобы расположить объект на нескольких строках, следует перед символом перевода строки указать символ \, поместить две строки внутри скобок или использовать конкатенацию внутри скобок:

```
>>> "string1\
string2"             # После \ не должно быть никаких символов
'string1string2'
>>> ("string1"
"string2")           # Неявная конкатенация строк
'string1string2'
>>> ("string1" +
"string2")           # Явная конкатенация строк
'string1string2'
```

Кроме того, если в конце строки расположен символ \, то его необходимо экранировать, иначе будет выведено сообщение об ошибке:

```
>>> print("string\")

SyntaxError: EOL while scanning string literal
>>> print("string\\")
string\
```

- указав строку между утроенными апострофами или утроенными кавычками. Такие объекты можно разместить на нескольких строках, а также одновременно использовать кавычки и апострофы без необходимости их экранировать. В остальном такие объекты эквивалентны строкам в апострофах и кавычках. Все специальные символы в таких строках интерпретируются. Примеры:

```
>>> print('''Строка1
Строка2''')
Строка1
Строка2
>>> print("""Строка1
Строка2""")
Строка1
Строка2
```

Если строка не присваивается переменной, то она считается строкой документирования. Такая строка сохраняется в атрибуте __doc__ того объекта, в котором расположена. В качестве примера создадим функцию со строкой документирования, а затем выведем содержимое строки:

```
>>> def test():
        """Это описание функции"""
        pass

>>> print(test.__doc__)
Это описание функции
```

Так как выражения внутри таких строк не выполняются, то утроенные кавычки (или утроенные апострофы) очень часто используются для комментирования больших фрагментов кода на этапе отладки программы.

Если перед строкой разместить модификатор `r`, то специальные символы внутри строки выводятся как есть. Например, символ `\n` не будет преобразован в символ перевода строки. Иными словами, он будет считаться последовательностью двух символов: `\` и `n`:

```
>>> print("Строка1\nСтрока2")
Строка1
Строка2
>>> print(r"Строка1\nСтрока2")
Строка1\nСтрока2
>>> print(r"""Строка1\nСтрока2""")
Строка1\nСтрока2
```

Такие неформатированные строки удобно использовать в шаблонах регулярных выражений, а также при указании пути к файлу или каталогу:

```
>>> print(r"C:\Python32\lib\site-packages")
C:\Python32\lib\site-packages
```

Если модификатор не указать, то все слэши в пути необходимо экранировать:

```
>>> print("C:\\Python32\\lib\\site-packages")
C:\Python32\lib\site-packages
```

Если в конце неформатированной строки расположен слэш, то его необходимо экранировать. Однако следует учитывать, что этот слэш будет добавлен в исходную строку. Пример:

```
>>> print(r"C:\Python32\lib\site-packages\")

SyntaxError: EOL while scanning string literal
>>> print(r"C:\Python32\lib\site-packages\\")
C:\Python32\lib\site-packages\\
```

Чтобы избавиться от лишнего слэша, можно использовать операцию конкатенации строк, обычные строки или удалить слэш явным образом:

```
>>> print(r"C:\Python32\lib\site-packages" + "\\") # Конкатенация
C:\Python32\lib\site-packages\
>>> print("C:\\Python32\\lib\\site-packages\\")    # Обычная строка
C:\Python32\lib\site-packages\
>>> print(r"C:\Python32\lib\site-packages\\"[:-1]) # Удаление слэша
C:\Python32\lib\site-packages\
```

6.2. Специальные символы

Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы, которые невозможно вставить обычным способом. Перечислим специальные символы, допустимые внутри строки, перед которой нет модификатора `r`:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\t` — знак табуляции;
- `\v` — вертикальная табуляция;
- `\a` — звонок;
- `\b` — забой;
- `\f` — перевод формата;
- `\0` — нулевой символ (не является концом строки);
- `\"` — кавычка;
- `\'` — апостроф;
- `\N` — восьмеричное значение `N`. Например, `\74` соответствует символу `<`;
- `\xN` — шестнадцатеричное значение `N`. Например, `\x6a` соответствует символу `j`;
- `\\` — обратный слэш;
- `\uxxxx` — 16-битный символ Unicode. Например, `\u043a` соответствует русской букве `к`;
- `\Uxxxxxxxx` — 32-битный символ Unicode.

Если после слэша не стоит символ, который вместе со слэшем интерпретируется как спецсимвол, то слэш сохраняется в составе строки:

```
>>> print("Этот символ \не специальный")
Этот символ \не специальный
```

Тем не менее, лучше экранировать слэш явным образом:

```
>>> print("Этот символ \\не специальный")
Этот символ \не специальный
```

6.3. Операции над строками

Как вы уже знаете, строки относятся к последовательностям. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение. Рассмотрим эти операции подробно.

К любому символу строки можно обратиться как к элементу списка. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля:

```
>>> s = "Python"
>>> s[0], s[1], s[2], s[3], s[4], s[5]
('P', 'y', 't', 'h', 'o', 'n')
```

Если символ, соответствующий указанному индексу, отсутствует в строке, то возбуждается исключение `IndexError`:

```
>>> s = "Python"
>>> s[10]
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    s[10]
IndexError: string index out of range
```

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца строки, а точнее, значение вычитается из длины строки, чтобы получить положительный индекс:

```
>>> s = "Python"
>>> s[-1], s[len(s)-1]
('n', 'n')
```

Так как строки относятся к неизменяемым типам данных, то изменить символ по индексу нельзя:

```
>>> s = "Python"
>>> s[0] = "J"                          # Изменить строку нельзя
Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    s[0] = "J"                         # Изменить строку нельзя
TypeError: 'str' object does not support item assignment
```

Чтобы выполнить изменение, можно воспользоваться операцией извлечения среза, которая возвращает указанный фрагмент строки. Формат операции:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры являются необязательными. Если параметр `<Начало>` не указан, то используется значение `0`. Если параметр `<Конец>` не указан, то возвращается фрагмент до конца строки. Следует также заметить, что символ с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве значения параметров можно указать отрицательные значения.

Теперь рассмотрим несколько примеров. Сначала получим копию строки:

```
>>> s = "Python"
>>> s[:] # Возвращается фрагмент от позиции 0 до конца строки
'Python'
```

Теперь выведем символы в обратном порядке:

```
>>> s[::-1] # Указываем отрицательное значение в параметре <Шаг>
'nohtyP'
```

Заменим первый символ в строке:

```
>>> "J" + s[1:] # Извлекаем фрагмент от символа 1 до конца строки
'Jython'
```

Удалим последний символ:

```
>>> s[:-1] # Возвращается фрагмент от 0 до len(s)-1
'Pytho'
```

Получим первый символ в строке:

```
>>> s[0:1] # Символ с индексом 1 не входит в диапазон
'P'
```

А теперь получим последний символ:

```
>>> s[-1:] # Получаем фрагмент от len(s)-1 до конца строки
'n'
```

И, наконец, выведем символы с индексами 2, 3 и 4:

```
>>> s[2:5] # Возвращаются символы с индексами 2, 3 и 4
'tho'
```

Узнать количество символов в строке позволяет функция `len()`:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
(6, 3, 6)
```

Теперь, когда мы знаем количество символов, можно перебрать все символы с помощью цикла `for`:

```
>>> s = "Python"
>>> for i in range(len(s)): print(s[i], end=" ")
```

Результат выполнения:

```
P y t h o n
```

Так как строки поддерживают итерации, мы можем просто указать строку в качестве параметра цикла:

```
>>> s = "Python"
>>> for i in s: print(i, end=" ")
```

Результат выполнения будет таким же:

```
P y t h o n
```

Соединить две строки в одну строку позволяет оператор `+`:

```
>>> print("Строка1" + "Строка2")
Строка1Строка2
```

Кроме того, можно выполнить неявную конкатенацию строк. В этом случае две строки указываются рядом без оператора между ними:

```
>>> print("Строка1" "Строка2")
Строка1Строка2
```

Обратите внимание на то, что если между строками указать запятую, то мы получим кортеж, а не строку:

```
>>> s = "Строка1", "Строка2"
>>> type(s)                          # Получаем кортеж, а не строку
<class 'tuple'>
```

Если соединяются, например, переменная и строка, то следует обязательно указывать символ конкатенации строк, иначе будет выведено сообщение об ошибке:

```
>>> s = "Строка1"
>>> print(s + "Строка2")             # Нормально
Строка1Строка2
>>> print(s "Строка2")               # Ошибка
SyntaxError: invalid syntax
```

При необходимости соединить строку с другим типом данных (например, с числом) следует произвести явное преобразование типов с помощью функции `str()`:

```
>>> "string" + str(10)
'string10'
```

Кроме рассмотренных операций, строки поддерживают операцию повторения и проверку на вхождение. Повторить строку указанное количество раз можно с помощью оператора `*`, а выполнить проверку на вхождение фрагмента в строку позволяет оператор `in`:

```
>>> "-" * 20
'--------------------'
>>> "yt" in "Python"              # Найдено
True
>>> "yt" in "Perl"                # Не найдено
False
```

6.4. Форматирование строк

Вместо соединения строк с помощью оператора `+` лучше использовать форматирование. Данная операция позволяет соединять строку с любым другим типом данных и выполняется быстрее конкатенации.

> ***Примечание***
>
> В последующих версиях Python оператор форматирования `%` может быть удален. Вместо этого оператора в новом коде следует использовать метод `format()`, который рассматривается в следующем разделе.

Форматирование имеет следующий синтаксис:

```
<Строка специального формата> % <Значения>
```

Внутри параметра `<Строка специального формата>` могут быть указаны спецификаторы, имеющие следующий синтаксис:

```
%[(<Ключ>)][<Флаг>][<Ширина>][.<Точность>]<Тип преобразования>
```

Количество спецификаторов внутри строки должно быть равно количеству элементов в параметре `<Значения>`. Если используется только один спецификатор, то параметр `<Значения>` может содержать одно значение, в противном случае необходимо перечислить значения через запятую внутри круглых скобок, создавая тем самым кортеж. Пример:

```
>>> "%s" % 10                          # Один элемент
'10'
>>> "%s — %s — %s" % (10, 20, 30)      # Несколько элементов
'10 — 20 — 30'
```

Параметры внутри спецификатора имеют следующий смысл:

- `<Ключ>` — ключ словаря. Если задан ключ, то в параметре `<Значения>` необходимо указать словарь, а не кортеж. Пример:

  ```
  >>> "%(name)s — %(year)s" % {"year": 1978, "name": "Nik"}
  'Nik — 1978'
  ```

- <Флаг> — флаг преобразования. Может содержать следующие значения:
  - # — для восьмеричных значений добавляет в начало комбинацию символов 0o, для шестнадцатеричных значений добавляет комбинацию символов 0x (если используется тип x) или 0X (если используется тип X), для вещественных чисел указывает всегда выводить дробную точку, даже если задано значение 0 в параметре <Точность>:

```
>>> print("%#o %#o %#o" % (0o77, 10, 10.5))
0o77 0o12 0o12
>>> print("%#x %#x %#x" % (0xff, 10, 10.5))
0xff 0xa 0xa
>>> print("%#X %#X %#X" % (0xff, 10, 10.5))
0XFF 0XA 0XA
>>> print("%#.0F %.0F" % (300, 300))
300. 300
```

  - 0 — задает наличие ведущих нулей для числового значения:

```
>>> "'%d' — '%05d'" % (3, 3) # 5 — ширина поля
"'3' — '00003'"
```

  - - — задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Если флаг указан одновременно с флагом 0, то действие флага 0 будет отменено. Пример:

```
>>> "'%5d' — '%-5d'" % (3, 3) # 5 — ширина поля
"'    3' — '3    '"
>>> "'%05d' — '%-05d'" % (3, 3)
"'00003' — '3    '"
```

  - пробел — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```
>>> "'% d' — '% d'" % (-3, 3)
"'-3' — ' 3'"
```

  - + — задает обязательный вывод знака, как для отрицательных, так и для положительных чисел. Если флаг + указан одновременно с флагом пробел, то действие флага пробел будет отменено. Пример:

```
>>> "'%+d' — '%+d'" % (-3, 3)
"'-3' — '+3'"
```

- <Ширина> — минимальная ширина поля. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
>>> "'%10d' — '%-10d'" % (3, 3)
"'         3' — '3         '"
>>> "'%3s''%10s'" % ("string", "string")
"'string''    string'"
```

  Вместо значения можно указать символ "*". В этом случае значение следует задать внутри кортежа:

```
>>> "'%*s''%10s'" % (10, "string", "str")
"'    string''       str'"
```

- <Точность> — количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка. Пример:

```
>>> import math
>>> "%s %f %.2f" % (math.pi, math.pi, math.pi)
'3.141592653589793 3.141593 3.14'
```

  Вместо значения можно указать символ "*". В этом случае значение следует задать внутри кортежа:

```
>>> "'%*.*f'" % (8, 5, math.pi)
"' 3.14159'"
```

- <Тип преобразования> — задает тип преобразования. Параметр является обязательным.

В параметре <Тип преобразования> могут быть указаны следующие символы:

- s — преобразует любой объект в строку с помощью функции str():

```
>>> print("%s" % ("Обычная строка"))
Обычная строка
>>> print("%s %s %s" % (10, 10.52, [1, 2, 3]))
10 10.52 [1, 2, 3]
```

- r — преобразует любой объект в строку с помощью функции repr():

```
>>> print("%r" % ("Обычная строка"))
'Обычная строка'
```

- a — преобразует объект в строку с помощью функции ascii():

```
>>> print("%a" % ("строка"))
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

- c — выводит одиночный символ или преобразует числовое значение в символ. В качестве примера выведем числовое значение и соответствующий этому значению символ:

```
>>> for i in range(33, 127): print("%s => %c" % (i, i))
```

- d и i — возвращают целую часть числа:

```
>>> print("%d %d %d" % (10, 25.6, -80))
10 25 -80
>>> print("%i %i %i" % (10, 25.6, -80))
10 25 -80
```

- o — восьмеричное значение:

```
>>> print("%o %o %o" % (0o77, 10, 10.5))
77 12 12
>>> print("%#o %#o %#o" % (0o77, 10, 10.5))
0o77 0o12 0o12
```

- x — шестнадцатеричное значение в нижнем регистре:

```
>>> print("%x %x %x" % (0xff, 10, 10.5))
ff a a
>>> print("%#x %#x %#x" % (0xff, 10, 10.5))
0xff 0xa 0xa
```

- `X` — шестнадцатеричное значение в верхнем регистре:

```
>>> print("%X %X %X" % (0xff, 10, 10.5))
FF A A
>>> print("%#X %#X %#X" % (0xff, 10, 10.5))
0XFF 0XA 0XA
```

- `f` и `F` — вещественное число в десятичном представлении:

```
>>> print("%f %f %f" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
>>> print("%F %F %F" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
>>> print("%#.0F %.0F" % (300, 300))
300. 300
```

- `e` — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):

```
>>> print("%e %e" % (3000, 18657.81452))
3.000000e+03 1.865781e+04
```

- `E` — вещественное число в экспоненциальной форме (буква "e" в верхнем регистре):

```
>>> print("%E %E" % (3000, 18657.81452))
3.000000E+03 1.865781E+04
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
>>> print("%g %g %g" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578e-05 1.865e-05
```

- `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
>>> print("%G %G %G" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578E-05 1.865E-05
```

Если внутри строки необходимо использовать символ процента, то этот символ следует удвоить, иначе будет выведено сообщение об ошибке:

```
>>> print("% %s" % ("- это символ процента")) # Ошибка
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    print("% %s" % ("- это символ процента")) # Ошибка
TypeError: not all arguments converted during string formatting
>>> print("%% %s" % ("- это символ процента")) # Нормально
% — это символ процента
```

Форматирование строк очень удобно использовать при передаче данных в шаблон HTML-страницы. Для этого заполняем словарь данными и указываем его справа от символа `%`, а сам шаблон — слева. Продемонстрируем это на примере (листинг 6.2).

Листинг 6.2. Пример использования форматирования строк

```
# -*- coding: utf-8 -*-
html = """<html>
<head><title>%(title)s</title>
</head>
```

```
<body>
<h1>%(h1)s</h1>
<div>%(content)s</div>
</body>
</html>"""
arr = {"title":    "Это название документа",
       "h1":       "Это заголовок первого уровня",
       "content":  "Это основное содержание страницы"}
print(html % arr) # Подставляем значения и выводим шаблон
input()
```

Результат выполнения:

```
<html>
<head><title>Это название документа</title>
</head>
<body>
<h1>Это заголовок первого уровня</h1>
<div>Это основное содержание страницы</div>
</body>
</html>
```

Для форматирования строк можно также использовать следующие методы:

- `expandtabs([<Ширина поля>])` — заменяет символ табуляции пробелами таким образом, чтобы общая ширина фрагмента вместе с текстом (расположенным перед символом табуляции) была равна указанной величине. Если параметр не указан, то ширина поля предполагается равной 8 символам. Пример:

```
>>> s = "1\t12\t123\t"
>>> "'%s'" % s.expandtabs(4)
"'1   12  123 '"
```

  В этом примере ширина задана равной четырем символам. Поэтому во фрагменте `"1\t"` табуляция будет заменена тремя пробелами, во фрагменте `"12\t"` — двумя пробелами, а во фрагменте `"123\t"` — одним пробелом. Во всех трех фрагментах ширина будет равна четырем символам.

  Если перед символом табуляции нет текста или количество символов перед табуляцией равно ширине, то табуляция заменяется указанным количеством пробелов:

```
>>> s = "\t"
>>> "'%s' — '%s'" % (s.expandtabs(), s.expandtabs(4))
"'        ' — '    '"
>>> s = "1234\t"
>>> "'%s'" % s.expandtabs(4)
"'1234    '"
```

  Если количество символов перед табуляцией больше ширины, то табуляция заменяется пробелами таким образом, чтобы ширина фрагмента вместе с текстом делилась без остатка на указанную ширину:

```
>>> s = "12345\t123456\t1234567\t1234567890\t"
>>> "'%s'" % s.expandtabs(4)
"'12345   123456  1234567 1234567890  '"
```

Таким образом, если количество символов перед табуляцией больше 4, но менее 8, то фрагмент дополняется пробелами до 8 символов. Если количество символов больше 8, но менее 12, то фрагмент дополняется пробелами до 12 символов и т. д. Все это справедливо при указании в качестве параметра числа 4;

- `center(<Ширина>[, <Символ>])` — производит выравнивание строки по центру внутри поля, указанной ширины. Если второй параметр не указан, то справа и слева от исходной строки будут добавлены пробелы. Пример:

```
>>> s = "str"
>>> s.center(15), s.center(11, "-")
('      str      ', '----str----')
```

Теперь произведем выравнивание трех фрагментов шириной 15 символов. Первый фрагмент будет выровнен по правому краю, второй — по левому, а третий — по центру:

```
>>> s = "str"
>>> "'%15s' '%-15s' '%s'" % (s, s, s.center(15))
"'            str' 'str            ' '      str      '"
```

Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью:

```
>>> s = "string"
>>> s.center(6), s.center(5)
('string', 'string')
```

- `ljust(<Ширина>[, <Символ>])` — производит выравнивание строки по левому краю внутри поля указанной ширины. Если второй параметр не указан, то справа от исходной строки будут добавлены пробелы. Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> s = "string"
>>> s.ljust(15), s.ljust(15, "-")
('string         ', 'string---------')
>>> s.ljust(6), s.ljust(5)
('string', 'string')
```

- `rjust(<Ширина>[, <Символ>])` — производит выравнивание строки по правому краю внутри поля указанной ширины. Если второй параметр не указан, то слева от исходной строки будут добавлены пробелы. Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> s = "string"
>>> s.rjust(15), s.rjust(15, "-")
('         string', '---------string')
>>> s.rjust(6), s.rjust(5)
('string', 'string')
```

- `zfill(<Ширина>)` — производит выравнивание фрагмента по правому краю внутри поля указанной ширины. Слева от фрагмента будут добавлены нули. Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> "5".zfill(20), "123456".zfill(5)
('00000000000000000005', '123456')
```

6.5. Метод *format()*

Начиная с Python 2.6, помимо операции форматирования, строки поддерживают метод `format()`. Метод имеет следующий синтаксис:

```
<Строка> = <Строка специального формата>.format(*args, **kwargs)
```

В параметре `<Строка специального формата>` внутри символов `{` и `}` указываются спецификаторы, имеющие следующий синтаксис:

```
{[<Поле>][!<Функция>][:<Формат>]}
```

Все символы, расположенные вне фигурных скобок, выводятся без преобразований. Если внутри строки необходимо использовать символы `{` и `}`, то эти символы следует удвоить, иначе возбуждается исключение `ValueError`. Пример:

```
>>> print("Символы {{ и }} — {0}".format("специальные"))
Символы { и } — специальные
```

В параметре `<Поле>` можно указать индекс позиции (нумерация начинается с нуля) или ключ. Допустимо комбинировать позиционные и именованные параметры. В этом случае в методе `format()` именованные параметры указываются в самом конце. Пример:

```
>>> "{0} — {1} — {2}".format(10, 12.3, "string")      # Индексы
'10 — 12.3 — string'
>>> arr = [10, 12.3, "string"]
>>> "{0} — {1} — {2}".format(*arr)                    # Индексы
'10 — 12.3 — string'
>>> "{model} — {color}".format(color="red", model="BMW") # Ключи
'BMW — red'
>>> d = {"color": "red", "model": "BMW"}
>>> "{model} — {color}".format(**d)                   # Ключи
'BMW — red'
>>> "{color} — {0}".format(2010, color="red")         # Комбинация
'red — 2010'
```

В качестве параметра в методе `format()` можно указать объект. Для доступа к элементам по индексу внутри строки формата применяются квадратные скобки, а для доступа к атрибутам объекта используется точечная нотация:

```
>>> arr = [10, [12.3, "string"] ]
>>> "{0[0]} — {0[1][0]} — {0[1][1]}".format(arr)      # Индексы
'10 — 12.3 — string'
>>> "{arr[0]} — {arr[1][1]}".format(arr=arr)          # Индексы
'10 — string'
>>> class Car: color, model = "red", "BMW"

>>> car = Car()
>>> "{0.model} — {0.color}".format(car)               # Атрибуты
'BMW — red'
```

Существует также краткая форма записи, при которой параметр `<Поле>` не указывается. В этом случае скобки без указанного индекса нумеруются слева направо, начиная с нуля:

```
>>> "{} — {} — {} — {n}".format(1, 2, 3, n=4) # "{0} — {1} — {2} — {n}"
'1 — 2 — 3 — 4'
>>> "{} — {} — {n} — {}".format(1, 2, 3, n=4) # "{0} — {1} — {n} — {2}"
'1 — 2 — 4 — 3'
```

Параметр <Функция> задает функцию, с помощью которой обрабатываются данные перед вставкой в строку. Если указано значение "s", то данные обрабатываются функцией str(), если значение "r", то функцией repr(), а если значение "a", то функцией ascii(). Если параметр не указан, то для преобразования данных в строку используется функция str(). Пример:

```
>>> print("{0!s}".format("строка"))                    # str()
строка
>>> print("{0!r}".format("строка"))                    # repr()
'строка'
>>> print("{0!a}".format("строка"))                    # ascii()
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

В параметре <Формат> указывается значение, имеющее следующий синтаксис:

```
[[<Заполнитель>]<Выравнивание>][<Знак>][#][0][<Ширина>][,]
[.<Точность>][<Преобразование>]
```

Параметр <Ширина> задает минимальную ширину поля. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
>>> "'{0:10}' '{1:3}'".format(3, "string")
"'         3' 'string'"
```

Ширину поля можно передать в качестве параметра в методе format(). В этом случае вместо числа указывается индекс параметра внутри фигурных скобок:

```
>>> "'{0:{1}}'".format(3, 10) # 10 — это ширина поля
"'         3'"
```

По умолчанию значение внутри поля выравнивается по правому краю. Управлять выравниванием позволяет параметр <Выравнивание>. Можно указать следующие значения:

- < — по левому краю;
- > — по правому краю;
- ^ — по центру поля. Пример:

```
>>> "'{0:<10}' '{1:>10}' '{2:^10}'".format(3, 3, 3)
"'3         ' '         3' '    3     '"
```

- = — знак числа выравнивается по левому краю, а число по правому краю:

```
>>> "'{0:=10}' '{1:=10}'".format(-3, 3)
"'-        3' '         3'"
```

Как видно из предыдущего примера, пространство между знаком и числом по умолчанию заполняется пробелами, а знак положительного числа не указывается. Чтобы вместо пробелов пространство заполнялось нулями, необходимо указать нуль перед шириной поля:

```
>>> "'{0:=010}' '{1:=010}'".format(-3, 3)
"'-000000003' '0000000003'"
```

Такого же эффекта можно достичь, указав нуль в параметре <Заполнитель>. В этом параметре допускаются и другие символы, которые будут выводиться вместо пробелов:

```
>>> "'{0:0=10}' '{1:0=10}'".format(-3, 3)
"'-000000003' '0000000003'"
>>> "'{0:*<10}' '{1:+>10}' '{2:.^10}'".format(3, 3, 3)
"'3*********' '+++++++++3' '....3.....'"
```

Управлять выводом знака числа позволяет параметр <Знак>. Допустимые значения:

- + — задает обязательный вывод знака как для отрицательных, так и для положительных чисел;
- - — вывод знака только для отрицательных чисел (значение по умолчанию);
- пробел — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```
>>> "'{0:+}' '{1:+}' '{0:-}' '{1:-}'".format(3, -3)
"'+3' '-3' '3' '-3'"
>>> "'{0: }' '{1: }'".format(3, -3)        # Пробел
"' 3' '-3'"
```

Для целых чисел в параметре <Преобразование> могут быть указаны следующие опции:

- b — двоичное значение:

```
>>> "'{0:b}' '{0:#b}'".format(3)
"'11' '0b11'"
```

- c — преобразует целое число в соответствующий символ:

```
>>> "'{0:c}'".format(100)
"'d'"
```

- d — десятичное значение;
- n — аналогично опции d, но учитывает настройки локали. Например, выведем большое число с разделением тысячных разрядов пробелом:

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, 'Russian_Russia.1251')
'Russian_Russia.1251'
>>> print("{0:n}".format(100000000).replace("\uffa0", " "))
100 000 000
```

В Python 3.2.0 между разрядами вставляется символ с кодом \uffa0, который отображается квадратиком. Чтобы вывести символ пробела, мы производим замену в строке с помощью метода replace(). В версии 2.6 поведение было другим. Там вставлялся символ с кодом \xa0 и не нужно было производить замену. Чтобы в Python 3.2.0 вставлялся символ с кодом \xa0, следует воспользоваться функцией format() из модуля locale:

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> print(locale.format("%d", 100000000, grouping=True))
100 000 000
>>> locale.localeconv()["thousands_sep"]
'\xa0'
```

Можно также разделить тысячные разряды запятой, указав ее в строке формата:

```
>>> print("{0:,d}".format(100000000))
100,000,000
```

- `o` — восьмеричное значение:

```
>>> "'{0:d}' '{0:o}' '{0:#o}'".format(511)
"'511' '777' '0o777'"
```

- `x` — шестнадцатеричное значение в нижнем регистре:

```
>>> "'{0:x}' '{0:#x}'".format(255)
"'ff' '0xff'"
```

- `X` — шестнадцатеричное значение в верхнем регистре:

```
>>> "'{0:X}' '{0:#X}'".format(255)
"'FF' '0XFF'"
```

Для вещественных чисел в параметре `<Преобразование>` могут быть указаны следующие опции:

- `f` и `F` — вещественное число в десятичном представлении:

```
>>> "'{0:f}' '{1:f}' '{2:f}'".format(30, 18.6578145, -2.5)
"'30.000000' '18.657815' '-2.500000'"
```

Задать количество знаков после запятой позволяет параметр `<Точность>`:

```
>>> "'{0:.7f}' '{1:.2f}'".format(18.6578145, -2.5)
"'18.6578145' '-2.50'"
```

- `e` — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):

```
>>> "'{0:e}' '{1:e}'".format(3000, 18657.81452)
"'3.000000e+03' '1.865781e+04'"
```

- `E` — вещественное число в экспоненциальной форме (буква "e" в верхнем регистре):

```
>>> "'{0:E}' '{1:E}'".format(3000, 18657.81452)
"'3.000000E+03' '1.865781E+04'"
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
>>> "'{0:g}' '{1:g}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578e-05'"
```

- `n` — аналогично опции `g`, но учитывает настройки локали;
- `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
>>> "'{0:G}' '{1:G}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578E-05'"
```

- `%` — умножает число на 100 и добавляет символ процента в конец. Значение отображается в соответствии с опцией `f`. Пример:

```
>>> "'{0:%}' '{1:.4%}'".format(0.086578, 0.000086578)
"'8.657800%' '0.0087%'"
```

6.6. Функции и методы для работы со строками

Рассмотрим основные функции для работы со строками:

- `str([<Объект>])` — преобразует любой объект в строку. Если параметр не указан, то возвращается пустая строка. Используется функцией `print()` для вывода объектов. Пример:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
>>> print("строка1\nстрока2")
строка1
строка2
```

- `repr(<Объект>)` — возвращает строковое представление объекта. Используется при выводе объектов в окне **Python Shell** редактора IDLE. Пример:

```
>>> repr("Строка"), repr([1, 2, 3]), repr({"x": 5})
("'Строка'", '[1, 2, 3]', "{'x': 5}")
>>> repr("строка1\nстрока2")
"'строка1\\nстрока2'"
```

- `ascii(<Объект>)` — возвращает строковое представление объекта. В строке могут быть символы только из кодировки ASCII. Пример:

```
>>> ascii([1, 2, 3]), ascii({"x": 5})
('[1, 2, 3]', "{'x': 5}")
>>> ascii("строка")
"'\\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
```

- `len(<Строка>)` — возвращает количество символов в строке:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
(6, 3, 6)
>>> len("строка")
6
```

Перечислим основные методы:

- `strip([<Символы>])` — удаляет пробельные или указанные символы в начале и конце строки. Пробельными символами считаются: пробел, символ перевода строки (`\n`), символ возврата каретки (`\r`), символы горизонтальной (`\t`) и вертикальной (`\v`) табуляции:

```
>>> s1, s2 = "     str\n\r\v\t", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.strip(), s2.strip("tsr"))
"'str' — 'ok'"
```

- `lstrip([<Символы>])` — удаляет пробельные или указанные символы в начале строки:

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.lstrip(), s2.lstrip("tsr"))
"'str     ' — 'okstrstrstr'"
```

- `rstrip([<Символы>])` — удаляет пробельные или указанные символы в конце строки:

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.rstrip(), s2.rstrip("tsr"))
"'     str' — 'strstrstrok'"
```

- `split([<Разделитель>[, <Лимит>]])` — разделяет строку на подстроки по указанному разделителю и добавляет их в список. Если первый параметр не указан или имеет значение `None`, то в качестве разделителя используется символ пробела. Если во втором параметре задано число, то в списке будет указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент, в котором будет остаток строки. Примеры:

```
>>> s = "word1 word2 word3"
>>> s.split(), s.split(None, 1)
(['word1', 'word2', 'word3'], ['word1', 'word2 word3'])
>>> s = "word1\nword2\nword3"
>>> s.split("\n")
['word1', 'word2', 'word3']
```

  Если в строке содержатся несколько пробелов подряд и разделитель не указан, то пустые элементы не будут добавлены в список:

```
>>> s = "word1           word2 word3     "
>>> s.split()
['word1', 'word2', 'word3']
```

  При использовании другого разделителя могут быть пустые элементы:

```
>>> s = ",,word1,,word2,,word3,,"
>>> s.split(",")
['', '', 'word1', '', 'word2', '', 'word3', '', '']
>>> "1,,2,,3".split(",")
['1', '', '2', '', '3']
```

  Если разделитель не найден в строке, то список будет состоять из одного элемента, представляющего исходную строку:

```
>>> "word1 word2 word3".split("\n")
['word1 word2 word3']
```

- `rsplit([<Разделитель>[, <Лимит>]])` — метод аналогичен методу `split()`, но поиск символа-разделителя производится не слева направо, а, наоборот, справа налево. Примеры:

```
>>> s = "word1 word2 word3"
>>> s.rsplit(), s.rsplit(None, 1)
(['word1', 'word2', 'word3'], ['word1 word2', 'word3'])
>>> "word1\nword2\nword3".rsplit("\n")
['word1', 'word2', 'word3']
```

- `splitlines([True])` — разделяет строку на подстроки по символу перевода строки (`\n`) и добавляет их в список. Символы новой строки включаются в результат, только если необязательный параметр имеет значение `True`. Если разделитель не найден в строке, то список будет содержать только один элемент. Примеры:

```
>>> "word1\nword2\nword3".splitlines()
['word1', 'word2', 'word3']
>>> "word1\nword2\nword3".splitlines(True)
['word1\n', 'word2\n', 'word3']
>>> "word1\nword2\nword3".splitlines(False)
```

```
['word1', 'word2', 'word3']
>>> "word1 word2 word3".splitlines()
['word1 word2 word3']
```

- `partition(<Разделитель>)` — находит первое вхождение символа-разделителя в строку и возвращает кортеж из трех элементов. Первый элемент будет содержать фрагмент, расположенный перед разделителем, второй элемент — символ-разделитель, а третий элемент — фрагмент, расположенный после символа-разделителя. Поиск производится слева направо. Если символ-разделитель не найден, то первый элемент кортежа будет содержать всю строку, а остальные элементы будут пустыми. Пример:

```
>>> "word1 word2 word3".partition(" ")
('word1', ' ', 'word2 word3')
>>> "word1 word2 word3".partition("\n")
('word1 word2 word3', '', '')
```

- `rpartition(<Разделитель>)` — метод аналогичен методу `partition()`, но поиск символа-разделителя производится не слева направо, а, наоборот, справа налево. Если символ-разделитель не найден, то первые два элемента кортежа будут пустыми, а третий элемент будет содержать всю строку. Пример:

```
>>> "word1 word2 word3".rpartition(" ")
('word1 word2', ' ', 'word3')
>>> "word1 word2 word3".rpartition("\n")
('', '', 'word1 word2 word3')
```

- `join()` — преобразует последовательность в строку. Элементы добавляются через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

В качестве примера преобразуем список и кортеж в строку:

```
>>> " => ".join(["word1", "word2", "word3"])
'word1 => word2 => word3'
>>> " ".join(("word1", "word2", "word3"))
'word1 word2 word3'
```

Обратите внимание на то, что элементы последовательностей должны быть строками, иначе возбуждается исключение `TypeError`:

```
>>> " ".join(("word1", "word2", 5))
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    " ".join(("word1", "word2", 5))
TypeError: sequence item 2: expected str instance, int found
```

Как вы уже знаете, строки относятся к неизменяемым типам данных. Если попытаться изменить символ по индексу, то возникнет ошибка. Чтобы изменить символ по индексу можно преобразовать строку в список с помощью функции `list()`, произвести изменения, а затем с помощью метода `join()` преобразовать список обратно в строку. Пример:

```
>>> s = "Python"
>>> arr = list(s); arr        # Преобразуем строку в список
['P', 'y', 't', 'h', 'o', 'n']
>>> arr[0] = "J"; arr         # Изменяем элемент по индексу
```

```
['J', 'y', 't', 'h', 'o', 'n']
>>> s = "".join(arr); s       # Преобразуем список в строку
'Jython'
```

В Python 3 можно также преобразовать строку в тип `bytearray`, а затем изменить символ по индексу:

```
>>> s = "Python"
>>> b = bytearray(s, "cp1251"); b
bytearray(b'Python')
>>> b[0] = ord("J"); b
bytearray(b'Jython')
>>> s = b.decode("cp1251"); s
'Jython'
```

6.7. Настройка локали

Для установки локали (совокупности локальных настроек системы) используется функция `setlocale()` из модуля `locale`. Прежде чем использовать функцию, необходимо подключить модуль с помощью выражения:

```
import locale
```

Функция `setlocale()` имеет следующий формат:

```
setlocale(<Категория>[, <Локаль>]);
```

Параметр `<Категория>` может принимать следующие значения:

- `locale.LC_ALL` — устанавливает локаль для всех режимов;
- `locale.LC_COLLATE` — для сравнения строк;
- `locale.LC_CTYPE` — для перевода символов в нижний или верхний регистр;
- `locale.LC_MONETARY` — для отображения денежных единиц;
- `locale.LC_NUMERIC` — для форматирования чисел;
- `locale.LC_TIME` — для форматирования вывода даты и времени.

Получить текущее значение локали позволяет функция `getlocale([<Категория>])`. В качестве примера настроим локаль под Windows вначале на кодировку Windows-1251, потом на кодировку UTF-8, а затем на кодировку по умолчанию. Далее выведем текущее значение локали для всех категорий и только для `locale.LC_COLLATE` (листинг 6.3).

Листинг 6.3. Настройка локали

```
>>> import locale
>>> # Для кодировки windows-1251
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> # Устанавливаем локаль по умолчанию
>>> locale.setlocale(locale.LC_ALL, "")
'Russian_Russia.1251'
>>> # Получаем текущее значение локали для всех категорий
>>> locale.getlocale()
('Russian_Russia', '1251')
```

```
>>> # Получаем текущее значение категории locale.LC_COLLATE
>>> locale.getlocale(locale.LC_COLLATE)
('Russian_Russia', '1251')
```

Получить настройки локали позволяет функция `localeconv()`. Функция возвращает словарь с настройками. Результат выполнения функции для локали `Russian_Russia.1251` выглядит следующим образом:

```
>>> locale.localeconv()
{'mon_decimal_point': ',', 'int_frac_digits': 2, 'p_sep_by_space': 0,
'frac_digits': 2, 'thousands_sep': '\xa0', 'n_sign_posn': 1,
'decimal_point': ',', 'int_curr_symbol': 'RUR', 'n_cs_precedes': 0,
'p_sign_posn': 1, 'mon_thousands_sep': '\xa0', 'negative_sign': '-',
'currency_symbol': 'p.', 'n_sep_by_space': 0, 'mon_grouping': [3, 0],
'p_cs_precedes': 0, 'positive_sign': '', 'grouping': [3, 0]}
```

6.8. Изменение регистра символов

Для изменения регистра символов предназначены следующие методы:

- `upper()` — заменяет все символы строки соответствующими прописными буквами:

```
>>> print("строка".upper())
СТРОКА
```

- `lower()` — заменяет все символы строки соответствующими строчными буквами:

```
>>> print("СТРОКА".lower())
строка
```

- `swapcase()` — заменяет все строчные символы соответствующими прописными буквами, а все прописные символы — строчными:

```
>>> print("СТРОКА строка".swapcase())
строка СТРОКА
```

- `capitalize()` — делает первую букву прописной:

```
>>> print("строка строка".capitalize())
Строка строка
```

- `title()` — делает первую букву каждого слова прописной:

```
>>> s = "первая буква каждого слова станет прописной"
>>> print(s.title())
Первая Буква Каждого Слова Станет Прописной
```

6.9. Функции для работы с символами

Для работы с отдельными символами предназначены следующие функции:

- `chr(<Код символа>)` — возвращает символ по указанному коду:

```
>>> print(chr(1055))
П
```

- `ord(<Символ>)` — возвращает код указанного символа:

```
>>> print(ord("П"))
1055
```

6.10. Поиск и замена в строке

Для поиска и замены в строке используются следующие методы:

- `find()` — ищет подстроку в строке. Возвращает номер позиции, с которой начинается вхождение подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.find(<Подстрока>[, <Начало>[, <Конец>]])
```

  Если начальная позиция не указана, то поиск будет осуществляться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то производится операция извлечения среза

```
<Строка>[<Начало>:<Конец>]
```

  и поиск подстроки будет выполняться в этом фрагменте. Пример:

```
>>> s = "пример пример Пример"
>>> s.find("при"), s.find("При"), s.find("тест")
(0, 14, -1)
>>> s.find("при", 9), s.find("при", 0, 6), s.find("при", 7, 12)
(-1, 0, 7)
```

- `index()` — метод аналогичен методу `find()`, но если подстрока в строку не входит, то возбуждается исключение `ValueError`. Формат метода:

```
<Строка>.index(<Подстрока>[, <Начало>[, <Конец>]])
```

  Пример:

```
>>> s = "пример пример Пример"
>>> s.index("при"), s.index("при", 7, 12), s.index("При", 1)
(0, 7, 14)
>>> s.index("тест")
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    s.index("тест")
ValueError: substring not found
```

- `rfind()` — ищет подстроку в строке. Возвращает позицию последнего вхождения подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.rfind(<Подстрока>[, <Начало>[, <Конец>]])
```

  Если начальная позиция не указана, то поиск будет производиться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза, и поиск подстроки будет производиться в этом фрагменте. Пример:

```
>>> s = "пример пример Пример Пример"
>>> s.rfind("при"), s.rfind("При"), s.rfind("тест")
```

```
(7, 21, -1)
>>> s.find("при", 0, 6), s.find("При", 10, 20)
(0, 14)
```

- `rindex()` — метод аналогичен методу `rfind()`, но если подстрока в строку не входит, то возбуждается исключение `ValueError`. Формат метода:

```
<Строка>.rindex(<Подстрока>[, <Начало>[, <Конец>]])
```

Пример:

```
>>> s = "пример пример Пример Пример"
>>> s.rindex("при"), s.rindex("При"), s.rindex("при", 0, 6)
(7, 21, 0)
>>> s.rindex("тест")
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    s.rindex("тест")
ValueError: substring not found
```

- `count()` — возвращает число вхождений подстроки в строку. Если подстрока в строку не входит, то возвращается значение `0`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.count(<Подстрока>[, <Начало>[, <Конец>]])
```

Пример:

```
>>> s = "пример пример Пример Пример"
>>> s.count("при"), s.count("при", 6), s.count("При")
(2, 1, 2)
>>> s.count("тест")
0
```

- `startswith()` — проверяет, начинается ли строка с указанной подстроки. Если начинается, то возвращается значение `True`, в противном случае — `False`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.startswith(<Подстрока>[, <Начало>[, <Конец>]])
```

Если начальная позиция не указана, сравнение будет производиться с началом строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза, и сравнение будет производиться с началом фрагмента. Пример:

```
>>> s = "пример пример Пример Пример"
>>> s.startswith("при"), s.startswith("При")
(True, False)
>>> s.startswith("при", 6), s.startswith("При", 14)
(False, True)
```

Начиная с версии 2.5, параметр `<Подстрока>` может быть кортежем:

```
>>> s = "пример пример Пример Пример"
>>> s.startswith(("при", "При"))
True
```

- `endswith()` — проверяет, заканчивается ли строка указанной подстрокой. Если заканчивается, то возвращается значение `True`, в противном случае — `False`. Метод зависит от регистра символов.

Формат метода:

```
<Строка>.endswith(<Подстрока>[, <Начало>[, <Конец>]])
```

Если начальная позиция не указана, то сравнение будет производиться с концом строки. Если параметры <Начало> и <Конец> указаны, то выполняется операция извлечения среза, и сравнение будет производиться с концом фрагмента. Пример:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith("ока"), s.endswith("ОКА")
(False, True)
>>> s.endswith("ока", 0, 9)
True
```

Начиная с версии 2.5, параметр <Подстрока> может быть кортежем:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith(("ока", "ОКА"))
True
```

- replace() — производит замену всех вхождений подстроки в строку на другую подстроку и возвращает результат в виде новой строки. Метод зависит от регистра символов. Формат метода:

```
<Строка>.replace(<Подстрока для замены>, <Новая подстрока>[,
                 <Максимальное количество замен>])
```

Пример:

```
>>> s = "Привет, Петя"
>>> print(s.replace("Петя", "Вася"))
Привет, Вася
>>> print(s.replace("петя", "вася")) # Зависит от регистра
Привет, Петя
>>> s = "strstrstrstrstr"
>>> s.replace("str", ""), s.replace("str", "", 3)
('', 'strstr')
```

- translate(<Таблица символов>) — заменяет символы в соответствии с параметром <Таблица символов>. Параметр <Таблица символов> должен быть словарем, ключами которого являются Unicode-коды заменяемых символов, а значениями — Unicode-коды вставляемых символов. Если в качестве значения указать None, то символ будет удален. Для примера удалим букву "П", а также изменим регистр всех букв "р":

```
>>> s = "Пример"
>>> d = {ord("П"): None, ord("р"): ord("Р")}
>>> d
{1088: 1056, 1055: None}
>>> s.translate(d)
'РимеР'
```

Упростить создание <Таблицы символов> позволяет статический метод maketrans(). Формат метода:

```
str.maketrans(<X>[, <Y>[, <Z>]])
```

Если указан только первый параметр, то он должен быть словарем:

```
>>> t = str.maketrans({"a": "A", "o": "O", "c": None})
>>> t
{1072: 'A', 1089: None, 1086: 'O'}
>>> "строка".translate(t)
'трОкА'
```

Если указаны два первых параметра, то они должны быть строками одинаковой длины. В результате будет создан словарь с ключами из строки `<X>` и значениями, расположенными в той же позиции, из строки `<Y>`. Изменим регистр некоторых символов:

```
>>> t = str.maketrans("абвгдежзи", "АБВГДЕЖЗИ")
>>> t
{1072: 1040, 1073: 1041, 1074: 1042, 1075: 1043, 1076: 1044,
1077: 1045, 1078: 1046, 1079: 1047, 1080: 1048}
>>> "абвгдежзи".translate(t)
'АБВГДЕЖЗИ'
```

В третьем параметре можно дополнительно указать строку из символов, которым будет сопоставлено значение `None`. После выполнения метода `translate()` эти символы будут удалены из строки. Заменим все цифры на `0`, а некоторые буквы удалим из строки:

```
>>> t = str.maketrans("123456789", "0" * 9, "str")
>>> t
{116: None, 115: None, 114: None, 49: 48, 50: 48, 51: 48,
52: 48, 53: 48, 54: 48, 55: 48, 56: 48, 57: 48}
>>> "str123456789str".translate(t)
'000000000'
```

6.11. Проверка типа содержимого строки

Для проверки типа содержимого предназначены следующие методы:

- `isdigit()` — возвращает `True`, если строка содержит только цифры, в противном случае — `False`:

```
>>> "0123".isdigit(), "123abc".isdigit(), "abc123".isdigit()
(True, False, False)
```

- `isdecimal()` — возвращает `True`, если строка содержит только десятичные символы, в противном случае — `False`. Обратите внимание на то, что к десятичным символам относятся не только десятичные цифры в кодировке ASCII, но и надстрочные и подстрочные десятичные цифры в других языках. Пример:

```
>>> "123".isdecimal(), "123стр".isdecimal()
(True, False)
```

- `isnumeric()` — возвращает `True`, если строка содержит только числовые символы, в противном случае — `False`. Обратите внимание на то, что к числовым символам относятся не только десятичные цифры в кодировке ASCII, но символы римских чисел, дробные числа и др. Пример:

```
>>> "\u2155".isnumeric(), "\u2155".isdigit()
(True, False)
>>> print("\u2155") # Выведет символ "1/5"
```

- `isalpha()` — возвращает `True`, если строка содержит только буквы, в противном случае — `False`. Если строка пустая, то возвращается значение `False`. Примеры:

```
>>> "string".isalpha(), "строка".isalpha(), "".isalpha()
(True, True, False)
>>> "123abc".isalpha(), "str str".isalpha(), "st,st".isalpha()
(False, False, False)
```

- `isspace()` — возвращает `True`, если строка содержит только пробельные символы, в противном случае — `False`:

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
(False, True, False)
```

- `isalnum()` — возвращает `True`, если строка содержит только буквы и (или) цифры, в противном случае — `False`. Если строка пустая, то возвращается значение `False`. Примеры:

```
>>> "0123".isalnum(), "123abc".isalnum(), "abc123".isalnum()
(True, True, True)
>>> "строка".isalnum()
True
>>> "".isalnum(), "123 abc".isalnum(), "abc, 123.".isalnum()
(False, False, False)
```

- `islower()` — возвращает `True`, если строка содержит буквы, и они все в нижнем регистре, в противном случае — `False`. Помимо букв строка может иметь другие символы, например цифры. Примеры:

```
>>> "srting".islower(), "строка".islower(), "".islower()
(True, True, False)
>>> "srting1".islower(), "str, 123".islower(), "123".islower()
(True, True, False)
>>> "STRING".islower(), "Строка".islower()
(False, False)
```

- `isupper()` — возвращает `True`, если строка содержит буквы, и они все в верхнем регистре, в противном случае — `False`. Помимо букв строка может иметь другие символы, например цифры. Примеры:

```
>>> "STRING".isupper(), "СТРОКА".isupper(), "".isupper()
(True, True, False)
>>> "STRING1".isupper(), "СТРОКА, 123".isupper(), "123".isupper()
(True, True, False)
>>> "string".isupper(), "STRing".isupper()
(False, False)
```

- `istitle()` — возвращает `True`, если строка содержит буквы, и первые буквы всех слов являются заглавными, в противном случае — `False`. Помимо букв строка может иметь другие символы, например цифры. Примеры:

```
>>> "Str Str".istitle(), "Стр Стр".istitle()
(True, True)
>>> "Str Str 123".istitle(), "Стр Стр 123".istitle()
(True, True)
>>> "Str str".istitle(), "Стр стр".istitle()
(False, False)
```

```
>>> "".istitle(), "123".istitle()
(False, False)
```

Переделаем нашу программу (см. листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 6.4). Кроме того, предусмотрим возможность ввода отрицательных целых чисел.

Листинг 6.4. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    x = x.rstrip("\r")              # Для версии 3.2.0 (см. разд. 1.7)
    if x == "stop":
        break # Выход из цикла
    if x == "":
        print("Вы не ввели значение!")
        continue
    if x[0] == "-": # Если первым символом является минус
        if not x[1:].isdigit(): # Если фрагмент не состоит из цифр
            print("Необходимо ввести число, а не строку!")
            continue
    else: # Если минуса нет, то проверяем всю строку
        if not x.isdigit(): # Если строка не состоит из цифр
           print("Необходимо ввести число, а не строку!")
           continue
    x = int(x) # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода значений и получения результата выглядит так:

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число:
Вы не ввели значение!
Введите число: str
Необходимо ввести число, а не строку!
Введите число: -5
Введите число: -str
Необходимо ввести число, а не строку!
Введите число: stop
Сумма чисел равна: 5
```

Значения, введенные пользователем, выделены полужирным шрифтом.

6.12. Тип данных *bytes*

Тип данных `str` отлично подходит для хранения текстовой информации, но что делать, если необходимо обрабатывать изображение? Ведь изображение не имеет кодировки, а значит, оно не может быть преобразовано в Unicode-строку. Для решения этой проблемы в Python 3 были введены типы `bytes` и `bytearray`, которые позволяют хранить последовательность целых чисел в диапазоне от `0` до `255`. Каждое такое число обозначает код символа. Тип данных `bytes` относится к неизменяемым типам, как и строки, а тип данных `bytearray` — к изменяемым, как и списки.

Создать объект типа `bytes` можно следующими способами:

- с помощью функции `bytes([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустой объект. Чтобы преобразовать строку в объект типа `bytes`, необходимо передать минимум два первых параметра. Если строка указана только в первом параметре, то возбуждается исключение `TypeError`. Пример:

```
>>> bytes()
b''
>>> bytes("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка")
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    bytes("строка")
TypeError: string argument without an encoding
```

  В третьем параметре могут быть указаны значения "strict" (при ошибке возбуждается исключение `UnicodeEncodeError`; значение по умолчанию), "replace" (неизвестный символ заменяется символом вопроса) или "ignore" (неизвестные символы игнорируются). Пример:

```
>>> bytes("string\uFFFD", "cp1251", "strict")
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    bytes("string\uFFFD", "cp1251", "strict")
  File "C:\Python32\lib\encodings\cp1251.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\ufffd' in position 6: character maps to <undefined>
>>> bytes("string\uFFFD", "cp1251", "replace")
b'string?'
>>> bytes("string\uFFFD", "cp1251", "ignore")
b'string'
```

- с помощью метода строк `encode([encoding="utf-8"][, errors="strict"])`. Если кодировка не указана, то строка преобразуется в последовательность байтов в кодировке UTF-8. В параметре `errors` могут быть указаны значения "strict" (значение по умолчанию), "replace", "ignore", "xmlcharrefreplace" или "backslashreplace". Пример:

```
>>> "строка".encode()
b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
>>> "строка".encode(encoding="cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
```

```
>>> "строка\uFFFD".encode(encoding="cp1251",
                          errors="xmlcharrefreplace")
b'\xf1\xf2\xf0\xee\xea\xe0&#65533;'
>>> "строка\uFFFD".encode(encoding="cp1251",
                          errors="backslashreplace")
b'\xf1\xf2\xf0\xee\xea\xe0\\ufffd'
```

- указав букву `b` (регистр не имеет значения) перед строкой в апострофах, кавычках, тройных апострофах или тройных кавычках. Обратите внимание на то, что в строке могут быть только символы с кодами, входящими в кодировку ASCII. Все остальные символы должны быть представлены специальными последовательностями:

```
>>> b"string", b'string', b"""string""", b'''string'''
(b'string', b'string', b'string', b'string')
>>> b"строка"
SyntaxError: bytes can only contain ASCII literal characters.
>>> b"\xf1\xf2\xf0\xee\xea\xe0"
b'\xf1\xf2\xf0\xee\xea\xe0'
```

- с помощью функции `bytes(<Последовательность>)`, которая преобразует последовательность целых чисел от `0` до `255` в объект типа `bytes`. Если число не попадает в диапазон, то возбуждается исключение `ValueError`. Пример:

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

- с помощью функции `bytes(<Число>)`, которая задает количество элементов в последовательности. Каждый элемент будет содержать нулевой символ:

```
>>> bytes(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

- с помощью метода `bytes.fromhex(<Строка>)`. Строка в этом случае должна содержать шестнадцатеричные значения символов:

```
>>> b = bytes.fromhex(" e1 e2e0ae aaa0 ")
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

Объекты типа `bytes` относятся к последовательностям. Каждый элемент такой последовательности может хранить целое число от `0` до `255`, которое обозначает код символа. Как и все последовательности, объекты поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение:

```
>>> b = bytes("string", "cp1251")
>>> b
b'string'
>>> b[0]                          # Обращение по индексу
115
>>> b[1:3]                        # Получение среза
b'tr'
```

```
>>> b + b"123"                    # Конкатенация
b'string123'
>>> b * 3                         # Повторение
b'stringstringstring'
>>> 115 in b, b"tr" in b, b"as" in b
(True, True, False)
```

Как видно из примера, при выводе объекта целиком, а также при извлечении среза производится попытка отображения символов. Однако доступ по индексу возвращает целое число, а не символ. Если преобразовать объект в список, то мы получим последовательность целых чисел:

```
>>> list(bytes("string", "cp1251"))
[115, 116, 114, 105, 110, 103]
```

Тип `bytes` относится к неизменяемым типам. Это означает, что можно получить значение по индексу, но изменить его нельзя:

```
>>> b = bytes("string", "cp1251")
>>> b[0] = 168
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    b[0] = 168
TypeError: 'bytes' object does not support item assignment
```

Объекты типа `bytes` поддерживают большинство строковых методов, которые мы рассматривали в предыдущих разделах. Однако некоторые из этих методов могут некорректно работать с русскими буквами, например, все мои попытки изменить регистр символов окончились неудачей в версии 3.2.0, даже настройка локали не помогла. В этих случаях следует использовать тип `str`, а не тип `bytes`. Не поддерживаются строковые методы `encode()`, `isidentifier()`, `isprintable()`, `isnumeric()`, `isdecimal()`, `format_map()` и `format()`, а также операция форматирования. При использовании методов следует учитывать, что в параметрах нужно указывать объекты типа `bytes`, а не строки:

```
>>> b = bytes("string", "cp1251")
>>> b.replace(b"s", b"S")
b'String'
```

Необходимо также помнить, что смешивать строки и объекты типа `bytes` в выражениях нельзя. Предварительно необходимо явно преобразовать объекты к одному типу, а лишь затем производить операцию:

```
>>> b"string" + "string"
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    b"string" + "string"
TypeError: can't concat bytes to str
>>> b"string" + "string".encode("ascii")
b'stringstring'
```

Объект типа `bytes` может содержать как однобайтовые символы, так и многобайтовые. При использовании многобайтовых символов некоторые функции могут работать не так, как вы думаете, например, функция `len()` вернет количество байтов, а не символов:

```
>>> len("строка")
6
>>> len(bytes("строка", "cp1251"))
6
>>> len(bytes("строка", "utf-8"))
12
```

Преобразовать объект типа `bytes` в строку позволяет метод `decode()`. Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр `encoding` задает кодировку символов (по умолчанию UTF-8) в объекте `bytes`, а параметр `errors` — способ обработки ошибок при преобразовании. В параметре `errors` можно указать значения "strict" (значение по умолчанию), "replace" или "ignore". Пример преобразования:

```
>>> b = bytes("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('строка', 'строка')
```

Для преобразования можно также воспользоваться функцией `str()`:

```
>>> b = bytes("строка", "cp1251")
>>> str(b, "cp1251")
'строка'
```

Чтобы изменить кодировку данных, следует сначала преобразовать тип `bytes` в строку, а затем произвести обратное преобразование, указав нужную кодировку. Преобразуем данные из кодировки Windows-1251 в кодировку KOI8-R, а затем обратно (листинг 6.5).

Листинг 6.5. Преобразование кодировок

```
>>> w = bytes("Строка", "cp1251") # Данные в кодировке windows-1251
>>> k = w.decode("cp1251").encode("koi8-r")
>>> k, str(k, "koi8-r")           # Данные в кодировке KOI8-R
(b'\xf3\xd4\xd2\xcf\xcb\xc1', 'Строка')
>>> w = k.decode("koi8-r").encode("cp1251")
>>> w, str(w, "cp1251")           # Данные в кодировке windows-1251
(b'\xd1\xf2\xf0\xee\xea\xe0', 'Строка')
```

6.13. Тип данных *bytearray*

Тип данных `bytearray` является разновидностью типа `bytes` и поддерживает те же самые методы и операции. В отличие от типа `bytes`, тип `bytearray` допускает возможность непосредственного изменения объекта и содержит дополнительные методы, позволяющие выполнять эти изменения.

Создать объект типа `bytearray` можно следующими способами:

- с помощью функции `bytearray([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустой объект. Чтобы преобразовать строку в объект типа `bytearray`, необходимо передать минимум два первых параметра. Если строка указана только в первом параметре, то возбуждается исключение `TypeError`.

Пример:

```
>>> bytearray()
bytearray(b'')
>>> bytearray("строка", "cp1251")
bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
>>> bytearray("строка")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    bytearray("строка")
TypeError: string argument without an encoding
```

В третьем параметре могут быть указаны значения "strict" (при ошибке возбуждается исключение `UnicodeEncodeError`; значение по умолчанию), "replace" (неизвестный символ заменяется символом вопроса) или "ignore" (неизвестные символы игнорируются). Пример:

```
>>> bytearray("string\uFFFD", "cp1251", "strict")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    bytearray("string\uFFFD", "cp1251", "strict")
  File "C:\Python32\lib\encodings\cp1251.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\ufffd' in position 6: character maps to <undefined>
>>> bytearray("string\uFFFD", "cp1251", "replace")
bytearray(b'string?')
>>> bytearray("string\uFFFD", "cp1251", "ignore")
bytearray(b'string')
```

- с помощью функции `bytearray(<Последовательность>)`, которая преобразует последовательность целых чисел от `0` до `255` в объект типа `bytearray`. Если число не попадает в диапазон, то возбуждается исключение `ValueError`. Пример:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

- с помощью функции `bytearray(<Число>)`, которая задает количество элементов в последовательности. Каждый элемент будет содержать нулевой символ:

```
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
```

- с помощью метода `bytearray.fromhex(<Строка>)`. Строка в этом случае должна содержать шестнадцатеричные значения символов:

```
>>> b = bytearray.fromhex(" e1 e2e0ae aaa0 ")
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
```

```
>>> str(b, "cp866")
'строка'
```

Тип bytearray относится к изменяемым типам. Поэтому можно не только получить значение по индексу, но и изменить его (что не свойственно строкам):

```
>>> b = bytearray("Python", "ascii")
>>> b[0]                              # Можем получить значение
80
>>> b[0] = b"J"[0]                    # Можем изменить значение
>>> b
bytearray(b'Jython')
```

При изменении значения важно помнить, что присваиваемое значение должно быть целым числом в диапазоне от 0 до 255. Чтобы получить число в предыдущем примере, мы создали объект типа bytes, а затем присвоили значение, расположенное по индексу 0 (b[0] = b"J"[0]). Можно, конечно, сразу указать код символа, но ведь держать все коды символов в памяти свойственно компьютеру, а не человеку.

Для изменения объекта можно также использовать следующие методы:

- append(<Число>) — добавляет один элемент в конец объекта. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> b = bytearray("string", "ascii")
>>> b.append(b"1"[0]); b
bytearray(b'string1')
```

- extend(<Последовательность>) — добавляет элементы последовательности в конец объекта. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> b = bytearray("string", "ascii")
>>> b.extend(b"123"); b
bytearray(b'string123')
```

Добавить несколько элементов можно с помощью операторов + и +=:

```
>>> b = bytearray("string", "ascii")
>>> b + b"123"                  # Возвращает новый объект
bytearray(b'string123')
>>> b += b"456"; b              # Изменяет текущий объект
bytearray(b'string456')
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> b = bytearray("string", "ascii")
>>> b[len(b):] = b"123"         # Изменяет текущий список
>>> b
bytearray(b'string123')
```

- insert(<Индекс>, <Число>) — добавляет один элемент в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий объект и ничего не возвращает. Добавим элемент в начало объекта:

```
>>> b = bytearray("string", "ascii")
>>> b.insert(0, b"1"[0]); b
bytearray(b'1string')
```

Метод `insert()` позволяет добавить только один элемент. Чтобы добавить несколько элементов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало объекта:

```
>>> b = bytearray("string", "ascii")
>>> b[:0] = b"123"; b
bytearray(b'123string')
```

- `pop([<Индекс>])` — удаляет элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, то удаляет и возвращает последний элемент. Примеры:

```
>>> b = bytearray("string", "ascii")
>>> b.pop()                # Удаляем последний элемент
103
>>> b
bytearray(b'strin')
>>> b.pop(0)               # Удаляем первый элемент
115
>>> b
bytearray(b'trin')
```

Удалить элемент списка позволяет также оператор `del`:

```
>>> b = bytearray("string", "ascii")
>>> del b[5]; b            # Удаляем последний элемент
bytearray(b'strin')
>>> del b[:2]; b           # Удаляем первый и второй элементы
bytearray(b'rin')
```

- `remove(<Число>)` — удаляет первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> b = bytearray("strstr", "ascii")
>>> b.remove(b"s"[0]); b   # Удаляет только первый элемент
bytearray(b'trstr')
```

- `reverse()` — изменяет порядок следования элементов на противоположный. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> b = bytearray("string", "ascii")
>>> b.reverse(); b
bytearray(b'gnirts')
```

Преобразовать объект типа `bytearray` в строку позволяет метод `decode()`. Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр `encoding` задает кодировку символов (по умолчанию UTF-8) в объекте `bytearray`, а параметр `errors` — способ обработки ошибок при преобразовании. В параметре `errors` можно указать значения "strict" (значение по умолчанию), "replace" или "ignore". Пример преобразования:

```
>>> b = bytearray("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('строка', 'строка')
```

Для преобразования можно также воспользоваться функцией `str()`:

```
>>> b = bytearray("строка", "cp1251")
>>> str(b, "cp1251")
'строка'
```

6.14. Преобразование объекта в последовательность байтов

Преобразовать объект в последовательность байтов, а затем восстановить объект позволяет модуль `pickle`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import pickle
```

Для преобразования предназначены две функции:

- `dumps(<Объект>[, <Протокол>][, fix_imports=True])` — производит сериализацию объекта и возвращает последовательность байтов специального формата. Формат зависит от указанного протокола (число от `0` до `3`). Пример преобразования списка и кортежа:

```
>>> import pickle
>>> obj1 = [1, 2, 3, 4, 5]      # Список
>>> obj2 = (6, 7, 8, 9, 10)     # Кортеж
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

- `loads(<Последовательность байтов>[, fix_imports=True][, encoding="ASCII"][, errors="strict"])` — преобразует последовательность байтов специального формата обратно в объект. Пример восстановления списка и кортежа:

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

6.15. Шифрование строк

Для шифрования строк предназначен модуль `hashlib`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import hashlib
```

Модуль предоставляет следующие функции: `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()` и `sha512()`. В качестве необязательного параметра функциям можно передать шифруемую последовательность байтов. Пример:

```
>>> import hashlib
>>> h = hashlib.sha1(b"password")
```

Передать последовательность байтов можно также с помощью метода `update()`. В этом случае объект присоединяется к предыдущему значению:

```
>>> h = hashlib.sha1()
>>> h.update(b"password")
```

Получить зашифрованную последовательность байтов и строку позволяют два метода — `digest()` и `hexdigest()`:

```
>>> h = hashlib.sha1(b"password")
>>> h.digest()
b'[\xaaa\xe4\xc9\xb9??\x06\x82%\x0bl\xf83\x1b~\xe6\x8f\xd8'
>>> h.hexdigest()
'5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'
```

Наиболее часто применяемой функцией является функция `md5()`, которая шифрует строку с помощью алгоритма MD5. Она используется для шифрования паролей, т. к. не существует алгоритма для дешифровки. Для сравнения введенного пользователем пароля с сохраненным в базе необходимо зашифровать введенный пароль, а затем произвести сравнение (листинг 6.6).

Листинг 6.6. Проверка правильности ввода пароля

```
>>> import hashlib
>>> h = hashlib.md5(b"password")
>>> p = h.hexdigest()
>>> p                                  # Пароль, сохраненный в базе
'5f4dcc3b5aa765d61d8327deb882cf99'
>>> h2 = hashlib.md5(b"password")     # Пароль, введенный пользователем
>>> if p == h2.hexdigest(): print("Пароль правильный")

Пароль правильный
```

ГЛАВА 7

Регулярные выражения

Регулярные выражения предназначены для выполнения сложного поиска или замены в строке. В языке Python использовать регулярные выражения позволяет модуль `re`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import re
```

7.1. Синтаксис регулярных выражений

Создать откомпилированный шаблон регулярного выражения позволяет функция `compile()`. Функция имеет следующий формат:

```
<Шаблон> = re.compile(<Регулярное выражение>[, <Модификатор>])
```

В параметре `<Модификатор>` могут быть указаны следующие флаги (или их комбинация через оператор `|`):

- `L` или `LOCALE` — учитываются настройки текущей локали;
- `I` или `IGNORECASE` — поиск без учета регистра. Пример:

```
>>> import re
>>> p = re.compile(r"^[а-яё]+$", re.I | re.U)
>>> print("Найдено" if p.search("АБВГДЕЁ") else "Нет")
Найдено
>>> p = re.compile(r"^[а-яё]+$", re.U)
>>> print("Найдено" if p.search("АБВГДЕЁ") else "Нет")
Нет
```

- `M` или `MULTILINE` — поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`"\n"`). Символ `^` соответствует привязке к началу каждой подстроки, а символ `$` соответствует позиции перед символом перевода строки;
- `S` или `DOTALL` — метасимвол "точка" будет соответствовать любому символу, включая символ перевода строки (`\n`). По умолчанию метасимвол "точка" не соответствует символу перевода строки. Символ `^` будет соответствовать привязке к началу всей строки, а символ `$` — привязке к концу всей строки. Пример:

```
>>> p = re.compile(r"^.$")
>>> print("Найдено" if p.search("\n") else "Нет")
Нет
```

```
>>> p = re.compile(r"^.$", re.M)
>>> print("Найдено" if p.search("\n") else "Нет")
Нет
>>> p = re.compile(r"^.$", re.S)
>>> print("Найдено" if p.search("\n") else "Нет")
Найдено
```

- `X` или `VERBOSE` — если флаг указан, то пробелы и символы перевода строки будут игнорированы. Кроме того, внутри регулярного выражения можно использовать комментарии. Пример:

```
>>> p = re.compile(r"""^ # Привязка к началу строки
        [0-9]+ # Строка должна содержать одну цифру (или более)
        $      # Привязка к концу строки
        """, re.X | re.S)
>>> print("Найдено" if p.search("1234567890") else "Нет")
Найдено
>>> print("Найдено" if p.search("abcd123") else "Нет")
Нет
```

- `U` или `UNICODE` — классы `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` будут соответствовать Unicode-символам. В Python 3 флаг установлен по умолчанию;
- `A` или `ASCII` — классы `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` будут соответствовать обычным символам.

Как видно из примеров, перед всеми строками, содержащими регулярные выражения, указан модификатор `r`. Иными словами, мы используем неформатированные строки. Если модификатор не указать, то все слэши необходимо будет экранировать. Например, строку

```
p = re.compile(r"^\w+$")
```

нужно было бы записать так:

```
p = re.compile("^\\w+$")
```

Внутри регулярного выражения символы `.`, `^`, `$`, `*`, `+`, `?`, `{`, `[`, `]`, `\`, `|`, `(` и `)` имеют специальное значение. Если эти символы должны трактоваться как есть, то их следует экранировать с помощью слэша. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок. В этом случае экранировать их не нужно. Например, метасимвол "точка" по умолчанию соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой нужно указать символ `\` или разместить точку внутри квадратных скобок (`[.]`). Продемонстрируем это на примере проверки правильности введенной даты (листинг 7.1).

Листинг 7.1. Проверка правильности ввода даты

```
# -*- coding: utf-8 -*-
import re          # Подключаем модуль

d = "29,12.2009"  # Вместо точки указана запятая

p = re.compile(r"^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$")
# Символ "\" не указан перед точкой
```

```
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Так как точка означает любой символ,
# выведет: Дата введена правильно

p = re.compile(r"^[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]$")
# Символ "\" указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Так как перед точкой указан символ "\",
# выведет: Дата введена неправильно

p = re.compile(r"^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$")
# Точка внутри квадратных скобок
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Выведет: Дата введена неправильно
input()
```

В этом примере мы осуществляли привязку к началу и концу строки с помощью следующих метасимволов:

- `^` — привязка к началу строки или подстроки (зависит от флагов `M` (или `MULTILINE`) и `S` (или `DOTALL`));
- `$` — привязка к концу строки или подстроки (зависит от флагов `M` (или `MULTILINE`) и `S` (или `DOTALL`));
- `\A` — привязка к началу строки (не зависит от модификатора);
- `\Z` — привязка к концу строки (не зависит от модификатора).

Если в параметре `<Модификатор>` указан флаг `M` (или `MULTILINE`), то поиск производится в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`\n`). В этом случае символ `^` соответствует привязке к началу каждой подстроки, а символ `$` соответствует позиции перед символом перевода строки (листинг 7.2).

Листинг 7.2. Пример использования многострочного режима

```
>>> p = re.compile(r"^.+$")       # Точка не соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Ничего не найдено
[]
>>> p = re.compile(r"^.+$", re.S) # Теперь точка соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Строка полностью соответствует
['str1\nstr2\nstr3']
>>> p = re.compile(r"^.+$", re.M) # Многострочный режим
>>> p.findall("str1\nstr2\nstr3") # Получили каждую подстроку
['str1', 'str2', 'str3']
```

Привязку к началу и концу строки следует использовать, если строка должна полностью соответствовать регулярному выражению. Например, привязку нужно использовать для проверки, содержит ли строка число (листинг 7.3).

Листинг 7.3. Проверка наличия целого числа в строке

```
# -*- coding: utf-8 -*-
import re                       # Подключаем модуль

p = re.compile(r"^[0-9]+$", re.S)
if p.search("245"):
    print("Число")              # Выведет: Число
else:
    print("Не число")
if p.search("Строка245"):
    print("Число")
else:
    print("Не число")           # Выведет: Не число
input()
```

Если убрать привязку к началу и концу строки, то любая строка, содержащая хотя бы одну цифру, будет распознана как "Число" (листинг 7.4).

Листинг 7.4. Отсутствие привязки к началу или концу строки

```
# -*- coding: utf-8 -*-
import re                       # Подключаем модуль

p = re.compile(r"[0-9]+", re.S)
if p.search("Строка245"):
    print("Число")              # Выведет: Число
else:
    print("Не число")
input()
```

Кроме того, можно указать привязку только к началу или только к концу строки (листинг 7.5).

Листинг 7.5. Привязка к началу и концу строки

```
# -*- coding: utf-8 -*-
import re                       # Подключаем модуль

p = re.compile(r"[0-9]+$", re.S)
if p.search("Строка245"):
    print("Есть число в конце строки")
else:
    print("Нет числа в конце строки")
# Выведет: Есть число в конце строки
```

```
p = re.compile(r"^[0-9]+", re.S)
if p.search("Строка245"):
    print("Есть число в начале строки")
else:
    print("Нет числа в начале строки")
# Выведет: Нет числа в начале строки
input()
```

В квадратных скобках `[]` можно указать символы, которые могут встречаться на этом месте в строке. Можно перечислять символы подряд или указать диапазон через тире:

- `[09]` — соответствует числу 0 или 9;
- `[0-9]` — соответствует любому числу от 0 до 9;
- `[абв]` — соответствует буквам "а", "б" и "в";
- `[а-г]` — соответствует буквам "а", "б", "в" и "г";
- `[а-яё]` — соответствует любой букве от "а" до "я";
- `[АБВ]` — соответствует буквам "А", "Б" и "В";
- `[А-ЯЁ]` — соответствует любой букве от "А" до "Я";
- `[а-яА-ЯёЁ]` — соответствует любой русской букве в любом регистре;
- `[0-9а-яА-ЯёЁa-zA-Z]` — любая цифра и любая буква независимо от регистра и языка.

> ***Внимание!***
> Буква "ё" не входит в диапазон `[а-я]`.

Значение можно инвертировать, если после первой скобки указать символ `^`. Таким образом можно указать символы, которых не должно быть на этом месте в строке:

- `[^09]` — не цифра 0 или 9;
- `[^0-9]` — не цифра от 0 до 9;
- `[^а-яА-ЯёЁa-zA-Z]` — не буква.

Как вы уже знаете, точка теряет свое специальное значение, если ее заключить в квадратные скобки. Кроме того, внутри квадратных скобок могут встретиться символы, которые имеют специальное значение (например, `^` и `-`). Символ `^` теряет свое специальное значение, если он не расположен сразу после открывающей квадратной скобки. Чтобы отменить специальное значение символа `-`, его необходимо указать после перечисления всех символов, перед закрывающей квадратной скобкой, или сразу после открывающей квадратной скобки. Все специальные символы можно сделать обычными, если перед ними указать символ `\`.

Вместо перечисления символов можно использовать стандартные классы:

- `\d` — соответствует любой цифре. При указании флага `A` (`ASCII`) эквивалентно `[0-9]`;
- `\w` — соответствует любой букве, цифре или символу подчеркивания. При указании флага `A` (`ASCII`) эквивалентно `[a-zA-Z0-9_]`;
- `\s` — любой пробельный символ. При указании флага `A` (`ASCII`) эквивалентно `[ \t\n\r\f\v]`;
- `\D` — не цифра. При указании флага `A` (`ASCII`) эквивалентно `[^0-9]`;

- `\W` — не буква, не цифра и не символ подчеркивания. При указании флага `A` (`ASCII`) эквивалентно `[^a-zA-Z0-9_]`;
- `\S` — не пробельный символ. При указании флага `A` (`ASCII`) эквивалентно `[^ \t\n\r\f\v]`.

> ***Примечание***
>
> В Python 3 по умолчанию устанавливается флаг `U` (`UNICODE`). При этом флаге все классы трактуются гораздо шире. Например, класс `\d` соответствует не только десятичным цифрам, но и другим цифрам из кодировки Unicode, класс `\w` включает не только латинские буквы, но и любые другие, а класс `\s` охватывает также неразрывные пробелы. Поэтому на практике лучше явно указывать символы внутри квадратных скобок, а не использовать классы.

Количество вхождений символа в строку задается с помощью *квантификаторов*:

- `{n}` — `n` вхождений символа в строку. Например, шаблон `r"^[0-9]{2}$"` соответствует двум вхождениям любой цифры;
- `{n,}` — `n` или более вхождений символа в строку. Например, шаблон `r"^[0-9]{2,}$"` соответствует двум и более вхождениям любой цифры;
- `{n,m}` — не менее `n` и не более `m` вхождений символа в строку. Числа указываются через запятую без пробела. Например, шаблон `r"^[0-9]{2,4}$"` соответствует от двух до четырех вхождениям любой цифры;
- `*` — ноль или большее число вхождений символа в строку. Эквивалентно комбинации `{0,}`;
- `+` — одно или большее число вхождений символа в строку. Эквивалентно комбинации `{1,}`;
- `?` — ни одного или одно вхождение символа в строку. Эквивалентно комбинации `{0,1}`.

Все квантификаторы являются "жадными". При поиске соответствия ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия. Рассмотрим это на примере. Получим содержимое всех тегов `<b>`, вместе с тегами:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*</b>", re.S)
>>> p.findall(s)
['<b>Text1</b>Text2<b>Text3</b>']
```

Вместо желаемого результата мы получили полностью строку. Чтобы ограничить "жадность", необходимо после квантификатора указать символ `?` (листинг 7.6).

Листинг 7.6. Ограничение "жадности" квантификаторов

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*?</b>", re.S)
>>> p.findall(s)
['<b>Text1</b>', '<b>Text3</b>']
```

Этот код вывел то, что мы искали. Если необходимо получить содержимое без тегов, то нужный фрагмент внутри шаблона следует разместить внутри круглых скобок (листинг 7.7).

Листинг 7.7. Получение значения определенного фрагмента

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>(.*?)</b>", re.S)
>>> p.findall(s)
['Text1', 'Text3']
```

Круглые скобки часто используются для группировки фрагментов внутри шаблона. В этом случае не требуется, чтобы фрагмент запоминался и был доступен в результатах поиска. Чтобы избежать захвата фрагмента, следует после открывающей круглой скобки разместить символы `?:` (листинг 7.8).

Листинг 7.8. Ограничение захвата фрагмента

```
>>> s = "test text"
>>> p = re.compile(r"([a-z]+((st)|(xt)))", re.S)
>>> p.findall(s)
[('test', 'st', 'st', ''), ('text', 'xt', '', 'xt')]
>>> p = re.compile(r"([a-z]+(?:(?:st)|(?:xt)))", re.S)
>>> p.findall(s)
['test', 'text']
```

В первом примере мы получили список с двумя элементами. Каждый элемент списка является кортежем, содержащим четыре элемента. Все эти элементы соответствуют фрагментам, заключенным в шаблоне в круглые скобки. Первый элемент кортежа содержит фрагмент, расположенный в первых круглых скобках, второй — во вторых круглых скобках и т. д. Три последних элемента кортежа являются лишними. Чтобы они не выводились в результатах, мы добавили символы `?:` после каждой открывающей круглой скобки. В результате список состоит только из фрагментов, полностью соответствующих регулярному выражению.

Обратите внимание на регулярное выражение в предыдущем примере:

```
r"([a-z]+((st)|(xt)))"
```

Здесь мы использовали метасимвол `|`, который позволяет сделать выбор между альтернативными значениями. Выражение `n|m` соответствует одному из символов: `n` или `m`. Пример:

```
красн((ая)|(ое)) — красная или красное, но не красный.
```

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью механизма *обратных ссылок*. Для этого порядковый номер круглых скобок в шаблоне указывается после слэша, например, `\1`. Нумерация скобок внутри шаблона начинается с `1`. Для примера получим текст между одинаковыми парными тегами (листинг 7.9).

Листинг 7.9. Обратные ссылки

```
>>> s = "<b>Text1</b>Text2<I>Text3</I><b>Text4</b>"
>>> p = re.compile(r"<([a-z]+)>(.*?)</\1>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3'), ('b', 'Text4')]
```

Фрагментам внутри круглых скобок можно дать имена. Для этого после открывающей круглой скобки следует указать комбинацию символов `?P<name>`. В качестве примера разберем e-mail на составные части (листинг 7.10).

Листинг 7.10. Именованные фрагменты

```
>>> email = "test@mail.ru"
>>> p = re.compile(r"""(?P<name>[a-z0-9_.-]+) # Название ящика
        @                                    # Символ "@"
        (?P<host>(?:[a-z0-9-]+\.)+[a-z]{2,6}) # Домен
        """, re.I | re.VERBOSE)
>>> r = p.search(email)
>>> r.group("name")                          # Название ящика
'test'
>>> r.group("host")                          # Домен
'mail.ru'
```

Чтобы внутри шаблона обратиться к именованным фрагментам, используется следующий синтаксис: `(?P=name)`. Для примера получим текст между одинаковыми парными тегами (листинг 7.11).

Листинг 7.11. Обращение к именованным фрагментам внутри шаблона

```
>>> s = "<b>Text1</b>Text2<I>Text3</I>"
>>> p = re.compile(r"<(?P<tag>[a-z]+)>(.*?)</(?P=tag)>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3')]
```

Кроме того, внутри круглых скобок могут быть расположены следующие конструкции:

- `(?aiLmsux)` — позволяет установить опции регулярного выражения. Буквы "a", "i", "L", "m", "s", "u" и "x" имеют такое же назначение, что и одноименные модификаторы в функции `compile()`;
- `(?#...)` — комментарий. Текст внутри круглых скобок игнорируется;
- `(?=...)` — положительный просмотр вперед. Выведем все слова, после которых расположена запятая:

  ```
  >>> s = "text1, text2, text3 text4"
  >>> p = re.compile(r"\w+(?=[,])", re.S | re.I)
  >>> p.findall(s)
  ['text1', 'text2']
  ```

- `(?!...)` — отрицательный просмотр вперед. Выведем все слова, после которых нет запятой:

  ```
  >>> s = "text1, text2, text3 text4"
  >>> p = re.compile(r"[a-z]+[0-9](?![,])", re.S | re.I)
  >>> p.findall(s)
  ['text3', 'text4']
  ```

- `(?<=...)` — положительный просмотр назад. Выведем все слова, перед которыми расположена запятая с пробелом:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"(?<=[,][ ])[a-z]+[0-9]", re.S | re.I)
>>> p.findall(s)
['text2', 'text3']
```

- `(?<!...)` — отрицательный просмотр назад. Выведем все слова, перед которыми расположен пробел, но перед пробелом нет запятой:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"(?<![,]) ([a-z]+[0-9])", re.S | re.I)
>>> p.findall(s)
['text4']
```

- `(?(id` или `name)шаблон1|шаблон2)` — если группа с номером или названием найдена, то должно выполняться условие из параметра `шаблон1`, в противном случае должно выполняться условие из параметра `шаблон2`. Выведем все слова, которые расположены внутри апострофов. Если перед словом нет апострофа, то в конце слова должна быть запятая:

```
>>> s = "text1 'text2' 'text3 text4, text5"
>>> p = re.compile(r"(')?([a-z]+[0-9])(?(1)'|,)", re.S | re.I)
>>> p.findall(s)
[("'", 'text2'), ('', 'text4')]
```

Рассмотрим небольшой пример. Предположим, необходимо получить все слова, расположенные после тире, причем перед тире и после слов должны следовать пробельные символы:

```
>>> s = "-word1 -word2 -word3 -word4 -word5"
>>> re.findall(r"\s\-([a-z0-9]+)\s", s, re.S | re.I)
['word2', 'word4']
```

Как видно из примера, мы получили только два слова вместо пяти. Первое и последнее слово не попали в результат, т. к. расположены в начале и конце строки. Чтобы эти слова попали в результат, необходимо добавить альтернативный выбор `(^|\s)` для начала строки и `(\s|$)` для конца строки. Чтобы найденные выражения внутри круглых скобок не попали в результат, следует добавить символы `?:` после открывающей скобки:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?:\s|$)", s, re.S | re.I)
['word1', 'word3', 'word5']
```

Первое и последнее слово успешно попали в результат. Почему же слова "word2" и "word4" не попали в список совпадений? Ведь перед тире есть пробел и после слова есть пробел. Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед тире расположено начало строки, и после слова есть пробел. После поиска указатель перемещается, и строка для дальнейшего поиска примет следующий вид:

```
"-word1 <Указатель>-word2 -word3 -word4 -word5"
```

Обратите внимание на то, что перед фрагментом `"-word2"` больше нет пробела, и тире не расположено вначале строки. Поэтому следующим совпадением будет слово "word3", и указатель снова будет перемещен:

```
"-word1 -word2 -word3 <Указатель>-word4 -word5"
```

Опять перед фрагментом `"-word4"` нет пробела, и тире не расположено вначале строки. Поэтому следующим совпадением будет слово "word5", и поиск будет завершен. Таким образом, слова "word2" и "word4" не попадают в результат, т. к. пробел до фрагмента уже был

использован в предыдущем поиске. Чтобы этого избежать, следует воспользоваться положительным просмотром вперед `(?=...)`:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?=\s|$)", s, re.S | re.I)
['word1', 'word2', 'word3', 'word4', 'word5']
```

В этом примере мы заменили фрагмент `(?:\s|$)` на `(?=\s|$)`. Поэтому все слова успешно попали в список совпадений.

7.2. Поиск первого совпадения с шаблоном

Для поиска первого совпадения с шаблоном предназначены следующие функции и методы:

- `match()` — проверяет соответствие с началом строки. Формат метода:

  ```
  match(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствие найдено, то возвращается объект `Match`, в противном случае возвращается значение `None`. Пример:

  ```
  >>> import re
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if p.match("str123") else "Нет")
  Нет
  >>> print("Найдено" if p.match("str123", 3) else "Нет")
  Найдено
  >>> print("Найдено" if p.match("123str") else "Нет")
  Найдено
  ```

  Вместо метода `match()` можно воспользоваться функцией `match()`. Формат функции:

  ```
  re.match(<Шаблон>, <Строка>[, <Модификатор>])
  ```

  В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, то возвращается объект `Match`, в противном случае возвращается значение `None`. Пример:

  ```
  >>> p = r"[0-9]+"
  >>> print("Найдено" if re.match(p, "str123") else "Нет")
  Нет
  >>> print("Найдено" if re.match(p, "123str") else "Нет")
  Найдено
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if re.match(p, "123str") else "Нет")
  Найдено
  ```

- `search()` — проверяет соответствие с любой частью строки. Формат метода:

  ```
  search(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствие найдено, то возвращается объект `Match`, в противном случае возвращается значение `None`. Пример:

  ```
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if p.search("str123") else "Нет")
  Найдено
  ```

```
>>> print("Найдено" if p.search("123str") else "Нет")
Найдено
>>> print("Найдено" if p.search("123str", 3) else "Нет")
Нет
```

Вместо метода `search()` можно воспользоваться функцией `search()`. Формат функции:

```
re.search(<Шаблон>, <Строка>[, <Модификатор>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, то возвращается объект `Match`, в противном случае возвращается значение `None`. Пример:

```
>>> p = r"[0-9]+"
>>> print("Найдено" if re.search(p, "str123") else "Нет")
Найдено
>>> p = re.compile(r"[0-9]+")
>>> print("Найдено" if re.search(p, "str123") else "Нет")
Найдено
```

В качестве примера переделаем нашу программу (см. листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 7.12). Кроме того, предусмотрим возможность ввода отрицательных целых чисел.

Листинг 7.12. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
import re
print("Введите слово 'stop' для получения результата")
summa = 0
p = re.compile(r"^[-]?[0-9]+$", re.S)
while True:
    x = input("Введите число: ")
    x = x.rstrip("\r")              # Для версии 3.2.0 (см. разд. 1.7)
    if x == "stop":
        break    # Выход из цикла
    if not p.search(x):
        print("Необходимо ввести число, а не строку!")
        continue # Переходим на следующую итерацию цикла
    x = int(x)   # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Объект `Match`, возвращаемый методами (функциями) `match()` и `search()`, имеет следующие свойства и методы:

- `re` — ссылка на скомпилированный шаблон, указанный в методах (функциях) `match()` и `search()`. Через эту ссылку доступны следующие свойства:
  - `groups` — количество групп в шаблоне;
  - `groupindex` — словарь с названиями групп и их номерами;

- `string` — значение параметра `<Строка>` в методах (функциях) `match()` и `search()`;
- `pos` — значение параметра `<Начальная позиция>` в методах `match()` и `search()`;
- `endpos` — значение параметра `<Конечная позиция>` в методах `match()` и `search()`;
- `lastindex` — возвращает номер последней группы или значение `None`;
- `lastgroup` — возвращает название последней группы или значение `None`. Пример:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m
<_sre.SRE_Match object at 0x00FC9DE8>
>>> m.re.groups, m.re.groupindex
(2, {'num': 1, 'str': 2})
>>> p.groups, p.groupindex
(2, {'num': 1, 'str': 2})
>>> m.string
'123456string 67890text'
>>> m.lastindex, m.lastgroup
(2, 'str')
>>> m.pos, m.endpos
(0, 22)
```

- `group([<id1 или name1>[, ..., <idN или nameN>]])` — возвращает фрагменты, соответствующие шаблону. Если параметр не задан или указано значение `0`, то возвращается фрагмент, полностью соответствующий шаблону. Если указан номер или название группы, то возвращается фрагмент, совпадающий с этой группой. Через запятую можно указать несколько номеров или названий групп. В этом случае возвращается кортеж, содержащий фрагменты, соответствующие группам. Если нет группы с указанным номером или названием, то возбуждается исключение `IndexError`. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m.group(), m.group(0) # Полное соответствие шаблону
('123456string', '123456string')
>>> m.group(1), m.group(2)          # Обращение по индексу
('123456', 'string')
>>> m.group("num"), m.group("str") # Обращение по названию
('123456', 'string')
>>> m.group(1, 2), m.group("num", "str") # Несколько параметров
(('123456', 'string'), ('123456', 'string'))
```

- `groupdict([<Значение по умолчанию>])` — возвращает словарь, содержащий значения именованных групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения `None`, для групп, не имеющих совпадений:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groupdict()
{'num': '123456', 'str': None}
>>> m.groupdict("")
{'num': '123456', 'str': ''}
```

- `groups([<Значение по умолчанию>])` — возвращает кортеж, содержащий значения всех групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения `None`, для групп, не имеющих совпадений:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groups()
('123456', None)
>>> m.groups("")
('123456', '')
```

- `start([<Номер группы или название>])` — возвращает индекс начала фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращается значение `-1`;
- `end([<Номер группы или название>])` — возвращает индекс конца фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращается значение `-1`;
- `span([<Номер группы или название>])` — возвращает кортеж, содержащий начальный и конечный индексы фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращается значение `(-1, -1)`. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> s = "str123456str"
>>> m = p.search(s)
>>> m.start(), m.end(), m.span()
(3, 12, (3, 12))
>>> m.start(1), m.end(1), m.start("num"), m.end("num")
(3, 9, 3, 9)
>>> m.start(2), m.end(2), m.start("str"), m.end("str")
(9, 12, 9, 12)
>>> m.span(1), m.span("num"), m.span(2), m.span("str")
((3, 9), (3, 9), (9, 12), (9, 12))
>>> s[m.start(1):m.end(1)], s[m.start(2):m.end(2)]
('123456', 'str')
```

- `expand(<Шаблон>)` — производит замену в строке. Внутри указанного шаблона можно использовать обратные ссылки: `\номер`, `\g<номер>` и `\g<название>`. В качестве примера поменяем два тега местами:

```
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> m = p.search("<br><hr>")
>>> m.expand(r"<\2><\1>")              # \номер
'<hr><br>'
>>> m.expand(r"<\g<2>><\g<1>>")        # \g<номер>
'<hr><br>'
>>> m.expand(r"<\g<tag2>><\g<tag1>>") # \g<название>
'<hr><br>'
```

В качестве примера использования метода `search()` проверим адрес электронной почты, введенный пользователем, на соответствие шаблону (листинг 7.13).

Листинг 7.13. Проверка e-mail на соответствие шаблону

```
# -*- coding: utf-8 -*-
import re
email = input("Введите e-mail: ")
email = email.rstrip("\r")        # Для версии 3.2.0 (см. разд. 1.7)
pe = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pe, re.I | re.S)
m = p.search(email)
if not m:
    print("E-mail не соответствует шаблону")
else:
    print("E-mail", m.group(0), "соответствует шаблону")
    print("ящик:", m.group(1), "домен:", m.group(2))
input()
```

Результат выполнения:

```
Введите e-mail: user@mail.ru
E-mail user@mail.ru соответствует шаблону
ящик: user домен: mail.ru
```

7.3. Поиск всех совпадений с шаблоном

Для поиска всех совпадений с шаблоном предназначено несколько функций и методов.

Метод `findall()` ищет все совпадения с шаблоном. Формат метода:

```
findall(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если соответствия найдены, то возвращается список с фрагментами, в противном случае возвращается пустой список. Если внутри шаблона есть более одной группы, то каждый элемент списка будет кортежем, а не строкой. Примеры:

```
>>> import re
>>> p = re.compile(r"[0-9]+")
>>> p.findall("2007, 2008, 2009, 2010, 2011")
['2007', '2008', '2009', '2010', '2011']
>>> p = re.compile(r"[a-z]+")
>>> p.findall("2007, 2008, 2009, 2010, 2011")
[]
>>> t = r"(([0-9]{3})-([0-9]{2})-([0-9]{2}))"
>>> p = re.compile(t)
>>> p.findall("322-77-20, 528-22-98")
[('322-77-20', '322', '77', '20'),
 ('528-22-98', '528', '22', '98')]
```

Вместо метода `findall()` можно воспользоваться функцией `findall()`. Формат функции:

```
re.findall(<Шаблон>, <Строка>[, <Модификатор>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`.

Пример:

```
>>> re.findall(r"[0-9]+", "1 2 3 4 5 6")
['1', '2', '3', '4', '5', '6']
>>> p = re.compile(r"[0-9]+")
>>> re.findall(p, "1 2 3 4 5 6")
['1', '2', '3', '4', '5', '6']
```

Метод `finditer()` аналогичен методу `findall()`, но возвращает итератор, а не список. На каждой итерации цикла возвращается объект `Match`. Формат метода:

```
finditer(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Пример:

```
>>> p = re.compile(r"[0-9]+")
>>> for m in p.finditer("2007, 2008, 2009, 2010, 2011"):
        print(m.group(0), "start:", m.start(), "end:", m.end())

2007 start: 0 end: 4
2008 start: 6 end: 10
2009 start: 12 end: 16
2010 start: 18 end: 22
2011 start: 24 end: 28
```

Вместо метода `finditer()` можно воспользоваться функцией `finditer()`. Формат функции:

```
re.finditer(<Шаблон>, <Строка>[, <Модификатор>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Получим содержимое между тегами:

```
>>> p = re.compile(r"<b>(.+?)</b>", re.I | re.S)
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> for m in re.finditer(p, s):
        print(m.group(1))

Text1
Text3
```

7.4. Замена в строке

Метод `sub()` ищет все совпадения с шаблоном и заменяет их указанным значением. Если совпадения не найдены, возвращается исходная строка. Метод имеет следующий формат:

```
sub(<Новый фрагмент или ссылка на функцию>, <Строка для замены>
    [, <Максимальное количество замен>])
```

Внутри нового фрагмента можно использовать обратные ссылки `\номер`, `\g<номер>` и `\g<название>`, соответствующие группам внутри шаблона. В качестве примера поменяем два тега местами:

```
>>> import re
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
```

```
>>> p.sub(r"<\2><\1>", "<br><hr>")              # \номер
'<hr><br>'
>>> p.sub(r"<\g<2>><\g<1>>", "<br><hr>")        # \g<номер>
'<hr><br>'
>>> p.sub(r"<\g<tag2>><\g<tag1>>", "<br><hr>") # \g<название>
'<hr><br>'
```

В качестве первого параметра можно указать ссылку на функцию. В эту функцию будет передаваться объект `Match`, соответствующий найденному фрагменту. Результат, возвращаемый этой функцией, служит фрагментом для замены. Для примера найдем все числа в строке и прибавим к ним число `10`:

```
# -*- coding: utf-8 -*-
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    x = int(m.group(0))
    x += 10
    return "{0}".format(x)

p = re.compile(r"[0-9]+")
# Заменяем все вхождения
print(p.sub(repl, "2008, 2009, 2010, 2011"))
# Заменяем только первые два вхождения
print(p.sub(repl, "2008, 2009, 2010, 2011", 2))
input()
```

Результат выполнения:

```
2018, 2019, 2020, 2021
2018, 2019, 2010, 2011
```

> ***Внимание!***
> Название функции указывается без круглых скобок.

Вместо метода `sub()` можно воспользоваться функцией `sub()`. Формат функции:

```
re.sub(<Шаблон>, <Новый фрагмент или ссылка на функцию>,
       <Строка для замены>[, <Максимальное количество замен>
       [, flags=0]])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Поменяем два тега местами, а также изменим регистр букв:

```
# -*- coding: utf-8 -*-
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    tag1 = m.group("tag1").upper()
    tag2 = m.group("tag2").upper()
    return "<{0}><{1}>".format(tag2, tag1)
```

```
p = r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>"
print(re.sub(p, repl, "<br><hr>"))
input()
```

Результат выполнения:

```
<HR><BR>
```

Метод `subn()` аналогичен методу `sub()`, но возвращает не строку, а кортеж из двух элементов — измененной строки и количества произведенных замен. Метод имеет следующий формат:

```
subn(<Новый фрагмент или ссылка на функцию>, <Строка для замены>
    [, <Максимальное количество замен>])
```

Заменим все числа в строке на `0`:

```
>>> p = re.compile(r"[0-9]+")
>>> p.subn("0", "2008, 2009, 2010, 2011")
('0, 0, 0, 0', 4)
```

Вместо метода `subn()` можно воспользоваться функцией `subn()`. Формат функции:

```
re.subn(<Шаблон>, <Новый фрагмент или ссылка на функцию>,
        <Строка для замены>[, <Максимальное количество замен>
       [, flags=0]])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Пример:

```
>>> p = r"200[79]"
>>> re.subn(p, "2001", "2007, 2008, 2009, 2010")
('2001, 2008, 2001, 2010', 2)
```

7.5. Прочие функции и методы

Метод `split()` разбивает строку по шаблону и возвращает список подстрок. Если во втором параметре задано число, то в списке будет указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент, в котором будет остаток строки. Метод имеет следующий формат:

```
split(<Исходная строка>[, <Лимит>])
```

Пример:

```
>>> import re
>>> p = re.compile(r"[\s,.]+")
>>> p.split("word1, word2\nword3\r\nword4.word5")
['word1', 'word2', 'word3', 'word4', 'word5']
>>> p.split("word1, word2\nword3\r\nword4.word5", 2)
['word1', 'word2', 'word3\r\nword4.word5']
```

Если разделитель не найден в строке, то список будет состоять только из одного элемента, содержащего исходную строку:

```
>>> p = re.compile(r"[0-9]+")
>>> p.split("word, word\nword")
['word, word\nword']
```

Вместо метода `split()` можно воспользоваться функцией `split()`. Формат функции:

```
re.split(<Шаблон>, <Исходная строка>[, <Лимит>[, flags=0]])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Пример:

```
>>> p = re.compile(r"[\s,.]+")
>>> re.split(p, "word1, word2\nword3")
['word1', 'word2', 'word3']
>>> re.split(r"[\s,.]+", "word1, word2\nword3")
['word1', 'word2', 'word3']
```

С помощью функции `escape(<Строка>)` можно экранировать все специальные символы в строке, полученной от пользователя. Эту строку в дальнейшем можно безопасно использовать внутри регулярного выражения. Пример:

```
>>> print(re.escape(r"[]().*"))
\[\]\(\)\.\*
```

ГЛАВА 8

Списки и кортежи

Списки и *кортежи* — это нумерованные наборы объектов. Каждый элемент набора содержит лишь ссылку на объект. По этой причине списки и кортежи могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Позиция элемента в наборе задается *индексом*. Обратите внимание на то, что нумерация элементов начинается с `0`, а не с `1`.

Списки и кортежи являются упорядоченными последовательностями элементов. Как и все последовательности, они поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`).

Списки относятся к изменяемым типам данных. Это означает, что мы можем не только получить элемент по индексу, но и изменить его:

```
>>> arr = [1, 2, 3]          # Создаем список
>>> arr[0]                   # Получаем элемент по индексу
1
>>> arr[0] = 50              # Изменяем элемент по индексу
>>> arr
[50, 2, 3]
```

Кортежи относятся к неизменяемым типам данных. Иными словами, можно получить элемент по индексу, но изменить его нельзя:

```
>>> t = (1, 2, 3)            # Создаем кортеж
>>> t[0]                     # Получаем элемент по индексу
1
>>> t[0] = 50                # Изменить элемент по индексу нельзя!
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t[0] = 50               # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

Рассмотрим списки и кортежи подробно.

8.1. Создание списка

Создать список можно следующими способами:

- с помощью функции `list([<Последовательность>])`. Функция позволяет преобразовать любую последовательность в список. Если параметр не указан, то создается пустой список.

Пример:

```
>>> list()                    # Создаем пустой список
[]
>>> list("String")            # Преобразуем строку в список
['S', 't', 'r', 'i', 'n', 'g']
>>> list((1, 2, 3, 4, 5))     # Преобразуем кортеж в список
[1, 2, 3, 4, 5]
```

- указав все элементы списка внутри квадратных скобок:

```
>>> arr = [1, "str", 3, "4"]
>>> arr
[1, 'str', 3, '4']
```

- заполнив список поэлементно с помощью метода `append()`:

```
>>> arr = []           # Создаем пустой список
>>> arr.append(1)      # Добавляем элемент1 (индекс 0)
>>> arr.append("str")  # Добавляем элемент2 (индекс 1)
>>> arr
[1, 'str']
```

В некоторых языках программирования (например, в PHP) можно добавить элемент, указав пустые квадратные скобки или индекс больше последнего индекса. В языке Python все эти способы приведут к ошибке:

```
>>> arr = []
>>> arr[] = 10
SyntaxError: invalid syntax
>>> arr[0] = 10
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    arr[0] = 10
IndexError: list assignment index out of range
```

При создании списка в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для списков этого делать нельзя. Рассмотрим пример:

```
>>> x = y = [1, 2]       # Якобы создали два объекта
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов и присвоили значение переменным `x` и `y`. Теперь попробуем изменить значение в переменной `y`:

```
>>> y[1] = 100           # Изменяем второй элемент
>>> x, y                 # Изменилось значение сразу в двух переменных
([1, 100], [1, 100])
```

Как видно из примера, изменение значения в переменной `y` привело также к изменению значения в переменной `x`. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x, y = [1, 2], [1, 2]
>>> y[1] = 100           # Изменяем второй элемент
>>> x, y
([1, 2], [1, 100])
```

Точно такая же ситуация возникает при использовании оператора повторения `*`. Например, в следующей инструкции производится попытка создания двух вложенных списков с помощью оператора `*`:

```
>>> arr = [ [] ] * 2     # Якобы создали два вложенных списка
>>> arr
[[], []]
>>> arr[0].append(5)     # Добавляем элемент
>>> arr                  # Изменились два элемента
[[5], [5]]
```

Создавать вложенные списки следует с помощью метода `append()` внутри цикла:

```
>>> arr = []
>>> for i in range(2): arr.append([])

>>> arr
[[], []]
>>> arr[0].append(5); arr
[[5], []]
```

Можно также воспользоваться генераторами списков:

```
>>> arr = [ [] for i in range(2) ]
>>> arr
[[], []]
>>> arr[0].append(5); arr
[[5], []]
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор `is`. Если переменные ссылаются на один и тот же объект, то оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]        # Неправильно
>>> x is y # Переменные содержат ссылку на один и тот же список
True
>>> x, y = [1, 2], [1, 2] # Правильно
>>> x is y                # Это разные объекты
False
```

Как вы уже знаете, операция присваивания сохраняет лишь ссылку на объект, а не сам объект. Что же делать, если необходимо создать копию списка? Первый способ заключается в применении операции извлечения среза, а второй способ — в использовании функции `list()` (листинг 8.1).

Листинг 8.1. Создание поверхностной копии списка

```
>>> x = [1, 2, 3, 4, 5] # Создали список
>>> # Создаем копию списка
>>> y = list(x) # или с помощью среза y = x[:]
```

```
>>> y
[1, 2, 3, 4, 5]
>>> x is y # Оператор показывает, что это разные объекты
False
>>> y[1] = 100 # Изменяем второй элемент
>>> x, y       # Изменился только список в переменной y
([1, 2, 3, 4, 5], [1, 100, 3, 4, 5])
```

На первый взгляд может показаться, что мы получили копию. Оператор `is` показывает, что это разные объекты, а изменение элемента затронуло лишь значение переменной `y`. В данном случае вроде все нормально. Но проблема заключается в том, что списки в языке Python могут иметь неограниченную степень вложенности. Рассмотрим это на примере:

```
>>> x = [1, [2, 3, 4, 5]] # Создали вложенный список
>>> y = list(x)           # Якобы сделали копию списка
>>> x is y                # Разные объекты
False
>>> y[1][1] = 100         # Изменяем элемент
>>> x, y                  # Изменение затронуло переменную x!!!
([1, [2, 100, 4, 5]], [1, [2, 100, 4, 5]])
```

В этом примере мы создали список, в котором второй элемент является вложенным списком. Далее с помощью функции `list()` попытались создать копию списка. Как и в предыдущем примере, оператор `is` показывает, что это разные объекты, но посмотрите на результат. Изменение переменной `y` затронуло и значение переменной `x`. Таким образом, функция `list()` и операция извлечения среза создают лишь *поверхностную копию* списка.

Чтобы получить полную копию списка, следует воспользоваться функцией `deepcopy()` из модуля `copy` (листинг 8.2).

Листинг 8.2. Создание полной копии списка

```
>>> import copy             # Подключаем модуль copy
>>> x = [1, [2, 3, 4, 5]]
>>> y = copy.deepcopy(x)    # Делаем полную копию списка
>>> y[1][1] = 100           # Изменяем второй элемент
>>> x, y                    # Изменился только список в переменной y
([1, [2, 3, 4, 5]], [1, [2, 100, 4, 5]])
```

Функция `deepcopy()` создает копию каждого объекта, при этом сохраняя внутреннюю структуру списка. Иными словами, если в списке существуют два элемента, ссылающиеся на один объект, то будет создана копия объекта, и элементы будут ссылаться на этот новый объект, а не на разные объекты. Пример:

```
>>> import copy          # Подключаем модуль copy
>>> x = [1, 2]
>>> y = [x, x]           # Два элемента ссылаются на один объект
>>> y
[[1, 2], [1, 2]]
>>> z = copy.deepcopy(y) # Сделали копию списка
>>> z[0] is x, z[1] is x, z[0] is z[1]
(False, False, True)
```

```
>>> z[0][0] = 300        # Изменили один элемент
>>> z                    # Значение изменилось сразу в двух элементах!
[[300, 2], [300, 2]]
>>> x                    # Начальный список не изменился
[1, 2]
```

8.2. Операции над списками

Обращение к элементам списка осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Нумерация элементов списка начинается с нуля. Выведем все элементы списка:

```
>>> arr = [1, "str", 3.2, "4"]
>>> arr[0], arr[1], arr[2], arr[3]
(1, 'str', 3.2, '4')
```

С помощью позиционного присваивания можно присвоить значения элементов списка каким-либо переменным. Количество элементов справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = [1, 2, 3] # Позиционное присваивание
>>> x, y, z
(1, 2, 3)
>>> x, y = [1, 2, 3]    # Количество элементов должно совпадать
Traceback (most recent call last):
  File "<pyshell#86>", line 1, in <module>
    x, y = [1, 2, 3]    # Количество элементов должно совпадать
ValueError: too many values to unpack (expected 2)
```

В Python 3 при позиционном присваивании перед одной из переменных слева от оператора = можно указать звездочку. В этой переменной будет сохраняться список, состоящий из "лишних" элементов. Если таких элементов нет, то список будет пустым:

```
>>> x, y, *z = [1, 2, 3]; x, y, z
(1, 2, [3])
>>> x, y, *z = [1, 2, 3, 4, 5]; x, y, z
(1, 2, [3, 4, 5])
>>> x, y, *z = [1, 2]; x, y, z
(1, 2, [])
>>> *x, y, z = [1, 2]; x, y, z
([], 1, 2)
>>> x, *y, z = [1, 2, 3, 4, 5]; x, y, z
(1, [2, 3, 4], 5)
>>> *z, = [1, 2, 3, 4, 5]; z
[1, 2, 3, 4, 5]
```

Так как нумерация элементов списка начинается с `0`, индекс последнего элемента будет на единицу меньше количества элементов. Получить количество элементов списка позволяет функция `len()`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> len(arr)              # Получаем количество элементов
5
```

```
>>> arr[len(arr)-1]       # Получаем последний элемент
5
```

Если элемент, соответствующий указанному индексу, отсутствует в списке, то возбуждается исключение `IndexError`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[5]                # Обращение к несуществующему элементу
Traceback (most recent call last):
  File "<pyshell#99>", line 1, in <module>
    arr[5]               # Обращение к несуществующему элементу
IndexError: list index out of range
```

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца списка, а точнее сказать, значение вычитается из общего количества элементов списка, чтобы получить положительный индекс:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[-1], arr[len(arr)-1] # Обращение к последнему элементу
(5, 5)
```

Так как списки относятся к изменяемым типам данных, то мы можем изменить элемент по индексу:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[0] = 600             # Изменение элемента по индексу
>>> arr
[600, 2, 3, 4, 5]
```

Кроме того, списки поддерживают операцию извлечения среза, которая возвращает указанный фрагмент списка. Формат операции:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры являются необязательными. Если параметр `<Начало>` не указан, то используется значение `0`. Если параметр `<Конец>` не указан, то возвращается фрагмент до конца списка. Следует также заметить, что элемент с индексом, указанном в этом параметре, не входит в возвращаемый фрагмент. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве значения параметров можно указать отрицательные значения.

Теперь рассмотрим несколько примеров. Сначала получим поверхностную копию списка:

```
>>> arr = [1, 2, 3, 4, 5]
>>> m = arr[:]; m # Создаем поверхностную копию и выводим значения
[1, 2, 3, 4, 5]
>>> m is arr      # Оператор is показывает, что это разные объекты
False
```

Теперь выведем символы в обратном порядке:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[::-1]             # Шаг -1
[5, 4, 3, 2, 1]
```

Выведем список без первого и последнего элементов:

```
>>> arr[1:]               # Без первого элемента
[2, 3, 4, 5]
```

```
>>> arr[:-1]              # Без последнего элемента
[1, 2, 3, 4]
```

Получим первые два элемента списка:

```
>>> arr[0:2]              # Символ с индексом 2 не входит в диапазон
[1, 2]
```

А теперь получим последний элемент:

```
>>> arr[-1:]              # Последний элемент списка
[5]
```

И, наконец, выведем фрагмент от второго элемента до четвертого включительно:

```
>>> arr[1:4] # Возвращаются элементы с индексами 1, 2 и 3
[2, 3, 4]
```

С помощью среза можно изменить фрагмент списка. Если срезу присвоить пустой список, то элементы, попавшие в срез, будут удалены:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[1:3] = [6, 7] # Изменяем значения элементов с индексами 1 и 2
>>> arr
[1, 6, 7, 4, 5]
>>> arr[1:3] = []     # Удаляем элементы с индексами 1 и 2
>>> arr
[1, 4, 5]
```

Соединить два списка в один список позволяет оператор +. Результатом объединения будет новый список:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = [6, 7, 8, 9]
>>> arr3 = arr1 + arr2
>>> arr3
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Вместо оператора + можно использовать оператор +=. Следует учитывать, что в этом случае элементы добавляются в текущий список:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr += [6, 7, 8, 9]
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Кроме рассмотренных операций, списки поддерживают операцию повторения и проверку на вхождение. Повторить список указанное количество раз можно с помощью оператора *, а выполнить проверку на вхождение элемента в список позволяет оператор in:

```
>>> [1, 2, 3] * 3                              # Операция повторения
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
```

8.3. Многомерные списки

Любой элемент списка может содержать объект произвольного типа. Например, элемент списка может быть числом, строкой, списком, кортежем, словарем и т. д. Создать вложенный список можно, например, так:

```
>>> arr = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Как вы уже знаете, выражение внутри скобок может располагаться на нескольких строках. Следовательно, предыдущий пример можно записать иначе:

```
>>> arr = [
              [1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]
          ]
```

Чтобы получить значение элемента во вложенном списке, следует указать два индекса:

```
>>> arr[1][1]
5
```

Элементы вложенного списка также могут иметь элементы произвольного типа. Количество вложений не ограничено. Таким образом, мы может создать объект любой степени сложности. В этом случае для доступа к элементам указывается несколько индексов подряд. Примеры:

```
>>> arr = [ [1, ["a", "b"], 3], [4, 5, 6], [7, 8, 9] ]
>>> arr[0][1][0]
'a'
>>> arr = [ [1, { "a": 10, "b": ["s", 5] } ] ]
>>> arr[0][1]["b"][0]
's'
```

8.4. Перебор элементов списка

Перебрать все элементы списка можно с помощью цикла `for`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> for i in arr: print(i, end=" ")
```

Результат выполнения:

```
1 2 3 4 5
```

Следует заметить, что переменную `i` внутри цикла можно изменить, но если она ссылается на неизменяемый тип данных (например, число или строку), то это не отразится на исходном списке:

```
>>> arr = [1, 2, 3, 4]       # Элементы имеют неизменяемый тип (число)
>>> for i in arr: i += 10

>>> arr                      # Список не изменился
[1, 2, 3, 4]
>>> arr = [ [1, 2], [3, 4] ] # Элементы имеют изменяемый тип (список)
>>> for i in arr: i[0] += 10
```

```
>>> arr                          # Список изменился
[[11, 2], [13, 4]]
```

Чтобы получить доступ к каждому элементу, можно, например, воспользоваться функцией `range()` для генерации индексов. Функция возвращает объект, поддерживающий итерации. С помощью этого объекта внутри цикла `for` можно получить текущий индекс. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон значений. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве примера умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

Можно также воспользоваться функцией `enumerate(<Объект>[, start=0])`, которая на каждой итерации цикла `for` возвращает кортеж из индекса и значения текущего элемента списка. Умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i, elem in enumerate(arr):
    arr[i] *= 2
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

Кроме того, перебрать элементы можно с помощью цикла `while`. Но в этом случае следует помнить, что цикл `while` работает медленнее цикла `for`. В качестве примера умножим каждый элемент списка на 2, используя цикл `while`:

```
arr = [1, 2, 3, 4]
i, c = 0, len(arr)
while i < c:
    arr[i] *= 2
    i += 1
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

8.5. Генераторы списков и выражения-генераторы

В предыдущем разделе мы изменяли элементы списка следующим образом:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                      # Результат выполнения: [2, 4, 6, 8]
```

С помощью генераторов списков тот же самый код можно записать более компактно. Помимо компактного отображения генераторы списков работают быстрее цикла `for`. Однако вместо изменения исходного списка возвращается новый список:

```
arr = [1, 2, 3, 4]
arr = [ i * 2 for i in arr ]
print(arr)                    # Результат выполнения: [2, 4, 6, 8]
```

Как видно из примера, мы поместили цикл `for` внутри квадратных скобок, а также изменили порядок следования параметров: инструкция, выполняемая внутри цикла, находится перед циклом. Обратите внимание на то, что выражение внутри цикла не содержит оператора присваивания. На каждой итерации цикла будет генерироваться новый элемент, которому неявным образом присваивается результат выполнения выражения внутри цикла. В итоге будет создан новый список, содержащий измененные значения элементов исходного списка.

Генераторы списков могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. В качестве примера получим четные элементы списка и умножим их на 10:

```
arr = [1, 2, 3, 4]
arr = [ i * 10 for i in arr if i % 2 == 0 ]
print(arr)                    # Результат выполнения: [20, 40]
```

Последовательность выполнения этого кода эквивалентна последовательности выполнения следующего кода:

```
arr = []
for i in [1, 2, 3, 4]:
    if i % 2 == 0:            # Если число четное
        arr.append(i * 10)    # Добавляем элемент
print(arr)                    # Результат выполнения: [20, 40]
```

Усложним наш пример. Получим четные элементы вложенного списка и умножим их на 10:

```
arr = [[1, 2], [3, 4], [5, 6]]
arr = [ j * 10 for i in arr for j in i if j % 2 == 0 ]
print(arr)                    # Результат выполнения: [20, 40, 60]
```

Последовательность выполнения этого кода эквивалентна последовательности выполнения следующего кода:

```
arr = []
for i in [[1, 2], [3, 4], [5, 6]]:
    for j in i:
        if j % 2 == 0:          # Если число четное
            arr.append(j * 10)  # Добавляем элемент
print(arr)                      # Результат выполнения: [20, 40, 60]
```

Если выражение разместить не внутри квадратных скобок, а внутри круглых скобок, то будет возвращаться не список, а итератор. Такие конструкции называются *выражениями-генераторами*. В качестве примера просуммируем четные числа в списке:

```
>>> arr = [1, 4, 12, 45, 10]
>>> sum( ( i for i in arr if i % 2 == 0 ) )
26
```

8.6. Функции *map()*, *zip()*, *filter()* и *reduce()*

Встроенная функция `map()` позволяет применить функцию к каждому элементу последовательности. Функция имеет следующий формат:

```
map(<Функция>, <Последовательность1>[, ..., <ПоследовательностьN>])
```

Функция `map()` возвращает объект, поддерживающий итерации, а не список, как это было ранее в Python 2. Чтобы получить список в версии Python 3, необходимо результат передать в функцию `list()`.

В качестве параметра `<Функция>` указывается ссылка на функцию (название функции без круглых скобок), которой будет передаваться текущий элемент последовательности. Внутри функции обратного вызова необходимо вернуть новое значение. Прибавим к каждому элементу списка число 10 (листинг 8.3).

Листинг 8.3. Функция `map()`

```
def func(elem):
    """ Увеличение значения каждого элемента списка """
    return elem + 10 # Возвращаем новое значение

arr = [1, 2, 3, 4, 5]
print( list( map(func, arr) ) )
# Результат выполнения: [11, 12, 13, 14, 15]
```

Функции `map()` можно передать несколько последовательностей. В этом случае в функцию обратного вызова будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковом смещении. Просуммируем элементы трех списков (листинг 8.4).

Листинг 8.4. Суммирование элементов трех списков

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3 # Возвращаем новое значение

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222, 333, 444, 555]
```

Если количество элементов в последовательностях будет разным, то в качестве ограничения выбирается последовательность с минимальным количеством элементов:

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20]
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222]
```

Встроенная функция `zip()` на каждой итерации возвращает кортеж, содержащий элементы последовательностей, которые расположены на одинаковом смещении. Функция возвраща-

ет объект, поддерживающий итерации, а не список, как это было ранее в Python 2. Чтобы получить список в версии Python 3, необходимо результат передать в функцию `list()`. Формат функции:

```
zip(<Последовательность1>[, ..., <ПоследовательностьN>])
```

Пример:

```
>>> zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
<zip object at 0x00FCAC88>
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Если количество элементов в последовательностях будет разным, то в результат попадут только элементы, которые существуют во всех последовательностях на одинаковом смещении:

```
>>> list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10]))
[(1, 4, 7), (2, 6, 8)]
```

В качестве еще одного примера переделаем нашу программу (листинг 8.4) суммирования элементов трех списков и используем функцию `zip()` вместо функции `map()` (листинг 8.5).

Листинг 8.5. Суммирование элементов трех списков с помощью функции `zip()`

```
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
arr = [x + y + z for (x, y, z) in zip(arr1, arr2, arr3)]
print(arr)
# Результат выполнения: [111, 222, 333, 444, 555]
```

Функция `filter()` позволяет выполнить проверку элементов последовательности. Формат функции:

```
filter(<Функция>, <Последовательность>)
```

Если в первом параметре вместо названия функции указать значение `None`, то каждый элемент последовательности будет проверен на соответствие значению `True`. Если элемент в логическом контексте возвращает значение `False`, то он не будет добавлен в возвращаемый результат. Функция возвращает объект, поддерживающий итерации, а не список или кортеж, как это было ранее в Python 2. Чтобы получить список в версии Python 3, необходимо результат передать в функцию `list()`. Пример:

```
>>> filter(None, [1, 0, None, [], 2])
<filter object at 0x00FD58B0>
>>> list(filter(None, [1, 0, None, [], 2]))
[1, 2]
```

Аналогичная операция с использованием генераторов списков выглядит так:

```
>>> [ i for i in [1, 0, None, [], 2] if i ]
[1, 2]
```

В первом параметре можно указать ссылку на функцию. В эту функцию в качестве параметра будет передаваться текущий элемент последовательности. Если элемент следует до-

бавить в возвращаемое функцией `filter()` значение, то внутри функции обратного вызова следует вернуть значение `True`, в противном случае — значение `False`. Удалим все отрицательные значения из списка (листинг 8.6).

Листинг 8.6. Пример использования функции `filter()`

```
def func(elem):
    return elem >= 0

arr = [-1, 2, -3, 4, 0, -20, 10]
arr = list(filter(func, arr))
print(arr)                              # Результат: [2, 4, 0, 10]
# Использование генераторов списков
arr = [-1, 2, -3, 4, 0, -20, 10]
arr = [ i for i in arr if func(i) ]
print(arr)                              # Результат: [2, 4, 0, 10]
```

Функция `reduce()` из модуля `functools` применяет указанную функцию к парам элементов и накапливает результат. Имеет следующий формат:

```
reduce(<Функция>, <Последовательность>[, <Начальное значение>])
```

В функцию обратного вызова в качестве параметров будут передаваться два элемента. Первый элемент будет содержать результат предыдущих вычислений, а второй — значение текущего элемента. Получим сумму всех элементов списка (листинг 8.7).

Листинг 8.7. Пример использования функции `reduce()`

```
from functools import reduce # Подключаем модуль

def func(x, y):
    print("({0}, {1})".format(x, y), end=" ")
    return x + y

arr = [1, 2, 3, 4, 5]
summa = reduce(func, arr)
# Последовательность: (1, 2) (3, 3) (6, 4) (10, 5)
print(summa) # Результат выполнения: 15
summa = reduce(func, arr, 10)
# Последовательность: (10, 1) (11, 2) (13, 3) (16, 4) (20, 5)
print(summa) # Результат выполнения: 25
summa = reduce(func, [], 10)
print(summa) # Результат выполнения: 10
```

8.7. Добавление и удаление элементов списка

Для добавления и удаления элементов списка используются следующие методы:

- `append(<Объект>)` — добавляет один объект в конец списка. Метод изменяет текущий список и ничего не возвращает.

Пример:

```
>>> arr = [1, 2, 3]
>>> arr.append(4); arr          # Добавляем число
[1, 2, 3, 4]
>>> arr.append([5, 6]); arr     # Добавляем список
[1, 2, 3, 4, [5, 6]]
>>> arr.append((7, 8)); arr     # Добавляем кортеж
[1, 2, 3, 4, [5, 6], (7, 8)]
```

- `extend(<Последовательность>)` — добавляет элементы последовательности в конец списка. Метод изменяет текущий список и ничего не возвращает. Пример:

```
>>> arr = [1, 2, 3]
>>> arr.extend([4, 5, 6])       # Добавляем список
>>> arr.extend((7, 8, 9))       # Добавляем кортеж
>>> arr.extend("abc")           # Добавляем буквы из строки
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c']
```

Добавить несколько элементов можно с помощью операции конкатенации или оператора +=:

```
>>> arr = [1, 2, 3]
>>> arr + [4, 5, 6]             # Возвращает новый список
[1, 2, 3, 4, 5, 6]
>>> arr += [4, 5, 6]            # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6]
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> arr = [1, 2, 3]
>>> arr[len(arr):] = [4, 5, 6]  # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6]
```

- `insert(<Индекс>, <Объект>)` — добавляет один объект в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий список и ничего не возвращает. Примеры:

```
>>> arr = [1, 2, 3]
>>> arr.insert(0, 0); arr   # Вставляем 0 в начало списка
[0, 1, 2, 3]
>>> arr.insert(-1, 20); arr # Можно указать отрицательные числа
[0, 1, 2, 20, 3]
>>> arr.insert(2, 100); arr # Вставляем 100 в позицию 2
[0, 1, 100, 2, 20, 3]
>>> arr.insert(10, [4, 5]); arr # Добавляем список
[0, 1, 100, 2, 20, 3, [4, 5]]
```

Метод `insert()` позволяет добавить только один объект. Чтобы добавить несколько объектов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало списка:

```
>>> arr = [1, 2, 3]
>>> arr[:0] = [-2, -1, 0]
>>> arr
[-2, -1, 0, 1, 2, 3]
```

- `pop([<Индекс>])` — удаляет элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, то удаляет и возвращает последний элемент списка. Если элемента с указанным индексом нет или список пустой, возбуждается исключение `IndexError`. Примеры:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.pop()        # Удаляем последний элемент списка
5
>>> arr              # Список изменился
[1, 2, 3, 4]
>>> arr.pop(0)       # Удаляем первый элемент списка
1
>>> arr              # Список изменился
[2, 3, 4]
```

Удалить элемент списка позволяет также оператор `del`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> del arr[4]; arr  # Удаляем последний элемент списка
[1, 2, 3, 4]
>>> del arr[:2]; arr # Удаляем первый и второй элементы
[3, 4]
```

- `remove(<Значение>)` — удаляет первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод изменяет текущий список и ничего не возвращает. Примеры:

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.remove(1)    # Удаляет только первый элемент
>>> arr
[2, 3, 1, 1]
>>> arr.remove(5)    # Такого элемента нет
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    arr.remove(5)    # Такого элемента нет
ValueError: list.remove(x): x not in list
```

Если необходимо удалить все повторяющиеся элементы списка, то можно список преобразовать во множество, а затем множество обратно преобразовать в список. Обратите внимание на то, что список должен содержать только неизменяемые объекты (например, числа, строки или кортежи). В противном случае возбуждается исключение `TypeError`. Пример:

```
>>> arr = [1, 2, 3, 1, 1, 2, 2, 3, 3]
>>> s = set(arr)          # Преобразуем список во множество
>>> s
{1, 2, 3}
>>> arr = list(s)         # Преобразуем множество в список
>>> arr                   # Все повторы были удалены
[1, 2, 3]
```

8.8. Поиск элемента в списке

Как вы уже знаете, выполнить проверку на вхождение элемента в список позволяет оператор `in`. Если элемент входит в список, то возвращается значение `True`, в противном случае — `False`. Пример:

```
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
```

Тем не менее, оператор `in` не дает никакой информации о местонахождении элемента внутри списка. Чтобы узнать индекс элемента внутри списка, следует воспользоваться методом `index()`. Формат метода:

```
index(<Значение>[, <Начало>[, <Конец>]])
```

Метод `index()` возвращает индекс элемента, имеющего указанное значение. Если значение не входит в список, то возбуждается исключение `ValueError`. Если второй и третий параметры не указаны, то поиск будет производиться с начала списка. Пример:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.index(1), arr.index(2)
(0, 1)
>>> arr.index(1, 1), arr.index(1, 3, 5)
(2, 4)
>>> arr.index(3)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    arr.index(3)
ValueError: 3 is not in list
```

Узнать общее количество элементов с указанным значением позволяет метод `count(<Значение>)`. Если элемент не входит в список, то возвращается значение `0`. Пример:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.count(1), arr.count(2)
(3, 2)
>>> arr.count(3)                    # Элемент не входит в список
0
```

С помощью функций `max()` и `min()` можно узнать максимальное и минимальное значение списка соответственно. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> max(arr), min(arr)
(5, 1)
```

Функция `any(<Последовательность>)` возвращает значение `True`, если в последовательности существует хоть один элемент, который в логическом контексте возвращает значение `True`. Если последовательность не содержит элементов, возвращается значение `False`. Пример:

```
>>> any([0, None]), any([0, None, 1]), any([])
(False, True, False)
```

Функция `all(<Последовательность>)` возвращает значение `True`, если все элементы последовательности в логическом контексте возвращают значение `True` или последовательность не содержит элементов.

Пример:

```
>>> all([0, None]), all([0, None, 1]), all([]), all(["str", 10])
(False, False, True, True)
```

8.9. Переворачивание и перемешивание списка

Метод `reverse()` изменяет порядок следования элементов списка на противоположный. Метод изменяет текущий список и ничего не возвращает. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.reverse()               # Изменяется текущий список
>>> arr
[5, 4, 3, 2, 1]
```

Если необходимо изменить порядок следования и получить новый список, то следует воспользоваться функцией `reversed(<Последовательность>)`. Функция возвращает итератор, который можно преобразовать в список с помощью функции `list()` или генератора списков:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> reversed(arr)
<list_reverseiterator object at 0x00FD5150>
>>> list(reversed(arr))              # Использование функции list()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> for i in reversed(arr): print(i, end=" ") # Вывод с помощью цикла

10 9 8 7 6 5 4 3 2 1
>>> [i for i in reversed(arr)] # Использование генератора списков
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Функция `shuffle(<Список>[, <Число от 0.0 до 1.0>])` из модуля `random` "перемешивает" список случайным образом. Функция перемешивает сам список и ничего не возвращает. Если второй параметр не указан, то используется значение, возвращаемое функцией `random()`. Пример:

```
>>> import random         # Подключаем модуль random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)   # Перемешиваем список случайным образом
>>> arr
[2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
```

8.10. Выбор элементов случайным образом

Получить элементы из списка случайным образом позволяют следующие функции из модуля `random`:

- `choice(<Последовательность>)` — возвращает случайный элемент из любой последовательности (строки, списка, кортежа):

```
>>> import random # Подключаем модуль random
>>> random.choice(["s", "t", "r"]) # Список
's'
```

- `sample(<Последовательность>, <Количество элементов>)` — возвращает список из указанного количества элементов. В этот список попадут элементы из последовательности, выбранные случайным образом. В качестве последовательности можно указать любые объекты, поддерживающие итерации. Пример:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
[7, 10]
>>> arr                          # Сам список не изменяется
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

8.11. Сортировка списка

Отсортировать список позволяет метод `sort()`. Метод имеет следующий формат:

```
sort([key=None][, reverse=False])
```

Все параметры являются необязательными. Метод изменяет текущий список и ничего не возвращает. Отсортируем список по возрастанию с параметрами по умолчанию:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort()                              # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Чтобы отсортировать список по убыванию, следует в параметре `reverse` указать значение `True`:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort(reverse=True)                  # Сортировка по убыванию
>>> arr
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Следует заметить, что стандартная сортировка зависит от регистра символов (листинг 8.8).

Листинг 8.8. Стандартная сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort()
for i in arr:
    print(i, end=" ")
# Результат выполнения: Единица2 Единый единица1
```

В результате мы получили неправильную сортировку, ведь `Единый` и `Единица2` больше `единица1`. Чтобы регистр символов не учитывался, можно указать ссылку на функцию для изменения регистра символов в параметре `key` (листинг 8.9).

Листинг 8.9. Пользовательская сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=str.lower)                  # Указываем метод lower()
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

Метод `sort()` сортирует сам список и не возвращает никакого значения. В некоторых случаях необходимо получить отсортированный список, а текущий список оставить без изменений. Для этого следует воспользоваться функцией `sorted()`. Функция имеет следующий формат:

```
sorted(<Последовательность>[, key=None][, reverse=False])
```

В первом параметре указывается список, который необходимо отсортировать. Остальные параметры эквивалентны параметрам метода `sort()`. Пример использования функции `sorted()` приведен в листинге 8.10.

Листинг 8.10. Пример использования функции `sorted()`

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> sorted(arr)               # Возвращает новый список!
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sorted(arr, reverse=True) # Возвращает новый список!
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr = ["единица1", "Единый", "Единица2"]
>>> sorted(arr, key=str.lower)
['единица1', 'Единица2', 'Единый']
```

8.12. Заполнение списка числами

Создать список, содержащий диапазон чисел, можно с помощью функции `range()`. Функция возвращает объект, поддерживающий итерации. Чтобы из этого объекта получить список, следует воспользоваться функцией `list()`. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый список значений. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве примера заполним список числами от `0` до `10`:

```
>>> list(range(11))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Создадим список, состоящий из диапазона чисел от `1` до `15`:

```
>>> list(range(1, 16))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Теперь изменим порядок следования чисел на противоположный:

```
>>> list(range(15, 0, -1))
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Если необходимо получить список со случайными числами (или случайными элементами из другого списка), то следует воспользоваться функцией `sample(<Последовательность>, <Количество элементов>)` из модуля `random`. Пример:

```
>>> import random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> random.sample(arr, 3)
[1, 9, 5]
>>> random.sample(range(300), 5)
[259, 294, 142, 292, 245]
```

8.13. Преобразование списка в строку

Преобразовать список в строку позволяет метод `join()`. Элементы добавляются через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

Пример:

```
>>> arr = ["word1", "word2", "word3"]
>>> " — ".join(arr)
'word1 — word2 — word3'
```

Обратите внимание на то, что элементы списка должны быть строками, иначе возвращается исключение `TypeError`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " — ".join(arr)
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    " — ".join(arr)
TypeError: sequence item 3: expected str instance, int found
```

Избежать этого исключения можно с помощью выражения-генератора, внутри которого текущий элемент списка преобразуется в строку с помощью функции `str()`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " — ".join( ( str(i) for i in arr ) )
'word1 — word2 — word3 — 2'
```

Кроме того, с помощью функции `str()` можно сразу получить строковое представление списка:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> str(arr)
"['word1', 'word2', 'word3', 2]"
```

8.14. Кортежи

Кортежи, так же как и списки, являются упорядоченными последовательностями элементов. Кортежи во многом аналогичны спискам, но имеют одно очень важное отличие — изменить кортеж нельзя. Можно сказать, что кортеж — это список, доступный "только для чтения". Создать кортеж можно следующими способами:

- с помощью функции `tuple([<Последовательность>])`. Функция позволяет преобразовать любую последовательность в кортеж. Если параметр не указан, то создается пустой кортеж. Пример:

```
>>> tuple()                # Создаем пустой кортеж
()
```

```
>>> tuple("String")        # Преобразуем строку в кортеж
('S', 't', 'r', 'i', 'n', 'g')
>>> tuple([1, 2, 3, 4, 5]) # Преобразуем список в кортеж
(1, 2, 3, 4, 5)
```

- указав все элементы через запятую внутри круглых скобок (или без скобок):

```
>>> t1 = ()         # Создаем пустой кортеж
>>> t2 = (5,)       # Создаем кортеж из одного элемента
>>> t3 = (1, "str", (3, 4)) # Кортеж из трех элементов
>>> t4 = 1, "str", (3, 4)   # Кортеж из трех элементов
>>> t1, t2, t3, t4
((), (5,), (1, 'str', (3, 4)), (1, 'str', (3, 4)))
```

Обратите особое внимание на вторую строку примера. Чтобы создать кортеж из одного элемента, необходимо в конце указать запятую. Именно запятые формируют кортеж, а не круглые скобки. Если внутри круглых скобок нет запятых, то будет создан объект другого типа. Пример:

```
>>> t = (5); type(t)      # Получили число, а не кортеж!
<class 'int'>
>>> t = ("str"); type(t)  # Получили строку, а не кортеж!
<class 'str'>
```

Четвертая строка в предыдущем примере также доказывает, что не скобки формируют кортеж, а запятые. Помните, что любое выражение в языке Python можно заключить в круглые скобки, а чтобы получить кортеж, необходимо указать запятые.

Позиция элемента в кортеже задается *индексом*. Обратите внимание на то, что нумерация элементов кортежа (как и списка) начинается с `0`, а не с `1`. Как и все последовательности, кортежи поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`). Пример:

```
>>> t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[0]                  # Получаем значение первого элемента кортежа
1
>>> t[::-1]               # Изменяем порядок следования элементов
(9, 8, 7, 6, 5, 4, 3, 2, 1)
>>> t[2:5]                # Получаем срез
(3, 4, 5)
>>> 8 in t, 0 in t        # Проверка на вхождение
(True, False)
>>> (1, 2, 3) * 3         # Повторение
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> (1, 2, 3) + (4, 5, 6) # Конкатенация
(1, 2, 3, 4, 5, 6)
```

Кортежи относятся к неизменяемым типам данных. Иными словами, можно получить элемент по индексу, но изменить его нельзя:

```
>>> t = (1, 2, 3)         # Создаем кортеж
>>> t[0]                  # Получаем элемент по индексу
1
>>> t[0] = 50             # Изменить элемент по индексу нельзя!
```

```
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in <module>
    t[0] = 50             # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

Получить количество элементов кортежа позволяет функция `len()`:

```
>>> t = (1, 2, 3)         # Создаем кортеж
>>> len(t)                # Получаем количество элементов
3
```

Начиная с Python 2.6, кортежи поддерживают два метода:

- `index(<Значение>[, <Начало>[, <Конец>]])` — возвращает индекс элемента, имеющего указанное значение. Если значение не входит в кортеж, возбуждается исключение `ValueError`. Если второй и третий параметры не указаны, то поиск будет производиться с начала кортежа. Пример:

```
>>> t = (1, 2, 1, 2, 1)
>>> t.index(1), t.index(2)
(0, 1)
>>> t.index(1, 1), t.index(1, 3, 5)
(2, 4)
>>> t.index(3)
... Фрагмент опущен ...
ValueError: tuple.index(x): x not in tuple
```

- `count(<Значение>)` — возвращает количество элементов с указанным значением. Если элемент не входит в кортеж, то возвращается значение `0`. Пример:

```
>>> t = (1, 2, 1, 2, 1)
>>> t.count(1), t.count(2)
(3, 2)
>>> t.count(3)              # Элемент не входит в кортеж
0
```

Других методов у кортежей нет. Чтобы произвести операции над кортежами, следует воспользоваться встроенными функциями, предназначенными для работы с последовательностями. Все эти функции мы уже рассматривали при изучении списков.

8.15. Модуль *itertools*

Модуль `itertools` содержит функции, позволяющие генерировать различные последовательности на основе других последовательностей, производить фильтрацию элементов и др. Все функции возвращают объекты, поддерживающие итерации. Прежде чем использовать функции, необходимо подключить модуль с помощью инструкции:

```
import itertools
```

8.15.1. Генерация неопределенного количества значений

Для генерации неопределенного количества значений предназначены следующие функции:

- `count([start=0][, step=1])` — создает бесконечно нарастающую последовательность значений. Начальное значение задается параметром `start`, а шаг — параметром `step`.

Пример:

```
>>> import itertools
>>> for i in itertools.count():
        if i > 10: break
        print(i, end=" ")

0 1 2 3 4 5 6 7 8 9 10
>>> list(zip(itertools.count(), "абвгд"))
[(0, 'а'), (1, 'б'), (2, 'в'), (3, 'г'), (4, 'д')]
>>> list(zip(itertools.count(start=2, step=2), "абвгд"))
[(2, 'а'), (4, 'б'), (6, 'в'), (8, 'г'), (10, 'д')]
```

- `cycle(<Последовательность>)` — на каждой итерации возвращается очередной элемент последовательности. Когда будет достигнут конец последовательности, перебор начнется сначала. И так бесконечно. Пример:

```
>>> n = 1
>>> for i in itertools.cycle("абв"):
        if n > 10: break
        print(i, end=" ")
        n += 1

а б в а б в а б в а
>>> list(zip(itertools.cycle([0, 1]), "абвгд"))
[(0, 'а'), (1, 'б'), (0, 'в'), (1, 'г'), (0, 'д')]
```

- `repeat(<Объект>[, <Количество повторов>])` — возвращает объект указанное количество раз. Если количество повторов не указано, то объект возвращается бесконечно. Пример:

```
>>> list(itertools.repeat(1, 10))
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> list(zip(itertools.repeat(5), "абвгд"))
[(5, 'а'), (5, 'б'), (5, 'в'), (5, 'г'), (5, 'д')]
```

8.15.2. Генерация комбинаций значений

Получить различные комбинации значений позволяют следующие функции:

- `combinations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. Формат функции:

```
combinations(<Последовательность>, <Количество элементов>)
```

Пример:

```
>>> import itertools
>>> list(itertools.combinations('абвг', 2))
[('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'в'), ('б', 'г'),
 ('в', 'г')]
>>> ["".join(i) for i in itertools.combinations('абвг', 2)]
['аб', 'ав', 'аг', 'бв', 'бг', 'вг']
```

```
>>> list(itertools.combinations('вгаб', 2))
[('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'а'), ('г', 'б'),
 ('а', 'б')]
>>> list(itertools.combinations('абвг', 3))
[('а', 'б', 'в'), ('а', 'б', 'г'), ('а', 'в', 'г'),
 ('б', 'в', 'г')]
```

- `combinations_with_replacement()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. Формат функции:

```
combinations_with_replacement(<Последовательность>,
                              <Количество элементов>)
```

Пример:

```
>>> list(itertools.combinations_with_replacement('абвг', 2))
[('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'б'),
 ('б', 'в'), ('б', 'г'), ('в', 'в'), ('в', 'г'), ('г', 'г')]
>>> list(itertools.combinations_with_replacement('вгаб', 2))
[('в', 'в'), ('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'г'),
 ('г', 'а'), ('г', 'б'), ('а', 'а'), ('а', 'б'), ('б', 'б')]
```

- `permutations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. Если количество элементов не указано, то используется длина последовательности. Формат функции:

```
permutations(<Последовательность>[, <Количество элементов>])
```

Пример:

```
>>> list(itertools.permutations('абвг', 2))
[('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'), ('б', 'в'),
 ('б', 'г'), ('в', 'а'), ('в', 'б'), ('в', 'г'), ('г', 'а'),
 ('г', 'б'), ('г', 'в')]
>>> ["".join(i) for i in itertools.permutations('абвг')]
['абвг', 'абгв', 'авбг', 'авгб', 'агбв', 'агвб', 'бавг',
 'багв', 'бваг', 'бвга', 'бгав', 'бгва', 'вабг', 'вагб',
 'вбаг', 'вбга', 'вгаб', 'вгба', 'габв', 'гавб', 'гбав',
 'гбва', 'гваб', 'гвба']
```

- `product()` — на каждой итерации возвращает кортеж, содержащий комбинацию из элементов одной или нескольких последовательностей. Формат функции:

```
product(<Последовательность1>[,..., <ПоследовательностьN>][,
        repeat=1])
```

Пример:

```
>>> from itertools import product
>>> list(product('абвг', repeat=2))
[('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'),
 ('б', 'б'), ('б', 'в'), ('б', 'г'), ('в', 'а'), ('в', 'б'),
 ('в', 'в'), ('в', 'г'), ('г', 'а'), ('г', 'б'), ('г', 'в'),
 ('г', 'г')]
>>> ["".join(i) for i in product('аб', 'вг', repeat=1)]
['ав', 'аг', 'бв', 'бг']
```

```
>>> ["".join(i) for i in product('аб', 'вг', repeat=2)]
['авав', 'аваг', 'авбв', 'авбг', 'агав', 'агаг', 'агбв',
 'агбг', 'бвав', 'бваг', 'бвбв', 'бвбг', 'бгав', 'бгаг',
 'бгбв', 'бгбг']
```

8.15.3. Фильтрация элементов последовательности

Для фильтрации элементов последовательности предназначены следующие функции:

- `filterfalse(<Функция>, <Последовательность>)` — возвращает элементы последовательности (по одному на каждой итерации), для которых функция, указанная в первом параметре, вернет значение `False`. Пример:

```
>>> import itertools
>>> def func(x): return x > 3

>>> list(itertools.filterfalse(func, [4, 5, 6, 0, 7, 2, 3]))
[0, 2, 3]
>>> list(filter(func, [4, 5, 6, 0, 7, 2, 3]))
[4, 5, 6, 7]
```

  Если в первом параметре вместо названия функции указать значение `None`, то каждый элемент последовательности будет проверен на соответствие значению `False`. Если элемент в логическом контексте возвращает значение `True`, то он не будет входить в возвращаемый результат. Пример:

```
>>> list(itertools.filterfalse(None, [0, 5, 6, 0, 7, 0, 3]))
[0, 0, 0]
>>> list(filter(None, [0, 5, 6, 0, 7, 0, 3]))
[5, 6, 7, 3]
```

- `dropwhile(<Функция>, <Последовательность>)` — возвращает элементы последовательности (по одному на каждой итерации), начиная с элемента, для которого функция, указанная в первом параметре, вернет значение `False`. Пример:

```
>>> def func(x): return x > 3

>>> list(itertools.dropwhile(func, [4, 5, 6, 0, 7, 2, 3]))
[0, 7, 2, 3]
>>> list(itertools.dropwhile(func, [4, 5, 6, 7, 8]))
[]
>>> list(itertools.dropwhile(func, [1, 2, 4, 5, 6, 7, 8]))
[1, 2, 4, 5, 6, 7, 8]
```

- `takewhile(<Функция>, <Последовательность>)` — возвращает элементы последовательности (по одному на каждой итерации), пока не встретится элемент, для которого функция, указанная в первом параметре, вернет значение `False`. Пример:

```
>>> def func(x): return x > 3

>>> list(itertools.takewhile(func, [4, 5, 6, 0, 7, 2, 3]))
[4, 5, 6]
>>> list(itertools.takewhile(func, [4, 5, 6, 7, 8]))
[4, 5, 6, 7, 8]
>>> list(itertools.takewhile(func, [1, 2, 4, 5, 6, 7, 8]))
[]
```

- `compress()` — производит фильтрацию последовательности, указанной в первом параметре. Элемент возвращается, только если соответствующий элемент (с таким же индексом) из второй последовательности трактуется как истина. Сравнение заканчивается, когда достигнут конец одной из последовательностей. Формат функции:

```
compress(<Фильтруемая последовательность>,
        <Последовательность логических значений>)
```

Пример:

```
>>> list(itertools.compress('абвгде', [1, 0, 0, 0, 1, 1]))
['а', 'д', 'е']
>>> list(itertools.compress('абвгде', [True, False, True]))
['а', 'в']
```

8.15.4. Прочие функции

Помимо функций, которые мы рассмотрели в предыдущих подразделах, модуль `itertools` содержит несколько дополнительных функций:

- `starmap(<Функция>, <Последовательность>)` — передает значение в функцию и возвращает результат ее выполнения. Обратите внимание на то, что каждый элемент должен быть последовательностью. При передаче в функцию производится распаковка последовательности. Иными словами, в функцию передаются отдельные элементы последовательности, а не последовательность целиком. Пример суммирования значений:

```
>>> import itertools
>>> def func1(x, y): return x + y

>>> list(itertools.starmap(func1, [(1, 2), (4, 5), (6, 7)]))
[3, 9, 13]
>>> def func2(x, y, z): return x + y + z

>>> list(itertools.starmap(func2, [(1, 2, 3), (4, 5, 6)]))
[6, 15]
```

- `zip_longest()` — на каждой итерации возвращает кортеж, содержащий элементы последовательностей, которые расположены на одинаковом смещении. Если последовательности имеют разное количество элементов, то вместо отсутствующего элемента вставляется объект, указанный в параметре `fillvalue`. Формат функции:

```
zip_longest(<Последовательность1>[, ..., <ПоследовательностьN>]
           [, fillvalue=None])
```

Пример:

```
>>> list(itertools.zip_longest([1, 2, 3], [4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
>>> list(itertools.zip_longest([1, 2, 3], [4]))
[(1, 4), (2, None), (3, None)]
>>> list(itertools.zip_longest([1, 2, 3], [4], fillvalue=0))
[(1, 4), (2, 0), (3, 0)]
>>> list(zip([1, 2, 3], [4]))
[(1, 4)]
```

- `accumulate(<Последовательность>)` — на каждой итерации возвращает сумму предыдущих элементов последовательности. Начальное значение равно `0`. Пример:

```
>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6]))
[1, 3, 6, 10, 15, 21]
>>> # [0+1, 1+2, 3+3, 6+4, 10+5, 15+6]
```

- `chain()` — на каждой итерации возвращает элементы сначала из первой последовательности, затем из второй последовательности и т. д. Формат функции:

```
chain(<Последовательность1>[, ..., <ПоследовательностьN>])
```

Пример:

```
>>> arr1, arr2, arr3 = [1, 2, 3], [4, 5], [6, 7, 8, 9]
>>> list(itertools.chain(arr1, arr2, arr3))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(itertools.chain("abc", "defg", "hij"))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> list(itertools.chain("abc", ["defg", "hij"]))
['a', 'b', 'c', 'defg', 'hij']
```

- `chain.from_iterable(<Последовательность>)` — аналогична функции `chain()`, но принимает одну последовательность, каждый элемент которой считается отдельной последовательностью. Пример:

```
>>> list(itertools.chain.from_iterable(["abc", "defg", "hij"]))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- `tee(<Последовательность>[, <Количество>])` — возвращает кортеж, содержащий несколько итераторов для последовательности. Если второй параметр не указан, то возвращается кортеж из двух итераторов. Пример:

```
>>> arr = [1, 2, 3]
>>> itertools.tee(arr)
(<itertools.tee object at 0x00FD8760>,
<itertools.tee object at 0x00FD8738>)
>>> itertools.tee(arr, 3)
(<itertools.tee object at 0x00FD8710>,
<itertools.tee object at 0x00FD87D8>,
<itertools.tee object at 0x00FD87B0>)
```

ГЛАВА 9

Словари и множества

Словари — это наборы объектов, доступ к которым осуществляется не по индексу, а по ключу. В качестве ключа можно указать неизменяемый объект, например число, строку или кортеж. Элементы словаря могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Следует также заметить, что элементы в словарях располагаются в произвольном порядке. Чтобы получить элемент, необходимо указать ключ, который использовался при сохранении значения.

Словари относятся к отображениям, а не к последовательностям. По этой причине функции, предназначенные для работы с последовательностями, а также операции извлечения среза, конкатенации, повторения и др., к словарям не применимы. Так же как и списки, словари относятся к изменяемым типам данных. Иными словами, мы можем не только получить значение по ключу, но и изменить его.

9.1. Создание словаря

Создать словарь можно следующими способами:

- с помощью функции `dict()`. Форматы функции:

```
dict(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
dict(<Словарь>)
dict(<Список кортежей с двумя элементами (Ключ, Значение)>)
dict(<Список списков с двумя элементами [Ключ, Значение]>)
```

Если параметры не указаны, то создается пустой словарь. Примеры:

```
>>> d = dict(); d                  # Создаем пустой словарь
{}
>>> d = dict(a=1, b=2); d
{'a': 1, 'b': 2}
>>> d = dict({"a": 1, "b": 2}); d     # Словарь
{'a': 1, 'b': 2}
>>> d = dict([("a", 1), ("b", 2)]); d # Список кортежей
{'a': 1, 'b': 2}
>>> d = dict([["a", 1], ["b", 2]]); d # Список списков
{'a': 1, 'b': 2}
```

Объединить два списка в список кортежей позволяет функция `zip()`:

```
>>> k = ["a", "b"]          # Список с ключами
>>> v = [1, 2]              # Список со значениями
>>> list(zip(k, v))         # Создание списка кортежей
[('a', 1), ('b', 2)]
>>> d = dict(zip(k, v)); d # Создание словаря
{'a': 1, 'b': 2}
```

- указав все элементы словаря внутри фигурных скобок. Это наиболее часто используемый способ создания словаря. Между ключом и значением указывается двоеточие, а пары "ключ/значение" перечисляются через запятую. Пример:

```
>>> d = {}; d               # Создание пустого словаря
{}
>>> d = { "a": 1, "b": 2 }; d
{'a': 1, 'b': 2}
```

- заполнив словарь поэлементно. В этом случае ключ указывается внутри квадратных скобок:

```
>>> d = {}             # Создаем пустой словарь
>>> d["a"] = 1         # Добавляем элемент1 (ключ "a")
>>> d["b"] = 2         # Добавляем элемент2 (ключ "b")
>>> d
{'a': 1, 'b': 2}
```

- с помощью метода `dict.fromkeys(<Последовательность>[, <Значение>])`. Метод создает новый словарь, ключами которого будут элементы последовательности. Если второй параметр не указан, то значением элементов словаря будет значение `None`. Пример:

```
>>> d = dict.fromkeys(["a", "b", "c"])
>>> d
{'a': None, 'c': None, 'b': None}
>>> d = dict.fromkeys(["a", "b", "c"], 0) # Указан список
>>> d
{'a': 0, 'c': 0, 'b': 0}
>>> d = dict.fromkeys(("a", "b", "c"), 0) # Указан кортеж
>>> d
{'a': 0, 'c': 0, 'b': 0}
```

При создании словаря в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для списков и словарей этого делать нельзя. Рассмотрим пример:

```
>>> d1 = d2 = { "a": 1, "b": 2 } # Якобы создали два объекта
>>> d2["b"] = 10
>>> d1, d2                  # Изменилось значение в двух переменных !!!
({'a': 1, 'b': 10}, {'a': 1, 'b': 10})
```

Как видно из примера, изменение значения в переменной `d2` привело также к изменению значения в переменной `d1`. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 1, "b": 2 }
>>> d2["b"] = 10
>>> d1, d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Создать поверхностную копию словаря позволяет функция `dict()` (листинг 9.1).

Листинг 9.1. Создание поверхностной копии словаря с помощью функции `dict()`

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = dict(d1)           # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Кроме того, для создания поверхностной копии можно воспользоваться методом `copy()` (листинг 9.2).

Листинг 9.2. Создание поверхностной копии словаря с помощью метода `copy()`

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = d1.copy()          # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy` (листинг 9.3).

Листинг 9.3. Создание полной копии словаря

```
>>> d1 = { "a": 1, "b": [20, 30, 40] }
>>> d2 = dict(d1)          # Создаем поверхностную копию
>>> d2["b"][0] = "test"
>>> d1, d2                 # Изменились значения в двух переменных!!!
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 30, 40]})
>>> import copy
>>> d3 = copy.deepcopy(d1) # Создаем полную копию
>>> d3["b"][1] = 800
>>> d1, d3                 # Изменилось значение только в переменной d3
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 800, 40]})
```

9.2. Операции над словарями

Обращение к элементам словаря осуществляется с помощью квадратных скобок, в которых указывается ключ. В качестве ключа можно указать неизменяемый объект, например число, строку или кортеж.

Выведем все элементы словаря:

```
>>> d = { 1: "int", "a": "str", (1, 2): "tuple" }
>>> d[1], d["a"], d[(1, 2)]
('int', 'str', 'tuple')
```

Если элемент, соответствующий указанному ключу, отсутствует в словаре, то возбуждается исключение `KeyError`:

```
>>> d = { "a": 1, "b": 2 }
>>> d["c"]              # Обращение к несуществующему элементу
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    d["c"]             # Обращение к несуществующему элементу
KeyError: 'c'
```

Проверить существование ключа можно с помощью оператора `in`. Если ключ найден, то возвращается значение `True`, в противном случае — `False`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d           # Ключ существует
True
>>> "c" in d           # Ключ не существует
False
```

Метод `get(<Ключ>[, <Значение по умолчанию>])` позволяет избежать вывода сообщения об ошибке при отсутствии указанного ключа. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение `None` или значение, указанное во втором параметре. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

Кроме того, можно воспользоваться методом `setdefault(<Ключ>[, <Значение по умолчанию>])`. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый элемент, со значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

Так как словари относятся к изменяемым типам данных, мы можем изменить элемент по ключу. Если элемент с указанным ключом отсутствует в словаре, то он будет добавлен в словарь:

```
>>> d = { "a": 1, "b": 2 }
>>> d["a"] = 800              # Изменение элемента по ключу
>>> d["c"] = "string"         # Будет добавлен новый элемент
>>> d
{'a': 800, 'c': 'string', 'b': 2}
```

Получить количество ключей в словаре позволяет функция `len()`:

```
>>> d = { "a": 1, "b": 2 }
>>> len(d)          # Получаем количество ключей в словаре
2
```

Удалить элемент из словаря можно с помощью оператора `del`:

```
>>> d = { "a": 1, "b": 2 }
>>> del d["b"]; d   # Удаляем элемент с ключом "b" и выводим словарь
{'a': 1}
```

9.3. Перебор элементов словаря

Перебрать все элементы словаря можно с помощью цикла `for`, хотя словари и не являются последовательностями. В качестве примера выведем элементы словаря двумя способами. Первый способ использует метод `keys()`, возвращающий объект с ключами словаря. Во втором случае мы просто указываем словарь в качестве параметра. На каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу (листинг 9.4).

Листинг 9.4. Перебор элементов словаря

```
d = {"x": 1, "y": 2, "z": 3}
for key in d.keys():          # Использование метода keys()
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (y => 2) (x => 1) (z => 3)
print()                       # Вставляем символ перевода строки
for key in d:                 # Словари также поддерживают итерации
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (y => 2) (x => 1) (z => 3)
```

Так как словари являются неупорядоченными структурами, элементы словаря выводятся в произвольном порядке. Чтобы вывести элементы с сортировкой по ключам, следует получить список ключей, а затем воспользоваться методом `sort()`. Пример:

```
d = {"x": 1, "y": 2, "z": 3}
k = list(d.keys())            # Получаем список ключей
k.sort()                      # Сортируем список ключей
for key in k:
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Для сортировки ключей вместо метода `sort()` можно воспользоваться функцией `sorted()`. Пример:

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d.keys()):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Так как на каждой итерации возвращается ключ словаря, можно функции `sorted()` сразу передать объект словаря, а не результат выполнения метода `keys()`:

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

9.4. Методы для работы со словарями

Для работы со словарями предназначены следующие методы:

- `keys()` — возвращает объект `dict_keys`, содержащий все ключи словаря. Этот объект поддерживает итерации, а также операции над множествами. Пример:

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 3, "c": 4, "d": 5 }
>>> d1.keys(), d2.keys() # Получаем объект dict_keys
(dict_keys(['a', 'b']), dict_keys(['a', 'c', 'd']))
>>> list(d1.keys()), list(d2.keys()) # Получаем список ключей
(['a', 'b'], ['a', 'c', 'd'])
>>> for k in d1.keys(): print(k, end=" ")

a b
>>> d1.keys() | d2.keys()   # Объединение
{'a', 'c', 'b', 'd'}
>>> d1.keys() - d2.keys()   # Разница
{'b'}
>>> d2.keys() - d1.keys()   # Разница
{'c', 'd'}
>>> d1.keys() & d2.keys()   # Одинаковые ключи
{'a'}
>>> d1.keys() ^ d2.keys()   # Уникальные ключи
{'c', 'b', 'd'}
```

- `values()` — возвращает объект `dict_values`, содержащий все значения словаря. Этот объект поддерживает итерации. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.values()                  # Получаем объект dict_values
dict_values([1, 2])
>>> list(d.values())            # Получаем список значений
[1, 2]
>>> [ v for v in d.values() ]
[1, 2]
```

- `items()` — возвращает объект `dict_items`, содержащий все ключи и значения в виде кортежей. Этот объект поддерживает итерации. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.items()                   # Получаем объект dict_items
dict_items([('a', 1), ('b', 2)])
>>> list(d.items())             # Получаем список кортежей
[('a', 1), ('b', 2)]
```

- `<Ключ> in <Словарь>` — проверяет существование указанного ключа в словаре. Если ключ найден, то возвращается значение `True`, в противном случае — `False`.

Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d                    # Ключ существует
True
>>> "c" in d                    # Ключ не существует
False
```

- `get(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение `None` или значение, указанное во втором параметре. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

- `setdefault(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый элемент со значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"),d.setdefault("c"),d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

- `pop(<Ключ>[, <Значение по умолчанию>])` — удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение из второго параметра. Если ключ отсутствует и второй параметр не указан, то возбуждается исключение `KeyError`. Пример:

```
>>> d = { "a": 1, "b": 2, "c": 3 }
>>> d.pop("a"), d.pop("n", 0)
(1, 0)
>>> d.pop("n") # Ключ отсутствует и нет второго параметра
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    d.pop("n") # Ключ отсутствует и нет второго параметра
KeyError: 'n'
>>> d
{'c': 3, 'b': 2}
```

- `popitem()` — удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если словарь пустой, возбуждается исключение `KeyError`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.popitem() # Удаляем произвольный элемент
('a', 1)
>>> d.popitem() # Удаляем произвольный элемент
('b', 2)
>>> d.popitem() # Словарь пустой. Возбуждается исключение
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    d.popitem() # Словарь пустой. Возбуждается исключение
KeyError: 'popitem(): dictionary is empty'
```

- `clear()` — удаляет все элементы словаря. Метод ничего не возвращает в качестве значения. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.clear()               # Удаляем все элементы
>>> d                       # Словарь теперь пустой
{}
```

- `update()` — добавляет элементы в словарь. Метод изменяет текущий словарь и ничего не возвращает. Форматы метода:

```
update(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
update(<Словарь>)
update(<Список кортежей с двумя элементами>)
update(<Список списков с двумя элементами>)
```

Если элемент с указанным ключом уже присутствует в словаре, то его значение будет перезаписано. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.update(c=3, d=4)
>>> d
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> d.update({"c": 10, "d": 20})          # Словарь
>>> d # Значения элементов перезаписаны
{'a': 1, 'c': 10, 'b': 2, 'd': 20}
>>> d.update([("d", 80), ("e", 6)])       # Список кортежей
>>> d
{'a': 1, 'c': 10, 'b': 2, 'e': 6, 'd': 80}
>>> d.update([["a", "str"], ["i", "t"]])  # Список списков
>>> d
{'a': 'str', 'c': 10, 'b': 2, 'e': 6, 'd': 80, 'i': 't'}
```

- `copy()` — создает поверхностную копию словаря:

```
>>> d1 = { "a": 1, "b": [10, 20] }
>>> d2 = d1.copy() # Создаем поверхностную копию
>>> d1 is d2       # Это разные объекты
False
>>> d2["a"] = 800  # Изменяем значение
>>> d1, d2         # Изменилось значение только в d2
({'a': 1, 'b': [10, 20]}, {'a': 800, 'b': [10, 20]})
>>> d2["b"][0] = "new" # Изменяем значение вложенного списка
>>> d1, d2         # Изменились значения и в d1, и в d2!!!
({'a': 1, 'b': ['new', 20]}, {'a': 800, 'b': ['new', 20]})
```

Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy`.

9.5. Генераторы словарей

Помимо генераторов списков язык Python 3 поддерживает генераторы словарей. Синтаксис генераторов словарей похож на синтаксис генераторов списков, но имеет два отличия:

- выражение заключается в фигурные скобки, а не в квадратные;
- внутри выражения перед циклом `for` указываются два значения через двоеточие, а не одно. Значение, расположенное слева от двоеточия, становится ключом, а значение, расположенное справа от двоеточия, — значением элемента.

Пример:

```
>>> keys = ["a", "b"]          # Список с ключами
>>> values = [1, 2]            # Список со значениями
>>> {k: v for (k, v) in zip(keys, values)}
{'a': 1, 'b': 2}
>>> {k: 0 for k in keys}
{'a': 0, 'b': 0}
```

Генераторы словарей могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. Создадим новый словарь, содержащий только элементы с четными значениями, из исходного словаря:

```
>>> d = { "a": 1, "b": 2, "c": 3, "d": 4 }
>>> {k: v for (k, v) in d.items() if v % 2 == 0}
{'b': 2, 'd': 4}
```

9.6. Множества

Множество — это неупорядоченная коллекция уникальных элементов, с которой можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. Обратите внимание на то, что множество может содержать только элементы неизменяемых типов, например числа, строки, кортежи. Объявить множество можно с помощью функции `set()`:

```
>>> s = set()
>>> s
set()
```

Функция `set()` позволяет также преобразовать элементы последовательности во множество:

```
>>> set("string")                    # Преобразуем строку
{'g', 'i', 'n', 's', 'r', 't'}
>>> set([1, 2, 3, 4, 5])             # Преобразуем список
{1, 2, 3, 4, 5}
>>> set((1, 2, 3, 4, 5))             # Преобразуем кортеж
{1, 2, 3, 4, 5}
>>> set([1, 2, 3, 1, 2, 3]) # Остаются только уникальные элементы
{1, 2, 3}
```

В Python 3 можно также создать множество, указав элементы внутри фигурных скобок. Обратите внимание на то, что при указании пустых фигурных скобок будет создан словарь, а не множество. Чтобы создать пустое множество, следует использовать функцию `set()`. Пример:

```
>>> {1, 2, 3, 1, 2, 3}
{1, 2, 3}
>>> x, y = {}, set()
```

```
>>> type(x)                          # Создан словарь!!!
<class 'dict'>
>>> type(y)                          # Создано множество
<class 'set'>
```

Перебрать элементы множества можно с помощью цикла `for`:

```
>>> for i in set([1, 2, 3]): print(i, end=" ")

1 2 3
```

Получить количество элементов множества позволяет функция `len()`:

```
>>> len(set([1, 2, 3]))
3
```

Для работы с множествами предназначены следующие операторы и соответствующие им методы:

- `|` и `union()` — объединение множеств:

  ```
  >>> s = {1, 2, 3}
  >>> s.union(set([4, 5, 6])), s | set([4, 5, 6])
  ({1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6})
  ```

  Если элемент уже содержится во множестве, то он повторно добавлен не будет:

  ```
  >>> set([1, 2, 3]) | set([1, 2, 3])
  {1, 2, 3}
  ```

- `a |= b` и `a.update(b)` — добавляют элементы множества `b` во множество `a`:

  ```
  >>> s = {1, 2, 3}
  >>> s.update(set([4, 5, 6]))
  >>> s
  {1, 2, 3, 4, 5, 6}
  >>> s |= set([7, 8, 9])
  >>> s
  {1, 2, 3, 4, 5, 6, 7, 8, 9}
  ```

- `-` и `difference()` — разница множеств:

  ```
  >>> set([1, 2, 3]) - set([1, 2, 4])
  {3}
  >>> {1, 2, 3}.difference(set([1, 2, 4]))
  {3}
  ```

- `a -= b` и `a.difference_update(b)` — удаляют элементы из множества `a`, которые существуют и во множестве `a`, и во множестве `b`:

  ```
  >>> s = {1, 2, 3}
  >>> s.difference_update(set([1, 2, 4]))
  >>> s
  {3}
  >>> s -= set([3, 4, 5])
  >>> s
  set()
  ```

- `&` и `intersection()` — пересечение множеств. Позволяет получить элементы, которые существуют в обоих множествах:

```
>>> set([1, 2, 3]) & set([1, 2, 4])
{1, 2}
>>> s = {1, 2, 3}
>>> s.intersection(set([1, 2, 4]))
{1, 2}
```

- `a &= b` и `a.intersection_update(b)` — во множестве `a` останутся элементы, которые существуют и во множестве `a`, и во множестве `b`:

```
>>> s = {1, 2, 3}
>>> s.intersection_update(set([1, 2, 4]))
>>> s
{1, 2}
>>> s &= {1, 6, 7}
>>> s
{1}
```

- `^` и `symmetric_difference()` — возвращают все элементы обоих множеств, исключая одинаковые элементы:

```
>>> s = {1, 2, 3}
>>> s ^ set([1, 2, 4]), s.symmetric_difference(set([1, 2, 4]))
({3, 4}, {3, 4})
>>> s ^ set([1, 2, 3]), s.symmetric_difference(set([1, 2, 3]))
(set(), set())
>>> s ^ set([4, 5, 6]), s.symmetric_difference(set([4, 5, 6]))
({1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6})
```

- `a ^= b` и `a.symmetric_difference_update(b)` — во множестве `a` будут все элементы обоих множеств, исключая одинаковые элементы:

```
>>> s = {1, 2, 3}
>>> s.symmetric_difference_update(set([1, 2, 4]))
>>> s
{3, 4}
>>> s ^= {3, 5, 6}
>>> s
{4, 5, 6}
```

Операторы сравнения множеств:

- `in` — проверка наличия элемента во множестве:

```
>>> s = {1, 2, 3, 4, 5}
>>> 1 in s, 12 in s, 12 not in s
(True, False, True)
```

- `==` — проверка на равенство:

```
>>> set([1, 2, 3]) == set([1, 2, 3])
True
>>> {1, 2, 3} == {3, 2, 1}
True
```

```
>>> set([1, 2, 3]) == set([1, 2, 3, 4])
False
```

- `a <= b` и `a.issubset(b)` — проверяют, входят ли все элементы множества `a` во множество `b`:

```
>>> s = {1, 2, 3}
>>> s <= set([1, 2]), s <= set([1, 2, 3, 4])
(False, True)
>>> s.issubset(set([1, 2])), s.issubset({1, 2, 3, 4})
(False, True)
```

- `a < b` — проверяет, входят ли все элементы множества `a` во множество `b`. Причем множество `a` не должно быть равно множеству `b`:

```
>>> s = {1, 2, 3}
>>> s < set([1, 2, 3]), s < set([1, 2, 3, 4])
(False, True)
```

- `a >= b` и `a.issuperset(b)` — проверяют, входят ли все элементы множества `b` во множество `a`:

```
>>> s = {1, 2, 3}
>>> s >= set([1, 2]), s >= set([1, 2, 3, 4])
(True, False)
>>> s.issuperset(set([1, 2])), s.issuperset({1, 2, 3, 4})
(True, False)
```

- `a > b` — проверяет, входят ли все элементы множества `b` во множество `a`. Причем множество `a` не должно быть равно множеству `b`:

```
>>> s = {1, 2, 3}
>>> s > set([1, 2]), s > set([1, 2, 3])
(True, False)
```

- `a.isdisjoint(b)` — возвращает `True`, если результатом пересечения множеств `a` и `b` является пустое множество (это означает, что множества не имеют одинаковых элементов):

```
>>> s = {1, 2, 3}
>>> s.isdisjoint(set([4, 5, 6]))
True
>>> s.isdisjoint(set([4, 1, 6]))
False
```

Для работы с множествами предназначены следующие методы:

- `copy()` — создает копию множества. Обратите внимание на то, что оператор = присваивает лишь ссылку на тот же объект, а не копирует его. Пример:

```
>>> s = {1, 2, 3}
>>> c = s; s is c # С помощью = копию создать нельзя!
True
>>> c = s.copy()  # Создаем копию объекта
>>> c
{1, 2, 3}
>>> s is c        # Теперь это разные объекты
False
```

- `add(<Элемент>)` — добавляет `<Элемент>` во множество:

```
>>> s = {1, 2, 3}
>>> s.add(4); s
{1, 2, 3, 4}
```

- `remove(<Элемент>)` — удаляет `<Элемент>` из множества. Если элемент не найден, то возбуждается исключение `KeyError`:

```
>>> s = {1, 2, 3}
>>> s.remove(3); s        # Элемент существует
{1, 2}
>>> s.remove(5)           # Элемент НЕ существует
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    s.remove(5)           # Элемент НЕ существует
KeyError: 5
```

- `discard(<Элемент>)` — удаляет `<Элемент>` из множества, если он присутствует:

```
>>> s = {1, 2, 3}
>>> s.discard(3); s       # Элемент существует
{1, 2}
>>> s.discard(5); s       # Элемент НЕ существует
{1, 2}
```

- `pop()` — удаляет произвольный элемент из множества и возвращает его. Если элементов нет, то возбуждается исключение `KeyError`:

```
>>> s = {1, 2}
>>> s.pop(), s
(1, {2})
>>> s.pop(), s
(2, set())
>>> s.pop() # Если нет элементов, то будет ошибка
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    s.pop() # Если нет элементов, то будет ошибка
KeyError: 'pop from an empty set'
```

- `clear()` — удаляет все элементы из множества:

```
>>> s = {1, 2, 3}
>>> s.clear(); s
set()
```

В языке Python существует еще один тип множеств — `frozenset`. В отличие от типа `set`, множество типа `frozenset` нельзя изменить. Объявить множество можно с помощью функции `frozenset()`:

```
>>> f = frozenset()
>>> f
frozenset()
```

Функция `frozenset()` позволяет также преобразовать элементы последовательности во множество:

```
>>> frozenset("string")                    # Преобразуем строку
frozenset({'g', 'i', 'n', 's', 'r', 't'})
>>> frozenset([1, 2, 3, 4, 4])             # Преобразуем список
frozenset({1, 2, 3, 4})
>>> frozenset((1, 2, 3, 4, 4))             # Преобразуем кортеж
frozenset({1, 2, 3, 4})
```

Множества `frozenset` поддерживают операторы, которые не изменяют само множество, а также следующие методы: `copy()`, `difference()`, `isdisjoint()`, `intersection()`, `issubset()`, `issuperset()`, `symmetric_difference()` и `union()`.

9.7. Генераторы множеств

Помимо генераторов списков и генераторов словарей язык Python 3 поддерживает генераторы множеств. Синтаксис генераторов множеств похож на синтаксис генераторов списков, но выражение заключается в фигурные скобки, а не в квадратные. Так как результатом является множество, все повторяющиеся элементы будут удалены. Пример:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3]}
{1, 2, 3}
```

Генераторы множеств могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. Создадим множество, содержащее только уникальные элементы с четными значениями, из элементов исходного списка:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3] if x % 2 == 0}
{2}
```

ГЛАВА 10

Работа с датой и временем

Для работы с датой и временем в языке Python предназначены следующие модули:

- `time` — позволяет получить текущую дату и время, а также произвести форматированный вывод;
- `datetime` — позволяет манипулировать датой и временем. Например, производить арифметические операции, сравнивать даты, выводить дату и время в различных форматах и др.;
- `calendar` — позволяет вывести календарь в виде простого текста или в HTML-формате;
- `timeit` — позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы.

10.1. Получение текущей даты и времени

Получить текущую дату и время позволяют следующие функции из модуля `time`:

- `time()` — возвращает вещественное число, представляющее количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.):

```
>>> import time              # Подключаем модуль time
>>> time.time()              # Получаем количество секунд
1303520699.093
```

- `gmtime([<Количество секунд>])` — возвращает объект `struct_time`, представляющий универсальное время (UTC). Если параметр не указан, то возвращается текущее время. Если параметр указан, то время будет не текущим, а соответствующим количеству секунд, прошедших с начала эпохи. Пример:

```
>>> time.gmtime(0)                  # Начало эпохи
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
>>> time.gmtime()                   # Текущая дата и время
time.struct_time(tm_year=2011, tm_mon=4, tm_mday=23, tm_hour=1,
tm_min=8, tm_sec=9, tm_wday=5, tm_yday=113, tm_isdst=0)
>>> time.gmtime(1283307823.0)       # Дата 01-09-2010
time.struct_time(tm_year=2010, tm_mon=9, tm_mday=1, tm_hour=2,
tm_min=23, tm_sec=43, tm_wday=2, tm_yday=244, tm_isdst=0)
```

Получить значение конкретного атрибута можно, указав его название или индекс внутри объекта:

```
>>> d = time.gmtime()
>>> d.tm_year, d[0]
(2011, 2011)
>>> tuple(d)              # Преобразование в кортеж
(2011, 4, 23, 1, 10, 24, 5, 113, 0)
```

- `localtime([<Количество секунд>])` — возвращает объект `struct_time`, представляющий локальное время. Если параметр не указан, то возвращается текущее время. Если параметр указан, то время будет не текущим, а соответствующим количеству секунд, прошедших с начала эпохи. Пример:

```
>>> time.localtime()              # Текущая дата и время
time.struct_time(tm_year=2011, tm_mon=4, tm_mday=23, tm_hour=5,
tm_min=12, tm_sec=55, tm_wday=5, tm_yday=113, tm_isdst=1)
>>> time.localtime(1283307823.0) # Дата 01-09-2010
time.struct_time(tm_year=2010, tm_mon=9, tm_mday=1, tm_hour=6,
tm_min=23, tm_sec=43, tm_wday=2, tm_yday=244, tm_isdst=1)
```

- `mktime(<Объект struct_time>)` — возвращает вещественное число, представляющее количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` или кортеж из девяти элементов. Если указанная дата некорректна, возбуждается исключение `OverflowError`. Пример:

```
>>> d = time.localtime(1303520699.0)
>>> time.mktime(d)
1303520699.0
>>> tuple(time.localtime(1303520699.0))
(2011, 4, 23, 5, 4, 59, 5, 113, 1)
>>> time.mktime((2011, 4, 23, 5, 4, 59, 5, 113, 1))
1303520699.0
>>> time.mktime((1940, 0, 31, 5, 23, 43, 5, 31, 0))
... Фрагмент опущен ...
OverflowError: mktime argument out of range
```

Объект `struct_time`, возвращаемый функциями `gmtime()` и `localtime()`, содержит следующие атрибуты:

- `tm_year` — `0` — год;
- `tm_mon` — `1` — месяц (число от `1` до `12`);
- `tm_mday` — `2` — день месяца (число от `1` до `31`);
- `tm_hour` — `3` — час (число от `0` до `23`);
- `tm_min` — `4` — минуты (число от `0` до `59`);
- `tm_sec` — `5` — секунды (число от `0` до `59`, изредка до `61`);
- `tm_wday` — `6` — день недели (число от `0` (для понедельника) до `6` (для воскресенья));
- `tm_yday` — `7` — количество дней, прошедшее с начала года (число от `1` до `366`);
- `tm_isdst` — `8` — флаг коррекции летнего времени (значения `0`, `1` или `-1`).

Выведем текущую дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 10.1).

Листинг 10.1. Вывод текущей даты и времени

```
# -*- coding: utf-8 -*-
import time # Подключаем модуль time
d = [ "понедельник", "вторник", "среда", "четверг",
      "пятница", "суббота", "воскресенье" ]
m = [ "", "января", "февраля", "марта", "апреля", "мая",
      "июня", "июля", "августа", "сентября", "октября",
      "ноября", "декабря" ]
t = time.localtime() # Получаем текущее время
print( "Сегодня:\n%s %s %s %s %02d:%02d:%02d\n%02d.%02d.%02d" %
     ( d[t[6]], t[2], m[t[1]], t[0], t[3], t[4], t[5],
       t[2], t[1], t[0] ) )
input()
```

Примерный результат выполнения:

```
Сегодня:
среда 24 августа 2011 17:07:29
24.08.2011
```

10.2. Форматирование даты и времени

Получить форматированный вывод даты и времени позволяют следующие функции из модуля `time`:

- `strftime(<Строка формата>[, <Объект struct_time>])` — возвращает строковое представление даты в соответствии со строкой формата. Если второй параметр не указан, будет использоваться текущая дата и время. Если во втором параметре указан объект `struct_time` или кортеж из девяти элементов, то дата будет соответствовать указанному значению. Функция зависит от настройки локали. Пример:

```
>>> import time
>>> time.strftime("%d.%m.%Y") # Форматирование даты
'24.08.2011'
>>> time.strftime("%H:%M:%S") # Форматирование времени
'05:26:51'
>>> time.strftime("%d.%m.%Y", time.localtime(1283307823.0))
'01.09.2010'
```

- `strptime(<Строка с датой>[, <Строка формата>])` — разбирает строку, указанную в первом параметре, в соответствии со строкой формата. Возвращает объект `struct_time`. Если строка не соответствует формату, возбуждается исключение `ValueError`. Если строка формата не указана, используется строка `"%a %b %d %H:%M:%S %Y"`. Пример:

```
>>> time.strptime("Sat Apr 23 05:31:50 2011")
time.struct_time(tm_year=2011, tm_mon=4, tm_mday=23, tm_hour=5,
tm_min=31, tm_sec=50, tm_wday=5, tm_yday=113, tm_isdst=-1)
>>> time.strptime("23.04.2011", "%d.%m.%Y")
time.struct_time(tm_year=2011, tm_mon=4, tm_mday=23, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=5, tm_yday=113, tm_isdst=-1)
```

```
>>> time.strptime("23-04-2011", "%d.%m.%Y")
... Фрагмент опущен ...
ValueError: time data '23-04-2011' does not match format '%d.%m.%Y'
```

- `asctime([<Объект struct_time>])` — возвращает строку специального формата. Если параметр не указан, будет использоваться текущая дата и время. Если в параметре указан объект `struct_time` или кортеж из девяти элементов, то дата будет соответствовать указанному значению. Пример:

```
>>> time.asctime()             # Текущая дата
'Wed Aug 24 17:11:37 2011'
>>> time.asctime(time.localtime(1283307823.0))
'Wed Sep  1 06:23:43 2010'
```

- `ctime([<Количество секунд>])` — функция аналогична `asctime()`, но в качестве параметра принимает не объект `struct_time`, а количество секунд, прошедших с начала эпохи. Пример:

```
>>> time.ctime()                    # Текущая дата
'Wed Aug 24 17:12:30 2011'
>>> time.ctime(1283307823.0)        # Дата в прошлом
'Wed Sep  1 06:23:43 2010'
```

В параметре `<Строка формата>` в функциях `strftime()` и `strptime()` могут быть использованы следующие комбинации специальных символов:

- `%y` — год из двух цифр (от `"00"` до `"99"`);
- `%Y` — год из четырех цифр (например, `"2011"`);
- `%m` — номер месяца с предваряющим нулем (от `"01"` до `"12"`);
- `%b` — аббревиатура месяца в зависимости от настроек локали (например, `"янв"` для января);
- `%B` — название месяца в зависимости от настроек локали (например, `"Январь"`);
- `%d` — номер дня в месяце с предваряющим нулем (от `"01"` до `"31"`);
- `%j` — день с начала года (от `"001"` до `"366"`);
- `%U` — номер недели в году (от `"00"` до `"53"`). Неделя начинается с воскресенья. Все дни с начала года до первого воскресенья относятся к неделе с номером `0`;
- `%W` — номер недели в году (от `"00"` до `"53"`). Неделя начинается с понедельника. Все дни с начала года до первого понедельника относятся к неделе с номером `0`;
- `%w` — номер дня недели (`"0"` — для воскресенья, `"6"` — для субботы);
- `%a` — аббревиатура дня недели в зависимости от настроек локали (например, `"Пн"` для понедельника);
- `%A` — название дня недели в зависимости от настроек локали (например, `"понедельник"`);
- `%H` — часы в 24-часовом формате (от `"00"` до `"23"`);
- `%I` — часы в 12-часовом формате (от `"01"` до `"12"`);
- `%M` — минуты (от `"00"` до `"59"`);
- `%S` — секунды (от `"00"` до `"59"`, изредка до `"61"`);
- `%p` — эквивалент значениям `AM` и `PM` в текущей локали;
- `%c` — представление даты и времени в текущей локали;

- `%x` — представление даты в текущей локали;
- `%X` — представление времени в текущей локали. Пример:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> print(time.strftime("%x"))   # Представление даты
24.08.2011
>>> print(time.strftime("%X"))   # Представление времени
5:46:20
>>> print(time.strftime("%c"))   # Дата и время
24.08.2011 5:46:33
```

- `%Z` — название часового пояса или пустая строка (например, "`Московское время (лето)`");
- `%%` — символ "`%`".

В качестве примера выведем текущую дату и время с помощью функции `strftime()` (листинг 10.2).

Листинг 10.2. Форматирование даты и времени

```
# -*- coding: utf-8 -*-
import time
import locale
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
s = "Сегодня:\n%A %d %b %Y %H:%M:%S\n%d.%m.%Y"
print(time.strftime(s))
input()
```

Примерный результат выполнения:

```
Сегодня:
среда 24 авг 2011 17:15:00
24.08.2011
```

10.3. "Засыпание" скрипта

Функция `sleep()` из модуля `time` прерывает выполнение скрипта на указанное время. По истечении срока скрипт продолжит работу. Формат функции:

```
sleep(<Время в секундах>)
```

В качестве параметра можно указать целое или вещественное число. Прежде чем использовать функцию, необходимо подключить модуль `time` с помощью инструкции `import time`:

```
>>> import time                # Подключаем модуль time
>>> time.sleep(5)              # "Засыпаем" на 5 секунд
```

или инструкции `from time import sleep`:

```
>>> from time import sleep     # Подключаем модуль time
>>> sleep(2.5)                 # "Засыпаем" на две с половиной секунды
```

10.4. Модуль *datetime*. Манипуляции датой и временем

Модуль `datetime` позволяет манипулировать датой и временем. Например, выполнять арифметические операции, сравнивать даты, выводить дату и время в различных форматах и др. Прежде чем использовать классы из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import datetime
```

Модуль содержит пять классов:

- `timedelta` — дата в виде количества дней, секунд и микросекунд. Экземпляр этого класса можно складывать с экземплярами классов `date` и `datetime`. Кроме того, результат вычитания двух дат будет экземпляром класса `timedelta`;
- `date` — представление даты в виде объекта;
- `time` — представление времени в виде объекта;
- `datetime` — представление комбинации даты и времени в виде объекта;
- `tzinfo` — абстрактный класс, отвечающий за зону времени. За подробной информацией по этому классу обращайтесь к документации по модулю `datetime`.

10.4.1. Класс *timedelta*

Класс `timedelta` из модуля `datetime` позволяет выполнять операции над датами, например, складывать, вычитать, сравнивать и др. Конструктор класса имеет следующий формат:

```
timedelta([days][, seconds][, microseconds][, milliseconds][, minutes]
          [, hours][, weeks])
```

Все параметры являются необязательными и по умолчанию имеют значение `0`. Первые три параметра являются основными:

- `days` — дни (диапазон `-999999999 <= days <= 999999999`);
- `seconds` — секунды (диапазон `0 <= seconds < 3600*24`);
- `microseconds` — микросекунды (диапазон `0 <= microseconds < 1000000`).

Все остальные параметры автоматически преобразуются в следующие значения:

- `milliseconds` — миллисекунды (одна миллисекунда преобразуется в `1000` микросекунд):

```
>>> import datetime
>>> datetime.timedelta(milliseconds=1)
datetime.timedelta(0, 0, 1000)
```

- `minutes` — минуты (одна минута преобразуется в `60` секунд):

```
>>> datetime.timedelta(minutes=1)
datetime.timedelta(0, 60)
```

- `hours` — часы (один час преобразуется в `3600` секунд):

```
>>> datetime.timedelta(hours=1)
datetime.timedelta(0, 3600)
```

- `weeks` — недели (одна неделя преобразуется в `7` дней):

```
>>> datetime.timedelta(weeks=1)
datetime.timedelta(7)
```

Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера укажем один час:

```
>>> datetime.timedelta(0, 0, 0, 0, 0, 1)
datetime.timedelta(0, 3600)
>>> datetime.timedelta(hours=1)
datetime.timedelta(0, 3600)
```

Получить результат можно с помощью следующих атрибутов:

- `days` — дни;
- `seconds` — секунды;
- `microseconds` — микросекунды.

Пример:

```
>>> d = datetime.timedelta(hours=1, days=2, milliseconds=1)
>>> d
datetime.timedelta(2, 3600, 1000)
>>> d.days, d.seconds, d.microseconds
(2, 3600, 1000)
>>> repr(d), str(d)
('datetime.timedelta(2, 3600, 1000)', '2 days, 1:00:00.001000')
```

Получить результат в секундах позволяет метод `total_seconds()` (метод доступен начиная с версии Python 3.2):

```
>>> d = datetime.timedelta(minutes=1)
>>> d.total_seconds()
60.0
```

Над экземплярами класса `timedelta` можно производить арифметические операции `+`, `-`, `/`, `//`, `%` и `*`, использовать унарные операторы `+` и `-`, а также получать абсолютное значение с помощью функции `abs()`. Примеры:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d1 + d2, d2 - d1                          # Сложение и вычитание
(datetime.timedelta(9), datetime.timedelta(5))
>>> d2 / d1                                   # Деление
3.5
>>> d1 / 2, d2 / 2.5                          # Деление
(datetime.timedelta(1), datetime.timedelta(2, 69120))
>>> d2 // d1                                  # Деление
3
>>> d1 // 2, d2 // 2                          # Деление
(datetime.timedelta(1), datetime.timedelta(3, 43200))
>>> d2 % d1                                   # Остаток
datetime.timedelta(1)
>>> d1 * 2, d2 * 2                            # Умножение
(datetime.timedelta(4), datetime.timedelta(14))
```

```
>>> 2 * d1, 2 * d2                              # Умножение
(datetime.timedelta(4), datetime.timedelta(14))
>>> d3 = -d1
>>> d3, abs(d3)
(datetime.timedelta(-2), datetime.timedelta(2))
```

Кроме того, можно использовать операторы сравнения ==, !=, <, <=, > и >=. Пример:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d3 = datetime.timedelta(weeks=1)
>>> d1 == d2, d2 == d3                # Проверка на равенство
(False, True)
>>> d1 != d2, d2 != d3                # Проверка на неравенство
(True, False)
>>> d1 < d2, d2 <= d3                 # Меньше, меньше или равно
(True, True)
>>> d1 > d2, d2 >= d3                 # Больше, больше или равно
(False, True)
```

10.4.2. Класс *date*

Класс `date` из модуля `datetime` позволяет выполнять операции над датами. Конструктор класса имеет следующий формат:

```
date(<Год>, <Месяц>, <День>)
```

Все параметры являются обязательными. В параметрах можно указать следующий диапазон значений:

- `<Год>` — от `MINYEAR` до `MAXYEAR`. Выведем значения этих констант:

  ```
  >>> import datetime
  >>> datetime.MINYEAR, datetime.MAXYEAR
  (1, 9999)
  ```

- `<Месяц>` — от `1` до `12` включительно;
- `<День>` — от `1` до количества дней в месяце.

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> datetime.date(2011, 8, 24)
datetime.date(2011, 8, 24)
>>> datetime.date(2011, 13, 24)   # Неправильное значение для месяца
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.date(2011, 8, 24)
>>> repr(d), str(d)
('datetime.date(2011, 8, 24)', '2011-08-24')
```

Для создания экземпляра класса можно также воспользоваться следующими методами:

- `today()` — возвращает текущую дату:

  ```
  >>> datetime.date.today() # Получаем текущую дату
  datetime.date(2011, 8, 24)
  ```

- `fromtimestamp(<Количество секунд>)` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.date.fromtimestamp(time.time())  # Текущая дата
datetime.date(2011, 8, 24)
>>> datetime.date.fromtimestamp(1233368623.0) # Дата 31-01-2009
datetime.date(2009, 1, 31)
```

- `fromordinal(<Количество дней с 1 года>)` — возвращает дату, соответствующую количеству дней, прошедших с `1` года. В качестве параметра указывается число от `1` до `datetime.date.max.toordinal()`. Пример:

```
>>> datetime.date.max.toordinal()
3652059
>>> datetime.date.fromordinal(3652059)
datetime.date(9999, 12, 31)
>>> datetime.date.fromordinal(1)
datetime.date(1, 1, 1)
```

Получить результат можно с помощью следующих атрибутов:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от `1` до `12`);
- `day` — день (число от `1` до количества дней в месяце).

Пример:

```
>>> d = datetime.date.today() # Текущая дата (24-08-2011)
>>> d.year, d.month, d.day
(2011, 8, 24)
```

Над экземплярами класса `date` можно производить следующие операции:

- `date2 = date1 + timedelta` — прибавляет к дате указанный период в днях. Значения атрибутов `timedelta.seconds` и `timedelta.microseconds` игнорируются;
- `date2 = date1 – timedelta` — вычитает из даты указанный период в днях. Значения атрибутов `timedelta.seconds` и `timedelta.microseconds` игнорируются;
- `timedelta = date1 – date2` — возвращает разницу между датами (период в днях). Атрибуты `timedelta.seconds` и `timedelta.microseconds` будут иметь значение `0`;
- можно также сравнивать две даты с помощью операторов сравнения.

Примеры:

```
>>> d1 = datetime.date(2011, 1, 19)
>>> d2 = datetime.date(2011, 1, 1)
>>> t = datetime.timedelta(days=10)
>>> d1 + t, d1 – t             # Прибавляем и вычитаем 10 дней
(datetime.date(2011, 1, 29), datetime.date(2011, 1, 9))
>>> d1 – d2                    # Разница между датами
datetime.timedelta(18)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
(False, True, False, True)
>>> d1 == d2, d1 != d2
(False, True)
```

Экземпляры класса `date` поддерживают следующие методы:

- `replace([year][, month][, day])` — возвращает дату с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. Примеры:

```
>>> d = datetime.date(2011, 6, 5)
>>> d.replace(2012, 3) # Заменяем год и месяц
datetime.date(2012, 3, 5)
>>> d.replace(year=2009, month=3, day=1)
datetime.date(2009, 3, 1)
>>> d.replace(day=7)   # Заменяем только день
datetime.date(2011, 6, 7)
```

- `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно задавать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

```
>>> d = datetime.date(2011, 6, 5)
>>> d.strftime("%d.%m.%Y")
'05.06.2011'
```

- `isoformat()` — возвращает дату в формате `ГГГГ-ММ-ДД`:

```
>>> d = datetime.date(2011, 6, 5)
>>> d.isoformat()
'2011-06-05'
```

- `ctime()` — возвращает строку специального формата:

```
>>> d = datetime.date(2011, 6, 5)
>>> d.ctime()
'Sun Jun  5 00:00:00 2011'
```

- `timetuple()` — возвращает объект `struct_time` с датой и временем:

```
>>> d = datetime.date(2011, 6, 5)
>>> d.timetuple()
time.struct_time(tm_year=2011, tm_mon=6, tm_mday=5, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=6, tm_yday=156, tm_isdst=-1)
```

- `toordinal()` — возвращает количество дней, прошедших с 1 года:

```
>>> d = datetime.date(2011, 6, 5)
>>> d.toordinal()
734293
>>> datetime.date.fromordinal(734293)
datetime.date(2011, 6, 5)
```

- `weekday()` — возвращает порядковый номер дня в неделе (`0` — для понедельника, `6` — для воскресенья):

```
>>> d = datetime.date(2011, 6, 5)
>>> d.weekday() # 6 — это воскресенье
6
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (`1` — для понедельника, `7` — для воскресенья):

```
>>> d = datetime.date(2011, 6, 4)
>>> d.isoweekday() # 6 — это суббота
6
```

- `isocalendar()` — возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):

```
>>> d = datetime.date(2011, 6, 5)
>>> d.isocalendar()
(2011, 22, 7)
```

10.4.3. Класс *time*

Класс `time` из модуля `datetime` позволяет выполнять операции над временем. Конструктор класса имеет следующий формат:

```
time([hour][, minute][, second][, microsecond][, tzinfo])
```

Все параметры являются необязательными. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В параметрах можно указать следующий диапазон значений:

- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> import datetime
>>> datetime.time(23, 12, 38, 375000)
datetime.time(23, 12, 38, 375000)
>>> t = datetime.time(hour=23, second=38, minute=12)
>>> repr(t), str(t)
('datetime.time(23, 12, 38)', '23:12:38')
>>> datetime.time(25, 12, 38, 375000)
... Фрагмент опущен ...
ValueError: hour must be in 0..23
```

Получить результат можно с помощью следующих атрибутов:

- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.hour, t.minute, t.second, t.microsecond
(23, 12, 38, 375000)
```

Над экземплярами класса time нельзя выполнять арифметические операции. Можно только производить сравнения. Пример:

```
>>> t1 = datetime.time(23, 12, 38, 375000)
>>> t2 = datetime.time(12, 28, 17)
>>> t1 < t2, t1 > t2, t1 <= t2, t1 >= t2
(False, True, False, True)
>>> t1 == t2, t1 != t2
(False, True)
```

Экземпляры класса time поддерживают следующие методы:

- replace([hour][, minute][, second][, microsecond][, tzinfo]) — возвращает время с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.replace(10, 52)       # Заменяем часы и минуты
datetime.time(10, 52, 38, 375000)
>>> t.replace(second=21)    # Заменяем только секунды
datetime.time(23, 12, 21, 375000)
```

- isoformat() — возвращает время в формате ISO 8601:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.isoformat()
'23:12:38.375000'
```

- strftime(<Строка формата>) — возвращает отформатированную строку. В строке формата можно указывать комбинации специальных символов, которые используются в функции strftime() из модуля time. Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.strftime("%H:%M:%S")
'23:12:38'
```

> ***Примечание***
>
> Экземпляры класса time поддерживают также методы dst(), utcoffset() и tzname(). За подробной информацией по этим методам, а также по абстрактному классу tzinfo обращайтесь к документации по модулю datetime.

10.4.4. Класс *datetime*

Класс datetime из модуля datetime позволяет выполнять операции над комбинацией даты и времени. Конструктор класса имеет следующий формат:

```
datetime(<Год>, <Месяц>, <День>[, hour][, minute][, second]
        [, microsecond][, tzinfo])
```

Первые три параметра являются обязательными. Остальные значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В параметрах можно указать следующий диапазон значений:

- <Год> — число от MINYEAR (1) до MAXYEAR (9999);
- <Месяц> — число от 1 до 12 включительно;

- `<День>` — число от `1` до количества дней в месяце;
- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> import datetime
>>> datetime.datetime(2011, 6, 5)
datetime.datetime(2011, 6, 5, 0, 0)
>>> datetime.datetime(2011, 6, 5, hour=12, minute=55)
datetime.datetime(2011, 6, 5, 12, 55)
>>> datetime.datetime(2011, 32, 20)
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.datetime(2011, 6, 5, 5, 19, 21)
>>> repr(d), str(d)
('datetime.datetime(2011, 6, 5, 5, 19, 21)', '2011-06-05 05:19:21')
```

Для создания экземпляра класса можно также воспользоваться следующими методами:

- `today()` — возвращает текущую дату и время:

  ```
  >>> datetime.datetime.today()
  datetime.datetime(2011, 8, 24, 7, 51, 47, 468000)
  ```

- `now([<Зона>])` — возвращает текущую дату и время. Если параметр не задан, то метод аналогичен методу `today()`. Пример:

  ```
  >>> datetime.datetime.now()
  datetime.datetime(2011, 8, 24, 7, 52, 14, 765000)
  ```

- `utcnow()` — возвращает текущее универсальное время (UTC):

  ```
  >>> datetime.datetime.utcnow()
  datetime.datetime(2011, 8, 24, 3, 52, 39, 906000)
  ```

- `fromtimestamp(<Количество секунд>[, <Зона>])` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

  ```
  >>> import datetime, time
  >>> datetime.datetime.fromtimestamp(time.time())
  datetime.datetime(2011, 8, 24, 7, 53, 14, 734000)
  >>> datetime.datetime.fromtimestamp(1233368623.0)
  datetime.datetime(2009, 1, 31, 5, 23, 43)
  ```

- `utcfromtimestamp(<Количество секунд>)` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи, в универсальном времени (UTC). Пример:

  ```
  >>> datetime.datetime.utcfromtimestamp(time.time())
  datetime.datetime(2011, 8, 24, 3, 54, 2, 93000)
  >>> datetime.datetime.utcfromtimestamp(1233368623.0)
  datetime.datetime(2009, 1, 31, 2, 23, 43)
  ```

- `fromordinal(<Количество дней с 1 года>)` — возвращает дату, соответствующую количеству дней, прошедших с 1 года. В качестве параметра указывается число от `1` до `datetime.datetime.max.toordinal()`. Пример:

```
>>> datetime.datetime.max.toordinal()
3652059
>>> datetime.datetime.fromordinal(3652059)
datetime.datetime(9999, 12, 31, 0, 0)
>>> datetime.datetime.fromordinal(1)
datetime.datetime(1, 1, 1, 0, 0)
```

- `combine(<Экземпляр класса date>, <Экземпляр класса time>)` — создает экземпляр класса `datetime` в соответствии со значениями экземпляров классов `date` и `time`:

```
>>> d = datetime.date(2011, 6, 5)  # Экземпляр класса date
>>> t = datetime.time(9, 12, 35)   # Экземпляр класса time
>>> datetime.datetime.combine(d, t)
datetime.datetime(2011, 6, 5, 9, 12, 35)
```

- `strptime(<Строка с датой>, <Строка формата>)` — разбирает строку, указанную в первом параметре, в соответствии со строкой формата. Если строка не соответствует формату, возбуждается исключение `ValueError`. Пример:

```
>>> datetime.datetime.strptime("05.06.2011", "%d.%m.%Y")
datetime.datetime(2011, 6, 5, 0, 0)
>>> datetime.datetime.strptime("05.06.2011", "%d-%m-%Y")
... Фрагмент опущен ...
ValueError: time data '05.06.2011'
does not match format '%d-%m-%Y'
```

Получить результат можно с помощью следующих атрибутов:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от `1` до `12`);
- `day` — день (число от `1` до количества дней в месяце);
- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Пример:

```
>>> d = datetime.datetime(2011, 6, 5, 5, 19, 21)
>>> d.year, d.month, d.day
(2011, 6, 5)
>>> d.hour, d.minute, d.second, d.microsecond
(5, 19, 21, 0)
```

Над экземплярами класса `datetime` можно производить следующие операции:

- `datetime2 = datetime1 + timedelta` — прибавляет к дате указанный период;
- `datetime2 = datetime1 – timedelta` — вычитает из даты указанный период;

- `timedelta = datetime1 – datetime2` — возвращает разницу между датами;
- можно также сравнивать две даты с помощью операторов сравнения.

Примеры:

```
>>> d1 = datetime.datetime(2011, 1, 20, 23, 48, 23)
>>> d2 = datetime.datetime(2011, 1, 1, 10, 15, 38)
>>> t = datetime.timedelta(days=10, minutes=10)
>>> d1 + t                      # Прибавляем 10 дней и 10 минут
datetime.datetime(2011, 1, 30, 23, 58, 23)
>>> d1 – t                      # Вычитаем 10 дней и 10 минут
datetime.datetime(2011, 1, 10, 23, 38, 23)
>>> d1 – d2                     # Разница между датами
datetime.timedelta(19, 48765)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
(False, True, False, True)
>>> d1 == d2, d1 != d2
(False, True)
```

Экземпляры класса `datetime` поддерживают следующие методы:

- `date()` — возвращает экземпляр класса `date`:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.date()
datetime.date(2011, 6, 5)
```

- `time()` — возвращает экземпляр класса `time`:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.time()
datetime.time(23, 48, 23)
```

- `timetz()` — возвращает экземпляр класса `time`. Метод учитывает параметр `tzinfo`;
- `replace([year][, month][, day][, hour][, minute][, second][, microsecond][, tzinfo])` — возвращает дату с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. Пример:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.replace(2008, 12)
datetime.datetime(2008, 12, 5, 23, 48, 23)
>>> d.replace(hour=12, month=10)
datetime.datetime(2011, 10, 5, 12, 48, 23)
```

- `timetuple()` — возвращает объект `struct_time` с датой и временем:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.timetuple()
time.struct_time(tm_year=2011, tm_mon=6, tm_mday=5, tm_hour=23,
tm_min=48, tm_sec=23, tm_wday=6, tm_yday=156, tm_isdst=-1)
```

- `utctimetuple()` — возвращает объект `struct_time` с датой в универсальном времени (UTC):

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.utctimetuple()
```

```
time.struct_time(tm_year=2011, tm_mon=6, tm_mday=5, tm_hour=23,
tm_min=48, tm_sec=23, tm_wday=6, tm_yday=156, tm_isdst=0)
```

- `toordinal()` — возвращает количество дней, прошедшее с 1 года:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.toordinal()
734293
```

- `weekday()` — возвращает порядковый номер дня в неделе (`0` — для понедельника, `6` — для воскресенья):

```
>>> d = datetime.datetime(2011, 6, 4, 23, 48, 23)
>>> d.weekday() # 5 — это суббота
5
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (`1` — для понедельника, `7` — для воскресенья):

```
>>> d = datetime.datetime(2011, 6, 4, 23, 48, 23)
>>> d.isoweekday() # 6 — это суббота
6
```

- `isocalendar()` — возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.isocalendar()
(2011, 22, 7)
```

- `isoformat([<Разделитель>])` — возвращает дату в формате ISO 8601:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.isoformat()           # Разделитель не указан
'2011-06-05T23:48:23'
>>> d.isoformat(" ")        # Пробел в качестве разделителя
'2011-06-05 23:48:23'
```

- `ctime()` — возвращает строку специального формата:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.ctime()
'Sun Jun  5 23:48:23 2011'
```

- `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно указывать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

```
>>> d = datetime.datetime(2011, 6, 5, 23, 48, 23)
>>> d.strftime("%d.%m.%Y %H:%M:%S")
'05.06.2011 23:48:23'
```

> ***Примечание***
>
> Экземпляры класса `datetime` поддерживают также методы `astimezone()`, `dst()`, `utcoffset()` и `tzname()`. За подробной информацией по этим методам, а также по абстрактному классу `tzinfo` обращайтесь к документации по модулю `datetime`.

10.5. Модуль *calendar*. Вывод календаря

Модуль `calendar` позволяет вывести календарь в виде простого текста или в HTML-формате. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import calendar
```

Модуль предоставляет следующие классы:

- `Calendar` — базовый класс, который наследуют все остальные классы. Формат конструктора:

  ```
  Calendar([<Первый день недели>])
  ```

  В качестве примера получим двумерный список всех дней в апреле 2011 года, распределенных по дням недели:

  ```
  >>> import calendar
  >>> c = calendar.Calendar(0)
  >>> c.monthdayscalendar(2011, 4) # 4 — это апрель
  [[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10],
   [11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29, 30, 0]]
  ```

- `TextCalendar` — позволяет вывести календарь в виде простого текста. Формат конструктора:

  ```
  TextCalendar([<Первый день недели>])
  ```

  Выведем календарь на весь 2011 год:

  ```
  >>> c = calendar.TextCalendar(0)
  >>> print(c.formatyear(2011)) # Текстовый календарь на 2011 год
  ```

- `LocaleTextCalendar` — позволяет вывести календарь в виде простого текста. Названия месяцев и дней недели выводятся в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleTextCalendar([<Первый день недели>[, <Название локали>]])
  ```

  Выведем календарь на весь 2011 год на русском языке:

  ```
  >>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
  >>> print(c.formatyear(2011))
  ```

- `HTMLCalendar` — позволяет вывести календарь в формате HTML. Формат конструктора:

  ```
  HTMLCalendar([<Первый день недели>])
  ```

  Выведем календарь на весь 2011 год:

  ```
  >>> c = calendar.HTMLCalendar(0)
  >>> print(c.formatyear(2011))
  ```

- `LocaleHTMLCalendar` — позволяет вывести календарь в формате HTML. Названия месяцев и дней недели выводятся в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleHTMLCalendar([<Первый день недели>[, <Название локали>]])
  ```

Выведем календарь на весь 2011 год на русском языке в виде отдельной XHTML-страницы:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2011, encoding="windows-1251")
>>> print(xhtml.decode("cp1251"))
```

В первом параметре всех конструкторов указывается число от `0` (для понедельника) до `6` (для воскресенья). Если параметр не указан, то значение равно `0`. Вместо чисел можно использовать встроенные константы `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`. Изменить значение параметра позволяет метод `setfirstweekday(<Первый день недели>)`. В качестве примера выведем текстовый календарь на январь 2011 года, где первым днем недели является воскресенье (листинг 10.3).

Листинг 10.3. Вывод текстового календаря

```
>>> c = calendar.TextCalendar()        # Первый день понедельник
>>> c.setfirstweekday(calendar.SUNDAY) # Первый день теперь воскресенье
>>> print(c.formatmonth(2011, 1)) # Текстовый календарь на январь 2011 г.
```

10.5.1. Методы классов *TextCalendar* и *LocaleTextCalendar*

Экземпляры классов `TextCalendar` и `LocaleTextCalendar` имеют следующие методы:

- `formatmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками. Выведем календарь на апрель 2011 года:

```
>>> import calendar
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatmonth(2011, 4))
    Апрель 2011
Пн Вт Ср Чт Пт Сб Вс
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

- `prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — метод аналогичен методу `formatmonth()`, но не возвращает календарь в виде строки, а сразу выводит его. Выведем календарь на апрель 2011 года и укажем ширину поля с днем равной 4 символам:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.prmonth(2011, 4, 4)
           Апрель 2011
 Пн   Вт   Ср   Чт   Пт   Сб   Вс
                    1    2    3
  4    5    6    7    8    9   10
 11   12   13   14   15   16   17
 18   19   20   21   22   23   24
 25   26   27   28   29   30
```

- `formatyear(<Год>[, w=2][, l=1][, c=6][, m=3])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - `w` — задает ширину поля с днем (по умолчанию `2`);
  - `l` — количество символов перевода строки между строками (по умолчанию `1`);
  - `c` — количество пробелов между месяцами (по умолчанию `6`);
  - `m` — количество месяцев на строке (по умолчанию `3`).

  Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера выведем календарь на 2011 год. На одной строке выведем сразу четыре месяца и установим количество пробелов между месяцами:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2011, m=4, c=2))
```

- `pryear(<Год>[, w=2][, l=1][, c=6][, m=3])` — метод аналогичен методу `formatyear()`, но не возвращает календарь в виде строки, а сразу выводит его. В качестве примера выведем календарь на 2011 год по два месяца на строке. Расстояние между месяцами установим равным 4 символам, ширину поля с датой равной 2 символам, а строки разделим одним символом перевода строки:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.pryear(2011, 2, 1, 4, 2)
```

10.5.2. Методы классов *HTMLCalendar* и *LocaleHTMLCalendar*

Экземпляры классов `HTMLCalendar` и `LocaleHTMLCalendar` имеют следующие методы:

- `formatmonth(<Год>, <Месяц>[, <True | False>])` — возвращает HTML-календарь на указанный месяц в году. Если в третьем параметре указано значение `True` (значение по умолчанию), то в заголовке таблицы после названия месяца будет указан год. Календарь будет отформатирован с помощью HTML-таблицы. Для каждой ячейки таблицы задается стилевой класс, с помощью которого можно управлять внешним видом календаря. Названия стилевых классов доступны через атрибут `cssclasses`, который содержит список названий для каждого дня недели:

```
>>> import calendar
>>> c = calendar.HTMLCalendar(0)
>>> print(c.cssclasses)
['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
```

  Выведем календарь на апрель 2011 года. Для будних дней укажем класс `"workday"`, а для выходных дней — класс `"week-end"`:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> c.cssclasses = ["workday", "workday", "workday", "workday",
                    "workday", "week-end", "week-end"]
>>> print(c.formatmonth(2011, 4, False))
```

- `formatyear(<Год>[, <Количество месяцев на строке>])` — возвращает HTML-календарь на указанный год. Календарь будет отформатирован с помощью нескольких HTML-таблиц. В качестве примера выведем календарь на 2011 год. На одной строке выведем сразу четыре месяца:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2011, 4))
```

- `formatyearpage(<Год>[, width][, css][, encoding])` — возвращает HTML-календарь на указанный год в виде отдельной XHTML-страницы. Параметры имеют следующее предназначение:
  - `width` — количество месяцев на строке (по умолчанию `3`);
  - `css` — название файла с таблицей стилей (по умолчанию `"calendar.css"`);
  - `encoding` — кодировка файла. Название кодировки будет указано в параметре `encoding` XML-пролога, а также в теге `<meta>`.

  Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера выведем календарь на 2011 год. На одной строке выведем сразу четыре месяца и укажем кодировку файла:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2011, 4, encoding="windows-1251")
>>> type(xhtml) # Возвращаемая строка имеет тип данных bytes
<class 'bytes'>
>>> print(xhtml.decode("cp1251"))
```

10.5.3. Другие полезные функции

Модуль `calendar` предоставляет еще и несколько функций, которые позволяют вывести текстовый календарь без создания экземпляра класса, а также возвращают дополнительную информацию о дате:

- `setfirstweekday(<Первый день недели>)` — устанавливает первый день недели для календаря. В качестве параметра указывается число от `0` (для понедельника) до `6` (для воскресенья). Вместо чисел можно использовать встроенные константы `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`. Получить текущее значение параметра можно с помощью функции `firstweekday()`. Установим воскресенье первым днем недели:

```
>>> import calendar
>>> calendar.firstweekday()      # По умолчанию 0
0
>>> calendar.setfirstweekday(6) # Изменяем значение
>>> calendar.firstweekday()      # Проверяем установку
6
```

- `month(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками. Выведем календарь на апрель 2011 года:

```
>>> calendar.setfirstweekday(0)
>>> print(calendar.month(2011, 4)) # Апрель 2011 года
     April 2011
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
```

```
18 19 20 21 22 23 24
25 26 27 28 29 30
```

- `prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — функция аналогична функции `month()`, но не возвращает календарь в виде строки, а сразу выводит его. Выведем календарь на апрель 2011 года:

```
>>> calendar.prmonth(2011, 4) # Апрель 2011 года
```

- `monthcalendar(<Год>, <Месяц>)` — возвращает двумерный список всех дней в указанном месяце, распределенных по дням недели. Дни, выходящие за пределы месяца, будут представлены нулями. Выведем массив для апреля 2011 года:

```
>>> calendar.monthcalendar(2011, 4)
[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10],
 [11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21, 22, 23, 24],
 [25, 26, 27, 28, 29, 30, 0]]
```

- `monthrange(<Год>, <Месяц>)` — возвращает кортеж из двух элементов: количество недель в месяце и число дней в месяце:

```
>>> print(calendar.monthrange(2011, 4))
(4, 30)
>>> # В апреле 2011 года 4 недели и 30 дней
```

- `calendar(<Год>[, w][, l][, c][, m])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - `w` — задает ширину поля с днем (по умолчанию `2`);
  - `l` — количество символов перевода строки между строками (по умолчанию `1`);
  - `c` — количество пробелов между месяцами (по умолчанию `6`);
  - `m` — количество месяцев на строке (по умолчанию `3`).

  Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера выведем календарь на 2011 год. На одной строке выведем сразу четыре месяца и установим количество пробелов между месяцами:

```
>>> print(calendar.calendar(2011, m=4, c=2))
```

- `prcal(<Год>[, w][, l][, c][, m])` — функция аналогична функции `calendar()`, но не возвращает календарь в виде строки, а сразу выводит его. В качестве примера выведем календарь на 2011 год по два месяца на строке. Расстояние между месяцами установим равным 4 символам, ширину поля с датой равной 2 символам, а строки разделим одним символом перевода строки:

```
>>> calendar.prcal(2011, 2, 1, 4, 2)
```

- `isleap(<Год>)` — возвращает значение `True`, если указанный год является високосным, в противном случае — `False`:

```
>>> calendar.isleap(2011), calendar.isleap(2012)
(False, True)
```

- `leapdays(<Год1>, <Год2>)` — возвращает количество високосных лет в диапазоне от `<Год1>` до `<Год2>` (`<Год2>` не учитывается):

```
>>> calendar.leapdays(2010, 2012) # 2012 не учитывается
0
>>> calendar.leapdays(2010, 2013) # 2012 — високосный год
1
```

- `weekday(<Год>, <Месяц>, <День>)` — возвращает номер дня недели (`0` — для понедельника, `6` — для воскресенья):

```
>>> calendar.weekday(2011, 6, 4)
5
```

- `timegm(<Объект struct_time>)` — возвращает число, представляющее количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` с датой и временем, возвращаемый функцией `gmtime()` из модуля `time`. Пример:

```
>>> import calendar, time
>>> d = time.gmtime(1303520699.0) # Дата 23-04-2011
>>> d
time.struct_time(tm_year=2011, tm_mon=4, tm_mday=23, tm_hour=1,
 tm_min=4, tm_sec=59, tm_wday=5, tm_yday=113, tm_isdst=0)
>>> tuple(d)
(2011, 4, 23, 1, 4, 59, 5, 113, 0)
>>> calendar.timegm(d)
1303520699
>>> calendar.timegm((2011, 4, 23, 1, 4, 59, 5, 113, 0))
1303520699
```

Модуль `calendar` предоставляет также несколько атрибутов:

- `day_name` — полные названия дней недели в текущей локали:

```
>>> [i for i in calendar.day_name]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
```

- `day_abbr` — аббревиатуры названий дней недели в текущей локали:

```
>>> [i for i in calendar.day_abbr]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

- `month_name` — полные названия месяцев в текущей локали:

```
>>> [i for i in calendar.month_name]
['', 'January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December']
```

- `month_abbr` — аббревиатуры названий месяцев в текущей локали:

```
>>> [i for i in calendar.month_abbr]
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> [i for i in calendar.day_abbr]
['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс']
```

```
>>> [i for i in calendar.month_name]
['', 'Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль',
 'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь']
>>> [i for i in calendar.month_abbr]
['', 'янв', 'фев', 'мар', 'апр', 'май', 'июн', 'июл', 'авг', 'сен',
 'окт', 'ноя', 'дек']
```

10.6. Измерение времени выполнения фрагментов кода

Модуль `timeit` позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
from timeit import Timer
```

Измерения производятся с помощью класса `Timer`. Конструктор класса имеет следующий формат:

```
Timer([stmt='pass'][, setup='pass'][, timer=<timer function>])
```

В параметре `stmt` указывается код (в виде строки), для которого измеряем время выполнения. Параметр `setup` позволяет указать код, который будет выполнен перед измерением времени выполнения кода в параметре `stmt`. Например, в параметре `setup` можно подключить модуль.

Получить время выполнения можно с помощью метода `timeit([number=1000000])`. В параметре `number` указывается количество повторений. Для примера просуммируем числа от `1` до `10000` тремя способами и выведем время выполнения каждого способа (листинг 10.4).

Листинг 10.4. Измерение времени выполнения

```
# -*- coding: utf-8 -*-
from timeit import Timer
code1 = """\
i, j = 1, 0
while i < 10001:
    j += i
    i += 1
"""
t1 = Timer(stmt=code1)
print("while:", t1.timeit(number=10000))
code2 = """\
j = 0
for i in range(1, 10001):
    j += i
"""
t2 = Timer(stmt=code2)
print("for:", t2.timeit(number=10000))
code3 = """\
j = sum(range(1, 10001))
"""
```

```
t3 = Timer(stmt=code3)
print("sum:", t3.timeit(number=10000))
input()
```

Примерный результат выполнения (зависит от мощности компьютера):

```
while: 17.26802443580117
for: 9.602403053818772
sum: 3.067899091205735
```

Как видно из результата, цикл `for` работает в два раза быстрее цикла `while`, а функция `sum()` в данном случае является самым оптимальным решением задачи.

Метод `repeat([repeat=3][, number=1000000])` вызывает метод `timeit()` указанное количество раз (задается в параметре `repeat`) и возвращает список значений. Аргумент `number` передается в качестве параметра методу `timeit()`. Для примера создадим список со строковыми представлениями чисел от `1` до `10000`. В первом случае для создания списка используем цикл `for` и метод `append()`, а во втором — генератор списков (листинг 10.5).

Листинг 10.5. Использование метода `repeat()`

```
# -*- coding: utf-8 -*-
from timeit import Timer
code1 = """\
arr1 = []
for i in range(1, 10001):
    arr1.append(str(i))
"""
t1 = Timer(stmt=code1)
print("append:", t1.repeat(repeat=3, number=2000))
code2 = """\
arr2 = [str(i) for i in range(1, 10001)]
"""
t2 = Timer(stmt=code2)
print("генератор:", t2.repeat(repeat=3, number=2000))
input()
```

Примерный результат выполнения:

```
append: [9.98154294362325, 9.997541285814483, 10.004275691347512]
генератор: [7.879446040575054, 7.924527742041953, 7.844603314867392]
```

Как видно из результата, генераторы списков работают быстрее.

ГЛАВА 11

Пользовательские функции

Функция — это фрагмент кода, который можно вызвать из любого места программы. В предыдущих главах мы уже не один раз использовали встроенные функции языка Python, например, с помощью функции `len()` получали количество элементов последовательности. В этой главе мы рассмотрим создание пользовательских функций, которые позволят уменьшить избыточность программного кода и повысить его структурированность.

11.1. Создание функции и ее вызов

Функция описывается с помощью ключевого слова `def` по следующей схеме:

```
def <Имя функции>([<Параметры>]):
    ["""  Строка документирования """]
    <Тело функции>
    [return <Значение>]
```

Имя функции должно быть уникальным идентификатором, состоящим из латинских букв, цифр и знаков подчеркивания, причем имя функции не может начинаться с цифры. В качестве имени нельзя использовать ключевые слова, кроме того, следует избегать совпадений с названиями встроенных идентификаторов. Регистр символов в названии функции имеет значение.

После имени функции в круглых скобках можно указать один или несколько параметров через запятую. Если функция не принимает параметры, то просто указываются круглые скобки. После круглых скобок ставится двоеточие.

Тело функции является составной конструкцией. Как и в любой составной конструкции, инструкции внутри функции выделяются одинаковым количеством пробелов слева. Концом функции считается инструкция, перед которой меньшее количество пробелов. Если тело функции не содержит инструкций, то внутри необходимо разместить оператор `pass`. Этот оператор удобно использовать на этапе отладки программы, когда мы определили функцию, а тело будем дописывать позже. Пример функции, которая ничего не делает:

```
def func():
    pass
```

Необязательная инструкция `return` позволяет вернуть значение из функции. После исполнения этой инструкции выполнение функции будет остановлено. Это означает, что инструкции после оператора `return` никогда не будут выполнены.

Пример:

```
def func():
    print("Текст до инструкции return")
    return "Возвращаемое значение"
    print("Эта инструкция никогда не будет выполнена")

print(func()) # Вызываем функцию
```

Результат выполнения:

```
Текст до инструкции return
Возвращаемое значение
```

Инструкции `return` может не быть вообще. В этом случае выполняются все инструкции внутри функции и возвращается значение `None`.

В качестве примера создадим три функции (листинг 11.1).

Листинг 11.1. Определения функций

```
def print_ok():
    """ Пример функции без параметров """
    print("Сообщение при удачно выполненной операции")

def echo(m):
    """ Пример функции с параметром """
    print(m)

def summa(x, y):
    """ Пример функции с параметрами,
        возвращающей сумму двух переменных """
    return x + y
```

При вызове функции значения передаются внутри круглых скобок через запятую. Если функция не принимает параметров, то указываются только круглые скобки. Необходимо также заметить, что количество параметров в определении функции должно совпадать с количеством параметров при вызове, иначе будет выведено сообщение об ошибке. Вызвать функции из листинга 11.1 можно способами, указанными в листинге 11.2.

Листинг 11.2. Вызов функций

```
print_ok()              # Вызываем функцию без параметров
echo("Сообщение")       # Функция выведет сообщение
x = summa(5, 2)         # Переменной x будет присвоено значение 7
a, b = 10, 50
y = summa(a, b)         # Переменной y будет присвоено значение 60
```

Как видно из последнего примера, имя переменной в вызове функции может не совпадать с именем переменной в определении функции. Кроме того, *глобальные* переменные `x` и `y` не конфликтуют с одноименными переменными в определении функции, т. к. они расположены в разных областях видимости. Переменные, указанные в определении функции, являют-

ся *локальными* и доступны только внутри функции. Более подробно области видимости мы рассмотрим в *разд. 11.9*.

Оператор +, используемый в функции `summa()`, применяется не только для сложения чисел, но и позволяет объединить последовательности. Таким образом, функция `summa()` может использоваться не только для сложения чисел. В качестве примера выполним конкатенацию строк и объединение списков:

```
def summa(x, y):
    return x + y

print(summa("str", "ing"))    # Выведет: string
print(summa([1, 2], [3, 4])) # Выведет: [1, 2, 3, 4]
```

Как вы уже знаете, все в языке Python является объектом, например строки, списки и даже сами типы данных. Функции не являются исключением. Инструкция `def` создает объект, имеющий тип `function`, и сохраняет ссылку на него в идентификаторе, указанном после инструкции `def`. Таким образом, мы можем сохранить ссылку на функцию в другой переменной. Для этого название функции указывается без круглых скобок. Сохраним ссылку в переменной и вызовем функцию через нее (листинг 11.3).

Листинг 11.3. Сохранение ссылки на функцию в переменной

```
def summa(x, y):
    return x + y

f = summa                      # Сохраняем ссылку в переменной f
v = f(10, 20)                  # Вызываем функцию через переменную f
```

Можно также передать ссылку на функцию в качестве параметра другой функции. Функции, передаваемые по ссылке, обычно называются *функциями обратного вызова* (листинг 11.4).

Листинг 11.4. Функции обратного вызова

```
def summa(x, y):
    return x + y

def func(f, a, b):
    """ Через переменную f будет доступна ссылка на
        функцию summa() """
    return f(a, b) # Вызываем функцию summa()

# Передаем ссылку на функцию в качестве параметра
v = func(summa, 10, 20)
```

Объекты функций поддерживают множество атрибутов, обратиться к которым можно указав атрибут после названия функции через точку. Например, через атрибут `__name__` можно получить название функции в виде строки, через атрибут `__doc__` — строку документирования и т. д. В качестве примера выведем названия всех атрибутов функции с помощью встроенной функции `dir()`:

```
>>> def summa(x, y):
        """ Суммирование двух чисел """
        return x + y

>>> dir(summa)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
 '__defaults__', '__delattr__', '__dict__', '__doc__', '__eq__',
 '__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
 '__module__', '__name__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
>>> summa.__name__
'summa'
>>> summa.__code__.co_varnames
('x', 'y')
>>> summa.__doc__
' Суммирование двух чисел '
```

11.2. Расположение определений функций

Все инструкции в программе выполняются последовательно сверху вниз. Это означает, что прежде чем использовать идентификатор в программе, его необходимо предварительно определить, присвоив ему значение. Поэтому определение функции должно быть расположено перед вызовом функции.

Правильно:

```
def summa(x, y):
    return x + y
v = summa(10, 20) # Вызываем после определения. Все нормально
```

Неправильно:

```
v = summa(10, 20) # Идентификатор еще не определен. Это ошибка!!!
def summa(x, y):
    return x + y
```

В последнем случае будет выведено сообщение об ошибке "`NameError: name 'summa' is not defined`". Чтобы избежать ошибки, определение функции размещают в самом начале программы после подключения модулей или в отдельном файле, который называется *модулем*.

С помощью оператора ветвления `if` можно изменить порядок выполнения программы. Таким образом, можно разместить внутри условия несколько определений функций с одинаковым названием, но разной реализацией (листинг 11.5).

Листинг 11.5. Определение функции в зависимости от условия

```
# -*- coding: utf-8 -*-
n = input("Введите 1 для вызова первой функции: ")
n = n.rstrip("\r")                  # Для версии 3.2.0 (см. разд. 1.7)
```

```
if n == "1":
    def echo():
        print("Вы ввели число 1")
else:
    def echo():
        print("Альтернативная функция")

echo() # Вызываем функцию
input()
```

При вводе числа 1 мы получим сообщение "Вы ввели число 1", в противном случае — "Альтернативная функция".

Помните, что инструкция def всего лишь присваивает ссылку на объект функции идентификатору, расположенному после ключевого слова def. Если определение одной функции встречается в программе несколько раз, то будет использоваться функция, которая расположена последней. Пример:

```
def echo():
    print("Вы ввели число 1")
def echo():
    print("Альтернативная функция")
echo() # Всегда выводит "Альтернативная функция"
```

11.3. Необязательные параметры и сопоставление по ключам

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение. Переделаем функцию суммирования двух чисел и сделаем второй параметр необязательным (листинг 11.6).

Листинг 11.6. Необязательные параметры

```
def summa(x, y=2):          # y — необязательный параметр
    return x + y
a = summa(5)               # Переменной a будет присвоено значение 7
b = summa(10, 50)          # Переменной b будет присвоено значение 60
```

Таким образом, если второй параметр не задан, то его значение будет равно 2. Обратите внимание на то, что необязательные параметры должны следовать после обязательных параметров, иначе будет выведено сообщение об ошибке.

До сих пор мы использовали позиционную передачу параметров в функцию:

```
def summa(x, y):
    return x + y
print(summa(10, 20))        # Выведет: 30
```

Переменной x при сопоставлении будет присвоено значение 10, а переменной y — значение 20. Язык Python позволяет также передать значения в функцию, используя сопоставление по ключам (листинг 11.7). Для этого при вызове функции параметрам присваиваются значения. Последовательность указания параметров может быть произвольной.

Листинг 11.7. Сопоставление по ключам

```
def summa(x, y):
    return x + y
print(summa(y=20, x=10))    # Сопоставление по ключам
```

Сопоставление по ключам очень удобно использовать, если функция имеет несколько необязательных параметров. В этом случае не нужно перечислять все значения, а достаточно присвоить значение нужному параметру. Пример:

```
def summa(a=2, b=3, c=4): # Все параметры являются необязательными
    return a + b + c
print(summa(2, 3, 20))    # Позиционное присваивание
print(summa(c=20))        # Сопоставление по ключам
```

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ `*`. Пример передачи значений из кортежа и списка приведен в листинге 11.8.

Листинг 11.8. Пример передачи значений из кортежа и списка

```
def summa(a, b, c):
    return a + b + c
t1, arr = (1, 2, 3), [1, 2, 3]
print(summa(*t1))                  # Распаковываем кортеж
print(summa(*arr))                 # Распаковываем список
t2 = (2, 3)
print(summa(1, *t2))               # Можно комбинировать значения
```

Если значения параметров содержатся в словаре, то распаковать словарь можно, указав перед ним две звездочки (`**`) (листинг 11.9).

Листинг 11.9. Пример передачи значений из словаря

```
def summa(a, b, c):
    return a + b + c
d1 = {"a": 1, "b": 2, "c": 3}
print(summa(**d1))                 # Распаковываем словарь
t, d2 = (1, 2), {"c": 3}
print(summa(*t, **d2))             # Можно комбинировать значения
```

Объекты в функцию передаются по ссылке. Если объект относится к неизменяемому типу, то изменение значения внутри функции не затронет значение переменной вне функции:

```
def func(a, b):
    a, b = 20, "str"
x, s = 80, "test"
func(x, s)              # Значения переменных x и s не изменяются
print(x, s)             # Выведет: 80 test
```

В этом примере значения в переменных `x` и `s` не изменились. Однако если объект относится к изменяемому типу, то ситуация будет другой:

```
def func(a, b):
    a[0], b["a"] = "str", 800
x = [1, 2, 3]            # Список
y = {"a": 1, "b": 2}     # Словарь
func(x, y)               # Значения будут изменены!!!
print(x, y)              # Выведет: ['str', 2, 3] {'a': 800, 'b': 2}
```

Как видно из примера, значения в переменных `x` и `y` изменились, т. к. список и словарь относятся к изменяемым типам. Чтобы избежать изменения значений, внутри функции следует создать копию объекта (листинг 11.10).

Листинг 11.10. Передача изменяемого объекта в функцию

```
def func(a, b):
    a = a[:]                  # Создаем поверхностную копию списка
    b = b.copy()              # Создаем поверхностную копию словаря
    a[0], b["a"] = "str", 800
x = [1, 2, 3]                 # Список
y = {"a": 1, "b": 2}          # Словарь
func(x, y)                    # Значения останутся прежними
print(x, y)                   # Выведет: [1, 2, 3] {'a': 1, 'b': 2}
```

Можно также сразу передавать копию объекта в вызове функции:

```
func(x[:], y.copy())
```

Если указать объект, имеющий изменяемый тип, в качестве значения по умолчанию, то этот объект будет сохраняться между вызовами функции. Пример:

```
def func(a=[]):
    a.append(2)
    return a
print(func())                 # Выведет: [2]
print(func())                 # Выведет: [2, 2]
print(func())                 # Выведет: [2, 2, 2]
```

Как видно из примера, значения накапливаются внутри списка. Обойти эту проблему можно, например, следующим образом:

```
def func(a=None):
    # Создаем новый список, если значение равно None
    if a is None:
        a = []
    a.append(2)
    return a
print(func())                 # Выведет: [2]
print(func([1]))              # Выведет: [1, 2]
print(func())                 # Выведет: [2]
```

11.4. Переменное число параметров в функции

Если перед параметром в определении функции указать символ `*`, то функции можно будет передать произвольное количество параметров. Все переданные параметры сохраняются

в кортеже. В качестве примера напишем функцию суммирования произвольного количества чисел (листинг 11.11).

Листинг 11.11. Сохранение переданных данных в кортеже

```
def summa(*t):
    """ Функция принимает произвольное количество параметров """
    res = 0
    for i in t:      # Перебираем кортеж с переданными параметрами
        res += i
    return res
print(summa(10, 20))                    # Выведет: 30
print(summa(10, 20, 30, 40, 50, 60))    # Выведет: 210
```

Можно также вначале указать несколько обязательных параметров и параметров, имеющих значения по умолчанию:

```
def summa(x, y=5, *t): # Комбинация параметров
    res = x + y
    for i in t:        # Перебираем кортеж с переданными параметрами
        res += i
    return res
print(summa(10))                        # Выведет: 15
print(summa(10, 20, 30, 40, 50, 60))    # Выведет: 210
```

Если перед параметром в определении функции указать две звездочки (`**`), то все именованные параметры будут сохранены в словаре (листинг 11.12).

Листинг 11.12. Сохранение переданных данных в словаре

```
def func(**d):
    for i in d:     # Перебираем словарь с переданными параметрами
        print("{0} => {1}".format(i, d[i]), end=" ")
func(a=1, b=2, c=3) # Выведет: a => 1 c => 3 b => 2
```

При комбинировании параметров параметр с двумя звездочками указывается самым последним. Если в определении функции указывается комбинация параметров с одной звездочкой и двумя звездочками, то функция примет любые переданные ей параметры (листинг 11.13).

Листинг 11.13. Комбинирование параметров

```
def func(*t, **d):
    """ Функция примет любые параметры """
    for i in t:
        print(i, end=" ")
    for i in d:    # Перебираем словарь с переданными параметрами
        print("{0} => {1}".format(i, d[i]), end=" ")
func(35, 10, a=1, b=2, c=3) # Выведет: 35 10 a => 1 c => 3 b => 2
func(10)                    # Выведет: 10
func(a=1, b=2)              # Выведет: a => 1 b => 2
```

В Python 3 в определении функции можно указать, что некоторые параметры передаются только по именам. Для этого параметры должны указываться после параметра с одной звездочкой, но перед параметром с двумя звездочками. Именованные параметры могут иметь значения по умолчанию. Пример:

```
def func(*t, a, b=10, **d):
    print(t, a, b, d)
func(35, 10, a=1, c=3) # Выведет: (35, 10) 1 10 {'c': 3}
func(10, a=5)          # Выведет: (10,) 5 10 {}
func(a=1, b=2)         # Выведет: () 1 2 {}
func(1, 2, 3)          # Ошибка. Параметр a обязателен!
```

В этом примере переменная `t` примет любое количество значений, которые будут объединены в кортеж. Переменные `a` и `b` должны передаваться только по именам, причем переменной `a` обязательно нужно передать значение при вызове функции. Переменная `b` имеет значение по умолчанию, поэтому при вызове допускается не передавать ей значение, но если значение передается, то оно должно быть указано после названия параметра и оператора `=`. Переменная `d` примет любое количество именованных параметров и сохранит их в словаре. Обратите внимание на то, что хотя переменные `a` и `b` являются именованными, они не попадут в этот словарь.

Параметра с двумя звездочками может не быть в определении функции, а вот параметр с одной звездочкой при указании параметров, передаваемых только по именам, должен быть обязательно. Если функция не должна принимать переменного количества параметров, но необходимо использовать переменные, передаваемые только по именам, то можно указать только звездочку без переменной:

```
def func(x=1, y=2, *, a, b=10):
    print(x, y, a, b)
func(35, 10, a=1)      # Выведет: 35 10 1 10
func(10, a=5)          # Выведет: 10 2 5 10
func(a=1, b=2)         # Выведет: 1 2 1 2
func(a=1, y=8, x=7)    # Выведет: 7 8 1 10
func(1, 2, 3)          # Ошибка. Параметр a обязателен!
```

В этом примере значения переменным `x` и `y` можно передавать как по позициям, так и по именам. Так как переменные имеют значения по умолчанию, допускается вообще не передавать им значений. Переменные `a` и `b` расположены после параметра с одной звездочкой, поэтому передать значения при вызове можно только по именам. Так как переменная `b` имеет значение по умолчанию, допускается не передавать ей значение при вызове, а вот переменная `a` обязательно должна получить значение, причем только по имени.

11.5. Анонимные функции

Помимо обычных функций язык Python позволяет использовать анонимные функции, которые называются *лямбда-функциями*. Анонимная функция описывается с помощью ключевого слова `lambda` по следующей схеме:

```
lambda [<Параметр1>[, ..., <ПараметрN>]]: <Возвращаемое значение>
```

После ключевого слова `lambda` можно указать передаваемые параметры. В качестве параметра `<Возвращаемое значение>` указывается выражение, результат выполнения которого будет возвращен функцией. Как видно из схемы, у лямбда-функций нет имени. По этой причине их и называют анонимными функциями.

В качестве значения лямбда-функция возвращает ссылку на объект-функцию, которую можно сохранить в переменной или передать в качестве параметра в другую функцию. Вызвать лямбда-функцию можно, как и обычную, с помощью круглых скобок, внутри которых расположены передаваемые параметры. Пример использования лямбда-функций приведен в листинге 11.14.

Листинг 11.14. Пример использования лямбда-функций

```
f1 = lambda: 10 + 20                # Функция без параметров
f2 = lambda x, y: x + y             # Функция с двумя параметрами
f3 = lambda x, y, z: x + y + z      # Функция с тремя параметрами
print(f1())                         # Выведет: 30
print(f2(5, 10))                    # Выведет: 15
print(f3(5, 10, 30))                # Выведет: 45
```

Как и в обычных функциях, некоторые параметры лямбда-функций могут быть необязательными. Для этого параметрам в определении функции присваивается значение по умолчанию (листинг 11.15).

Листинг 11.15. Необязательные параметры в лямбда-функциях

```
f = lambda x, y=2: x + y
print(f(5))                         # Выведет: 7
print(f(5, 6))                      # Выведет: 11
```

Наиболее часто не сохраняют ссылку в переменной, а сразу передают в качестве параметра в другую функцию. Например, метод списков `sort()` позволяет указать пользовательскую функцию в параметре `key`. Отсортируем список без учета регистра символов, указав в качестве параметра лямбда-функцию (листинг 11.16).

Листинг 11.16. Сортировка без учета регистра символов

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=lambda s: s.lower())
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

11.6. Функции-генераторы

Функцией-генератором называется функция, которая может возвращать одно значение из нескольких значений на каждой итерации. Приостановить выполнение функции и превратить функцию в генератор позволяет ключевое слово `yield`. В качестве примера напишем функцию, которая возводит элементы последовательности в указанную степень (листинг 11.17).

Листинг 11.17. Пример использования функций-генераторов

```
def func(x, y):
    for i in range(1, x+1):
        yield i ** y
```

```
for n in func(10, 2):
    print(n, end=" ")     # Выведет: 1 4 9 16 25 36 49 64 81 100
print()                   # Вставляем пустую строку
for n in func(10, 3):
    print(n, end=" ")     # Выведет: 1 8 27 64 125 216 343 512 729 1000
```

Функции-генераторы поддерживают метод `__next__()`, который позволяет получить следующее значение. Когда значения заканчиваются, метод возбуждает исключение `StopIteration`. Вызов метода `__next__()` в цикле `for` производится незаметно для нас. В качестве примера перепишем предыдущую программу и используем метод `__next__()` вместо цикла `for` (листинг 11.18).

Листинг 11.18. Использование метода `__next__()`

```
def func(x, y):
    for i in range(1, x+1):
       yield i ** y

i = func(3, 3)
print(i.__next__())       # Выведет: 1 (1 ** 3)
print(i.__next__())       # Выведет: 8 (2 ** 3)
print(i.__next__())       # Выведет: 27 (3 ** 3)
print(i.__next__())       # Исключение StopIteration
```

Таким образом, с помощью обычных функций мы можем вернуть все значения сразу в виде списка, а с помощью функций-генераторов только одно значение за раз. Эта особенность очень полезна при обработке большого количества значений, т. к. не нужно загружать весь список со значениями в память.

11.7. Декораторы функций

Декораторы позволяют изменить поведение обычных функций. Например, выполнить какие-либо действия перед выполнением функции. Рассмотрим это на примере (листинг 11.19).

Листинг 11.19. Декораторы функций

```
def deco(f):                          # Функция-декоратор
    print("Вызвана функция func()")
    return f                          # Возвращаем ссылку на функцию
@deco
def func(x):
    return "x = {0}".format(x)

print(func(10))
```

Выведет:

```
Вызвана функция func()
x = 10
```

В этом примере перед определением функции `func()` указывается название функции `deco()` с предваряющим символом `@`:

```
@deco
```

Таким образом, функция `deco()` становится декоратором функции `func()`. В качестве параметра функция-декоратор принимает ссылку на функцию, поведение которой необходимо изменить, и должна возвращать ссылку на ту же функцию или какую-либо другую. Наш предыдущий пример эквивалентен следующему коду:

```
def deco(f):
    print("Вызвана функция func()")
    return f
def func(x):
    return "x = {0}".format(x)
# Вызываем функцию func() через функцию deco()
print(deco(func)(10))
```

Перед определением функции можно указать сразу несколько функций-декораторов. В качестве примера обернем функцию `func()` в два декоратора: `deco1()` и `deco2()` (листинг 11.20).

Листинг 11.20. Указание нескольких декораторов

```
def deco1(f):
    print("Вызвана функция deco1()")
    return f
def deco2(f):
    print("Вызвана функция deco2()")
    return f
@deco1
@deco2
def func(x):
    return "x = {0}".format(x)
print(func(10))
```

Выведет:

```
Вызвана функция deco2()
Вызвана функция deco1()
x = 10
```

Использование двух декораторов эквивалентно следующему коду:

```
func = deco1(deco2(func))
```

Сначала будет вызвана функция `deco2()`, а затем функция `deco1()`. Результат выполнения будет присвоен идентификатору `func`.

В качестве еще одного примера использования декораторов рассмотрим выполнение функции только при правильно введенном пароле (листинг 11.21).

Листинг 11.21. Ограничение доступа с помощью декоратора

```
passw = input("Введите пароль: ")
passw = passw.rstrip("\r")   # Для версии 3.2.0
```

```
def test_passw(p):
    def deco(f):
        if p == "10":        # Сравниваем пароли
            return f
        else:
            return lambda: "Доступ закрыт"
    return deco              # Возвращаем функцию-декоратор

@test_passw(passw)
def func():
    return "Доступ открыт"
print(func())                # Вызываем функцию
```

В этом примере после символа `@` указана не ссылка на функцию, а выражение, которое возвращает декоратор. Иными словами, декоратором является не функция `test_passw()`, а результат ее выполнения (функция `deco()`). Если введенный пароль является правильным, то будет выполнена функция `func()`, в противном случае будет выведена надпись `"Доступ закрыт"`, которую возвращает анонимная функция.

11.8. Рекурсия. Вычисление факториала

Рекурсия — это возможность функции вызывать саму себя. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или выполнения неопределенного количества операций. В качестве примера рассмотрим вычисление факториала (листинг 11.22).

Листинг 11.22. Вычисление факториала

```
def factorial(n):
    if n == 0 or n == 1: return 1
    else:
        return n * factorial(n - 1)

while True:
    x = input("Введите число: ")
    x = x.rstrip("\r")          # Для версии 3.2.0 (см. разд. 1.7)
    if x.isdigit():             # Если строка содержит только цифры
        x = int(x)              # Преобразуем строку в число
        break                   # Выходим из цикла
    else:
        print("Вы ввели не число!")
print("Факториал числа {0} = {1}".format(x, factorial(x)))
```

Начиная с версии 2.6, для вычисления факториала можно воспользоваться функцией `factorial()` из модуля `math`. Пример:

```
>>> import math
>>> math.factorial(5), math.factorial(6)
(120, 720)
```

11.9. Глобальные и локальные переменные

Глобальные переменные — это переменные, объявленные в программе вне функции. В Python глобальные переменные видны в любой части модуля, включая функции (листинг 11.23).

Листинг 11.23. Глобальные переменные

```
def func(glob2):
    print("Значение глобальной переменной glob1 =", glob1)
    glob2 += 10
    print("Значение локальной переменной glob2 =", glob2)

glob1, glob2 = 10, 5
func(77) # Вызываем функцию
print("Значение глобальной переменной glob2 =", glob2)
```

Результат выполнения:

```
Значение глобальной переменной glob1 = 10
Значение локальной переменной glob2 = 87
Значение глобальной переменной glob2 = 5
```

Переменной `glob2` внутри функции присваивается значение, переданное при вызове функции. По этой причине создается новое имя `glob2`, которое является *локальным*. Все изменения этой переменной внутри функции не затронут значение одноименной глобальной переменной.

Локальные переменные — это переменные, которым внутри функции присваивается значение. Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной переменной не изменяется. Локальные переменные видны только внутри тела функции (листинг 11.24).

Листинг 11.24. Локальные переменные

```
def func():
    local1 = 77                    # Локальная переменная
    glob1 = 25                     # Локальная переменная
    print("Значение glob1 внутри функции =", glob1)
glob1 = 10                         # Глобальная переменная
func()                             # Вызываем функцию
print("Значение glob1 вне функции =", glob1)
try:
    print(local1)                  # Вызовет исключение NameError
except NameError:                  # Обрабатываем исключение
    print("Переменная local1 не видна вне функции")
```

Результат выполнения:

```
Значение glob1 внутри функции = 25
Значение glob1 вне функции = 10
Переменная local1 не видна вне функции
```

Как видно из примера, переменная `local1`, объявленная внутри функции `func()`, недоступна вне функции. Объявление внутри функции локальной переменной `glob1` не изменило значения одноименной глобальной переменной.

Если обращение к переменной производится до присваивания значения (даже если существует одноименная глобальная переменная), то будет возбуждено исключение `UnboundLocalError` (листинг 11.25).

Листинг 11.25. Ошибка при обращении к переменной до присваивания значения

```
def func():
    # Локальная переменная еще не определена
    print(glob1)                    # Эта строка вызовет ошибку!!!
    glob1 = 25                      # Локальная переменная
glob1 = 10                          # Глобальная переменная
func()                              # Вызываем функцию
# Результат выполнения:
# UnboundLocalError: local variable 'glob1' referenced before assignment
```

Для того чтобы значение глобальной переменной можно было изменить внутри функции, необходимо объявить переменную глобальной с помощью ключевого слова `global`. Продемонстрируем это на примере (листинг 11.26).

Листинг 11.26. Использование ключевого слова `global`

```
def func():
    # Объявляем переменную glob1 глобальной
    global glob1
    glob1 = 25        # Изменяем значение глобальной переменной
    print("Значение glob1 внутри функции =", glob1)
glob1 = 10            # Глобальная переменная
print("Значение glob1 вне функции =", glob1)
func()                # Вызываем функцию
print("Значение glob1 после функции =", glob1)
```

Результат выполнения:

```
Значение glob1 вне функции = 10
Значение glob1 внутри функции = 25
Значение glob1 после функции = 25
```

Таким образом, поиск идентификатора, используемого внутри функции, будет производиться в следующем порядке:

1. Поиск объявления идентификатора внутри функции (в локальной области видимости).
2. Поиск объявления идентификатора в глобальной области.
3. Поиск во встроенной области видимости (встроенные функции, классы и т. д.).

При использовании анонимных функций следует учитывать, что при указании внутри функции глобальной переменной будет сохранена ссылка на эту переменную, а не ее значение в момент определения функции:

```
x = 5
# Сохраняется ссылка, а не значение переменной x!!!
func = lambda: x
x = 80                          # Изменили значение
print(func())                   # Выведет: 80, а не 5
```

Если необходимо сохранить именно текущее значение переменной, то можно воспользоваться способом, приведенным в листинге 11.27.

Листинг 11.27. Сохранение значения переменной

```
x = 5
# Сохраняется значение переменной x
func = (lambda y: lambda: y)(x)
x = 80                                  # Изменили значение
print(func())                           # Выведет: 5
```

Обратите внимание на третью строку примера. В ней мы определили анонимную функцию с одним параметром, возвращающую ссылку на вложенную анонимную функцию. Далее мы вызываем первую функцию с помощью круглых скобок и передаем ей значение переменной `x`. В результате сохраняется текущее значение переменной, а не ссылка на нее.

Сохранить текущее значение переменной можно также, указав глобальную переменную в качестве значения параметра по умолчанию в определении функции (листинг 11.28).

Листинг 11.28. Сохранение значения с помощью параметра по умолчанию

```
x = 5
# Сохраняется значение переменной x
func = lambda x=x: x
x = 80                                   # Изменили значение
print(func())                            # Выведет: 5
```

Получить все идентификаторы и их значения позволяют следующие функции:

- `globals()` — возвращает словарь с глобальными идентификаторами;
- `locals()` — возвращает словарь с локальными идентификаторами. Пример:

```
def func():
    local1 = 54
    glob2 = 25
    print("Глобальные идентификаторы внутри функции")
    print(sorted(globals().keys()))
    print("Локальные идентификаторы внутри функции")
    print(sorted(locals().keys()))
glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(globals().keys()))
```

Результат выполнения:

```
Глобальные идентификаторы внутри функции
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
 '__package__', 'func', 'glob1', 'glob2']
Локальные идентификаторы внутри функции
['glob2', 'local1']
Глобальные идентификаторы вне функции
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
 '__package__', 'func', 'glob1', 'glob2']
```

- `vars([<Объект>])` — если вызывается без параметра внутри функции, то возвращает словарь с локальными идентификаторами. Если вызывается без параметра вне функции, то возвращает словарь с глобальными идентификаторами. При указании объекта в качестве параметра возвращает идентификаторы этого объекта (эквивалентно вызову `<Объект>.__dict__`). Пример:

```
def func():
    local1 = 54
    glob2 = 25
    print("Локальные идентификаторы внутри функции")
    print(sorted(vars().keys()))
glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(vars().keys()))
print("Указание объекта в качестве параметра")
print(sorted(vars(dict).keys()))
print("Альтернативный вызов")
print(sorted(dict.__dict__.keys()))
```

11.10. Вложенные функции

Одну функцию можно вложить в другую функцию, причем уровень вложенности неограничен. В этом случае вложенная функция получает свою собственную локальную область видимости и имеет доступ к идентификаторам внутри функции-родителя. Рассмотрим вложение функций на примере (листинг 11.29).

Листинг 11.29. Вложенные функции

```
def func1(x):
    def func2():
        print(x)
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                # Выведет: 10
f2()                # Выведет: 99
```

В этом примере мы определили функцию `func1()`, принимающую один параметр. Внутри функции `func1()` определена функция `func2()`. Результатом выполнения функции `func1()` будет ссылка на вложенную функцию. Внутри функции `func2()` мы производим вывод значения переменной `x`, которая является локальной в функции `func1()`. Таким образом, помимо локальной, глобальной и встроенной областей видимости добавляется *вложенная область видимости*. При этом поиск идентификаторов вначале производится внутри вложенной функции, затем внутри функции-родителя, далее в функциях более высокого уровня и лишь потом в глобальной и встроенных областях видимости. В нашем примере переменная `x` будет найдена в области видимости функции `func1()`.

Следует учитывать, что сохраняются лишь ссылки на переменные, а не их значения в момент определения функции. Например, если после определения функции `func2()` произвести изменение переменной `x`, то будет использоваться это новое значение:

```
def func1(x):
    def func2():
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                  # Выведет: 30
f2()                  # Выведет: 30
```

Обратите внимание на результат выполнения. В обоих случаях мы получили значение `30`. Если необходимо сохранить именно значение переменной при определении вложенной функции, следует передать значение как значение по умолчанию:

```
def func1(x):
    def func2(x=x): # Сохраняем текущее значение, а не ссылку
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                  # Выведет: 10
f2()                  # Выведет: 99
```

Теперь попробуем изменить значение переменной `x`, объявленной внутри функции `func1()`, из вложенной функции `func2()`. Если внутри функции `func2()` присвоить значение переменной `x`, то будет создана новая локальная переменная с таким же именем. Если внутри функции `func2()` объявить переменную как глобальную и присвоить ей значение, то изменится значение глобальной переменной, а не значение переменной `x` внутри функции `func1()`. Таким образом, ни один из изученных ранее способов не позволяет изменить значение переменной внутри функции-родителя из вложенной функции. Чтобы решить эту проблему, в Python 3 было введено ключевое слово `nonlocal` (листинг 11.30).

Листинг 11.30. Ключевое слово `nonlocal`

```
def func1(a):
    x = a
```

```
    def func2(b):
        nonlocal x    # Объявляем переменную как nonlocal
        print(x)
        x = b         # Можем изменить значение x в func1()
    return func2

f = func1(10)
f(5)                   # Выведет: 10
f(12)                  # Выведет: 5
f(3)                   # Выведет: 12
```

При использовании ключевого слова `nonlocal` следует помнить, что переменная обязательно должна существовать внутри функции-родителя. В противном случае будет выведено сообщение об ошибке.

11.11. Аннотации функций

В Python 3 функции могут содержать аннотации, которые вводят новый способ документирования. Теперь в заголовке функции допускается указывать предназначение каждого параметра и данные какого типа он может принимать, а также тип возвращаемого функцией значения. Аннотации имеют следующий формат:

```
def <Имя функции>(
    [<Параметр1>[: <Выражение>] [= <Значение по умолчанию>][, ...,
     <ПараметрN>[: <Выражение>] [= <Значение по умолчанию>]]]
    ) -> <Возвращаемое значение>:
    <Тело функции>
```

В параметрах `<Выражение>` и `<Возвращаемое значение>` можно указать любое допустимое выражение языка Python. Это выражение будет выполнено при создании функции. Пример указания аннотаций:

```
def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
    print(a, b)
```

В этом примере для переменной `a` создано описание `"Параметр1"`. Для переменной `b` описанием является выражение `10 + 5`, а число `3` — значением параметра по умолчанию. Кроме того, после закрывающей круглой скобки указан тип возвращаемого функцией значения (`None`). После создания функции все выражения будут выполнены и результаты сохранятся в виде словаря в атрибуте `__annotations__` объекта функции. В качестве примера выведем значение атрибута:

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3) -> None: pass

>>> func.__annotations__
{'a': 'Параметр1', 'b': 15, 'return': None}
```

ГЛАВА 12

Модули и пакеты

Модулем в языке Python называется любой файл с программой. Каждый модуль может импортировать другой модуль, получая, таким образом, доступ к идентификаторам внутри импортированного модуля. Следует заметить, что импортируемый модуль может содержать программу не только на языке Python. Например, можно импортировать скомпилированный модуль, написанный на языке C.

Все программы, которые мы запускали ранее, были расположены в модуле с названием `"__main__"`. Получить имя модуля позволяет предопределенный атрибут `__name__`. Атрибут `__name__` для запускаемого модуля содержит значение `"__main__"`, а для импортируемого модуля — его имя. Выведем название модуля:

```
print(__name__)                    # Выведет: __main__
```

Проверить, является модуль главной программой или импортированным модулем, позволяет код, приведенный в листинге 12.1.

Листинг 12.1. Проверка способа запуска модуля

```
if __name__ == "__main__":
    print("Это главная программа")
else:
    print("Импортированный модуль")
```

12.1. Инструкция *import*

Импортировать модуль позволяет инструкция `import`. Мы уже не раз использовали эту инструкцию для подключения встроенных модулей. Например, подключали модуль `time` для получения текущей даты с помощью функции `strftime()`:

```
import time                            # Импортируем модуль time
print(time.strftime("%d.%m.%Y"))       # Выводим текущую дату
```

Инструкция `import` имеет следующий формат:

```
import <Название модуля 1> [as <Псевдоним 1>][, ...,
       <Название модуля N> [as <Псевдоним N>]]
```

После ключевого слова `import` указывается название модуля. Обратите внимание на то, что название не должно содержать расширения и пути к файлу. При именовании модулей необ-

ходимо учитывать, что операция импорта создает одноименный идентификатор. Это означает, что название модуля должно полностью соответствовать правилам именований переменных. Можно создать модуль с именем, начинающимся с цифры, но подключить такой модуль будет нельзя. Кроме того, следует избегать совпадения с ключевыми словами, встроенными идентификаторами и названиями модулей, входящих в стандартную библиотеку.

За один раз можно импортировать сразу несколько модулей, перечислив их через запятую. В качестве примера подключим модули `time` и `math` (листинг 12.2).

Листинг 12.2. Подключение нескольких модулей сразу

```
import time, math                # Импортируем несколько модулей сразу
print(time.strftime("%d.%m.%Y")) # Текущая дата
print(math.pi)                   # Число pi
```

После импортирования модуля его название становится идентификатором, через который можно получить доступ к атрибутам, определенным внутри модуля. Доступ к атрибутам модуля осуществляется с помощью точечной нотации. Например, обратиться к константе `pi`, расположенной внутри модуля `math`, можно так:

```
math.pi
```

Функция `getattr()` позволяет получить значение атрибута модуля по его названию, заданному в виде строки. С помощью этой функции можно сформировать название атрибута динамически во время выполнения программы. Формат функции:

```
getattr(<Объект модуля>, <Атрибут>[, <Значение по умолчанию>])
```

Если указанный атрибут не найден, возбуждается исключение `AttributeError`. Чтобы избежать вывода сообщения об ошибке, можно в третьем параметре указать значение, которое будет возвращаться, если атрибут не существует. Пример использования функции приведен в листинге 12.3.

Листинг 12.3. Пример использования функции `getattr()`

```
import math
print(getattr(math, "pi"))       # Число pi
print(getattr(math, "x", 50))    # Число 50, т. к. x не существует
```

Проверить существование атрибута позволяет функция `hasattr(<Объект>, <Название атрибута>)`. Если атрибут существует, функция возвращает значение `True`. Напишем функцию проверки существования атрибута в модуле `math` (листинг 12.4).

Листинг 12.4. Проверка существования атрибута

```
import math
def hasattr_math(attr):
    if hasattr(math, attr):
        return "Атрибут существует"
    else:
        return "Атрибут не существует"
print(hasattr_math("pi"))       # Атрибут существует
print(hasattr_math("x"))        # Атрибут не существует
```

Если название модуля является слишком длинным и его неудобно указывать каждый раз для доступа к идентификаторам внутри модуля, то можно создать псевдоним. Псевдоним задается после ключевого слова `as`. Создадим псевдоним для модуля `math` (листинг 12.5).

Листинг 12.5. Использование псевдонимов

```
import math as m                # Создание псевдонима
print(m.pi)                     # Число pi
```

Теперь доступ к атрибутам модуля `math` может осуществляться только с помощью идентификатора `m`. Идентификатор `math` в этом случае использовать уже нельзя.

Все идентификаторы внутри импортированного модуля доступны только через идентификатор, указанный в инструкции `import`. Это означает, что любая глобальная переменная на самом деле является глобальной переменной модуля. По этой причине модули часто используются как пространства имен. В качестве примера создадим модуль под названием tests.py, в котором определим переменную `x` (листинг 12.6).

Листинг 12.6. Содержимое модуля tests.py

```
# -*- coding: utf-8 -*-
x = 50
```

В основной программе также определим переменную `x`, но с другим значением. Затем подключим файл tests.py и выведем значения переменных (листинг 12.7).

Листинг 12.7. Содержимое основной программы

```
# -*- coding: utf-8 -*-
import tests            # Подключаем файл tests.py
x = 22
print(tests.x)          # Значение переменной x внутри модуля
print(x)                # Значение переменной x в основной программе
input()
```

Оба файла размещаем в одной папке, а затем запускаем файл с основной программой с помощью двойного щелчка на значке файла. Как видно из результата, никакого конфликта имен нет, т. к. одноименные переменные расположены в разных пространствах имен.

Обратите внимание на содержимое папки с файлами после подключения модуля tests.py. Внутри папки автоматически был создан каталог `__pycache__` с файлом tests.cpython-32.pyc. Этот файл содержит скомпилированный байт-код одноименного модуля. Байт-код создается при первом импортировании модуля и изменяется только после изменения кода внутри модуля. При всех последующих подключениях модуля tests.py будет исполняться код из файла tests.cpython-32.pyc. Следует заметить, что для импортирования модуля достаточно иметь только файл с расширением pyc. Для примера переименуйте файл tests.py (например, в tests1.py), скопируйте файл tests.cpython-32.pyc из папки `__pycache__` в папку с основной программой и переименуйте его в tests.pyc, а затем запустите основную программу. Программа будет нормально выполняться. Таким образом, чтобы скрыть исходный код модулей, можно поставлять программу клиентам только с файлами, имеющими расширение pyc.

Примечание

Каталог __pycache__ создается только в версии Python 3.2. В предыдущих версиях файл с расширением pyc создавался в папке модуля и назывался так же, как и модуль. Например, если модуль имеет название tests.py, то скомпилированный байт-код модуля сохраняется в файле tests.pyc.

Существует еще одно обстоятельство, на которое следует обратить особое внимание. Импортирование модуля выполняется только при первом вызове инструкции `import` (или `from`). При каждом вызове инструкции `import` проверяется наличие объекта модуля в словаре `modules` из модуля `sys`. Если ссылка на модуль находится в этом словаре, то модуль повторно импортироваться не будет. В качестве примера выведем ключи словаря `modules`, предварительно отсортировав их (листинг 12.8).

Листинг 12.8. Вывод ключей словаря `modules`

```
# -*- coding: utf-8 -*-
import tests, sys                  # Подключаем модули tests и sys
print(sorted(sys.modules.keys()))
input()
```

Инструкция `import` требует явного указания объекта модуля. Например, передать название модуля в виде строки нельзя. Чтобы подключить модуль, название которого создается динамически в зависимости от определенных условий, следует воспользоваться функцией `__import__()`. В качестве примера подключим модуль tests.py с помощью функции `__import__()` (листинг 12.9).

Листинг 12.9. Использование функции `__import__()`

```
# -*- coding: utf-8 -*-
s = "test" + "s"             # Динамическое создание названия модуля
m = __import__(s)            # Подключение модуля tests
print(m.x)                   # Вывод значения атрибута x
input()
```

Получить список всех идентификаторов внутри модуля позволяет функция `dir()`. Кроме того, можно воспользоваться словарем `__dict__`, который содержит все идентификаторы и их значения (листинг 12.10).

Листинг 12.10. Вывод списка всех идентификаторов

```
# -*- coding: utf-8 -*-
import tests
print(dir(tests))
print(sorted(tests.__dict__.keys()))
input()
```

12.2. Инструкция *from*

Для импортирования определенных идентификаторов из модуля можно воспользоваться инструкцией `from`.

Инструкция имеет несколько форматов:

```
from <Название модуля> import <Идентификатор 1> [as <Псевдоним 1>]
                        [, ..., <Идентификатор N> [as <Псевдоним N>]]
from <Название модуля> import (<Идентификатор 1> [as <Псевдоним 1>],
                          [..., <Идентификатор N> [as <Псевдоним N>]])
from <Название модуля> import *
```

Первые два формата позволяют импортировать модуль и сделать доступными только указанные идентификаторы. Для длинных имен можно назначить псевдоним, указав его после ключевого слова `as`. В качестве примера сделаем доступными константу `pi` и функцию `floor()` из модуля `math`, а для названия функции создадим псевдоним (листинг 12.11).

Листинг 12.11. Инструкция `from`

```
# -*- coding: utf-8 -*-
from math import pi, floor as f
print(pi)                                # Вывод числа pi
# Вызываем функцию floor() через идентификатор f
print(f(5.49))                           # Выведет: 5
input()
```

Идентификаторы можно разместить на нескольких строках, указав их названия через запятую внутри круглых скобок:

```
from math import (pi, floor,
                  sin, cos)
```

Третий формат инструкции `from` позволяет импортировать все идентификаторы из модуля. Для примера импортируем все идентификаторы из модуля `math` (листинг 12.12).

Листинг 12.12. Импорт всех идентификаторов из модуля

```
# -*- coding: utf-8 -*-
from math import *    # Импортируем все идентификаторы из модуля math
print(pi)             # Вывод числа pi
print(floor(5.49))    # Вызываем функцию floor()
input()
```

Следует заметить, что идентификаторы, названия которых начинаются с символа подчеркивания, импортированы не будут. Кроме того, необходимо учитывать, что импортирование всех идентификаторов из модуля может нарушить пространство имен главной программы, т. к. идентификаторы, имеющие одинаковые имена, будут перезаписаны. Создадим два модуля и подключим их с помощью инструкций `from` и `import`. Содержимое файла module1.py приведено в листинге 12.13.

Листинг 12.13. Содержимое файла module1.py

```
# -*- coding: utf-8 -*-
s = "Значение из модуля module1"
```

Содержимое файла module2.py приведено в листинге 12.14.

Листинг 12.14. Содержимое файла module2.py

```
# -*- coding: utf-8 -*-
s = "Значение из модуля module2"
```

Исходный код основной программы приведен в листинге 12.15.

Листинг 12.15. Содержимое основной программы

```
# -*- coding: utf-8 -*-
from module1 import *
from module2 import *
import module1, module2
print(s)                  # Выведет: "Значение из модуля module2"
print(module1.s)          # Выведет: "Значение из модуля module1"
print(module2.s)          # Выведет: "Значение из модуля module2"
input()
```

Размещаем все файлы в одной папке, а затем запускаем основную программу с помощью двойного щелчка на значке файла. Итак, в обоих модулях определена переменная с именем `s`. При импортировании всех идентификаторов значением переменной `s` будет значение из модуля, который был импортирован последним. В нашем случае это значение из модуля module2.py. Получить доступ к обеим переменным можно только при использовании инструкции `import`. Благодаря точечной нотации пространство имен не нарушается.

В атрибуте `__all__` можно указать список идентификаторов, которые будут импортироваться с помощью выражения `from module import *`. Идентификаторы внутри списка указываются в виде строки. Создадим файл module1.py (листинг 12.16).

Листинг 12.16. Использование атрибута `__all__`

```
# -*- coding: utf-8 -*-
x, y, z, _s = 10, 80, 22, "Строка"
__all__ = ["x", "_s"]
```

Затем подключим его к основной программе (листинг 12.17).

Листинг 12.17. Содержимое основной программы

```
# -*- coding: utf-8 -*-
from module1 import *
print(sorted(vars().keys())) # Получаем список всех идентификаторов
input()
```

После запуска основной программы (с помощью двойного щелчка на значке файла) получим следующий результат:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
 '__package__', '_s', 'x']
```

Как видно из примера, были импортированы только переменные `_s` и `x`. Если бы мы не указали идентификаторы внутри списка `__all__`, то результат был бы другим:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
 '__package__', 'x', 'y', 'z']
```

Обратите внимание на то, что переменная `_s` в этом случае не копируется из модуля, т. к. ее имя начинается с символа подчеркивания.

12.3. Пути поиска модулей

До сих пор мы размещали модули в одной папке с исполняемым файлом. В этом случае нет необходимости настраивать пути поиска модулей, т. к. папка с исполняемым файлом автоматически добавляется в начало списка путей. Получить полный список путей поиска позволяет следующий код:

```
>>> import sys              # Подключаем модуль sys
>>> sys.path                # path содержит список путей поиска модулей
```

Список `sys.path` содержит пути поиска, получаемые из следующих источников:

- путь к текущему каталогу с исполняемым файлом;
- значение переменной окружения `PYTHONPATH`. Для добавления переменной в меню **Пуск** выбираем пункт **Панель управления** (или **Настройка | Панель управления**). В открывшемся окне выбираем пункт **Система**. Переходим на вкладку **Дополнительно** и нажимаем кнопку **Переменные среды**. В разделе **Переменные среды пользователя** нажимаем кнопку **Создать**. В поле **Имя переменной** вводим "PYTHONPATH", а в поле **Значение переменной** задаем пути к папкам с модулями через точку с запятой, например, `C:\folder1;C:\folder2`. После этого изменения перезагружать компьютер не нужно, достаточно заново запустить программу;
- пути поиска стандартных модулей;
- содержимое файлов с расширением pth, расположенных в каталогах поиска стандартных модулей, например в каталоге C:\Python32\Lib\site-packages. Название файла может быть произвольным, главное, чтобы расширение файла было pth. Каждый путь (абсолютный или относительный) должен быть расположен на отдельной строке. В качестве примера создайте файл mypath.pth в каталоге C:\Python32\Lib\site-packages со следующим содержимым:

  ```
  # Это комментарий
  C:\folder1
  C:\folder2
  ```

  Обратите внимание на то, что каталоги должны существовать, в противном случае они не будут добавлены в список `sys.path`.

При поиске модуля список `sys.path` просматривается слева направо. Поиск прекращается после первого найденного модуля. Таким образом, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, то будет использоваться модуль из папки C:\folder1, т. к. он расположен первым в списке путей поиска.

Список `sys.path` можно изменять из программы с помощью списковых методов. Например, добавить каталог в конец списка можно с помощью метода `append()`, а в начало списка — с помощью метода `insert()` (листинг 12.18).

Листинг 12.18. Изменение списка путей поиска модулей

```
# -*- coding: utf-8 -*-
import sys
sys.path.append(r"C:\folder1")          # Добавляем в конец списка
sys.path.insert(0, r"C:\folder2")       # Добавляем в начало списка
print(sys.path)
input()
```

В этом примере мы добавили папку C:\folder2 в начало списка. Теперь, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, то будет использоваться модуль из папки C:\folder2, а не из C:\folder1, как в предыдущем примере.

Обратите внимание на символ `r` перед открывающей кавычкой. В этом режиме специальные последовательности символов не интерпретируются. Если используются обычные строки, то необходимо удвоить каждый слэш в пути:

```
sys.path.append("C:\\folder1\\folder2\\folder3")
```

12.4. Повторная загрузка модулей

Как вы уже знаете, модуль загружается только один раз при первой операции импорта. Все последующие операции импортирования этого модуля будут возвращать уже загруженный объект модуля, даже если сам модуль был изменен. Чтобы повторно загрузить модуль, следует воспользоваться функцией `reload()` из модуля `imp`. Формат функции:

```
from imp import reload
reload(<Объект модуля>)
```

В качестве примера создадим модуль tests.py со следующим содержимым:

```
# -*- coding: utf-8 -*-
x = 150
```

Подключим этот модуль в окне **Python Shell** редактора IDLE и выведем текущее значение переменной `x`:

```
>>> import sys
>>> sys.path.append(r"C:\book") # Добавляем путь к папке с модулем
>>> import tests                # Подключаем модуль tests.py
>>> print(tests.x)              # Выводим текущее значение
150
```

Не закрывая окно **Python Shell**, изменим значение переменной `x` на `800`, а затем попробуем заново импортировать модуль и вывести текущее значение переменной:

```
>>> # Изменяем значение в модуле на 800
>>> import tests
>>> print(tests.x)              # Значение не изменилось
150
```

Как видно из примера, значение переменной `x` не изменилось. Теперь перезагрузим модуль с помощью функции `reload()` (листинг 12.19).

Листинг 12.19. Повторная загрузка модуля

```
>>> from imp import reload
>>> reload(tests)                # Перезагружаем модуль
<module 'tests' from 'C:\book\tests.py'>
>>> print(tests.x)               # Значение изменилось
800
```

При использовании функции `reload()` следует учитывать, что идентификаторы, импортированные с помощью инструкции `from`, перезагружены не будут. Кроме того, повторно не загружаются скомпилированные модули, написанные на других языках программирования, например языке C.

12.5. Пакеты

Пакетом называется каталог с модулями, в котором расположен файл инициализации __init__.py. Файл инициализации может быть пустым или содержать код, который будет выполнен при первом обращении к пакету. В любом случае он обязательно должен присутствовать внутри каталога с модулями.

В качестве примера создадим следующую структуру файлов и каталогов:

```
main.py                 # Основной файл с программой
folder1\                # Папка на одном уровне вложенности с main.py
    __init__.py         # Файл инициализации
    module1.py          # Модуль folder1\module1.py
    folder2\            # Вложенная папка
        __init__.py     # Файл инициализации
        module2.py      # Модуль folder1\folder2\module2.py
        module3.py      # Модуль folder1\folder2\module3.py
```

Содержимое файлов __init__.py приведено в листинге 12.20.

Листинг 12.20. Содержимое файлов __init__.py

```
# -*- coding: utf-8 -*-
print("__init__ из", __name__)
```

Содержимое модулей module1.py, module2.py и module3.py приведено в листинге 12.21.

Листинг 12.21. Содержимое модулей module1.py, module2.py и module3.py

```
# -*- coding: utf-8 -*-
msg = "Модуль {0}".format(__name__)
```

Теперь импортируем эти модули в основном файле main.py и получим значение переменной `msg` разными способами. Файл main.py будем запускать с помощью двойного щелчка на значке файла. Содержимое файла main.py приведено в листинге 12.22.

Листинг 12.22. Содержимое файла main.py

```
# -*- coding: utf-8 -*-

# Доступ к модулю folder1\module1.py
import folder1.module1 as m1
                        # Выведет: __init__ из folder1
print(m1.msg)           # Выведет: Модуль folder1.module1
from folder1 import module1 as m2
print(m2.msg)           # Выведет: Модуль folder1.module1
from folder1.module1 import msg
print(msg)              # Выведет: Модуль folder1.module1

# Доступ к модулю folder1\folder2\module2.py
import folder1.folder2.module2 as m3
                        # Выведет: __init__ из folder1.folder2
print(m3.msg)           # Выведет: Модуль folder1.folder2.module2
from folder1.folder2 import module2 as m4
print(m4.msg)           # Выведет: Модуль folder1.folder2.module2
from folder1.folder2.module2 import msg
print(msg)              # Выведет: Модуль folder1.folder2.module2

input()
```

Как видно из примера, пакеты позволяют распределить модули по каталогам. Чтобы импортировать модуль, расположенный во вложенном каталоге, необходимо указать путь к нему, перечислив имена каталогов через точку. Если модуль расположен в каталоге C:\folder1\folder2\, то путь к нему из C:\ будет выглядеть так: `folder1.folder2`. При использовании инструкции `import` путь к модулю должен включать не только названия каталогов, но и название модуля без расширения:

```
import folder1.folder2.module2
```

Получить доступ к идентификаторам внутри импортированного модуля можно следующим образом:

```
print(folder1.folder2.module2.msg)
```

Так как постоянно указывать такой длинный идентификатор очень неудобно, можно создать псевдоним, указав его после ключевого слова `as`, и обращаться к идентификаторам модуля через него:

```
import folder1.folder2.module2 as m
print(m.msg)
```

При использовании инструкции `from` можно импортировать как объект модуля, так и определенные идентификаторы из модуля. Чтобы импортировать объект модуля, его название следует указать после ключевого слова `import`:

```
from folder1.folder2 import module2
print(module2.msg)
```

Для импортирования только определенных идентификаторов название модуля указывается в составе пути, а после ключевого слова `import` через запятую перечисляются идентификаторы:

```
from folder1.folder2.module2 import msg
print(msg)
```

Если необходимо импортировать все идентификаторы из модуля, то после ключевого слова `import` указывается символ `*`:

```
from folder1.folder2.module2 import *
print(msg)
```

Инструкция `from` позволяет также импортировать сразу несколько модулей из пакета. Для этого внутри файла инициализации __init__.py в атрибуте `__all__` необходимо указать список модулей, которые будут импортироваться с помощью выражения `from` *пакет* `import *`. В качестве примера изменим содержимое файла __init__.py из каталога folder1\folder2\:

```
# -*- coding: utf-8 -*-
__all__ = ["module2", "module3"]
```

Теперь изменим содержимое основного файла main.py (листинг 12.23) и запустим его.

Листинг 12.23. Содержимое файла main.py

```
# -*- coding: utf-8 -*-
from folder1.folder2 import *
print(module2.msg)          # Выведет: Модуль folder1.folder2.module2
print(module3.msg)          # Выведет: Модуль folder1.folder2.module3
input()
```

Как видно из примера, после ключевого слова `from` указывается лишь путь к каталогу без имени модуля. В результате выполнения инструкции `from` все модули, указанные в списке `__all__`, будут импортированы в пространство имен модуля main.py.

До сих пор мы рассматривали импортирование модулей из основной программы. Теперь рассмотрим импорт модулей внутри пакета. В этом случае инструкция `from` поддерживает относительный импорт модулей. Чтобы импортировать модуль, расположенный в том же каталоге, перед названием модуля указывается точка:

```
from .module import *
```

Чтобы импортировать модуль, расположенный в родительском каталоге, перед названием модуля указываются две точки:

```
from ..module import *
```

Если необходимо обратиться еще уровнем выше, то указываются три точки:

```
from ...module import *
```

Чем выше уровень, тем больше точек необходимо указать. После ключевого слова `from` можно ставить только точки. В этом случае имя модуля вводится после ключевого слова `import`. Пример:

```
from .. import module
```

Рассмотрим относительный импорт на примере. Для этого изменим содержимое модуля module3.py, как показано в листинге 12.24.

Листинг 12.24. Содержимое модуля module3.py

```
# -*- coding: utf-8 -*-

# Импорт модуля module2.py из текущего каталога
from . import module2 as m1
var1 = "Значение из: {0}".format(m1.msg)
from .module2 import msg as m2
var2 = "Значение из: {0}".format(m2)

# Импорт модуля module1.py из родительского каталога
from .. import module1 as m3
var3 = "Значение из: {0}".format(m3.msg)
from ..module1 import msg as m4
var4 = "Значение из: {0}".format(m4)
```

Теперь изменим содержимое основного файла main.py (листинг 12.25) и запустим его с помощью двойного щелчка на значке файла.

Листинг 12.25. Содержимое файла main.py

```
# -*- coding: utf-8 -*-
from folder1.folder2 import module3 as m
print(m.var1)          # Значение из: Модуль folder1.folder2.module2
print(m.var2)          # Значение из: Модуль folder1.folder2.module2
print(m.var3)          # Значение из: Модуль folder1.module1
print(m.var4)          # Значение из: Модуль folder1.module1
input()
```

При импортировании модуля внутри пакета с помощью инструкции `import` важно помнить, что в Python 3 производится *абсолютный импорт*. Если при запуске с помощью двойного щелчка на значке файла автоматически добавляется путь к каталогу с исполняемым файлом, то при импорте внутри пакета этого не происходит. Поэтому если изменить содержимое модуля module3.py показанным ниже способом, то мы получим сообщение об ошибке или загрузим совсем другой модуль:

```
# -*- coding: utf-8 -*-
import module2       # Ошибка! Поиск модуля по абсолютному пути
var1 = "Значение из: {0}".format(module2.msg)
var2 = var3 = var4 = 0
```

В этом примере мы попытались импортировать модуль module2.py из модуля module3.py. При этом с помощью двойного щелчка запускаем файл main.py (см. листинг 12.25). Так как импорт внутри пакета выполняется по абсолютному пути, поиск модуля module2.py не будет производиться в папке folder1\folder2\. В результате модуль не будет найден. Если в путях поиска модулей находится модуль с таким же именем, то будет импортирован модуль, который мы и не предполагали подключать.

Чтобы подключить модуль, расположенный в той же папке внутри пакета, необходимо воспользоваться относительным импортом с помощью инструкции `from`:

```
from . import module2
```

Или указать полный путь относительно корневого каталога пакета:

```
import folder1.folder2.module2 as module2
```

ГЛАВА 13

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это способ организации программы, позволяющий использовать один и тот же код многократно. В отличие от функций и модулей ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального мира в виде объектов, а также организовать связи между этими объектами.

Основным "кирпичиком" ООП является класс. *Класс* — это объект, включающий набор переменных и функций для управления этими переменными. Переменные называют *атрибутами*, а функции — *методами*. Класс является фабрикой объектов, т. е. позволяет создать неограниченное количество экземпляров, основанных на этом классе.

13.1. Определение класса и создание экземпляра класса

Класс описывается с помощью ключевого слова `class` по следующей схеме:

```
class <Название класса>[(<Класс1>[, ..., <КлассN>])]:
    ["""  Строка документирования """]
    <Описание атрибутов и методов>
```

Инструкция создает новый объект и присваивает ссылку на него идентификатору, указанному после ключевого слова `class`. Это означает, что название класса должно полностью соответствовать правилам именований переменных. После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую. Если класс не наследует базовые классы, то круглые скобки можно не указывать. Следует заметить, что все выражения внутри инструкции `class` выполняются при создании объекта, а не при создании экземпляра класса. В качестве примера создадим класс, внутри которого просто выводится сообщение (листинг 13.1).

Листинг 13.1. Создание определения класса

```
# -*- coding: utf-8 -*-
class MyClass:
    """ Это строка документирования """
    print("Инструкции выполняются сразу")
input()
```

Этот пример содержит только определение класса `MyClass` и не создает экземпляр класса. Как только поток выполнения достигнет инструкции `class`, сообщение, указанное в функции `print()`, будет сразу выведено.

Создание переменной (атрибута) внутри класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, с помощью инструкции `def`. Методам класса в первом параметре автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть именем `self`, хотя это и не обязательно. Доступ к атрибутам и методам класса производится через переменную `self` с помощью точечной нотации. Например, к атрибуту `x` из метода класса можно обратиться так: `self.x`.

Чтобы использовать атрибуты и методы класса, необходимо создать экземпляр класса. Для этого используется следующий синтаксис:

```
<Экземпляр класса> = <Название класса>([<Параметры>])
```

Определим класс `MyClass` с атрибутом `x` и методом `print_x()`, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод (листинг 13.2).

Листинг 13.2. Создание атрибута и метода

```
class MyClass:
    def __init__(self):    # Конструктор
        self.x = 10        # Атрибут экземпляра класса
    def print_x(self):     # self — это ссылка на экземпляр класса
        print(self.x)      # Выводим значение атрибута
c = MyClass()              # Создание экземпляра класса
                           # Вызываем метод print_x()
c.print_x()                # self не указывается при вызове метода
print(c.x)                 # К атрибуту можно обратиться непосредственно
```

При обращении к методам класса используется следующий формат:

```
<Экземпляр класса>.<Имя метода>([<Параметры>])
```

Обратите внимание на то, что при вызове метода не нужно передавать ссылку на экземпляр класса в качестве параметра, как это делается в определении метода внутри класса. Ссылку на экземпляр класса интерпретатор передает автоматически.

Обращение к атрибутам класса осуществляется аналогично:

```
<Экземпляр класса>.<Имя атрибута>
```

Для доступа к атрибутам и методам можно также использовать следующие функции:

- `getattr()` — возвращает значение атрибута по его названию, заданному в виде строки. С помощью этой функции можно сформировать название атрибута динамически во время выполнения программы. Формат функции:

  ```
  getattr(<Объект>, <Атрибут>[, <Значение по умолчанию>])
  ```

  Если указанный атрибут не найден, возбуждается исключение `AttributeError`. Чтобы избежать вывода сообщения об ошибке можно в третьем параметре указать значение, которое будет возвращаться, если атрибут не существует;

- `setattr()` — задает значение атрибута. Название атрибута указывается в виде строки. Формат функции:

  ```
  setattr(<Объект>, <Атрибут>, <Значение>)
  ```

- `delattr(<Объект>, <Атрибут>)` — удаляет указанный атрибут. Название атрибута указывается в виде строки;
- `hasattr(<Объект>, <Атрибут>)` — проверяет наличие указанного атрибута. Если атрибут существует, функция возвращает значение `True`.

Продемонстрируем работу функций на примере (листинг 13.3).

Листинг 13.3. Функции `getattr()`, `setattr()` и `hasattr()`

```
class MyClass:
    def __init__(self):
        self.x = 10
    def get_x(self):
        return self.x
c = MyClass()                   # Создаем экземпляр класса
print(getattr(c, "x"))          # Выведет: 10
print(getattr(c, "get_x")())    # Выведет: 10
print(getattr(c, "y", 0))       # Выведет: 0, т. к. атрибут не найден
setattr(c, "y", 20)             # Создаем атрибут y
print(getattr(c, "y", 0))       # Выведет: 20
delattr(c, "y")                 # Удаляем атрибут y
print(getattr(c, "y", 0))       # Выведет: 0, т. к. атрибут не найден
print(hasattr(c, "x"))          # Выведет: True
print(hasattr(c, "y"))          # Выведет: False
```

Все атрибуты класса в языке Python являются открытыми (`public`), т. е. доступны для непосредственного изменения. Кроме того, атрибуты можно создавать динамически после создания класса. Можно создать как атрибут объекта класса, так и атрибут экземпляра класса. Рассмотрим это на примере (листинг 13.4).

Листинг 13.4. Атрибуты объекта класса и экземпляра класса

```
class MyClass:                  # Определяем пустой класс
    pass
MyClass.x = 50                  # Создаем атрибут объекта класса
c1, c2 = MyClass(), MyClass()   # Создаем два экземпляра класса
c1.y = 10                       # Создаем атрибут экземпляра класса
c2.y = 20                       # Создаем атрибут экземпляра класса
print(c1.x, c1.y)               # Выведет: 50 10
print(c2.x, c2.y)               # Выведет: 50 20
```

В этом примере мы определяем пустой класс, разместив в нем оператор `pass`. Далее создаем атрибут объекта класса (`x`). Этот атрибут будет доступен всем создаваемым экземплярам класса. Затем создаем два экземпляра класса и добавляем одноименные атрибуты (`y`). Значение этого атрибута будет разным в каждом экземпляре класса. Если создать новый экземпляр (например, `c3`), то атрибут `y` в нем определен не будет. Таким образом, с помощью классов можно имитировать типы данных, определенные в других языках программирования, например тип `struct` в языке C.

Очень важно понимать разницу между атрибутами объекта класса и атрибутами экземпляра класса. Атрибут объекта класса доступен всем экземплярам класса, но после изменения ат-

рибута значение изменится во всех экземплярах класса. Атрибут экземпляра класса может хранить уникальное значение для каждого экземпляра. Изменение атрибута экземпляра не затронет значение одноименного атрибута в других экземплярах. Рассмотрим это на примере. Создадим класс с атрибутом объекта класса (x) и атрибутом экземпляра класса (y):

```
class MyClass:
    x = 10                      # Атрибут объекта класса
    def __init__(self):
        self.y = 20             # Атрибут экземпляра класса
```

Теперь создадим два экземпляра этого класса:

```
c1 = MyClass()                  # Создаем экземпляр класса
c2 = MyClass()                  # Создаем экземпляр класса
```

Выведем значения атрибута x, а затем изменим значение и опять произведем вывод:

```
print(c1.x, c2.x)               # 10 10
MyClass.x = 88                  # Изменяем атрибут объекта класса
print(c1.x, c2.x)               # 88 88
```

Как видно из примера, изменение атрибута объекта класса затронуло значение в двух экземплярах класса сразу. Теперь произведем аналогичную операцию с атрибутом y:

```
print(c1.y, c2.y)               # 20 20
c1.y = 88                       # Изменяем атрибут экземпляра класса
print(c1.y, c2.y)               # 88 20
```

В этом случае изменилось значение атрибута только в экземпляре c1.

Следует также учитывать, что в одном классе могут одновременно существовать атрибут объекта и атрибут экземпляра с одним именем. Изменение атрибута объекта класса мы производили следующим образом:

```
MyClass.x = 88                  # Изменяем атрибут объекта класса
```

Если после этой инструкции вставить инструкцию

```
c1.x = 200                      # Создаем атрибут экземпляра
```

то будет создан атрибут экземпляра класса, а не изменено значение атрибута объекта класса. Чтобы увидеть разницу, выведем значения атрибута:

```
print(c1.x, MyClass.x)          # 200 88
```

13.2. Методы *__init__()* и *__del__()*

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации __init__(). В других языках программирования такой метод принято называть *конструктором класса*. Формат метода:

```
def __init__(self[, <Значение1>[, ..., <ЗначениеN>]]):
    <Инструкции>
```

С помощью метода __init__() можно присвоить начальные значения атрибутам класса. При создании экземпляра класса начальные значения указываются после имени класса в круглых скобках:

```
<Экземпляр класса> = <Имя класса>([<Значение1>[, ..., <ЗначениеN>]])
```

Пример использования метода `__init__()` приведен в листинге 13.5.

Листинг 13.5. Метод `__init__()`

```
class MyClass:
    def __init__(self, value1, value2): # Конструктор
        self.x = value1
        self.y = value2
c = MyClass(100, 300)                    # Создаем экземпляр класса
print(c.x, c.y)                          # Выведет: 100 300
```

Если конструктор вызывается при создании объекта, то перед уничтожением объекта автоматически вызывается метод, называемый *деструктором*. В языке Python деструктор реализуется в виде предопределенного метода `__del__()` (листинг 13.6). Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка. Кроме того, т. к. интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке Python не имеет особого смысла.

Листинг 13.6. Метод `__del__()`

```
class MyClass:
    def __init__(self): # Конструктор класса
        print("Вызван метод __init__()")
    def __del__(self):  # Деструктор класса
        print("Вызван метод __del__()")
c1 = MyClass()          # Выведет: Вызван метод __init__()
del c1                  # Выведет: Вызван метод __del__()
c2 = MyClass()          # Выведет: Вызван метод __init__()
c3 = c2                 # Создаем ссылку на экземпляр класса
del c2                  # Ничего не выведет, т. к. существует ссылка
del c3                  # Выведет: Вызван метод __del__()
```

13.3. Наследование

Наследование является, пожалуй, самым главным понятием ООП. Предположим, у нас есть класс (например, `Class1`). При помощи *наследования* мы можем создать новый класс (например, `Class2`), в котором будет доступ ко всем атрибутам и методам класса `Class1`, а также к некоторым новым атрибутам и методам (листинг 13.7).

Листинг 13.7. Наследование

```
class Class1:         # Базовый класс
    def func1(self):
        print("Метод func1() класса Class1")
    def func2(self):
        print("Метод func2() класса Class1")

class Class2(Class1): # Класс Class2 наследует класс Class1
    def func3(self):
        print("Метод func3() класса Class2")
```

```
c = Class2()          # Создаем экземпляр класса Class2
c.func1()             # Выведет: Метод func1() класса Class1
c.func2()             # Выведет: Метод func2() класса Class1
c.func3()             # Выведет: Метод func3() класса Class2
```

Как видно из примера, класс `Class1` указывается внутри круглых скобок в определении класса `Class2`. Таким образом, класс `Class2` наследует все атрибуты и методы класса `Class1`. Класс `Class1` называется *базовым классом* или *суперклассом*, а класс `Class2` — *производным классом* или *подклассом*.

Если имя метода в классе `Class2` совпадает с именем метода класса `Class1`, то будет использоваться метод из класса `Class2`. Чтобы вызвать одноименный метод из базового класса, следует указать перед методом название базового класса. Кроме того, в первом параметре метода необходимо явно указать ссылку на экземпляр класса. Рассмотрим это на примере (листинг 13.8).

Листинг 13.8. Переопределение методов

```
class Class1:                 # Базовый класс
    def __init__(self):
        print("Конструктор базового класса")
    def func1(self):
        print("Метод func1() класса Class1")

class Class2(Class1):         # Класс Class2 наследует класс Class1
    def __init__(self):
        print("Конструктор производного класса")
        Class1.__init__(self) # Вызываем конструктор базового класса
    def func1(self):
        print("Метод func1() класса Class2")
        Class1.func1(self)    # Вызываем метод базового класса

c = Class2()                  # Создаем экземпляр класса Class2
c.func1()                     # Вызываем метод func1()
```

Выведет:

```
Конструктор производного класса
Конструктор базового класса
Метод func1() класса Class2
Метод func1() класса Class1
```

> ***Внимание!***
>
> Конструктор базового класса автоматически не вызывается, если он переопределен в производном классе.

Чтобы вызвать одноименный метод из базового класса, можно также воспользоваться функцией `super()`. Формат функции:

```
super([<Класс>, <Указатель self>])
```

С помощью функции `super()` инструкцию

```
Class1.__init__(self)          # Вызываем конструктор базового класса
```

можно записать так:

```
super().__init__()              # Вызываем конструктор базового класса
```

или так:

```
super(Class2, self).__init__() # Вызываем конструктор базового класса
```

Обратите внимание на то, что при использовании функции `super()` не нужно явно передавать указатель `self` в вызываемый метод. Кроме того, в первом параметре функции `super()` указывается производный класс, а не базовый. Поиск идентификатора будет производиться во всех базовых классах. Результатом поиска станет первый найденный идентификатор в цепочке наследования.

> ***Примечание***
>
> В последних версиях Python 2 существовало два типа классов: "классические" классы и классы нового стиля. Классы нового стиля должны были явно наследовать класс `object`. В Python 3 все классы являются классами нового стиля и неявно наследуют класс `object`. Таким образом, все классы верхнего уровня являются наследниками класса `object`, даже если этот класс не указан в списке наследования. "Классические" классы (в понимании Python 2) в Python 3 больше не поддерживаются.

13.4. Множественное наследование

В определении класса в круглых скобках можно указать сразу несколько базовых классов через запятую. Результатом поиска будет первый найденный идентификатор. Рассмотрим множественное наследование на примере (листинг 13.9).

Листинг 13.9. Множественное наследование

```
class Class1:          # Базовый класс для класса Class2
    def func1(self):
        print("Метод func1() класса Class1")

class Class2(Class1): # Класс Class2 наследует класс Class1
    def func2(self):
        print("Метод func2() класса Class2")

class Class3(Class1): # Класс Class3 наследует класс Class1
    def func1(self):
        print("Метод func1() класса Class3")
    def func2(self):
        print("Метод func2() класса Class3")
    def func3(self):
        print("Метод func3() класса Class3")
    def func4(self):
        print("Метод func4() класса Class3")

class Class4(Class2, Class3): # Множественное наследование
    def func4(self):
        print("Метод func4() класса Class4")
```

```
c = Class4()              # Создаем экземпляр класса Class4
c.func1()                 # Выведет: Метод func1() класса Class3
c.func2()                 # Выведет: Метод func2() класса Class2
c.func3()                 # Выведет: Метод func3() класса Class3
c.func4()                 # Выведет: Метод func4() класса Class4
```

Метод `func1()` определен в двух классах — `Class1` и `Class3`. Так как вначале просматриваются все базовые классы первого уровня, метод `func1()` будет найден в классе `Class3`, а не в классе `Class1`.

Метод `func2()` также определен в двух классах — `Class2` и `Class3`. Так как класс `Class2` стоит первым в списке базовых классов, то метод будет найден именно в этом классе. Чтобы наследовать метод из класса `Class3`, следует указать это явным образом. Переделаем определение класса `Class4` из предыдущего примера и наследуем метод `func2()`из класса `Class3` (листинг 13.10).

Листинг 13.10. Указание класса при наследовании метода

```
class Class4(Class2, Class3): # Множественное наследование
    # Наследуем func2() из класса Class3, а не из класса Class2
    func2 = Class3.func2
    def func4(self):
        print("Метод func4() класса Class4")
```

Метод `func3()` определен только в классе `Class3`, поэтому метод наследуется от этого класса. Метод `func4()`, определенный в классе `Class3`, переопределяется в производном классе. Если метод найден в производном классе, то вся иерархия наследования просматриваться не будет.

Если необходимо получить перечень базовых классов, то можно воспользоваться атрибутом `__bases__`. В качестве значения атрибут возвращает кортеж. В качестве примера выведем базовые классы для всех классов из предыдущего примера:

```
print(Class1.__bases__)
print(Class2.__bases__)
print(Class3.__bases__)
print(Class4.__bases__)
```

Выведет:

```
(<class 'object'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class2'>, <class '__main__.Class3'>)
```

Рассмотрим порядок поиска идентификаторов при сложной иерархии множественного наследования (листинг 13.11).

Листинг 13.11. Поиск идентификаторов при множественном наследовании

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
```

```
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7()
print(c.x)
```

Последовательность поиска атрибута `x` будет такой:

```
Class7 -> Class4 -> Class3 — > Class6 -> Class5 -> Class2 -> Class1
```

Получить всю цепочку наследования позволяет атрибут `__mro__`:

```
print(Class7.__mro__)
```

Результат выполнения:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>,
 <class '__main__.Class3'>, <class '__main__.Class6'>,
 <class '__main__.Class5'>, <class '__main__.Class2'>,
 <class '__main__.Class1'>, <class 'object'>)
```

13.5. Специальные методы

Классы поддерживают следующие специальные методы:

- `__call__()` — позволяет обработать вызов экземпляра класса как вызов функции. Формат метода:

  ```
  __call__(self[, <Параметр1>[, ..., <ПараметрN>]])
  ```

  Пример:

  ```
  class MyClass:
      def __init__(self, m):
          self.msg = m
      def __call__(self):
          print(self.msg)
  c1 = MyClass("Значение1") # Создание экземпляра класса
  c2 = MyClass("Значение2") # Создание экземпляра класса
  c1()                      # Выведет: Значение1
  c2()                      # Выведет: Значение2
  ```

- `__setitem__(self, <Ключ>, <Значение>)` — вызывается в случае присваивания значения по индексу или ключу;

- `__getitem__(self, <Ключ>)` — вызывается при доступе к значению по индексу или ключу. Метод автоматически вызывается при использовании цикла `for`, а также при других операциях, применимых к последовательностям. Пример:

  ```
  class MyClass:
      def __init__(self, a):
          self.arr = a
      def __getitem__(self, index):
          return self.arr[index]
  ```

```
    def __setitem__(self, index, value):
        self.arr[index] = value
c = MyClass( [1, 2, 3, 4, 5] )
print(c[0])             # Выведет: 1
c[0] = 0                # Присваивание по индексу
print(c[0])             # Выведет: 0
for i in c:             # for автоматически вызывает __getitem__()
    print(i, end=" ")
# Выведет: 0 2 3 4 5
print(list(c))                      # Выведет: [0, 2, 3, 4, 5]
print("Есть" if 0 in c else "Нет") # Выведет: Есть
```

- `__delitem__(self, <Ключ>)` — вызывается в случае удаления элемента по индексу или ключу с помощью выражения `del <Экземпляр класса>[<Ключ>]`;
- `__getattr__(self, <Атрибут>)` — вызывается при обращении к несуществующему атрибуту класса. Пример:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print("Вызван метод __getattr__()")
        return 0
c = MyClass()
# Атрибут i существует
print(c.i)     # Выведет: 20. Метод __getattr__() не вызывается
# Атрибут s не существует
print(c.s)     # Выведет: Вызван метод __getattr__() 0
```

- `__getattribute__(self, <Атрибут>)` — вызывается при обращении к любому атрибуту класса. Необходимо учитывать, что использование точечной нотации (для обращения к атрибуту класса) внутри этого метода приведет к зацикливанию. Чтобы избежать зацикливания, следует вызывать метод `__getattribute__()` объекта `object`. Внутри метода нужно вернуть значение атрибута или возбудить исключение `AttributeError`. Пример:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattribute__(self, attr):
        print("Вызван метод __getattribute__()")
        return object.__getattribute__(self, attr) # Только так!!!
c = MyClass()
print(c.i)     # Выведет: Вызван метод __getattribute__() 20
```

- `__setattr__(self, <Атрибут>, <Значение>)` — вызывается при попытке присваивания значения атрибуту экземпляра класса. Если внутри метода необходимо присвоить значение атрибуту, то следует использовать словарь `__dict__`, иначе при точечной нотации метод `__setattr__()` будет вызван повторно, и это приведет к зацикливанию. Пример:

```
class MyClass:
    def __setattr__(self, attr, value):
        print("Вызван метод __setattr__()")
        self.__dict__[attr] = value                # Только так!!!
```

```
c = MyClass()
c.i = 10       # Выведет: Вызван метод __setattr__()
print(c.i)     # Выведет: 10
```

- `__delattr__(self, <Атрибут>)` — вызывается при удалении атрибута с помощью инструкции `del <Экземпляр класса>.<Атрибут>`;
- `__iter__(self)` — если метод определен, то считается, что объект поддерживает итерационный протокол. Если в классе одновременно определены методы `__iter__()` и `__getitem__()`, то предпочтение отдается методу `__iter__()`. Помимо метода `__iter__()` в классе должен быть определен метод `__next__()`, который будет вызываться на каждой итерации. Метод `__next__()` должен возвращать текущее значение или возбуждать исключение `StopIteration`, которое сообщает об окончании итераций. Пример:

```
class MyClass:
    def __init__(self, a):
        self.arr = a
        self.i = 0 # Текущий индекс
    def __iter__(self):
        return self
    def __next__(self):
        if self.i >= len(self.arr):
            self.i = 0              # Устанавливаем в нач. состояние
            raise StopIteration     # Возбуждаем исключение
        else:
            elem = self.arr[self.i]
            self.i += 1
            # Возвращаем элемент по текущему индексу
            return elem
c = MyClass( [1, 2, 3, 4, 5] )
for i in c:
    print(i, end=" ")      # Выведет: 1 2 3 4 5
print(c.__next__())        # Выведет: 1
print(c.__next__())        # Выведет: 2
for i in c:
    print(i, end=" ")      # Выведет: 3 4 5
```

- `__len__(self)` — вызывается при использовании функции `len()`, а также для проверки объекта на логическое значение при отсутствии метода `__bool__()`. Метод должен возвращать положительное целое число. Пример:

```
class MyClass:
    def __len__(self):
        return 50
c = MyClass()
print(len(c))              # Выведет: 50
```

- `__bool__(self)` — вызывается при использовании функции `bool()`;
- `__int__(self)` — вызывается при преобразовании объекта в целое число с помощью функции `int()`;
- `__float__(self)` — вызывается при преобразовании объекта в вещественное число с помощью функции `float()`;

- `__complex__(self)` — вызывается при использовании функции `complex()`;
- `__round__(self, n)` — вызывается при использовании функции `round()`;
- `__index__(self)` — вызывается при использовании функций `bin()`, `hex()` и `oct()`;
- `__repr__(self)` и `__str__(self)` — служат для преобразования объекта в строку. Метод `__repr__()` вызывается при выводе в интерактивной оболочке, а также при использовании функции `repr()`. Метод `__str__()` вызывается при выводе с помощью функции `print()`, а также при использовании функции `str()`. Если метод `__str__()` отсутствует, то будет вызван метод `__repr__()`. В качестве значения методы `__repr__()` и `__str__()` должны возвращать строку. Пример:

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Вызван метод __repr__() {0}".format(self.msg)
    def __str__(self):
        return "Вызван метод __str__() {0}".format(self.msg)
c = MyClass("Значение")
print(repr(c)) # Выведет: Вызван метод __repr__() Значение
print(str(c))  # Выведет: Вызван метод __str__() Значение
print(c)       # Выведет: Вызван метод __str__() Значение
```

- `__hash__(self)` — этот метод следует перегрузить, если экземпляр класса планируется использовать в качестве ключа словаря или внутри множества. Пример:

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __hash__(self):
        return hash(self.x)
c = MyClass(10)
d = {}
d[c] = "Значение"
print(d[c]) # Выведет: Значение
```

13.6. Перегрузка операторов

Перегрузка операторов позволяет экземплярам классов участвовать в обычных операциях. Чтобы перегрузить оператор, необходимо в классе определить метод со специальным названием. Для перегрузки математических операторов используются следующие методы:

- `x + y` — сложение — `x.__add__(y)`;
- `y + x` — сложение (экземпляр класса справа) — `x.__radd__(y)`;
- `x += y` — сложение и присваивание — `x.__iadd__(y)`;
- `x - y` — вычитание — `x.__sub__(y)`;
- `y - x` — вычитание (экземпляр класса справа) — `x.__rsub__(y)`;
- `x -= y` — вычитание и присваивание — `x.__isub__(y)`;
- `x * y` — умножение — `x.__mul__(y)`;

- `y * x` — умножение (экземпляр класса справа) — `x.__rmul__(y)`;
- `x *= y` — умножение и присваивание — `x.__imul__(y)`;
- `x / y` — деление — `x.__truediv__(y)`;
- `y / x` — деление (экземпляр класса справа) — `x.__rtruediv__(y)`;
- `x /= y` — деление и присваивание — `x.__itruediv__(y)`;
- `x // y` — деление с округлением вниз — `x.__floordiv__(y)`;
- `y // x` — деление с округлением вниз (экземпляр класса справа) — `x.__rfloordiv__(y)`;
- `x //= y` — деление с округлением вниз и присваивание — `x.__ifloordiv__(y)`;
- `x % y` — остаток от деления — `x.__mod__(y)`;
- `y % x` — остаток от деления (экземпляр класса справа) — `x.__rmod__(y)`;
- `x %= y` — остаток от деления и присваивание — `x.__imod__(y)`;
- `x ** y` — возведение в степень — `x.__pow__(y)`;
- `y ** x` — возведение в степень (экземпляр класса справа) — `x.__rpow__(y)`;
- `x **= y` — возведение в степень и присваивание — `x.__ipow__(y)`;
- `-x` — унарный - (минус) — `x.__neg__()`;
- `+x` — унарный + (плюс) — `x.__pos__()`;
- `abs(x)` — абсолютное значение — `x.__abs__()`.

Пример перегрузки математических операторов приведен в листинге 13.12.

Листинг 13.12. Пример перегрузки математических операторов

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __add__(self, y):            # Перегрузка оператора +
        print("Экземпляр слева")
        return self.x + y
    def __radd__(self, y):           # Перегрузка оператора +
        print("Экземпляр справа")
        return self.x + y
    def __iadd__(self, y):           # Перегрузка оператора +=
        print("Сложение с присваиванием")
        self.x += y
        return self
c = MyClass(50)
print(c + 10)                  # Выведет: Экземпляр слева 60
print(20 + c)                  # Выведет: Экземпляр справа 70
c += 30                        # Выведет: Сложение с присваиванием
print(c.x)                     # Выведет: 80
```

Методы перегрузки двоичных операторов:

- `~x` — двоичная инверсия — `x.__invert__()`;
- `x & y` — двоичное И — `x.__and__(y)`;

- `y & x` — двоичное И (экземпляр класса справа) — `x.__rand__(y)`;
- `x &= y` — двоичное И и присваивание — `x.__iand__(y)`;
- `x | y` — двоичное ИЛИ — `x.__or__(y)`;
- `y | x` — двоичное ИЛИ (экземпляр класса справа) — `x.__ror__(y)`;
- `x |= y` — двоичное ИЛИ и присваивание — `x.__ior__(y)`;
- `x ^ y` — двоичное исключающее ИЛИ — `x.__xor__(y)`;
- `y ^ x` — двоичное исключающее ИЛИ (экземпляр класса справа) — `x.__rxor__(y)`;
- `x ^= y` — двоичное исключающее ИЛИ и присваивание — `x.__ixor__(y)`;
- `x << y` — сдвиг влево — `x.__lshift__(y)`;
- `y << x` — сдвиг влево (экземпляр класса справа) — `x.__rlshift__(y)`;
- `x <<= y` — сдвиг влево и присваивание — `x.__ilshift__(y)`;
- `x >> y` — сдвиг вправо — `x.__rshift__(y)`;
- `y >> x` — сдвиг вправо (экземпляр класса справа) — `x.__rrshift__(y)`;
- `x >>= y` — сдвиг вправо и присваивание — `x.__irshift__(y)`.

Перегрузка операторов сравнения производится с помощью следующих методов:

- `x == y` — равно — `x.__eq__(y)`;
- `x != y` — не равно — `x.__ne__(y)`;
- `x < y` — меньше — `x.__lt__(y)`;
- `x > y` — больше — `x.__gt__(y)`;
- `x <= y` — меньше или равно — `x.__le__(y)`;
- `x >= y` — больше или равно — `x.__ge__(y)`;
- `y in x` — проверка на вхождение — `x.__contains__(y)`.

Пример перегрузки операторов сравнения приведен в листинге 13.13.

Листинг 13.13. Пример перегрузки операторов сравнения

```
class MyClass:
    def __init__(self):
        self.x = 50
        self.arr = [1, 2, 3, 4, 5]
    def __eq__(self, y):           # Перегрузка оператора ==
        return self.x == y
    def __contains__(self, y):     # Перегрузка оператора in
        return y in self.arr
c = MyClass()
print("Равно" if c == 50 else "Не равно") # Выведет: Равно
print("Равно" if c == 51 else "Не равно") # Выведет: Не равно
print("Есть" if 5 in c else "Нет")        # Выведет: Есть
```

13.7. Статические методы и методы класса

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса. Для этого перед определением метода внутри класса следует указать декоратор `@staticmethod`. Вызов статического метода без создания экземпляра класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать статический метод через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования статических методов приведен в листинге 13.14.

Листинг 13.14. Статические методы

```
class MyClass:
    @staticmethod
    def func1(x, y):                # Статический метод
        return x + y
    def func2(self, x, y):          # Обычный метод в классе
        return x + y
    def func3(self, x, y):
        return MyClass.func1(x, y) # Вызов из метода класса

print(MyClass.func1(10, 20))        # Вызываем статический метод
c = MyClass()
print(c.func2(15, 6))               # Вызываем метод класса
print(c.func1(50, 12))              # Вызываем статический метод
                                    # через экземпляр класса
print(c.func3(23, 5))               # Вызываем статический метод
                                    # внутри класса
```

Обратите внимание на то, что в определении статического метода нет параметра `self`. Это означает, что внутри статического метода нет доступа к атрибутам и методам экземпляра класса.

Методы класса создаются с помощью декоратора `@classmethod`. В качестве первого параметра в метод класса передается ссылка на класс, а не на экземпляр класса. Вызов метода класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать метод класса через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования методов класса приведен в листинге 13.15.

Листинг 13.15. Методы класса

```
class MyClass:
    @classmethod
    def func(cls, x): # Метод класса
        print(cls, x)
```

```
MyClass.func(10)      # Вызываем метод через название класса
c = MyClass()
c.func(50)            # Вызываем метод класса через экземпляр
```

13.8. Абстрактные методы

Абстрактные методы содержат только определение метода без реализации. Предполагается, что класс-потомок должен переопределить метод и реализовать его функциональность. Чтобы такое предположение сделать более очевидным, часто внутри абстрактного метода возбуждают исключение (листинг 13.16).

Листинг 13.16. Абстрактные методы

```
class Class1:
    def func(self, x):     # Абстрактный метод
        # Возбуждаем исключение с помощью raise
        raise NotImplementedError("Необходимо переопределить метод")

class Class2(Class1):      # Наследуем абстрактный метод
    def func(self, x):     # Переопределяем метод
        print(x)

class Class3(Class1):      # Класс не переопределяет метод
    pass

c2 = Class2()
c2.func(50)                # Выведет: 50
c3 = Class3()
try:                       # Перехватываем исключения
    c3.func(50)            # Ошибка. Метод func() не переопределен
except NotImplementedError as msg:
    print(msg)             # Выведет: Необходимо переопределить метод
```

Начиная с версии Python 2.6, в состав стандартной библиотеки входит модуль `abc`. В этом модуле определен декоратор `@abstractmethod`, который позволяет указать, что метод, перед которым расположен декоратор, является абстрактным. При попытке создать экземпляр класса-потомка, в котором не переопределен абстрактный метод, возбуждается исключение `TypeError`. Рассмотрим использование декоратора `@abstractmethod` на примере (листинг 13.17).

Листинг 13.17. Использование декоратора `@abstractmethod`

```
from abc import ABCMeta, abstractmethod
class Class1(metaclass=ABCMeta):
    @abstractmethod
    def func(self, x):     # Абстрактный метод
        pass

class Class2(Class1):      # Наследуем абстрактный метод
    def func(self, x):     # Переопределяем метод
        print(x)
```

```
class Class3(Class1):       # Класс не переопределяет метод
    pass

c2 = Class2()
c2.func(50)                 # Выведет: 50
try:
    c3 = Class3()           # Ошибка. Метод func() не переопределен
    c3.func(50)
except TypeError as msg:
    print(msg)              # Can't instantiate abstract class Class3
                            # with abstract methods func
```

13.9. Ограничение доступа к идентификаторам внутри класса

Все идентификаторы внутри класса в языке Python являются открытыми, т. е. доступны для непосредственного изменения. Для имитации частных идентификаторов можно воспользоваться методами `__getattr__()`, `__getattribute__()` и `__setattr__()`, которые перехватывают обращения к атрибутам класса. Кроме того, можно воспользоваться идентификаторами, названия которых начинаются с двух символов подчеркивания. Такие идентификаторы называются *псевдочастными*. Псевдочастные идентификаторы доступны внутри класса, но не доступны по имени через экземпляр класса. Тем не менее изменить идентификатор через экземпляр класса все равно можно, зная, каким образом искажается название идентификатора. Например, идентификатор `__privateVar` внутри класса `Class1` будет доступен по имени `_Class1__privateVar`. Как видно из примера, перед идентификатором добавляется название класса с предваряющим символом подчеркивания. Приведем пример использования псевдочастных идентификаторов (листинг 13.18).

Листинг 13.18. Псевдочастные идентификаторы

```
class MyClass:
    def __init__(self, x):
        self.__privateVar = x
    def set_var(self, x):          # Изменение значения
        self.__privateVar = x
    def get_var(self):             # Получение значения
        return self.__privateVar
c = MyClass(10)                    # Создаем экземпляр класса
print(c.get_var())                 # Выведет: 10
c.set_var(20)                      # Изменяем значение
print(c.get_var())                 # Выведет: 20
try:                               # Перехватываем ошибки
    print(c.__privateVar)          # Ошибка!!!
except AttributeError as msg:
    print(msg)                     # Выведет: 'MyClass' object has
                                   # no attribute '__privateVar'
c._MyClass__privateVar = 50        # Значение псевдочастных атрибутов
                                   # все равно можно изменить
print(c.get_var())                 # Выведет: 50
```

Можно также ограничить перечень атрибутов, разрешенных для экземпляров класса. Для этого разрешенные атрибуты перечисляются внутри класса в атрибуте `__slots__`. В качестве значения атрибуту можно присвоить строку или список строк с названиями идентификаторов. Если производится попытка обращения к атрибуту, не перечисленному в `__slots__`, то возбуждается исключение `AttributeError` (листинг 13.19).

Листинг 13.19. Атрибут `__slots__`

```
class MyClass:
    __slots__ = ["x", "y"]
    def __init__(self, a, b):
        self.x, self.y = a, b
c = MyClass(1, 2)
print(c.x, c.y)                 # Выведет: 1 2
c.x, c.y = 10, 20               # Изменяем значения атрибутов
print(c.x, c.y)                 # Выведет: 10 20
try:                            # Перехватываем исключения
    c.z = 50                    # Атрибут z не указан в __slots__,
                                # поэтому возбуждается исключение
except AttributeError as msg:
    print(msg)                  # 'MyClass' object has no attribute 'z'
```

13.10. Свойства класса

Внутри класса можно создать идентификатор, через который в дальнейшем будут производиться операции получения и изменения значения атрибута, а также удаления атрибута. Создается такой идентификатор с помощью функции `property()`. Формат функции:

```
<Свойство> = property(<Чтение>[, <Запись>[, <Удаление>
                      [, <Строка документирования>]]])
```

В первых трех параметрах указывается ссылка на соответствующий метод класса. При попытке получить значение будет вызван метод, указанный в первом параметре. При операции присваивания значения будет вызван метод, указанный во втором параметре. Этот метод должен принимать один параметр. В случае удаления атрибута вызывается метод, указанный в третьем параметре. Если в качестве какого-либо параметра задано значение `None`, то это означает, что соответствующий метод не поддерживается. Рассмотрим свойства класса на примере (листинг 13.20).

Листинг 13.20. Свойства класса

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    def get_var(self):        # Чтение
        return self.__var
    def set_var(self, value): # Запись
        self.__var = value
    def del_var(self):        # Удаление
        del self.__var
    v = property(get_var, set_var, del_var, "Строка документирования")
```

```
c = MyClass(5)
c.v = 35                    # Вызывается метод set_var()
print(c.v)                  # Вызывается метод get_var()
del c.v                     # Вызывается метод del_var()
```

В Python 2.6 были добавлены методы `getter()`, `setter()` и `deleter()`, позволяющие создавать свойства классов с помощью декораторов функций. Пример использования декораторов приведен в листинге 13.21.

Листинг 13.21. Методы `getter()`, `setter()` и `deleter()`

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    @property
    def v(self):                        # Чтение
        return self.__var
    @v.setter
    def v(self, value):                 # Запись
        self.__var = value
    @v.deleter
    def v(self):                        # Удаление
        del self.__var
c = MyClass(5)
c.v = 35                                # Запись
print(c.v)                              # Чтение
del c.v                                 # Удаление
```

13.11. Декораторы классов

В языке Python 3 помимо декораторов функций поддерживаются также *декораторы классов*, которые позволяют изменить поведение обычных классов. В качестве параметра декоратор принимает ссылку на объект класса, поведение которого необходимо изменить, и должен возвращать ссылку на тот же класс или какой-либо другой. Пример декорирования класса показан в листинге 13.22.

Листинг 13.22. Декораторы классов

```
def deco(C):                    # Принимает объект класса
    print("Внутри декоратора") # Производит какие-то действия
    return C                    # Возвращает объект класса

@deco
class MyClass:
    def __init__(self, value):
        self.v = value
c = MyClass(5)
print(c.v)
```

ГЛАВА 14

Обработка исключений

Исключения — это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, то программа прерывается и выводится сообщение об ошибке.

Существуют три типа ошибок в программе:

- *синтаксические* — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., т. е. ошибки в синтаксисе языка. Как правило, интерпретатор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

```
>>> print("Нет завершающей кавычки!)
SyntaxError: EOL while scanning string literal
```

- *логические* — это ошибки в логике работы программы, которые можно выявить только по результатам работы скрипта. Как правило, интерпретатор не предупреждает о наличии ошибки. А программа будет выполняться, т. к. не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить и исправить;
- *ошибки времени выполнения* — это ошибки, которые возникают во время работы скрипта. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль:

```
>>> def test(x, y): return x / y

>>> test(4, 2)                          # Нормально
2.0
>>> test(4, 0)                          # Ошибка
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    test(4, 0)                         # Ошибка
  File "<pyshell#2>", line 1, in test
    def test(x, y): return x / y
ZeroDivisionError: division by zero
```

Необходимо заметить, что в языке Python исключения возбуждаются не только при ошибке, но и как уведомление о наступлении каких-либо событий. Например, метод `index()` возбуждает исключение `ValueError`, если искомый фрагмент не входит в строку:

```
>>> "Строка".index("текст")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    "Строка".index("текст")
ValueError: substring not found
```

14.1. Инструкция *try...except...else...finally*

Для обработки исключений предназначена инструкция `try`. Формат инструкции:

```
try:
    <Блок, в котором перехватываются исключения>
[except [<Исключение1>[ as <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>
[...
except [<ИсключениеN>[ as <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>]]
[else:
    <Блок, выполняемый, если исключение не возникло>]
[finally:
    <Блок, выполняемый в любом случае>]
```

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока `try`. В блоке `except` в параметре `<Исключение1>` указывается класс обрабатываемого исключения. Например, обработать исключение, возникающее при делении на ноль, можно так, как показано в листинге 14.1.

Листинг 14.1. Обработка деления на ноль

```
try:                                # Перехватываем исключения
    x = 1 / 0                       # Ошибка: деление на 0
except ZeroDivisionError:           # Указываем класс исключения
    print("Обработали деление на 0")
    x = 0
print(x)                            # Выведет: 0
```

Если в блоке `try` возникло исключение, то управление передается блоку `except`. В случае, если исключение не соответствует указанному классу, управление передается следующему блоку `except`. Если ни один блок `except` не соответствует исключению, то исключение "всплывает" к обработчику более высокого уровня. Если исключение нигде не обрабатывается в программе, то управление передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке. Таким образом, в обработчике может быть несколько блоков `except` с разными классами исключений. Кроме того, один обработчик можно вложить в другой (листинг 14.2).

Листинг 14.2. Вложенные обработчики

```
try:                                # Обрабатываем исключения
    try:                            # Вложенный обработчик
        x = 1 / 0                   # Ошибка: деление на 0
```

```
    except NameError:
        print("Неопределенный идентификатор")
    except IndexError:
        print("Несуществующий индекс")
    print("Выражение после вложенного обработчика")
except ZeroDivisionError:
    print("Обработка деления на 0")
    x = 0
print(x)                                # Выведет: 0
```

В этом примере во вложенном обработчике не указано исключение `ZeroDivisionError`, поэтому исключение "всплывает" к обработчику более высокого уровня. После обработки исключения управление передается инструкции, расположенной сразу после обработчика. В нашем примере управление будет передано инструкции, выводящей значение переменной `x` (`print(x)`). Обратите внимание на то, что инструкция `print("Выражение после вложенного обработчика")` выполнена не будет.

В инструкции `except` можно указать сразу несколько исключений, перечислив их через запятую внутри круглых скобок (листинг 14.3).

Листинг 14.3. Обработка нескольких исключений

```
try:
    x = 1 / 0
except (NameError, IndexError, ZeroDivisionError):
    # Обработка сразу нескольких исключений
    x = 0
print(x) # Выведет: 0
```

Получить информацию об обрабатываемом исключении можно через второй параметр в инструкции `except` (листинг 14.4).

Листинг 14.4. Получение информации об исключении

```
try:
    x = 1 / 0                       # Ошибка деления на 0
except (NameError, IndexError, ZeroDivisionError) as err:
    print(err.__class__.__name__) # Название класса исключения
    print(err)                      # Текст сообщения об ошибке
```

Результат выполнения:

```
ZeroDivisionError
division by zero
```

Для получения информации об исключении можно воспользоваться функцией `exc_info()` из модуля `sys`, которая возвращает кортеж из трех элементов: типа исключения, значения и объекта с трассировочной информацией. Преобразовать эти значения в удобочитаемый вид позволяет модуль `traceback`. Пример использования функции `exc_info()` и модуля `traceback` приведен в листинге 14.5.

Листинг 14.5. Пример использования функции `exc_info()`

```
import sys, traceback
try:
    x = 1 / 0
except ZeroDivisionError:
    Type, Value, Trace = sys.exc_info()
    print("Type: ", Type)
    print("Value:", Value)
    print("Trace:", Trace)
    print("\n", "print_exception()".center(40, "-"))
    traceback.print_exception(Type, Value, Trace, limit=5,
                              file=sys.stdout)
    print("\n", "print_tb()".center(40, "-"))
    traceback.print_tb(Trace, limit=1, file=sys.stdout)
    print("\n", "format_exception()".center(40, "-"))
    print(traceback.format_exception(Type, Value, Trace, limit=5))
    print("\n", "format_exception_only()".center(40, "-"))
    print(traceback.format_exception_only(Type, Value))
```

Результат выполнения показан в листинге 14.6.

Листинг 14.6. Результат выполнения листинга 14.5

```
Type:  <class 'ZeroDivisionError'>
Value: division by zero
Trace: <traceback object at 0x00C0E030>

 -----------print_exception()------------
Traceback (most recent call last):
  File "C:\book\test.py", line 3, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

 ---------------print_tb()---------------
  File "C:\book\test.py", line 3, in <module>
    x = 1 / 0

 -----------format_exception()-----------
['Traceback (most recent call last):\n', '  File "C:\\book\\test.py",
line 3, in <module>\n    x = 1 / 0\n',
'ZeroDivisionError: division by zero\n']

 --------format_exception_only()---------
['ZeroDivisionError: division by zero\n']
```

Если в инструкции `except` не указан класс исключения, то такой блок перехватывает все исключения. На практике следует избегать пустых инструкций `except`, т. к. можно перехватить исключение, которое является лишь сигналом системе, а не ошибкой. Пример пустой инструкции `except` приведен в листинге 14.7.

Листинг 14.7. Пример перехвата всех исключений

```
try:
    x = 1 / 0                    # Ошибка деления на 0
except:                          # Обработка всех исключений
    x = 0
print(x)                         # Выведет: 0
```

Если в обработчике присутствует блок `else`, то инструкции внутри этого блока будут выполнены только при отсутствии ошибок. При необходимости выполнить какие-либо завершающие действия вне зависимости от того, возникло исключение или нет, следует воспользоваться блоком `finally`. В качестве примера выведем последовательность выполнения блоков (листинг 14.8).

Листинг 14.8. Блоки `else` и `finally`

```
try:
    x = 10 / 2                   # Нет ошибки
    #x = 10 / 0                  # Ошибка деления на 0
except ZeroDivisionError:
    print("Деление на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Результат выполнения при отсутствии исключения:

```
Блок else
Блок finally
```

Последовательность выполнения блоков при наличии исключения будет другой:

```
Деление на 0
Блок finally
```

Необходимо заметить, что при наличии исключения и отсутствии блока `except` инструкции внутри блока `finally` будут выполнены, но исключение не будет обработано. Оно продолжит "всплывание" к обработчику более высокого уровня. Если пользовательский обработчик отсутствует, то управление передается обработчику по умолчанию, который прерывает выполнение программы и выводит сообщение об ошибке. Пример:

```
>>> try:
        x = 10 / 0
finally: print("Блок finally")

Блок finally
Traceback (most recent call last):
  File "<pyshell#17>", line 2, in <module>
    x = 10 / 0
ZeroDivisionError: division by zero
```

В качестве примера переделаем нашу программу (см. листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 14.9).

Листинг 14.9. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    x = x.rstrip("\r")              # Для версии 3.2.0 (см. разд. 1.7)
    if x == "stop":
        break      # Выход из цикла
    try:
        x = int(x) # Преобразуем строку в число
    except ValueError:
        print("Необходимо ввести целое число!")
    else:
        summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода значений и получения результата выглядит так:

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: str
Необходимо ввести целое число!
Введите число: -5
Введите число:
Необходимо ввести целое число!
Введите число: stop
Сумма чисел равна: 5
```

Значения, введенные пользователем, выделены полужирным шрифтом.

14.2. Инструкция *with...as*

Начиная с версии 2.6, язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует выполнение завершающих действий (например, закрытие файла) вне зависимости от того, произошло исключение внутри блока кода или нет.

Для работы с протоколом предназначена инструкция `with...as`. Инструкция имеет следующий формат:

```
with <Выражение1>[ as <Переменная>][, ...,
     <ВыражениеN>[ as <Переменная>]]:
    <Блок, в котором перехватываем исключения>
```

Вначале вычисляется `<Выражение1>`, которое должно возвращать объект, поддерживающий протокол. Этот объект должен иметь два метода: `__enter__()` и `__exit__()`. Метод

`__enter__()` вызывается после создания объекта. Значение, возвращаемое этим методом, присваивается переменной, указанной после ключевого слова `as`. Если переменная не указана, возвращаемое значение игнорируется. Формат метода `__enter__()`:

```
__enter__(self)
```

Далее выполняются инструкции внутри тела инструкции `with`. Если при выполнении возникло исключение, то управление передается методу `__exit__()`. Метод имеет следующий формат:

```
__exit__(self, <Тип исключения>, <Значение>, <Объект traceback>)
```

Значения, доступные через последние три параметра, полностью эквивалентны значениям, возвращаемым функцией `exc_info()` из модуля `sys`. Если исключение обработано, метод должен вернуть значение `True`, в противном случае — `False`. Если метод возвращает `False`, то исключение передается вышестоящему обработчику.

Если при выполнении инструкций, расположенных внутри тела инструкции `with`, исключение не возникло, то управление все равно передается методу `__exit__()`. В этом случае последние три параметра будут содержать значение `None`.

Рассмотрим последовательность выполнения протокола на примере (листинг 14.10).

Листинг 14.10. Протокол менеджеров контекста

```
class MyClass:
    def __enter__(self):
        print("Вызван метод __enter__()")
        return self
    def __exit__(self, Type, Value, Trace):
        print("Вызван метод __exit__()")
        if Type is None:  # Если исключение не возникло
            print("Исключение не возникло")
        else:             # Если возникло исключение
            print("Value =", Value)
            return False  # False — исключение не обработано
                          # True  — исключение обработано
print("Последовательность при отсутствии исключения:")
with MyClass():
    print("Блок внутри with")
print("\nПоследовательность при наличии исключения:")
with MyClass() as obj:
    print("Блок внутри with")
    raise TypeError("Исключение TypeError")
```

Результат выполнения:

```
Последовательность при отсутствии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Исключение не возникло
```

```
Последовательность при наличии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Value = Исключение TypeError
Traceback (most recent call last):
  File "C:\book\test.py", line 20, in <module>
    raise TypeError("Исключение TypeError")
TypeError: Исключение TypeError
```

Некоторые встроенные объекты по умолчанию поддерживают протокол, например, файлы. Если в инструкции `with` указана функция `open()`, то после выполнения инструкций внутри блока файл автоматически будет закрыт. Пример использования инструкции `with` приведен в листинге 14.11.

Листинг 14.11. Инструкция `with...as`

```
with open("test.txt", "a", encoding="utf-8") as f:
    f.write("Строка\n") # Записываем строку в конец файла
```

В этом примере файл test.txt открывается на дозапись в конец файла. После выполнения функции `open()` переменной `f` будет присвоена ссылка на объект файла. С помощью этой переменной мы можем работать с файлом внутри тела инструкции `with`. После выхода из блока, вне зависимости от наличия исключения, файл будет закрыт.

14.3. Классы встроенных исключений

Все встроенные исключения в языке Python представлены в виде классов. Иерархия встроенных классов исключений показана в листинге 14.12.

Листинг 14.12. Иерархия встроенных классов исключений

```
BaseException
    GeneratorExit
    KeyboardInterrupt
    SystemExit
    Exception
        StopIteration
        Warning
            BytesWarning, ResourceWarning,
            DeprecationWarning, FutureWarning, ImportWarning,
            PendingDeprecationWarning, RuntimeWarning, SyntaxWarning,
            UnicodeWarning, UserWarning
        ArithmeticError
            FloatingPointError, OverflowError, ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
```

```
        EnvironmentError
            IOError
            OSError
                WindowsError
        EOFError
        ImportError
        LookupError
            IndexError, KeyError
        MemoryError
        NameError
            UnboundLocalError
        ReferenceError
        RuntimeError
            NotImplementedError
        SyntaxError
            IndentationError
                TabError
        SystemError
        TypeError
        ValueError
            UnicodeError
                UnicodeDecodeError, UnicodeEncodeError
                UnicodeTranslateError
```

Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих классов-потомков. Например, для перехвата деления на ноль мы использовали класс `ZeroDivisionError`. Если вместо этого класса указать базовый класс `ArithmeticError`, то будут перехватываться исключения классов `FloatingPointError`, `OverflowError` и `ZeroDivisionError`. Пример:

```
try:
    x = 1 / 0                       # Ошибка: деление на 0
except ArithmeticError:             # Указываем базовый класс
    print("Обработали деление на 0")
```

Рассмотрим основные классы встроенных исключений:

- `BaseException` — является классом самого верхнего уровня;
- `Exception` — именно этот класс, а не `BaseException`, необходимо наследовать при создании пользовательских классов исключений;
- `AssertionError` — возбуждается инструкцией `assert`;
- `AttributeError` — попытка обращения к несуществующему атрибуту объекта;
- `EOFError` — возбуждается функцией `input()` при достижении конца файла;
- `IOError` — ошибка доступа к файлу;
- `ImportError` — невозможно подключить модуль или пакет;
- `IndentationError` — неправильно расставлены отступы в программе;
- `IndexError` — указанный индекс не существует в последовательности;
- `KeyError` — указанный ключ не существует в словаре;

- `KeyboardInterrupt` — нажата комбинация клавиш <Ctrl>+<C>;
- `NameError` — попытка обращения к идентификатору до его определения;
- `StopIteration` — возбуждается методом `__next__()` как сигнал об окончании итераций;
- `SyntaxError` — синтаксическая ошибка;
- `TypeError` — тип объекта не соответствует ожидаемому;
- `UnboundLocalError` — внутри функции переменной присваивается значение после обращения к одноименной глобальной переменной;
- `UnicodeDecodeError` — ошибка преобразования последовательности байтов в строку;
- `UnicodeEncodeError` — ошибка преобразования строки в последовательность байтов;
- `ValueError` — переданный параметр не соответствует ожидаемому значению;
- `ZeroDivisionError` — попытка деления на ноль.

14.4. Пользовательские исключения

Для возбуждения пользовательских исключений предназначены две инструкции:

- `raise`;
- `assert`.

Инструкция `raise` возбуждает указанное исключение. Она имеет несколько форматов:

```
raise <Экземпляр класса>
raise <Название класса>
raise <Экземпляр или название класса> from <Объект исключения>
raise
```

В первом формате инструкции `raise` указывается экземпляр класса возбуждаемого исключения. При создании экземпляра можно передать данные конструктору класса. Эти данные будут доступны через второй параметр в инструкции `except`. Пример возбуждения встроенного исключения `ValueError`:

```
>>> raise ValueError("Описание исключения")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ValueError("Описание исключения")
ValueError: Описание исключения
```

Пример обработки этого исключения:

```
try:
    raise ValueError("Описание исключения")
except ValueError as msg:
    print(msg) # Выведет: Описание исключения
```

В качестве исключения можно указать экземпляр пользовательского класса:

```
class MyError(Exception):
    def __init__(self, value):
        self.msg = value
    def __str__(self):
        return self.msg
```

```
# Обработка пользовательского исключения
try:
    raise MyError("Описание исключения")
except MyError as err:
    print(err)        # Вызывается метод __str__()
    print(err.msg)    # Обращение к атрибуту класса
# Повторно возбуждаем исключение
raise MyError("Описание исключения")
```

Результат выполнения:

```
Описание исключения
Описание исключения
Traceback (most recent call last):
  File "C:\book\test.py", line 14, in <module>
    raise MyError("Описание исключения")
MyError: Описание исключения
```

Класс `Exception` содержит все необходимые методы для вывода сообщения об ошибке. Поэтому в большинстве случаев достаточно создать пустой класс, который наследует класс `Exception`:

```
class MyError(Exception): pass
try:
    raise MyError("Описание исключения")
except MyError as err:
    print(err)        # Выведет: Описание исключения
```

Во втором формате инструкции `raise` в первом параметре задается объект класса, а не экземпляр. Пример:

```
try:
    raise ValueError # Эквивалентно: raise ValueError()
except ValueError:
    print("Сообщение об ошибке")
```

В третьем формате инструкции `raise` в первом параметре задается экземпляр класса или просто название класса, а во втором параметре указывается объект исключения. В этом случае объект исключения сохраняется в атрибуте `__cause__`. При обработке вложенных исключений эти данные используются для вывода информации не только о последнем исключении, но и о первоначальном исключении. Пример:

```
try:
    x = 1 / 0
except Exception as err:
    raise ValueError() from err
```

Результат выполнения:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the
following exception:
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError
```

Как видно из результата, мы получили информацию не только по исключению `ValueError`, но и по исключению `ZeroDivisionError`. Следует заметить, что при отсутствии инструкции `from` информация сохраняется неявным образом. Если убрать инструкцию `from` в предыдущем примере, то получим следующий результат:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another
exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError
```

Четвертый формат инструкции `raise` позволяет повторно возбудить последнее исключение. Пример:

```
class MyError(Exception): pass
try:
    raise MyError("Сообщение об ошибке")
except MyError as err:
    print(err)
    raise              # Повторно возбуждаем исключение
```

Результат выполнения:

```
Сообщение об ошибке
Traceback (most recent call last):
  File "C:\book\test.py", line 5, in <module>
    raise MyError("Сообщение об ошибке")
MyError: Сообщение об ошибке
```

Инструкция `assert` возбуждает исключение `AssertionError`, если логическое выражение возвращает значение `False`. Инструкция имеет следующий формат:

```
assert <Логическое выражение>[, <Данные>]
```

Инструкция `assert` эквивалентна следующему коду:

```
if __debug__:
    if not <Логическое выражение>:
        raise AssertionError(<Данные>)
```

Если при запуске программы используется флаг `-O`, то переменная `__debug__` будет иметь ложное значение. Таким образом можно удалить все инструкции `assert` из байт-кода.

Пример использования инструкции `assert`:

```
try:
    x = -3
    assert x >= 0, "Сообщение об ошибке"
except AssertionError as err:
    print(err) # Выведет: Сообщение об ошибке
```

ГЛАВА 15

Работа с файлами и каталогами

Очень часто нужно сохранить какие-либо данные. Для этого существуют два способа: сохранение в файл и сохранение в базу данных. Первый способ используется при сохранении информации небольшого объема. Если объем велик, то лучше (и удобнее) воспользоваться базой данных.

15.1. Открытие файла

Прежде чем работать с файлом, необходимо создать объект файла с помощью функции `open()`. Функция имеет следующий формат:

```
open(<Путь к файлу>[, mode='r'][, buffering=-1][, encoding=None][,
     errors=None][, newline=None][, closefd=True])
```

В первом параметре указывается путь к файлу. Путь может быть абсолютным или относительным. При указании абсолютного пути в Windows следует учитывать, что слэш является специальным символом. По этой причине слэш необходимо удваивать или вместо обычных строк использовать неформатированные строки. Пример:

```
>>> "C:\\temp\\new\\file.txt"    # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"      # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"       # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Обратите внимание на последний пример. В этом пути присутствуют сразу три специальных символа: `\t`, `\n` и `\f`. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Если такую строку передать в функцию `open()`, то это приведет к исключению `IOError`:

```
>>> open("C:\temp\new\file.txt")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    open("C:\temp\new\file.txt")
IOError: [Errno 22] Invalid argument: 'C:\temp\new\x0cile.txt'
```

Вместо абсолютного пути к файлу можно указать относительный путь. В этом случае путь определяется с учетом местоположения текущего рабочего каталога. Относительный путь будет автоматически преобразован в абсолютный путь с помощью функции `abspath()` из модуля `os.path`. Возможны следующие варианты:

- если открываемый файл находится в текущем рабочем каталоге, то можно указать только название файла. Пример:

```
>>> import os.path # Подключаем модуль
>>> # Файл в текущем рабочем каталоге (C:\book\)
>>> os.path.abspath(r"file.txt")
'C:\\book\\file.txt'
```

- если открываемый файл расположен во вложенной папке, то перед названием файла перечисляются названия вложенных папок через слэш. Пример:

```
>>> # Открываемый файл в C:\book\folder1\
>>> os.path.abspath(r"folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> # Открываемый файл в C:\book\folder1\folder2\
>>> os.path.abspath(r"folder1/folder2/file.txt")
'C:\\book\\folder1\\folder2\\file.txt'
```

- если папка с файлом расположена ниже уровнем, то перед названием файла указываются две точки и слэш (`"../"`). Пример:

```
>>> # Открываемый файл в C:\
>>> os.path.abspath(r"../file.txt")
'C:\\file.txt'
```

- если в начале пути расположен слэш, то путь отсчитывается от корня диска. В этом случае местоположение текущего рабочего каталога не имеет значения. Пример:

```
>>> # Открываемый файл в C:\book\folder1\
>>> os.path.abspath(r"/book/folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> # Открываемый файл в C:\book\folder1\folder2\
>>> os.path.abspath(r"/book/folder1/folder2/file.txt")
'C:\\book\\folder1\\folder2\\file.txt'
```

В абсолютном и относительном пути можно указать как прямые, так и обратные слэши. Все слэши будут автоматически преобразованы с учетом значения атрибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение в операционной системе Windows:

```
>>> os.path.sep
'\\'
>>> os.path.abspath(r"C:/book/folder1/file.txt")
'C:\\book\\folder1\\file.txt'
```

При использовании относительного пути необходимо учитывать местоположение текущего рабочего каталога, т. к. рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый файл. Если файл запускается с помощью двойного щелчка на его значке, то каталоги будут совпадать. Если же файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Рассмотрим все на примере. В каталоге C:\book создадим следующую структуру файлов:

```
C:\book\
    test.py
    folder1\
        __init__.py
        module1.py
```

Содержимое файла C:\book\test.py приведено в листинге 15.1.

Листинг 15.1. Содержимое файла C:\book\test.py

```
# -*- coding: utf-8 -*-
import os, sys
print("%-25s%s" % ("Файл:", os.path.abspath(__file__)))
print("%-25s%s" % ("Текущий рабочий каталог:", os.getcwd()))
print("%-25s%s" % ("Каталог для импорта:", sys.path[0]))
print("%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt")))
print("-" * 40)
import folder1.module1 as m
m.get_cwd()
```

Файл C:\book\folder1__init__.py создаем пустым. Как вы уже знаете, этот файл позволяет преобразовать каталог в пакет с модулями. Содержимое файла C:\book\folder1\module1.py приведено в листинге 15.2.

Листинг 15.2. Содержимое файла C:\book\folder1\module1.py

```
# -*- coding: utf-8 -*-
import os, sys
def get_cwd():
    print("%-25s%s" % ("Файл:", os.path.abspath(__file__)))
    print("%-25s%s" % ("Текущий рабочий каталог:", os.getcwd()))
    print("%-25s%s" % ("Каталог для импорта:", sys.path[0]))
    print("%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt")))
```

Запускаем командную строку, переходим в каталог C:\book и запускаем файл test.py:

```
C:\>cd C:\book
C:\book>test.py
Файл:                     C:\book\test.py
Текущий рабочий каталог: C:\book
Каталог для импорта:      C:\book
Путь к файлу:             C:\book\file.txt
----------------------------------------
Файл:                     C:\book\folder1\module1.py
Текущий рабочий каталог: C:\book
Каталог для импорта:      C:\book
Путь к файлу:             C:\book\file.txt
```

В этом примере текущий рабочий каталог совпадает с каталогом, в котором расположен файл test.py. Однако обратите внимание на текущий рабочий каталог внутри модуля module1.py. Если внутри этого модуля в функции `open()` указать название файла без пути, то поиск файла будет произведен в каталоге C:\book, а не C:\book\folder1.

Теперь перейдем в корень диска C: и опять запустим файл test.py:

```
C:\book>cd C:\
C:\>C:\book\test.py
Файл:                       C:\book\test.py
Текущий рабочий каталог: C:\
Каталог для импорта:     C:\book
Путь к файлу:               C:\file.txt
----------------------------------------
Файл:                       C:\book\folder1\module1.py
Текущий рабочий каталог: C:\
Каталог для импорта:     C:\book
Путь к файлу:               C:\file.txt
```

В этом случае текущий рабочий каталог не совпадает с каталогом, в котором расположен файл test.py. Если внутри файлов test.py и module1.py в функции `open()` указать название файла без пути, то поиск файла будет производиться в корне диска C:, а не в каталогах с этими файлами.

Чтобы поиск файла всегда производился в каталоге с исполняемым файлом, необходимо этот каталог сделать текущим с помощью функции `chdir()` из модуля `os`. В качестве примера изменим содержимое файла test.py (листинг 15.3).

Листинг 15.3. Пример использования функции `chdir()`

```
# -*- coding: utf-8 -*-
import os, sys
# Делаем каталог с исполняемым файлом текущим
os.chdir(os.path.dirname(os.path.abspath(__file__)))
print("%-25s%s" % ("Файл:", __file__))
print("%-25s%s" % ("Текущий рабочий каталог:", os.getcwd()))
print("%-25s%s" % ("Каталог для импорта:", sys.path[0]))
print("%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt")))
```

Обратите внимание на четвертую строку. С помощью атрибута `__file__` мы получаем путь к исполняемому файлу вместе с названием файла. Атрибут `__file__` не всегда содержит полный путь к файлу. Например, если запуск осуществляется следующим образом

```
C:\book>C:\Python32\python test.py
```

то атрибут будет содержать только название файла без пути. Чтобы всегда получать полный путь к файлу, следует передать значение атрибута в функцию `abspath()` из модуля `os.path`. Далее мы извлекаем путь (без названия файла) с помощью функции `dirname()` и передаем его функции `chdir()`. Теперь если в функции `open()` указать название файла без пути, то поиск будет производиться в каталоге с этим файлом. Запустим файл test.py с помощью командной строки:

```
C:\>C:\book\test.py
Файл:                       C:\book\test.py
Текущий рабочий каталог: C:\book
Каталог для импорта:     C:\book
Путь к файлу:               C:\book\file.txt
```

Функции, предназначенные для работы с каталогами, мы еще рассмотрим подробно в следующих разделах. Сейчас важно запомнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог, в котором расположен исполняемый файл. Кроме того, пути поиска файлов не имеют никакого отношения к путям поиска модулей.

Необязательный параметр `mode` в функции `open()` может принимать следующие значения:

- `r` — только чтение (значение по умолчанию). После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение `IOError`;
- `r+` — чтение и запись. После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение `IOError`;
- `w` — запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `w+` — чтение и запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `a` — запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
- `a+` — чтение и запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется.

Кроме того, после режима может следовать модификатор:

- `b` — файл будет открыт в бинарном режиме. Файловые методы принимают и возвращают объекты типа `bytes`;
- `t` — файл будет открыт в текстовом режиме (значение по умолчанию в Windows). Файловые методы принимают и возвращают объекты типа `str`. В этом режиме в Windows при чтении символ `\r` будет удален, а при записи, наоборот, добавлен. В качестве примера создадим файл file.txt и запишем в него две строки:

```
>>> f = open(r"file.txt", "w")  # Открываем файл на запись
>>> f.write("String1\nString2") # Записываем две строки в файл
15
>>> f.close()                   # Закрываем файл
```

Так как мы указали режим `w`, если файл не существует, то он будет создан, а если существует, то файл будет перезаписан. Теперь выведем содержимое файла в бинарном и текстовом режимах:

```
>>> # Бинарный режим (символ \r остается)
>>> with open(r"file.txt", "rb") as f:
        for line in f:
            print(repr(line))

b'String1\r\n'
b'String2'
>>> # Текстовый режим (символ \r удаляется)
>>> with open(r"file.txt", "r") as f:
        for line in f:
            print(repr(line))
```

```
'String1\n'
'String2'
```

Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла. В необязательном параметре `buffering` можно указать размер буфера. Если в качестве значения указан `0`, то данные будут сразу записываться в файл (значение допустимо только в бинарном режиме). Значение `1` используется при построчной записи в файл (значение допустимо только в текстовом режиме), другое положительное число задает примерный размер буфера, а отрицательное значение (или отсутствие значения) означает установку размера, применяемого в системе по умолчанию.

При использовании текстового режима (используется по умолчанию) при чтении производится попытка преобразовать данные в кодировку Unicode, а при записи выполняется обратная операция: строка преобразуется в последовательность байтов в определенной кодировке. По умолчанию используется кодировка, используемая в системе. Если преобразование невозможно, то возбуждается исключение. Указать кодировку, которая будет использоваться при записи и чтении файла, позволяет параметр `encoding`. В качестве примера запишем данные в кодировке UTF-8:

```
>>> f = open(r"file.txt", "w", encoding="utf-8")
>>> f.write("Строка")  # Записываем строку в файл
6
>>> f.close()          # Закрываем файл
```

Для чтения этого файла следует явно указать кодировку при открытии файла:

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(line)

Строка
```

При работе с файлами в кодировках UTF-8, UTF-16 и UTF-32 следует учитывать, что в начале файла могут быть служебные символы, называемые сокращенно BOM (*Byte Order Mark*, метка порядка байтов). Для кодировки UTF-8 эти символы являются необязательными, и в предыдущем примере они не были добавлены в файл при записи. Чтобы символы были добавлены, в параметре `encoding` следует указать значение `utf-8-sig`. Запишем строку в файл в кодировке UTF-8 с BOM:

```
>>> f = open(r"file.txt", "w", encoding="utf-8-sig")
>>> f.write("Строка")  # Записываем строку в файл
6
>>> f.close()          # Закрываем файл
```

Теперь прочитаем файл с разными значениями в параметре `encoding`:

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(repr(line))

'\ufeffСтрока'
>>> with open(r"file.txt", "r", encoding="utf-8-sig") as f:
        for line in f:
            print(repr(line))

'Строка'
```

В первом примере мы указали значение `utf-8`, поэтому маркер BOM был прочитан из файла вместе с данными. Во втором примере указано значение `utf-8-sig`, поэтому маркер BOM не попал в результат. Если неизвестно есть маркер в файле или нет и необходимо получить данные без маркера, то следует всегда указывать значение `utf-8-sig` при чтении файла в кодировке UTF-8.

Для кодировок UTF-16 и UTF-32 маркер BOM является обязательным. При указании значений `utf-16` и `utf-32` в параметре `encoding` обработка маркера производится автоматически. При записи данных он автоматически вставляется в начало файла, а при чтении маркер не попадает в результат. Запишем строку в файл, а затем прочитаем ее из файла:

```
>>> with open(r"file.txt", "w", encoding="utf-16") as f:
        f.write("Строка")

6
>>> with open(r"file.txt", "r", encoding="utf-16") as f:
        for line in f:
            print(repr(line))

'Строка'
```

При использовании значений `utf-16-le`, `utf-16-be`, `utf-32-le` и `utf-32-be` маркер BOM необходимо самим добавить в начало файла, а при чтении удалить его.

В параметре `errors` можно указать уровень обработки ошибок. Возможные значения: "strict" (при ошибке возбуждается исключение; значение по умолчанию), "replace" (неизвестный символ заменяется символом вопроса или символом с кодом `\ufffd`), "ignore" (неизвестные символы игнорируются), "xmlcharrefreplace" (неизвестный символ заменяется последовательностью `&#xxxx;`) и "backslashreplace" (неизвестный символ заменяется последовательностью `\uxxxx`).

15.2. Методы для работы с файлами

После открытия файла функция `open()` возвращает объект, с помощью которого производится дальнейшая работа с файлом. Тип объекта зависит от режима открытия файла и буферизации. Перечислим основные методы:

- `close()` — закрывает файл. Так как интерпретатор автоматически удаляет объект, когда отсутствуют ссылки на него, можно явно не закрывать файл в небольших программах. Тем не менее явное закрытие файла является признаком хорошего стиля программирования. Кроме того, в Python 3.2 при наличии не закрытого файла генерируется предупреждающее сообщение: `"ResourceWarning: unclosed file"`.

  Начиная с версии 2.6, язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует закрытие файла вне зависимости от того, произошло исключение внутри блока кода или нет. Пример:

  ```
  with open(r"file.txt", "w", encoding="cp1251") as f:
      f.write("Строка") # Записываем строку в файл
  # Здесь файл уже закрыт автоматически
  ```

- `write(<Данные>)` — записывает строку или последовательность байтов в файл. Если в качестве параметра указана строка, то файл должен быть открыт в текстовом режиме. Для записи последовательности байтов необходимо открыть файл в бинарном режиме.

Помните, что нельзя записывать строку в бинарном режиме и последовательность байтов в текстовом режиме. Метод возвращает количество записанных символов или байтов. Пример записи в файл:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.write("Строка1\nСтрока2") # Записываем строку в файл
15
>>> f.close()                   # Закрываем файл
>>> # Бинарный режим
>>> f = open(r"file.txt", "wb")
>>> f.write(bytes("Строка1\nСтрока2", "cp1251"))
15
>>> f.write(bytearray("\nСтрока3", "cp1251"))
8
>>> f.close()
```

- `writelines(<Последовательность>)` — записывает последовательность в файл. Если все элементы последовательности являются строками, то файл должен быть открыт в текстовом режиме. Если все элементы являются последовательностями байтов, то файл должен быть открыт в бинарном режиме. Пример записи элементов списка:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.writelines(["Строка1\n", "Строка2"])
>>> f.close()
>>> # Бинарный режим
>>> f = open(r"file.txt", "wb")
>>> arr = [bytes("Строка1\n", "cp1251"), bytes("Строка2", "cp1251")]
>>> f.writelines(arr)
>>> f.close()
```

- `read([<Количество>])` — считывает данные из файла. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном — последовательность байтов. Если параметр не указан, то возвращается содержимое файла от текущей позиции указателя до конца файла:

```
>>> # Текстовый режим
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.read()

'Строка1\nСтрока2'
>>> # Бинарный режим
>>> with open(r"file.txt", "rb") as f:
        f.read()

b'\xd1\xf2\xf0\xee\xea\xe01\n\xd1\xf2\xf0\xee\xea\xe02'
```

Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество символов или байтов. Когда достигается конец файла, метод возвращает пустую строку. Пример:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
```

```
>>> f.read(8)                # Считываем 8 символов
'Строка1\n'
>>> f.read(8)                # Считываем 8 символов
'Строка2'
>>> f.read(8)                # Достигнут конец файла
''
>>> f.close()
```

- `readline([<Количество>])` — считывает из файла одну строку при каждом вызове. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном — последовательность байтов. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. При достижении конца файла возвращается пустая строка. Пример:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.readline(), f.readline()
('Строка1\n', 'Строка2')
>>> f.readline()             # Достигнут конец файла
''
>>> f.close()
>>> # Бинарный режим
>>> f = open(r"file.txt", "rb")
>>> f.readline(), f.readline()
(b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02')
>>> f.readline()             # Достигнут конец файла
b''
>>> f.close()
```

Если в необязательном параметре указано число, то считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца файла или из файла не будет прочитано указанное количество символов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество символов. Если количество символов в строке больше, то возвращается указанное количество символов. Пример:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.readline(2), f.readline(2)
('Ст', 'ро')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
'ка1\n'
>>> f.close()
```

- `readlines()` — считывает все содержимое файла в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. Если файл открыт в текстовом режиме, то возвращается список строк, а если в бинарном — список объектов типа `bytes`. Пример:

```
>>> # Текстовый режим
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.readlines()
```

```
['Строка1\n', 'Строка2']
>>> # Бинарный режим
>>> with open(r"file.txt", "rb") as f:
        f.readlines()

[b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02']
```

- `__next__()` — считывает одну строку при каждом вызове. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном — последовательность байтов. При достижении конца файла возбуждается исключение `StopIteration`. Пример:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.__next__(), f.__next__()
('Строка1\n', 'Строка2')
>>> f.__next__() # Достигнут конец файла
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    f.__next__() # Достигнут конец файла
StopIteration
>>> f.close()
```

Благодаря методу `__next__()` мы можем перебирать файл построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `__next__()`. В качестве примера выведем все строки, предварительно удалив символ перевода строки:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> for line in f: print(line.rstrip("\n"), end=" ")

Строка1 Строка2
>>> f.close()
```

- `flush()` — записывает данные из буфера на диск;
- `fileno()` — возвращает целочисленный дескриптор файла. Возвращаемое значение всегда будет больше числа 2, т. к. число 0 закреплено за стандартным вводом `stdin`, 1 — за стандартным выводом `stdout`, а 2 — за стандартным выводом сообщений об ошибках `stderr`. Пример:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.fileno()                    # Дескриптор файла
3
>>> f.close()
```

- `truncate([<Количество>])` — обрезает файл до указанного количества символов или байтов. Метод возвращает новый размер файла. Пример:

```
>>> f = open(r"file.txt", "r+", encoding="cp1251")
>>> f.read()
'Строка1\nСтрока2'
>>> f.truncate(5)
5
```

```
>>> f.close()
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.read()

'Строк'
```

- `tell()` — возвращает позицию указателя относительно начала файла в виде целого числа. Обратите внимание на то, что в Windows метод `tell()` считает символ `\r` как дополнительный байт, хотя этот символ удаляется при открытии файла в текстовом режиме. Пример:

```
>>> with open(r"file.txt", "w", encoding="cp1251") as f:
        f.write("String1\nString2")

15
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.tell()      # Указатель расположен в начале файла
0
>>> f.readline()  # Перемещаем указатель
'String1\n'
>>> f.tell()      # Возвращает 9 (8 + '\r'), а не 8 !!!
9
>>> f.close()
```

Чтобы избежать этого несоответствия, следует открывать файл в бинарном режиме, а не в текстовом:

```
>>> f = open(r"file.txt", "rb")
>>> f.readline()  # Перемещаем указатель
b'String1\r\n'
>>> f.tell()      # Теперь значение соответствует
9
>>> f.close()
```

- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель в позицию, имеющую смещение `<Смещение>` относительно позиции `<Позиция>`. В параметре `<Позиция>` могут быть указаны следующие атрибуты из модуля `io` или соответствующие им значения:
  - `io.SEEK_SET` или `0` — начало файла (значение по умолчанию);
  - `io.SEEK_CUR` или `1` — текущая позиция указателя;
  - `io.SEEK_END` или `2` — конец файла.

Выведем значения этих атрибутов:

```
>>> import io
>>> io.SEEK_SET, io.SEEK_CUR, io.SEEK_END
(0, 1, 2)
```

Пример использования метода `seek()`:

```
>>> import io
>>> f = open(r"file.txt", "rb")
>>> f.seek(9, io.SEEK_CUR) # 9 байтов от указателя
9
```

```
>>> f.tell()
9
>>> f.seek(0, io.SEEK_SET) # Перемещаем указатель в начало
0
>>> f.tell()
0
>>> f.seek(-9, io.SEEK_END) # -9 байтов от конца файла
7
>>> f.tell()
7
>>> f.close()
```

Помимо методов объекты файлов поддерживают несколько атрибутов:

- `name` — содержит название файла;
- `mode` — режим, в котором был открыт файл;
- `closed` — возвращает `True`, если файл был закрыт, и `False` в противном случае. Пример:

```
>>> f = open(r"file.txt", "r+b")
>>> f.name, f.mode, f.closed
('file.txt', 'rb+', False)
>>> f.close()
>>> f.closed
True
```

- `encoding` — название кодировки, которая будет использоваться для преобразования строк перед записью в файл или при чтении. Обратите внимание на то, что изменить значение атрибута нельзя, т. к. атрибут доступен только для чтения. Атрибут доступен только в текстовом режиме. Пример:

```
>>> f = open(r"file.txt", "a", encoding="cp1251")
>>> f.encoding
'cp1251'
>>> f.close()
```

  Стандартный вывод `stdout` также является файловым объектом. Атрибут `encoding` этого объекта всегда содержит кодировку устройства вывода, поэтому строка преобразуется в последовательность байтов в правильной кодировке. Например, при запуске с помощью двойного щелчка на значке файла атрибут `encoding` будет иметь значение `"cp866"`, а при запуске в окне **Python Shell** редактора IDLE — значение `"cp1251"`. Пример:

```
>>> import sys
>>> sys.stdout.encoding
'cp1251'
```

- `buffer` — позволяет получить доступ к буферу. Атрибут доступен только в текстовом режиме. С помощью этого объекта можно записать последовательность байтов в текстовый поток. Пример:

```
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.buffer.write(bytes("Строка", "cp1251"))
6
>>> f.close()
```

15.3. Доступ к файлам с помощью модуля *os*

Модуль `os` содержит дополнительные низкоуровневые функции, позволяющие работать с файлами. Функциональность этого модуля зависит от используемой операционной системы. Получить название используемой версии модуля можно с помощью атрибута `name`. В операционной системе Windows XP атрибут имеет значение `"nt"`:

```
>>> import os
>>> os.name               # Значение в ОС Windows XP
'nt'
```

Для доступа к файлам предназначены следующие функции из модуля `os`:

- `open(<Путь к файлу>, <Режим>[, mode=0o777])` — открывает файл и возвращает целочисленный дескриптор, с помощью которого производится дальнейшая работа с файлом. Если файл открыть не удалось, возбуждается исключение `OSError`. В параметре `<Режим>` в операционной системе Windows могут быть указаны следующие флаги (или их комбинация через символ `|`):
  - `os.O_RDONLY` — чтение;
  - `os.O_WRONLY` — запись;
  - `os.O_RDWR` — чтение и запись;
  - `os.O_APPEND` — добавление в конец файла;
  - `os.O_CREAT` — создать файл, если он не существует;
  - `os.O_TRUNC` — очистить содержимое файла;
  - `os.O_BINARY` — файл будет открыт в бинарном режиме;
  - `os.O_TEXT` — файл будет открыт в текстовом режиме. В Windows файлы по умолчанию открываются в текстовом режиме.

  Рассмотрим несколько примеров. Откроем файл на запись и запишем в него одну строку. Если файл не существует, то создадим его. Если файл существует, то очистим его:

```
>>> import os                # Подключаем модуль
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_TRUNC
>>> f = os.open(r"file.txt", mode)
>>> os.write(f, b"String1\n") # Записываем данные
8
>>> os.close(f)              # Закрываем файл
```

  Добавим еще одну строку в конец файла:

```
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_APPEND
>>> f = os.open(r"file.txt", mode)
>>> os.write(f, b"String2\n") # Записываем данные
8
>>> os.close(f)              # Закрываем файл
```

  Прочитаем содержимое файла в текстовом режиме:

```
>>> f = os.open(r"file.txt", os.O_RDONLY)
>>> os.read(f, 50)           # Читаем 50 байт
b'String1\nString2\n'
>>> os.close(f)              # Закрываем файл
```

Теперь прочитаем содержимое файла в бинарном режиме:

```
>>> f = os.open(r"file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.read(f, 50)             # Читаем 50 байт
b'String1\r\nString2\r\n'
>>> os.close(f)                # Закрываем файл
```

- `read(<Дескриптор>, <Количество байтов>)` — читает из файла указанное количество байтов. При достижении конца файла возвращается пустая строка. Пример:

```
>>> f = os.open(r"file.txt", os.O_RDONLY)
>>> os.read(f, 5), os.read(f, 5), os.read(f, 5), os.read(f, 5)
(b'Strin', b'g1\nS', b'tring', b'2\n')
>>> os.read(f, 5)              # Достигнут конец файла
b''
>>> os.close(f)                # Закрываем файл
```

- `write(<Дескриптор>, <Последовательность байтов>)` — записывает последовательность байтов в файл. Возвращает количество записанных байтов;
- `close(<Дескриптор>)` — закрывает файл;
- `lseek(<Дескриптор>, <Смещение>, <Позиция>)` — устанавливает указатель в позицию, имеющую смещение `<Смещение>` относительно позиции `<Позиция>`. В качестве значения функция возвращает новую позицию указателя. В параметре `<Позиция>` могут быть указаны следующие атрибуты или соответствующие им значения:
  - `os.SEEK_SET` или `0` — начало файла;
  - `os.SEEK_CUR` или `1` — текущая позиция указателя;
  - `os.SEEK_END` или `2` — конец файла.

  Пример:

```
>>> f = os.open(r"file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.lseek(f, 0, os.SEEK_END) # Перемещение в конец файла
18
>>> os.lseek(f, 0, os.SEEK_SET) # Перемещение в начало файла
0
>>> os.lseek(f, 9, os.SEEK_CUR) # Относительно указателя
9
>>> os.lseek(f, 0, os.SEEK_CUR) # Текущее положение указателя
9
>>> os.close(f)                   # Закрываем файл
```

- `dup(<Дескриптор>)` — возвращает дубликат файлового дескриптора;
- `fdopen(<Дескриптор>[, <Режим>[, <Размер буфера>]])` — возвращает файловый объект по указанному дескриптору. Параметры `<Режим>` и `<Размер буфера>` имеют тот же смысл, что и в функции `open()`. Пример:

```
>>> fd = os.open(r"file.txt", os.O_RDONLY)
>>> fd
3
>>> f = os.fdopen(fd, "r")
>>> f.fileno() # Объект имеет тот же дескриптор
3
```

```
>>> f.read()
'String1\nString2\n'
>>> f.close()
```

15.4. Классы *StringIO* и *BytesIO*

Класс `StringIO` из модуля `io` позволяет работать со строкой как с файловым объектом. Все операции с файловым объектом производятся в оперативной памяти. Формат конструктора класса:

```
StringIO([<Начальное значение>][, newline=None])
```

Если первый параметр не указан, то начальным значением будет пустая строка. После создания объекта указатель текущей позиции устанавливается на начало "файла". Объект, возвращаемый конструктором класса, имеет следующие методы:

- `close()` — закрывает "файл". Проверить, открыт "файл" или закрыт, позволяет атрибут `closed`. Атрибут возвращает `True`, если "файл" был закрыт, и `False` в противном случае;
- `getvalue()` — возвращает содержимое "файла" в виде строки:

```
>>> import io               # Подключаем модуль
>>> f = io.StringIO("String1\n")
>>> f.getvalue()            # Получаем содержимое файла
'String1\n'
>>> f.close()               # Закрываем файл
```

- `tell()` — возвращает текущую позицию указателя относительно начала "файла";
- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель в позицию, имеющую смещение `<Смещение>` относительно позиции `<Позиция>`. В параметре `<Позиция>` могут быть указаны следующие значения:
  - `0` — начало "файла" (значение по умолчанию);
  - `1` — текущая позиция указателя;
  - `2` — конец "файла".

  Пример использования методов `seek()` и `tell()`:

```
>>> f = io.StringIO("String1\n")
>>> f.tell()        # Позиция указателя
0
>>> f.seek(0, 2)    # Перемещаем указатель в конец файла
8
>>> f.tell()        # Позиция указателя
8
>>> f.seek(0)       # Перемещаем указатель в начало файла
0
>>> f.tell()        # Позиция указателя
0
>>> f.close()       # Закрываем файл
```

- `write(<Строка>)` — записывает строку в "файл":

```
>>> f = io.StringIO("String1\n")
>>> f.seek(0, 2)         # Перемещаем указатель в конец файла
8
```

```
>>> f.write("String2\n") # Записываем строку в файл
8
>>> f.getvalue()         # Получаем содержимое файла
'String1\nString2\n'
>>> f.close()            # Закрываем файл
```

- `writelines(<Последовательность>)` — записывает последовательность в "файл":

```
>>> f = io.StringIO()
>>> f.writelines(["String1\n", "String2\n"])
>>> f.getvalue()         # Получаем содержимое файла
'String1\nString2\n'
>>> f.close()            # Закрываем файл
```

- `read([<Количество символов>])` — считывает данные из "файла". Если параметр не указан, то возвращается содержимое "файла" от текущей позиции указателя до конца "файла". Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество символов. Когда достигается конец "файла", метод возвращает пустую строку. Пример:

```
>>> f = io.StringIO("String1\nString2\n")
>>> f.read()
'String1\nString2\n'
>>> f.seek(0) # Перемещаем указатель в начало файла
0
>>> f.read(5), f.read(5), f.read(5), f.read(5), f.read(5)
('Strin', 'g1\nSt', 'ring2', '\n', '')
>>> f.close() # Закрываем файл
```

- `readline([<Количество символов>])` — считывает из "файла" одну строку при каждом вызове. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. При достижении конца "файла" возвращается пустая строка. Пример:

```
>>> f = io.StringIO("String1\nString2")
>>> f.readline(), f.readline(), f.readline()
('String1\n', 'String2', '')
>>> f.close() # Закрываем файл
```

Если в необязательном параметре указано число, то считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца "файла" или из "файла" не будет прочитано указанное количество символов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество символов. Если количество символов в строке больше, то возвращается указанное количество символов. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3\n")
>>> f.readline(5), f.readline(5)
('Strin', 'g1\n')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
'String2\n'
>>> f.close()       # Закрываем файл
```

- `readlines([<Примерное количество символов>])` — считывает все содержимое "файла" в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines()
['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем файл
```

  Если в необязательном параметре указано число, то считывается указанное количество символов плюс фрагмент до символа конца строки `\n`. Затем эта строка разбивается и добавляется построчно в список. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines(14)
['String1\n', 'String2\n']
>>> f.seek(0) # Перемещаем указатель в начало файла
0
>>> f.readlines(17)
['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем файл
```

- `__next__()` — считывает одну строку при каждом вызове. При достижении конца "файла" возбуждается исключение `StopIteration`. Пример:

```
>>> f = io.StringIO("String1\nString2")
>>> f.__next__(), f.__next__()
('String1\n', 'String2')
>>> f.__next__()
... Фрагмент опущен ...
StopIteration
>>> f.close() # Закрываем файл
```

  Благодаря методу `__next__()` мы можем перебирать файл построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `__next__()`. Пример:

```
>>> f = io.StringIO("String1\nString2")
>>> for line in f: print(line.rstrip())

String1
String2
>>> f.close() # Закрываем файл
```

- `flush()` — сбрасывает данные из буфера в "файл";
- `truncate([<Количество символов>])` — обрезает "файл" до указанного количества символов. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.truncate(15)      # Обрезаем файл
15
>>> f.getvalue()        # Получаем содержимое файла
'String1\nString2'
>>> f.close()           # Закрываем файл
```

Если параметр не указан, то "файл" обрезается до текущей позиции указателя:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.seek(15)          # Перемещаем указатель
15
>>> f.truncate()        # Обрезаем файл до указателя
15
>>> f.getvalue()        # Получаем содержимое файла
'String1\nString2'
>>> f.close()           # Закрываем файл
```

Класс `StringIO` работает только со строками. Чтобы выполнять аналогичные операции с последовательностями байтов, следует использовать класс `BytesIO` из модуля `io`. Формат конструктора класса:

```
BytesIO([<Начальное значение>])
```

Класс `BytesIO` содержит такие же методы, что и класс `StringIO`, но в качестве значений методы принимают и возвращают последовательности байтов, а не строки. Рассмотрим основные операции на примере:

```
>>> import io              # Подключаем модуль
>>> f = io.BytesIO(b"String1\n")
>>> f.seek(0, 2)           # Перемещаем указатель в конец файла
8
>>> f.write(b"String2\n") # Пишем в файл
8
>>> f.getvalue()           # Получаем содержимое файла
b'String1\nString2\n'
>>> f.seek(0)              # Перемещаем указатель в начало файла
0
>>> f.read()               # Считываем данные
b'String1\nString2\n'
>>> f.close()              # Закрываем файл
```

Начиная с версии 3.2, класс `BytesIO` поддерживает метод `getbuffer()`, который возвращает ссылку на объект `memoryview`. С помощью этого объекта можно получать и изменять данные по индексу или срезу, преобразовывать данные в список целых чисел (с помощью метода `tolist()`) или в последовательность байтов (с помощью метода `tobytes()`). Пример:

```
>>> f = io.BytesIO(b"Python")
>>> buf = f.getbuffer()
>>> buf[0]                 # Получаем значение по индексу
b'P'
>>> buf[0] = b"J"          # Изменяем значение по индексу
>>> f.getvalue()           # Получаем содержимое
b'Jython'
>>> buf.tolist()           # Преобразуем в список чисел
[74, 121, 116, 104, 111, 110]
>>> buf.tobytes()          # Преобразуем в тип bytes
b'Jython'
>>> f.close()              # Закрываем файл
```

15.5. Права доступа к файлам и каталогам

В операционной системе семейства UNIX для каждого объекта (файла или каталога) назначаются права доступа для каждой разновидности пользователей — владельца, группы и прочих. Могут быть назначены следующие права доступа:

- чтение;
- запись;
- выполнение.

Права доступа обозначаются буквами:

- `r` — файл можно читать, а содержимое каталога можно просматривать;
- `w` — файл можно модифицировать, удалять и переименовывать, а в каталоге можно создавать или удалять файлы. Каталог можно переименовать или удалить;
- `x` — файл можно выполнять, а в каталоге можно выполнять операции над файлами, в том числе производить поиск файлов в нем.

Права доступа к файлу определяются записью типа:

```
-rw-r--r--
```

Первый символ `-` означает, что это файл, и не задает никаких прав доступа. Далее три символа (`rw-`) задают права доступа для владельца (чтение и запись). Символ `--` означает, что права доступа на выполнение нет. Следующие три символа задают права доступа для группы (`r--`) — только чтение. Ну и последние три символа (`r--`) задают права для всех остальных пользователей (только чтение).

Права доступа к каталогу определяются такой строкой:

```
drwxr-xr-x
```

Первая буква (`d`) означает, что это каталог. Владелец может выполнять в каталоге любые действия (`rwx`), а группа и все остальные пользователи — только читать и выполнять поиск (`r-x`). Для того чтобы каталог можно было просматривать, должны быть установлены права на выполнение (`x`).

Кроме того, права доступа обозначаются числом. Такие числа называются *маской прав доступа*. Число состоит из трех цифр от 0 до 7. Первая цифра задает права для владельца, вторая — для группы, а третья — для всех остальных пользователей. Например, права доступа `-rw-r--r--` соответствуют числу 644. Сопоставим числам, входящим в маску прав доступа, двоичную и буквенную записи (табл. 15.1).

Например, права доступа `rw-r--r--` можно записать так: `110 100 100`, что переводится в число `6 4 4`. Таким образом, если право предоставлено, то в соответствующей позиции стоит 1, а если нет — то 0.

Таблица 15.1. *Права доступа в разных записях*

Восьмеричная цифра	Двоичная запись	Буквенная запись	Восьмеричная цифра	Двоичная запись	Буквенная запись
0	000	`---`	4	100	`r--`
1	001	`--x`	5	101	`r-x`
2	010	`-w-`	6	110	`rw-`
3	011	`-wx`	7	111	`rwx`

Для определения прав доступа к файлу или каталогу предназначена функция access() из модуля os. Функция имеет следующий формат:

```
access(<Путь>, <Режим>)
```

Функция возвращает True, если проверка прошла успешно, или False в противном случае. В параметре <Режим> могут быть указаны следующие константы, определяющие тип проверки:

- os.F_OK — проверка наличия пути или файла:

```
>>> import os                        # Подключаем модуль os
>>> os.access(r"file.txt", os.F_OK) # Файл существует
True
>>> os.access(r"C:\book", os.F_OK)  # Каталог существует
True
>>> os.access(r"C:\book2", os.F_OK) # Каталог не существует
False
```

- os.R_OK — проверка на возможность чтения файла или каталога;
- os.W_OK — проверка на возможность записи в файл или каталог;
- os.X_OK — определение, является ли файл или каталог выполняемым.

Чтобы изменить права доступа из программы, необходимо воспользоваться функцией chmod() из модуля os. Функция имеет следующий формат:

```
chmod(<Путь>, <Права доступа>)
```

Права доступа задаются в виде числа, перед которым следует указать комбинацию символов 0o (это соответствует восьмеричной записи числа):

```
>>> os.chmod(r"file.txt", 0o777) # Полный доступ к файлу
```

Вместо числа можно указать комбинацию констант из модуля stat. За дополнительной информацией обращайтесь к документации по модулю.

15.6. Функции для манипулирования файлами

Для копирования и перемещения файлов предназначены следующие функции из модуля shutil:

- copyfile(<Копируемый файл>, <Куда копируем>) — позволяет скопировать содержимое файла в другой файл. Никакие метаданные (например, права доступа) не копируются. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение IOError. Пример:

```
>>> import shutil      # Подключаем модуль
>>> shutil.copyfile(r"file.txt", r"file2.txt")
>>> # Путь не существует:
>>> shutil.copyfile(r"file.txt", r"C:\book2\file2.txt")
... Фрагмент опущен ...
IOError: [Errno 2] No such file or directory:
'C:\\book2\\file2.txt'
```

- `copy(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать файл. Копируются также права доступа. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `IOError`. Пример:

```
>>> shutil.copy(r"file.txt", r"file3.txt")
```

- `copy2(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать файл вместе с метаданными. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `IOError`. Пример:

```
>>> shutil.copy2(r"file.txt", r"file4.txt")
```

- `move(<Путь к файлу>, <Куда перемещаем>)` — копирует файл в указанное место, а затем удаляет исходный файл. Если файл существует, то он будет перезаписан. Если файл не удалось переместить, возбуждается исключение `IOError`. Если файл удалить нельзя, то в операционной системе Windows возбуждается исключение `WindowsError`. Пример перемещения файла file4.txt в каталог C:\book\test:

```
>>> shutil.move(r"file4.txt", r"C:\book\test")
```

Для переименования и удаления файлов предназначены следующие функции из модуля `os`:

- `rename(<Старое имя>, <Новое имя>)` — переименовывает файл. Если исходный файл отсутствует или новое имя файла уже существует, то в операционной системе Windows возбуждается исключение `WindowsError`. Пример переименования файла с обработкой исключений:

```
import os # Подключаем модуль
try:
    os.rename(r"file3.txt", "file4.txt")
except OSError: # WindowsError наследует OSError
    print("Файл не удалось переименовать")
else:
    print("Файл успешно переименован")
```

- `remove(<Путь к файлу>)` и `unlink(<Путь к файлу>)` — позволяют удалить файл. Если файл удалить нельзя, то в операционной системе Windows возбуждается исключение `WindowsError`. Пример:

```
>>> os.remove(r"file2.txt")
>>> os.unlink(r"file4.txt")
```

Модуль `os.path` содержит дополнительные функции, позволяющие проверить наличие файла, получить размер файла и др. Перечислим эти функции:

- `exists(<Путь>)` — проверяет указанный путь на существование. Значением функции будет `True`, если путь существует, и `False` в противном случае:

```
>>> import os.path
>>> os.path.exists(r"file.txt"), os.path.exists(r"file2.txt")
(True, False)
>>> os.path.exists(r"C:\book"), os.path.exists(r"C:\book2")
(True, False)
```

- `getsize(<Путь к файлу>)` — возвращает размер файла. Если файл не существует, то в Windows возбуждается исключение `WindowsError`:

```
>>> os.path.getsize(r"file.txt")  # Файл существует
18
>>> os.path.getsize(r"file2.txt") # Файл не существует
... Фрагмент опущен ...
WindowsError: [Error 2] Не удается найти указанный файл: 'file2.txt'
```

- `getatime(<Путь к файлу>)` — служит для определения времени последнего доступа к файлу. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример:

```
>>> import time    # Подключаем модуль time
>>> t = os.path.getatime(r"file.txt")
>>> t
1304111982.96875
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'30.04.2011 01:19:42'
```

- `getctime(<Путь к файлу>)` — позволяет узнать дату создания файла. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример:

```
>>> t = os.path.getctime(r"file.txt")
>>> t
1304028509.015625
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 02:08:29'
```

- `getmtime(<Путь к файлу>)` — возвращает время последнего изменения файла. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример:

```
>>> t = os.path.getmtime(r"file.txt")
>>> t
1304044731.265625
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 06:38:51'
```

Получить размер файла и время создания, изменения и доступа к файлу, а также значения других метаданных позволяет функция `stat()` из модуля `os`. В качестве значения функция возвращает объект `stat_result`, содержащий десять атрибутов: `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime` и `st_ctime`. Пример использования функции `stat()` приведен в листинге 15.4.

Листинг 15.4. Пример использования функции `stat()`

```
>>> import os, time
>>> s = os.stat(r"file.txt")
>>> s
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304111982,
st_mtime=1304044731, st_ctime=1304028509)
>>> s.st_size      # Размер файла
18
```

```
>>> t = s.st_atime # Последний доступ
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'30.04.2011 01:19:42'
>>> t = s.st_ctime # Создание файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 02:08:29'
>>> t = s.st_mtime # Изменение файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 06:38:51'
```

Обновить время последнего доступа и время изменения файла позволяет функция `utime()` из модуля `os`. Функция имеет два формата:

```
utime(<Путь к файлу>, None)
utime(<Путь к файлу>, (<Последний доступ>, <Изменение файла>))
```

Если в качестве второго параметра указано значение `None`, то время доступа и изменения файла будет текущим. Во втором формате функции `utime()` указывается кортеж из новых значений в виде количества секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример использования функции `utime()` приведен в листинге 15.5.

Листинг 15.5. Пример использования функции `utime()`

```
>>> import os, time
>>> os.stat(r"file.txt")          # Первоначальные значения
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304111982,
st_mtime=1304044731, st_ctime=1304028509)
>>> t = time.time() — 600
>>> os.utime(r"file.txt", (t, t)) # Текущее время минус 600 сек
>>> os.stat(r"file.txt")
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304112906,
st_mtime=1304112906, st_ctime=1304028509)
>>> os.utime(r"file.txt", None)   # Текущее время
>>> os.stat(r"file.txt")
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304113557,
st_mtime=1304113557, st_ctime=1304028509)
```

15.7. Преобразование пути к файлу или каталогу

Преобразовать путь к файлу или каталогу позволяют следующие функции из модуля `os.path`:

- `abspath(<Относительный путь>)` — преобразует относительный путь в абсолютный, учитывая местоположение текущего рабочего каталога. Пример:

```
>>> import os.path
>>> os.path.abspath(r"file.txt")
'C:\\book\\file.txt'
```

```
>>> os.path.abspath(r"folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> os.path.abspath(r"../file.txt")
'C:\\file.txt'
```

В относительном пути можно указать как прямые, так и обратные слэши. Все слэши будут автоматически преобразованы с учетом значения атрибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение в операционной системе Windows:

```
>>> os.path.sep
'\\'
```

При указании пути в Windows следует учитывать, что слэш является специальным символом. По этой причине слэш необходимо удваивать или вместо обычных строк использовать неформатированные строки. Пример:

```
>>> "C:\\temp\\new\\file.txt"    # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"      # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"       # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Кроме того, если слэш расположен в конце строки, то его необходимо удваивать даже при использовании неформатированных строк:

```
>>> r"C:\temp\new\"              # Неправильно!!!
SyntaxError: EOL while scanning string literal
>>> r"C:\temp\new\\"
'C:\\temp\\new\\\\'
```

В первом случае последний слэш экранирует закрывающую кавычку, что приводит к синтаксической ошибке. Решить эту проблему можно, удвоив последний слэш. Однако посмотрите на результат. Два слэша превратились в четыре. От одной проблемы ушли, а к другой пришли. Поэтому в данном случае лучше использовать обычные строки:

```
>>> "C:\\temp\\new\\"            # Правильно
'C:\\temp\\new\\'
>>> r"C:\temp\new\\"[:-1]        # Можно и удалить слэш
'C:\\temp\\new\\'
```

- `isabs(<Путь>)` — возвращает `True`, если путь является абсолютным, и `False` в противном случае:

```
>>> os.path.isabs("file.txt")
False
>>> os.path.isabs(r"C:\book\file.txt")
True
```

- `basename(<Путь>)` — возвращает имя файла без пути к нему:

```
>>> os.path.basename(r"C:\book\folder1\file.txt")
'file.txt'
>>> os.path.basename(r"C:\book\folder")
'folder'
>>> os.path.basename("C:\\book\\folder\\")
''
```

- `dirname(<Путь>)` — возвращает путь к каталогу:

```
>>> os.path.dirname(r"C:\book\folder\file.txt")
'C:\\book\\folder'
>>> os.path.dirname(r"C:\book\folder")
'C:\\book'
>>> os.path.dirname("C:\\book\\folder\\")
'C:\\book\\folder'
```

- `split(<Путь>)` — возвращает кортеж из двух элементов: пути к каталогу и названия файла:

```
>>> os.path.split(r"C:\book\folder\file.txt")
('C:\\book\\folder', 'file.txt')
>>> os.path.split(r"C:\book\folder")
('C:\\book', 'folder')
>>> os.path.split("C:\\book\\folder\\")
('C:\\book\\folder', '')
```

- `splitdrive(<Путь>)` — разделяет путь на имя диска и остальную часть пути. В качестве значения возвращается кортеж из двух элементов:

```
>>> os.path.splitdrive(r"C:\book\folder\file.txt")
('C:', '\\book\\folder\\file.txt')
```

- `splitext(<Путь>)` — возвращает кортеж из двух элементов: пути с названием файла, но без расширения, и расширения файла (фрагмент после последней точки):

```
>>> os.path.splitext(r"C:\book\folder\file.tar.gz")
('C:\\book\\folder\\file.tar', '.gz')
```

- `join(<Путь1>[, ..., <ПутьN>])` — соединяет указанные элементы пути:

```
>>> os.path.join("C:\\", "book\\folder", "file.txt")
'C:\\book\\folder\\file.txt'
>>> os.path.join(r"C:\\", "book/folder/", "file.txt")
'C:\\\\book/folder/file.txt'
```

  Обратите внимание на последний пример. В пути используются разные слэши и в результате получен некорректный путь. Чтобы этот путь сделать корректным, необходимо воспользоваться функцией `normpath()`:

```
>>> p = os.path.join(r"C:\\", "book/folder/", "file.txt")
>>> os.path.normpath(p)
'C:\\book\\folder\\file.txt'
```

15.8. Перенаправление ввода/вывода

При рассмотрении методов для работы с файлами говорилось, что значение, возвращаемое методом `fileno()`, всегда будет больше числа 2, т. к. число 0 закреплено за стандартным вводом `stdin`, 1 — за стандартным выводом `stdout`, а 2 — за стандартным выводом сообщений об ошибках `stderr`. Все эти потоки имеют некоторое сходство с файловыми объектами. Например, потоки `stdout` и `stderr` имеют метод `write()`, предназначенный для вывода сообщений, а поток `stdin` имеет метод `readline()`, предназначенный для получения входящих данных. Если этим объектам присвоить ссылку на объект, поддерживающий

файловые методы, то можно перенаправить стандартные потоки в другое место. В качестве примера перенаправим вывод в файл (листинг 15.6).

Листинг 15.6. Перенаправление вывода в файл

```
>>> import sys                  # Подключаем модуль sys
>>> tmp_out = sys.stdout        # Сохраняем ссылку на sys.stdout
>>> f = open(r"file.txt", "a")  # Открываем файл на дозапись
>>> sys.stdout = f              # Перенаправляем вывод в файл
>>> print("Пишем строку в файл")
>>> sys.stdout = tmp_out        # Восстанавливаем стандартный вывод
>>> print("Пишем строку в стандартный вывод")
Пишем строку в стандартный вывод
>>> f.close()                   # Закрываем файл
```

В этом примере мы вначале сохранили ссылку на стандартный вывод в переменной `tmp_out`. С помощью этой переменной можно в дальнейшем восстановить вывод в стандартный поток.

Функция `print()` напрямую поддерживает перенаправление вывода. Для этого используется параметр `file`, который по умолчанию ссылается на стандартный поток вывода. Например, записать строку в файл можно так:

```
>>> f = open(r"file.txt", "a")
>>> print("Пишем строку в файл", file=f)
>>> f.close()
```

Стандартный ввод `stdin` также можно перенаправить. В этом случае функция `input()` будет читать одну строку из файла при каждом вызове. При достижении конца файла возбуждается исключение `EOFError`. В качестве примера выведем содержимое файла с помощью перенаправления потока ввода (листинг 15.7).

Листинг 15.7. Перенаправление потока ввода

```
# -*- coding: utf-8 -*-
import sys
tmp_in = sys.stdin           # Сохраняем ссылку на sys.stdin
f = open(r"file.txt", "r")   # Открываем файл на чтение
sys.stdin = f                # Перенаправляем ввод
while True:
    try:
        line = input()       # Считываем строку из файла
        print(line)          # Выводим строку
    except EOFError:         # Если достигнут конец файла,
        break                # выходим из цикла
sys.stdin = tmp_in           # Восстанавливаем стандартный ввод
f.close()                    # Закрываем файл
input()
```

Если необходимо узнать, ссылается ли стандартный ввод на терминал или нет, можно воспользоваться методом `isatty()`. Метод возвращает `True`, если объект ссылается на терминал, и `False` в противном случае.

Пример:

```
>>> tmp_in = sys.stdin           # Сохраняем ссылку на sys.stdin
>>> f = open(r"file.txt", "r")
>>> sys.stdin = f                # Перенаправляем ввод
>>> sys.stdin.isatty()           # Не ссылается на терминал
False
>>> sys.stdin = tmp_in           # Восстанавливаем стандартный ввод
>>> sys.stdin.isatty()           # Ссылается на терминал
True
>>> f.close()                    # Закрываем файл
```

Перенаправить стандартный ввод/вывод можно также с помощью командной строки. В качестве примера создадим файл tests.py в папке C:\book с кодом, приведенным в листинге 15.8.

Листинг 15.8. Содержимое файла tests.py

```
# -*- coding: utf-8 -*-
while True:
    try:
        line = input()
        print(line)
    except EOFError:
        break
```

Запускаем командную строку и переходим в папку со скриптом, выполнив команду `cd C:\book`. Теперь выведем содержимое файла file.txt, выполнив команду:

```
C:\Python32\python.exe tests.py < file.txt
```

Перенаправить стандартный вывод в файл можно аналогичным образом. Только в этом случае символ < необходимо заменить символом >. Изменим файл tests.py следующим образом:

```
# -*- coding: utf-8 -*-
print("String")            # Эта строка будет записана в файл
```

Теперь перенаправим вывод в файл file.txt, выполнив команду:

```
C:\Python32\python.exe tests.py > file.txt
```

В этом режиме файл file.txt будет перезаписан. Если необходимо добавить результат в конец файла, следует использовать символы >>. Пример дозаписи в файл:

```
C:\Python32\python.exe tests.py >> file.txt
```

С помощью стандартного вывода `stdout` можно создать индикатор выполнения процесса в окне консоли. Чтобы реализовать такой индикатор, нужно вспомнить, что символ перевода строки в Windows состоит из двух символов: `\r` (перевод каретки) и `\n` (перевод строки). Таким образом, используя только символ перевода каретки `\r`, можно перемещаться в начало строки и перезаписывать ранее выведенную информацию. Рассмотрим вывод индикатора процесса на примере (листинг 15.9).

Листинг 15.9. Индикатор выполнения процесса

```
# -*- coding: utf-8 -*-
import sys, time
for i in range(5, 101, 5):
    sys.stdout.write("\r ... %s%%" % i) # Обновляем индикатор
    sys.stdout.flush()                  # Сбрасываем содержимое буфера
    time.sleep(1)                       # Засыпаем на 1 секунду
sys.stdout.write("\rПроцесс завершен\n")
input()
```

Сохраняем код в файл и запускаем с помощью двойного щелчка на значке файла. В окне консоли записи будут заменять друг друга на одной строке каждую секунду. Так как данные перед выводом могут помещаться в буфер, мы сбрасываем их явным образом с помощью метода `flush()`.

15.9. Сохранение объектов в файл

Сохранить объекты в файл и в дальнейшем восстановить объекты из файла позволяют модули `pickle` и `shelve`. Модуль `pickle` предоставляет следующие функции:

- `dump(<Объект>, <Файл>[, <Протокол>][, fix_imports=True])` — производит сериализацию объекта и записывает данные в указанный файл. В параметре `<Файл>` указывается файловый объект, открытый на запись в бинарном режиме. Пример сохранения объекта в файл:

```
>>> import pickle
>>> f = open(r"file.txt", "wb")
>>> obj = ["Строка", (2, 3)]
>>> pickle.dump(obj, f)
>>> f.close()
```

- `load()` — читает данные из файла и преобразует их в объект. В параметре `<Файл>` указывается файловый объект, открытый на чтение в бинарном режиме. Формат функции:

```
load(<Файл>[, fix_imports=True][, encoding="ASCII"]
           [, errors="strict"])
```

  Пример восстановления объекта из файла:

```
>>> f = open(r"file.txt", "rb")
>>> obj = pickle.load(f)
>>> obj
['Строка', (2, 3)]
>>> f.close()
```

В один файл можно сохранить сразу несколько объектов, последовательно вызывая функцию `dump()`. Пример сохранения нескольких объектов приведен в листинге 15.10.

Листинг 15.10. Сохранение нескольких объектов

```
>>> obj1 = ["Строка", (2, 3)]
>>> obj2 = (1, 2)
>>> f = open(r"file.txt", "wb")
```

```
>>> pickle.dump(obj1, f)          # Сохраняем первый объект
>>> pickle.dump(obj2, f)          # Сохраняем второй объект
>>> f.close()
```

Для восстановления объектов необходимо несколько раз вызвать функцию `load()` (листинг 15.11).

Листинг 15.11. Восстановление нескольких объектов

```
>>> f = open(r"file.txt", "rb")
>>> obj1 = pickle.load(f)        # Восстанавливаем первый объект
>>> obj2 = pickle.load(f)        # Восстанавливаем второй объект
>>> obj1, obj2
(['Строка', (2, 3)], (1, 2))
>>> f.close()
```

Сохранить объект в файл можно также с помощью метода `dump(<Объект>)` класса `Pickler`. Конструктор класса имеет следующий формат:

```
Pickler(<Файл>[, <Протокол>][, fix_imports=True])
```

Пример сохранения объекта в файл:

```
>>> f = open(r"file.txt", "wb")
>>> obj = ["Строка", (2, 3)]
>>> pkl = pickle.Pickler(f)
>>> pkl.dump(obj)
>>> f.close()
```

Восстановить объект из файла позволяет метод `load()` из класса `Unpickler`. Формат конструктора класса:

```
Unpickler(<Файл>[, fix_imports=True][, encoding="ASCII"]
                 [, errors="strict"])
```

Пример восстановления объекта из файла:

```
>>> f = open(r"file.txt", "rb")
>>> obj = pickle.Unpickler(f).load()
>>> obj
['Строка', (2, 3)]
>>> f.close()
```

Модуль `pickle` позволяет также преобразовать объект в строку и восстановить объект из строки. Для этого предназначены две функции:

- `dumps(<Объект>[, <Протокол>][, fix_imports=True])` — производит сериализацию объекта и возвращает последовательность байтов специального формата. Формат зависит от указанного протокола (число от `0` до `3`). Пример преобразования списка и кортежа:

```
>>> obj1 = [1, 2, 3, 4, 5]      # Список
>>> obj2 = (6, 7, 8, 9, 10)     # Кортеж
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

- `loads(<Последовательность байтов>[, fix_imports=True][, encoding="ASCII"][, errors="strict"])` — преобразует последовательность байтов специального формата обратно в объект. Пример восстановления списка и кортежа:

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

Модуль `shelve` позволяет сохранять объекты под определенным ключом (задается в виде строки) и предоставляет интерфейс доступа, сходный со словарями. Для сериализации объекта используются возможности модуля `pickle`, а чтобы записать получившуюся строку по ключу в файл, применяется модуль `dbm`. Все эти действия модуль `shelve` производит незаметно для нас.

Чтобы открыть файл с базой объектов, используется функция `open()`. Функция имеет следующий формат:

```
open(<Путь к файлу>[, flag="c"][, protocol=None][, writeback=False])
```

В необязательном параметре `flag` можно указать один из режимов открытия файла:

- `r` — только чтение;
- `w` — чтение и запись;
- `c` — чтение и запись (значение по умолчанию). Если файл не существует, он будет создан;
- `n` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан.

Функция `open()` возвращает объект, с помощью которого производится дальнейшая работа с базой данных. Этот объект имеет следующие методы:

- `close()` — закрывает файл с базой данных. В качестве примера создадим файл и сохраним в нем список и кортеж:

```
>>> import shelve                    # Подключаем модуль
>>> db = shelve.open("db1")          # Открываем файл
>>> db["obj1"] = [1, 2, 3, 4, 5]     # Сохраняем список
>>> db["obj2"] = (6, 7, 8, 9, 10)    # Сохраняем кортеж
>>> db["obj1"], db["obj2"]           # Вывод значений
([1, 2, 3, 4, 5], (6, 7, 8, 9, 10))
>>> db.close()                       # Закрываем файл
```

- `keys()` — возвращает объект с ключами;
- `values()` — возвращает объект со значениями;
- `items()` — возвращает объект, поддерживающий итерации. На каждой итерации возвращается кортеж, содержащий ключ и значение. Пример:

```
>>> db = shelve.open("db1")
>>> db.keys(), db.values()
(KeysView(<shelve.DbfilenameShelf object at 0x00FE81B0>),
 ValuesView(<shelve.DbfilenameShelf object at 0x00FE81B0>))
>>> list(db.keys()), list(db.values())
(['obj1', 'obj2'], [[1, 2, 3, 4, 5], (6, 7, 8, 9, 10)])
```

```
>>> db.items()
ItemsView(<shelve.DbfilenameShelf object at 0x00FE81B0>)
>>> list(db.items())
[('obj1', [1, 2, 3, 4, 5]), ('obj2', (6, 7, 8, 9, 10))]
>>> db.close()
```

- `get(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение `None` или значение, указанное во втором параметре;
- `setdefault(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый элемент со значением, указанным во втором параметре, и возвращает это значение. Если второй параметр не указан, значением нового элемента будет `None`;
- `pop(<Ключ>[, <Значение по умолчанию>])` — удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение из второго параметра. Если ключ отсутствует, и второй параметр не указан, то возбуждается исключение `KeyError`;
- `popitem()` — удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если файл пустой, возбуждается исключение `KeyError`;
- `clear()` — удаляет все элементы. Метод ничего не возвращает в качестве значения;
- `update()` — добавляет элементы. Метод изменяет текущий объект и ничего не возвращает. Если элемент с указанным ключом уже присутствует, то его значение будет перезаписано. Форматы метода:

```
update(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
update(<Словарь>)
update(<Список кортежей с двумя элементами>)
update(<Список списков с двумя элементами>)
```

Помимо этих методов можно воспользоваться функцией `len()` для получения количества элементов и оператором `del` для удаления определенного элемента, а также оператором `in` для проверки существования ключа. Пример:

```
>>> db = shelve.open("db1")
>>> len(db)            # Количество элементов
2
>>> "obj1" in db
True
>>> del db["obj1"]     # Удаление элемента
>>> "obj1" in db
False
>>> db.close()
```

15.10. Функции для работы с каталогами

Для работы с каталогами используются следующие функции из модуля `os`:

- `getcwd()` — возвращает текущий рабочий каталог. От значения, возвращаемого этой функцией, зависит преобразование относительного пути в абсолютный. Кроме того,

важно помнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог с исполняемым файлом.

Пример:

```
>>> import os
>>> os.getcwd()                # Текущий рабочий каталог
'C:\\book'
```

- `chdir(<Имя каталога>)` — делает указанный каталог текущим:

```
>>> os.chdir("C:\\book\\folder1\\")
>>> os.getcwd()                # Текущий рабочий каталог
'C:\\book\\folder1'
```

- `mkdir(<Имя каталога>[, <Права доступа>])` — создает новый каталог с правами доступа, указанными во втором параметре. Права доступа задаются трехзначным числом, перед которым указывается `0o` (значение второго параметра по умолчанию `0o777`). Пример создания нового каталога в текущем рабочем каталоге:

```
>>> os.mkdir("newfolder")      # Создание каталога
```

- `rmdir(<Имя каталога>)` — удаляет пустой каталог. Если в каталоге есть файлы или указанный каталог не существует, то в Windows возбуждается исключение `WindowsError`. Удалим каталог `newfolder`:

```
>>> os.rmdir("newfolder") # Удаление каталога
```

- `listdir(<Путь>)` — возвращает список объектов в указанном каталоге:

```
>>> os.listdir("C:\\book\\folder1\\")
['file1.txt', 'file2.txt', 'file3.txt', 'folder1', 'folder2']
```

- `walk()` — позволяет обойти дерево каталогов. Формат функции:

```
walk(<Начальный каталог>[, topdown=True][, onerror=None]
     [, followlinks=False])
```

В качестве значения функция `walk()` возвращает объект. На каждой итерации через этот объект доступен кортеж из трех элементов: текущий каталог, список каталогов и список файлов. Если произвести изменения в списке каталогов во время выполнения, то это позволит изменить порядок обхода вложенных каталогов.

Необязательный параметр `topdown` задает последовательность обхода каталогов. Если в качестве значения указано `True` (значение по умолчанию), то последовательность обхода будет такой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\"): print(p)

C:\book\folder1\
C:\book\folder1\folder1_1
C:\book\folder1\folder1_1\folder1_1_1
C:\book\folder1\folder1_1\folder1_1_2
C:\book\folder1\folder1_2
```

Если в параметре `topdown` указано значение `False`, то последовательность обхода будет другой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\", False):
        print(p)
```

```
C:\book\folder1\folder1_1\folder1_1_1
C:\book\folder1\folder1_1\folder1_1_2
C:\book\folder1\folder1_1
C:\book\folder1\folder1_2
C:\book\folder1\
```

Благодаря такой последовательности обхода каталогов можно удалить все вложенные файлы и каталоги. Это особенно важно при удалении каталога, т. к. функция `rmdir()` позволяет удалить только пустой каталог. Пример очистки дерева каталогов:

```
import os
for (p, d, f) in os.walk("C:\\book\\folder1\\", False):
    for file_name in f: # Удаляем все файлы
        os.remove(os.path.join(p, file_name))
    for dir_name in d: # Удаляем все каталоги
        os.rmdir(os.path.join(p, dir_name))
```

Внимание!

Очень осторожно используйте этот код. Если в качестве первого параметра в функции `walk()` указать корневой каталог диска, то все файлы и каталоги будут удалены.

Удалить дерево каталогов позволяет также функция `rmtree()` из модуля `shutil`. Функция имеет следующий формат:

```
rmtree(<Путь>[, <Обработка ошибок>[, <Обработчик ошибок>]])
```

Если в параметре `<Обработка ошибок>` указано значение `True`, то ошибки будут проигнорированы. Если указано значение `False` (значение по умолчанию), то в третьем параметре можно указать ссылку на функцию-обработчик. Эта функция будет вызываться при возникновении исключения. Пример удаления дерева каталогов вместе с начальным каталогом:

```
import shutil
shutil.rmtree("C:\\book\\folder1\\")
```

Как вы уже знаете, функция `listdir()` возвращает список объектов в указанном каталоге. Проверить, на какой тип объекта ссылается элемент этого списка, можно с помощью следующих функций из модуля `os.path`:

- `isdir(<Объект>)` — возвращает `True`, если объект является каталогом, и `False` в противном случае:

```
>>> import os.path
>>> os.path.isdir(r"C:\book\file.txt")
False
>>> os.path.isdir("C:\\book\\")
True
```

- `isfile(<Объект>)` — возвращает `True`, если объект является файлом, и `False` в противном случае:

```
>>> os.path.isfile(r"C:\book\file.txt")
True
>>> os.path.isfile("C:\\book\\")
False
```

- `islink(<Объект>)` — возвращает `True`, если объект является символической ссылкой, и `False` в противном случае. Если символические ссылки не поддерживаются, функция возвращает `False`.

Функция `listdir()` возвращает список всех объектов в указанном каталоге. Если необходимо ограничить список определенными критериями, то следует воспользоваться функцией `glob(<Путь>)` из модуля `glob`. Функция `glob()` позволяет указать в пути следующие специальные символы:

- `?` — любой одиночный символ;
- `*` — любое количество символов;
- `[<Символы>]` — позволяет указать символы, которые должны быть на этом месте в пути. Можно перечислить символы или указать диапазон через тире.

В качестве значения функция возвращает список путей к объектам, совпадающим с шаблоном. Пример использования функции `glob()` приведен в листинге 15.12.

Листинг 15.12. Пример использования функции `glob()`

```
>>> import os, glob
>>> os.listdir("C:\\book\\folder1\\")
['file.txt', 'file1.txt', 'file2.txt', 'folder1_1', 'folder1_2',
'index.html']
>>> glob.glob("C:\\book\\folder1\\*.txt")
['C:\\book\\folder1\\file.txt', 'C:\\book\\folder1\\file1.txt',
'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*.html") # Абсолютный путь
['C:\\book\\folder1\\index.html']
>>> glob.glob("folder1/*.html")            # Относительный путь
['folder1\\index.html']
>>> glob.glob("C:\\book\\folder1\\*[0-9].txt")
['C:\\book\\folder1\\file1.txt', 'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*\\*.html")
['C:\\book\\folder1\\folder1_1\\index.html',
'C:\\book\\folder1\\folder1_2\\test.html']
```

Обратите внимание на последний пример. Специальные символы могут быть указаны не только в названии файла, но и в именах каталогов в пути. Это позволяет просматривать сразу несколько каталогов в поисках объектов, соответствующих шаблону.

ГЛАВА 16

Основы SQLite

В предыдущей главе мы рассмотрели работу с файлами и научились сохранять объекты с доступом по ключу с помощью модуля `shelve`. При сохранении объектов этот модуль использует возможности модуля `pickle`, для сериализации объекта, и модуля `dbm`, для записи получившейся строки по ключу в файл. Если необходимо сохранять в файл просто строки, то можно сразу воспользоваться модулем `dbm`. Однако если объем сохраняемых данных велик и требуется удобный доступ к ним, то вместо этого модуля лучше использовать базы данных.

Начиная с версии 2.5, в состав стандартной библиотеки Python входит модуль `sqlite3`, позволяющий работать с базой данных SQLite. Для использования этой базы данных нет необходимости устанавливать сервер, ожидающий запросы на каком-либо порту, т. к. SQLite напрямую работает с файлом базы данных. Все что нужно для работы с SQLite — это библиотека sqlite3.dll (расположена в папке C:\Python32\DLLs) и язык программирования, позволяющий использовать эту библиотеку (например, Python). Необходимо заметить, что база данных SQLite не предназначена для проектов, предъявляющих требования к защите данных и разграничению прав доступа для нескольких пользователей. Тем не менее для небольших проектов SQLite является хорошей заменой полноценных баз данных.

Так как SQLite входит в состав стандартной библиотеки Python, мы на некоторое время отвлечемся от изучения языка Python и рассмотрим особенности использования языка SQL (Structured Query Language — структурированный язык запросов) применительно к базе данных SQLite. Для выполнения SQL-запросов мы воспользуемся программой sqlite3.exe, позволяющей работать с SQLite из командной строки. Со страницы **http://www.sqlite.org/download.html** загружаем архив, а затем распаковываем его в текущую папку. Далее копируем файл sqlite3.exe в каталог, с которым будем в дальнейшем работать (например, C:\book).

16.1. Создание базы данных

Попробуем создать новую базу данных. Запускаем командную строку. Для этого в меню **Пуск** выбираем пункт **Выполнить**. В открывшемся окне набираем команду `cmd` и нажимаем кнопку **OK**. Откроется черное окно, в котором будет приглашение для ввода команд. Переходим в папку C:\book, выполнив команду:

```
cd C:\book
```

В командной строке должно быть приглашение:

```
C:\book>
```

По умолчанию в консоли используется кодировка cp866. Чтобы сменить кодировку на cp1251, в командной строке вводим команду:

```
chcp 1251
```

Теперь необходимо изменить название шрифта, т. к. точечные шрифты не поддерживают кодировку Windows-1251. Щелкаем правой кнопкой мыши на заголовке окна и из контекстного меню выбираем пункт **Свойства**. В открывшемся окне переходим на вкладку **Шрифт** и в списке выделяем пункт **Lucida Console**. На этой же вкладке можно также установить размер шрифта. Нажимаем кнопку **OK**, чтобы изменения вступили в силу. Для проверки правильности установки кодировки вводим команду `chcp`. Результат выполнения должен выглядеть так:

```
C:\book>chcp
Текущая кодовая страница: 1251
```

Для создания новой базы данных вводим команду:

```
C:\book>sqlite3.exe testdb.db
```

Если файл testdb.db не существует, то будет создана новая база данных и открыта для дальнейшей работы. Если база данных уже существует, то она просто открывается без удаления содержимого. Результат выполнения команды будет выглядеть так:

```
SQLite version 3.7.6
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

> ***Примечание***
>
> В примерах следующих разделов предполагается, что база данных была открыта указанным способом. Поэтому запомните способ изменения кодировки в консоли и способ создания (или открытия) базы данных.

Фрагмент `"sqlite> "` является приглашением для ввода SQL-команд. Каждая SQL-команда должна завершаться точкой с запятой. Если точку с запятой не указать и нажать клавишу <Enter>, то приглашение примет вид `"   ...> "`. В качестве примера получим версию SQLite:

```
sqlite> SELECT sqlite_version();
3.7.6
sqlite> SELECT sqlite_version()
   ...> ;
3.7.6
```

SQLite позволяет использовать комментарии. Однострочный комментарий начинается с двух тире и оканчивается в конце строки. В этом случае после комментария точку с запятой указывать не нужно. Многострочный комментарий начинается с комбинации символов `/*` и заканчивается комбинацией `*/`. Допускается отсутствие завершающей комбинации символов. В этом случае комментируется фрагмент до конца файла. Многострочные комментарии не могут быть вложенными. Если внутри многострочного комментария расположен однострочный комментарий, то он игнорируется. Пример использования комментариев:

```
sqlite> -- Это однострочный комментарий
sqlite> /* Это многострочный комментарий */
sqlite> SELECT sqlite_version(); -- Комментарий после SQL-команды
3.7.6
```

```
sqlite> SELECT sqlite_version(); /* Комментарий после SQL-команды */
3.7.6
```

Чтобы завершить работу с SQLite и закрыть базу данных, следует выполнить команду `.exit` или `.quit`.

16.2. Создание таблицы

Создать таблицу в базе данных позволяет следующая SQL-команда:

```
CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS]
[<Название базы данных>.]<Название таблицы> (
   <Название поля1> [<Тип данных>] [<Опции>],
   [...,
   <Название поляN> [<Тип данных>] [<Опции>],]
   [<Дополнительные опции>]
);
```

Если после ключевого слова `CREATE` указано слово `TEMP` или `TEMPORARY`, то будет создана временная таблица. После закрытия базы данных временные таблицы автоматически удаляются. Пример создания временных таблиц:

```
sqlite> CREATE TEMP TABLE tmp1 (pole1);
sqlite> CREATE TEMPORARY TABLE tmp2 (pole1);
sqlite> .tables
tmp1  tmp2
```

Обратите внимание на предпоследнюю строку. С помощью команды `.tables` мы получаем список всех таблиц в базе данных. Эта команда работает только в утилите sqlite3.exe и является сокращенной записью следующего SQL-запроса:

```
sqlite> SELECT name FROM sqlite_master
   ...> WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%'
   ...> UNION ALL
   ...> SELECT name FROM sqlite_temp_master
   ...> WHERE type IN ('table','view')
   ...> ORDER BY 1;
tmp1
tmp2
```

Необязательные ключевые слова `IF NOT EXISTS` означают, что если таблица уже существует, то создавать таблицу заново не нужно. Если таблица уже существует и ключевые слова `IF NOT EXISTS` не указаны, то будет выведено сообщение об ошибке. Пример:

```
sqlite> CREATE TEMP TABLE tmp1 (pole3);
Error: table tmp1 already exists
sqlite> CREATE TEMP TABLE IF NOT EXISTS tmp1 (pole3);
sqlite> PRAGMA table_info(tmp1);
0|pole1||0||0
```

В этом примере мы использовали SQL-команду `PRAGMA table_info(<Название таблицы>)`, позволяющую получить информацию о полях таблицы (название поля, тип данных, значение по умолчанию и др.). Как видно из результата, структура временной таблицы `tmp1` не изменилась после выполнения запроса на создание таблицы с таким же названием.

В параметрах <Название таблицы> и <Название поля> указывается идентификатор или строка. В идентификаторах лучше использовать только буквы латинского алфавита, цифры и символ подчеркивания. Имена, начинающиеся с префикса "sqlite_", зарезервированы для служебного использования. Если в этих параметрах указывается идентификатор, то название не должно содержать пробелов и не должно совпадать с ключевыми словами SQL. Например, при попытке назвать таблицу именем table будет выведено сообщение об ошибке:

```
sqlite> CREATE TEMP TABLE table (pole1);
Error: near "table": syntax error
```

Если вместо идентификатора указать строку, то сообщения об ошибке не возникнет:

```
sqlite> CREATE TEMP TABLE "table" (pole1);
sqlite> .tables
table  tmp1   tmp2
```

Кроме того, идентификатор можно разместить внутри квадратных скобок:

```
sqlite> DROP TABLE "table";
sqlite> CREATE TEMP TABLE [table] (pole1);
sqlite> .tables
table  tmp1   tmp2
```

> ***Примечание***
>
> Хотя ошибки и удается избежать, на практике не стоит использовать ключевые слова SQL в качестве названия таблицы или поля.

Обратите внимание на первую строку примера. С помощью SQL-команды DROP TABLE <Название таблицы> мы удаляем таблицу table из базы данных. Если этого не сделать, то попытка создать таблицу, при наличии уже существующей одноименной таблицы, приведет к выводу сообщения об ошибке. SQL-команда DROP TABLE позволяет удалить как обычную таблицу, так и временную таблицу.

В целях совместимости с другими базами данных, значение, указанное в параметре <Тип данных>, преобразуется в один из пяти классов родства:

- INTEGER — класс будет назначен, если значение содержит фрагмент "INT" в любом месте. Этому классу родства соответствуют типы данных INT, INTEGER, TINYINT, SMALLINT, MEDIUMINT, BIGINT и др.;
- TEXT — если значение содержит фрагменты "CHAR", "CLOB" или "TEXT". Например, TEXT, CHARACTER(30), VARCHAR(250), VARYING CHARACTER(100), CLOB и др. Все значения внутри круглых скобок игнорируются;
- NONE — если значение содержит фрагмент "BLOB" или тип данных не указан;
- REAL — если значение содержит фрагменты "REAL", "FLOA" или "DOUB". Например, REAL, DOUBLE, DOUBLE PRECISION, FLOAT;
- NUMERIC — если все предыдущие условия не выполняются, то назначается этот класс родства.

> ***Внимание!***
>
> Все классы указаны в порядке уменьшения приоритета определения родства. Например, если значение соответствует сразу двум классам INTEGER и TEXT, то будет назначен класс INTEGER, т. к. его приоритет выше.

Классы родства являются лишь обозначением предполагаемого типа данных, а не строго определенным значением. Иными словами, SQLite использует не статическую типизацию (как в большинстве баз данных), а динамическую типизацию. Например, если для поля указан класс `INTEGER`, то при вставке значения производится попытка преобразовать введенные данные в целое число. Если преобразовать не получилось, то производится попытка преобразовать введенные данные в вещественное число. Если данные нельзя преобразовать в целое или вещественное число, то будет произведена попытка преобразовать в строку и т. д. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1 INTEGER,  p2 INTEGER,
   ...>  p3 INTEGER,  p4 INTEGER,  p5 INTEGER);
sqlite> INSERT INTO tmp3 VALUES (10, "00547", 5.45, "Строка", NULL);
sqlite> SELECT * FROM tmp3;
10|547|5.45|Строка|
sqlite> SELECT typeof(p1), typeof(p2), typeof(p3), typeof(p4),
   ...> typeof(p5) FROM tmp3;
integer|integer|real|text|null
sqlite> DROP TABLE tmp3;
```

В этом примере мы воспользовались встроенной функцией `typeof()` для определения типа данных, хранящихся в ячейке таблицы. SQLite поддерживает следующие типы данных:

- `NULL` — значение `NULL`;
- `INTEGER` — целые числа;
- `REAL` — вещественные числа;
- `TEXT` — строки;
- `BLOB` — бинарные данные.

Если после `INTEGER` указаны ключевые слова `PRIMARY KEY` (т. е. поле является первичным ключом), то в это поле можно вставить только целые числа или значение `NULL`. При указании значения `NULL` будет вставлено число на единицу большее максимального числа в столбце. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1 INTEGER PRIMARY KEY);
sqlite> INSERT INTO tmp3 VALUES (10);   -- Нормально
sqlite> INSERT INTO tmp3 VALUES (5.78); -- Ошибка
Error: datatype mismatch
sqlite> INSERT INTO tmp3 VALUES ("Строка"); -- Ошибка
Error: datatype mismatch
sqlite> INSERT INTO tmp3 VALUES (NULL);
sqlite> SELECT * FROM tmp3;
10
11
sqlite> DROP TABLE tmp3;
```

Класс `NUMERIC` аналогичен классу `INTEGER`. Различие между этими классами проявляется только при явном преобразовании типов с помощью инструкции `CAST`. Если строку, содержащую вещественное число, преобразовать в класс `INTEGER`, то дробная часть будет отброшена. Если строку, содержащую вещественное число, преобразовать в класс `NUMERIC`, то возможны два варианта:

- если преобразование в целое число возможно без потерь, то данные будут иметь тип `INTEGER`;
- в противном случае — тип `REAL`.

Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1 TEXT);
sqlite> INSERT INTO tmp3 VALUES ("00012.86");
sqlite> INSERT INTO tmp3 VALUES ("52.0");
sqlite> SELECT p1, typeof(p1) FROM tmp3;
00012.86|text
52.0|text
sqlite> SELECT CAST (p1 AS INTEGER) FROM tmp3;
12
52
sqlite> SELECT CAST (p1 AS NUMERIC) FROM tmp3;
12.86
52
sqlite> DROP TABLE tmp3;
```

В параметре <Опции> могут быть указаны следующие конструкции:

- NOT NULL [<Обработка ошибок>] — означает, что поле обязательно должно иметь значение при вставке новой записи. Если опция не указана, то поле может содержать значение NULL;
- DEFAULT <Значение> — задает для поля значение по умолчанию, которое будет использовано, если при вставке записи для этого поля не было явно указано значение. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1, p2 INTEGER DEFAULT 0);
sqlite> INSERT INTO tmp3 (p1) VALUES (800);
sqlite> INSERT INTO tmp3 VALUES (5, 1204);
sqlite> SELECT * FROM tmp3;
800|0
5|1204
sqlite> DROP TABLE tmp3;
```

  В параметре <Значение> можно указать специальные значения:

  - CURRENT_TIME — текущее время UTC в формате ЧЧ:ММ:СС;
  - CURRENT_DATE — текущая дата UTC в формате ГГГГ-ММ-ДД;
  - CURRENT_TIMESTAMP — текущая дата и время UTC в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС.

  Пример указания специальных значений:

```
sqlite> CREATE TEMP TABLE tmp3 (id INTEGER,
   ...> t TEXT DEFAULT CURRENT_TIME,
   ...> d TEXT DEFAULT CURRENT_DATE,
   ...> dt TEXT DEFAULT CURRENT_TIMESTAMP);
sqlite> INSERT INTO tmp3 (id) VALUES (1);
sqlite> SELECT * FROM tmp3;
1|01:30:19|2011-04-30|2011-04-30 01:30:19
sqlite> /* Текущая дата на компьютере: 2011-04-30 05:30:19 */
sqlite> DROP TABLE tmp3;
```

- COLLATE <Функция> — задает функцию сравнения для класса TEXT. Могут быть указаны функции BINARY (значение по умолчанию), NOCASE (без учета регистра) и RTRIM.

Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1, p2 TEXT COLLATE NOCASE);
sqlite> INSERT INTO tmp3 VALUES ("abcd", "abcd");
sqlite> SELECT p1 = "ABCD" FROM tmp3; -- Не найдено
0
sqlite> SELECT p2 = "ABCD" FROM tmp3; -- Найдено
1
sqlite> DROP TABLE tmp3;
```

Примечание

При использовании NOCASE возможны проблемы с регистром русских букв.

- UNIQUE [<Обработка ошибок>] — указывает, что поле может содержать только уникальные значения;
- CHECK(<Условие>) — значение, вставляемое в поле, должно удовлетворять указанному условию. В качестве примера ограничим значения числами 10 и 20:

```
sqlite> CREATE TEMP TABLE tmp3 (
   ...> p1 INTEGER CHECK(p1 IN (10, 20)));
sqlite> INSERT INTO tmp3 VALUES (10); -- OK
sqlite> INSERT INTO tmp3 VALUES (30); -- Ошибка
Error: constraint failed
sqlite> DROP TABLE tmp3;
```

- PRIMARY KEY [ASC | DESC] [<Обработка ошибок>] [AUTOINCREMENT] — указывает, что поле является первичным ключом таблицы. Записи в таком поле должны быть уникальными. Если полю назначен класс INTEGER, то в это поле можно вставить только целые числа или значение NULL. При указании значения NULL будет вставлено число на единицу большее максимального числа в столбце. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (id INTEGER PRIMARY KEY, t TEXT);
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка1");
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка2");
sqlite> SELECT * FROM tmp3;
1|Строка1
2|Строка2
sqlite> DELETE FROM tmp3 WHERE id=2;
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка3");
sqlite> SELECT * FROM tmp3;
1|Строка1
2|Строка3
sqlite> DROP TABLE tmp3;
```

В этом примере мы вставили две записи. Так как при вставке для первого поля указано значение NULL, новая запись всегда будет иметь значение на единицу больше максимального числа в поле. Если удалить последнюю запись, а затем вставить новую запись, то запись будет иметь такой же индекс, что и удаленная. Чтобы индекс всегда был уникальным, необходимо дополнительно указать ключевое слово AUTOINCREMENT. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (
   ...> id INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> t TEXT);
```

```
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка1");
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка2");
sqlite> SELECT * FROM tmp3;
1|Строка1
2|Строка2
sqlite> DELETE FROM tmp3 WHERE id=2;
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка3");
sqlite> SELECT * FROM tmp3;
1|Строка1
3|Строка3
sqlite> DROP TABLE tmp3;
```

Обратите внимание на индекс последней вставленной записи. Индекс имеет значение 3, а не 2, как это было в предыдущем примере. Таким образом, индекс новой записи всегда будет уникальным.

Если в таблице не существует поля с первичным ключом, то получить индекс записи можно с помощью специальных названий полей: ROWID, OID или _ROWID_. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (t TEXT);
sqlite> INSERT INTO tmp3 VALUES ("Строка1");
sqlite> INSERT INTO tmp3 VALUES ("Строка2");
sqlite> SELECT ROWID, OID, _ROWID_, t FROM tmp3;
1|1|1|Строка1
2|2|2|Строка2
sqlite> DELETE FROM tmp3 WHERE OID=2;
sqlite> INSERT INTO tmp3 VALUES ("Строка3");
sqlite> SELECT ROWID, OID, _ROWID_, t FROM tmp3;
1|1|1|Строка1
2|2|2|Строка3
sqlite> DROP TABLE tmp3;
```

В необязательном параметре <Дополнительные опции> могут быть указаны следующие конструкции:

- PRIMARY KEY (<Список полей через запятую>) [<Обработка ошибок>] — позволяет задать первичный ключ для нескольких полей таблицы;
- UNIQUE (<Список полей через запятую>) [<Обработка ошибок>] — указывает, что заданные поля могут содержать только уникальные значения;
- CHECK(<Условие>) — значение должно удовлетворять указанному условию.

Необязательный параметр <Обработка ошибок> во всех рассмотренных в этом разделе конструкциях задает способ разрешения конфликтных ситуаций. Формат конструкции:

```
ON CONFLICT <Алгоритм>
```

В параметре <Алгоритм> указываются следующие значения:

- ROLLBACK — при ошибке транзакция завершается с откатом всех измененных ранее записей, дальнейшее выполнение прерывается и выводится сообщение об ошибке. Если активной транзакции нет, то используется алгоритм ABORT;
- ABORT — при возникновении ошибки аннулируются все изменения, произведенные текущей командой, и выводится сообщение об ошибке. Все изменения, сделанные предыдущими командами в транзакции, сохраняются. Алгоритм ABORT используется по умолчанию;

- FAIL — при возникновении ошибки все изменения, произведенные текущей командой, сохраняются, а не аннулируются как в алгоритме ABORT. Дальнейшее выполнение команды прерывается и выводится сообщение об ошибке. Все изменения, сделанные предыдущими командами в транзакции, сохраняются;
- IGNORE — проигнорировать ошибку и продолжить выполнение без вывода сообщения об ошибке;
- REPLACE — при нарушении условия UNIQUE существующая запись удаляется, а новая вставляется. Сообщение об ошибке не выводится. При нарушении условия NOT NULL значение NULL заменяется значением по умолчанию. Если значение по умолчанию не задано для поля, то используется алгоритм ABORT. Если нарушено условие CHECK, применяется алгоритм IGNORE. Пример обработки условия UNIQUE:

```
sqlite> CREATE TEMP TABLE tmp3 (
   ...> id UNIQUE ON CONFLICT REPLACE, t TEXT);
sqlite> INSERT INTO tmp3 VALUES (10, "s1");
sqlite> INSERT INTO tmp3 VALUES (10, "s2");
sqlite> SELECT * FROM tmp3;
10|s2
sqlite> DROP TABLE tmp3;
```

16.3. Вставка записей

Для добавления записей в таблицу используется инструкция INSERT. Формат инструкции:

```
INSERT [OR <Алгоритм>] INTO [<Название базы данных>.]<Название таблицы>
[(<Поле1>, <Поле2>, ...)] VALUES (<Значение1>, <Значение2>, ...);
```

Необязательный параметр OR <Алгоритм> задает алгоритм обработки ошибок (ROLLBACK, ABORT, FAIL, IGNORE или REPLACE). Все эти алгоритмы мы уже рассматривали в предыдущем разделе. После названия таблицы внутри круглых скобок могут быть перечислены поля, которым будут присваиваться значения, указанные в круглых скобках после ключевого слова VALUES. Количество параметров должно совпадать. Если в таблице существуют поля, которым в инструкции INSERT не присваивается значение, то они получат значения по умолчанию. Если список полей не указан, то значения задаются в том порядке, в котором поля перечислены в инструкции CREATE TABLE.

Создадим три таблицы user (данные о пользователе), rubr (название рубрики) и site (описание сайта):

```
sqlite> CREATE TABLE user (
   ...> id_user INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> email TEXT,
   ...> passw TEXT);
sqlite> CREATE TABLE rubr (
   ...> id_rubr INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> name_rubr TEXT);
sqlite> CREATE TABLE site (
   ...> id_site INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> id_user INTEGER,
   ...> id_rubr INTEGER,
```

```
   ...> url TEXT,
   ...> title TEXT,
   ...> msg TEXT);
```

Такая структура таблиц характерна для реляционных баз данных и позволяет избежать дублирования данных в таблицах, ведь одному пользователю может принадлежать несколько сайтов, а в одной рубрике можно зарегистрировать множество сайтов. Если в таблице `site` каждый раз указывать название рубрики, то при необходимости переименовать рубрику придется изменять названия во всех записях, где встречается старое название. Если же названия рубрик расположены в отдельной таблице, то изменить название можно будет только в одном месте. Все остальные записи будут связаны целочисленным идентификатором. Как получить данные сразу из нескольких таблиц, мы рассмотрим по мере изучения SQLite.

Теперь заполним таблицы связанными данными:

```
sqlite> INSERT INTO user (email, passw)
   ...> VALUES ('unicross@mail.ru', 'password1');
sqlite> INSERT INTO rubr VALUES (NULL, 'Программирование');
sqlite> SELECT * FROM user;
1|unicross@mail.ru|password1
sqlite> SELECT * FROM rubr;
1|Программирование
sqlite> INSERT INTO site (id_user, id_rubr, url, title, msg)
   ...> VALUES (1, 1, 'http://wwwadmin.ru', 'Название', 'Описание');
```

В первом примере перечислены только поля `email` и `passw`. Так как поле `id_user` не указано, то ему присваивается значение по умолчанию. В таблице `user` поле `id_user` объявлено как первичный ключ, поэтому будет вставлено значение на единицу большее максимального значения в поле. Такого же эффекта можно достичь, если в качестве значения передать `NULL`. Это демонстрируется во втором примере. В третьем примере вставляется запись в таблицу `site`. Поля `id_user` и `id_rubr` в этой таблице должны содержать идентификаторы соответствующих записей из таблиц `user` и `rubr`. Поэтому вначале мы делаем запросы на выборку данных и смотрим, какой идентификатор был присвоен вставленным записям в таблицы `user` и `rubr`. Обратите внимание на то, что мы опять указываем названия полей явным образом. Хотя перечислять поля и необязательно, но лучше всегда так делать. Тогда в дальнейшем можно будет изменить структуру таблицы (например, добавить поле) без необходимости изменять все SQL-запросы. Достаточно для нового поля указать значение по умолчанию, и все старые запросы будут по-прежнему рабочими.

Во всех этих примерах строковые значения указываются внутри одинарных кавычек. Однако бывают ситуации, когда внутри строки уже содержится одинарная кавычка. Попытка вставить такую строку приведет к ошибке:

```
sqlite> INSERT INTO rubr VALUES (NULL, 'Название 'в кавычках'');
Error: near "в": syntax error
```

Чтобы избежать этой ошибки, можно заключить строку в двойные кавычки или удвоить каждую одинарную кавычку внутри строки:

```
sqlite> INSERT INTO rubr VALUES (NULL, "Название 'в кавычках'");
sqlite> INSERT INTO rubr VALUES (NULL, 'Название ''в кавычках''');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Название 'в кавычках'
3|Название 'в кавычках'
```

Если предпринимается попытка вставить запись, а в таблице уже есть запись с таким же значением первичного ключа (или значение индекса UNIQUE не уникально), то такая SQL-команда приводит к ошибке. Если необходимо, чтобы такие неуникальные записи обновлялись без вывода сообщения об ошибке, можно указать алгоритм обработки ошибок REPLACE после ключевого слова OR. Заменим название рубрики с идентификатором 2:

```
sqlite> INSERT OR REPLACE INTO rubr
   ...> VALUES (2, 'Музыка');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Название 'в кавычках'
```

Вместо алгоритма REPLACE можно использовать инструкцию REPLACE INTO. Инструкция имеет следующий формат:

```
REPLACE INTO [<Название базы данных>.]<Название таблицы>
[(<Поле1>, <Поле2>, ...)] VALUES (<Значение1>, <Значение2>, ...);
```

Заменим название рубрики с идентификатором 3:

```
sqlite> REPLACE INTO rubr VALUES (3, 'Игры');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Игры
```

16.4. Обновление и удаление записей

Обновление записи осуществляется с помощью инструкции UPDATE. Формат инструкции:

```
UPDATE [OR <Алгоритм>] [<Название базы данных>.]<Название таблицы>
SET <Поле1>='<Значение>', <Поле2>='<Значение2>', ...
[WHERE <Условие>];
```

Необязательный параметр OR <Алгоритм> задает алгоритм обработки ошибок (ROLLBACK, ABORT, FAIL, IGNORE или REPLACE). Все эти алгоритмы мы уже рассматривали при изучении создания таблицы. После ключевого слова SET указываются названия полей и их новые значения после знака равенства. Чтобы ограничить набор изменяемых записей, применяется инструкция WHERE. Обратите внимание на то, что если не указано <Условие>, то будут обновлены все записи в таблице. Какие выражения можно указать в параметре <Условие>, мы рассмотрим немного позже.

В качестве примера изменим название рубрики с идентификатором 3:

```
sqlite> UPDATE rubr SET name_rubr='Кино' WHERE id_rubr=3;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Кино
```

Удаление записи осуществляется с помощью инструкции DELETE. Формат инструкции:

```
DELETE FROM [<Название базы данных>.]<Название таблицы>
[WHERE <Условие>];
```

Если условие не указано, то будут удалены все записи из таблицы. В противном случае удаляются только записи, соответствующие условию. В качестве примера удалим рубрику с идентификатором 3:

```
sqlite> DELETE FROM rubr WHERE id_rubr=3;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
```

Частое обновление и удаление записей приводит к дефрагментации таблицы. Чтобы освободить неиспользуемое пространство, можно воспользоваться SQL-командой VACUUM. Обратите внимание на то, что SQL-команда может изменить порядок нумерации в специальных полях (ROWID, OID и _ROWID_).

16.5. Изменение свойств таблицы

В некоторых случаях необходимо изменить структуру уже созданной таблицы. Для этого используется инструкция ALTER TABLE. В SQLite инструкция ALTER TABLE позволяет выполнить лишь ограниченное количество операций. Например, нельзя изменить свойство поля или удалить его из таблицы. Формат инструкции:

```
ALTER TABLE [<Название базы данных>.]<Название таблицы>
<Преобразование>;
```

В параметре <Преобразование> могут быть указаны следующие конструкции:

- RENAME TO <Новое имя таблицы> — переименовывает таблицу. Изменим название таблицы user на users:

```
sqlite> .tables
rubr             sqlite_sequence  tmp1             user
site             table            tmp2
sqlite> ALTER TABLE user RENAME TO users;
sqlite> .tables
rubr             sqlite_sequence  tmp1             users
site             table            tmp2
```

- ADD [COLUMN] <Имя нового поля> [<Тип данных>] [<Опции>] — добавляет новое поле после всех имеющихся полей. Обратите внимание на то, что в новом поле нужно задать значение по умолчанию, или значение NULL должно быть допустимым, т. к. в таблице уже есть записи. Кроме того, поле не может быть объявлено как PRIMARY KEY или UNIQUE. Добавим поле iq в таблицу site:

```
sqlite> ALTER TABLE site ADD COLUMN iq INTEGER DEFAULT 0;
sqlite> PRAGMA table_info(site);
0|id_site|INTEGER|0||1
1|id_user|INTEGER|0||0
2|id_rubr|INTEGER|0||0
3|url|TEXT|0||0
4|title|TEXT|0||0
5|msg|TEXT|0||0
6|iq|INTEGER|0|0|0
sqlite> SELECT * FROM site;
1|1|1|http://wwwadmin.ru|Название|Описание|0
```

Внимание!

При использовании SQLite версии 3.1.3 и ниже после добавления нового поля необходимо выполнить инструкцию VACUUM.

16.6. Выбор записей

Для извлечения данных из таблицы предназначена инструкция SELECT. Инструкция имеет следующий формат:

```
SELECT [ALL | DISTINCT]
[<Название таблицы>.]<Поле>[, ...]
[ FROM <Название таблицы> [AS <Псевдоним>][, ...] ]
[ WHERE <Условие> ]
[ [ GROUP BY <Название поля> ] [ HAVING <Условие> ] ]
[ ORDER BY <Название поля> [COLLATE BINARY | NOCASE] [ASC | DESC][,...] ]
[ LIMIT <Ограничение> ]
```

SQL-команда SELECT ищет все записи в указанной таблице, которые удовлетворяют условию в инструкции WHERE. Если инструкция WHERE не указана, то будут возвращены все записи из таблицы. Получим все записи из таблицы rubr:

```
sqlite> SELECT id_rubr, name_rubr FROM rubr;
1|Программирование
2|Музыка
```

Теперь выведем только запись с идентификатором 1:

```
sqlite> SELECT id_rubr, name_rubr FROM rubr WHERE id_rubr=1;
1|Программирование
```

Вместо перечисления полей можно указать символ *. В этом случае будут возвращены значения всех полей. Получим все записи из таблицы rubr:

```
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
```

SQL-команда SELECT позволяет вместо перечисления полей указать выражение. Это выражение будет вычислено, и возвращен результат:

```
sqlite> SELECT 10 + 5;
15
```

Чтобы из программы было легче обратиться к результату выполнения выражения, можно назначить псевдоним, указав его после выражения через ключевое слово AS:

```
sqlite> SELECT (10 + 5) AS expr1, (70 * 2) AS expr2;
15|140
```

Псевдоним можно назначить также таблицам. Это особенно полезно при выборке из нескольких таблиц сразу. В качестве примера заменим индекс рубрики в таблице site на соответствующее название из таблицы rubr:

```
sqlite> SELECT s.url, r.name_rubr FROM site AS s, rubr AS r
   ...> WHERE s.id_rubr = r.id_rubr;
http://wwwadmin.ru|Программирование
```

В этом примере мы назначили псевдонимы сразу двум таблицам. Теперь при указании списка полей достаточно задать псевдоним перед названием поля через точку, а не указывать полные названия таблиц. Более подробно выбор записей сразу из нескольких таблиц мы рассмотрим в следующем разделе.

После ключевого слова SELECT можно указать слово ALL или DISTINCT. Слово ALL является значением по умолчанию и говорит, что возвращаются все записи. Если указано слово DISTINCT, то в результат попадут только уникальные значения.

Инструкция GROUP BY позволяет сгруппировать несколько записей. Эта инструкция особенно полезна при использовании агрегатных функций. В качестве примера добавим одну рубрику и два сайта:

```
sqlite> INSERT INTO rubr VALUES (3, 'Поисковые порталы');
sqlite> INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
   ...> VALUES (1, 1, 'http://python.org', 'Python', '', 1000);
sqlite> INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
   ...> VALUES (1, 3, 'http://google.ru', 'Гугль', '', 3000);
```

Теперь выведем количество сайтов в каждой рубрике:

```
sqlite> SELECT id_rubr, COUNT(id_rubr) FROM site
   ...> GROUP BY id_rubr;
1|2
3|1
```

Если необходимо ограничить сгруппированный набор записей, то следует воспользоваться инструкцией HAVING. Эта инструкция выполняет те же функции, что и инструкция WHERE, но только для сгруппированного набора. В качестве примера выведем номера рубрик, в которых зарегистрировано более одного сайта:

```
sqlite> SELECT id_rubr FROM site
   ...> GROUP BY id_rubr HAVING COUNT(id_rubr)>1;
1
```

В этих примерах мы воспользовались агрегатной функцией COUNT(), которая возвращает количество записей. Перечислим агрегатные функции, используемые наиболее часто:

- COUNT(<Поле> | *) — количество записей в указанном поле. Выведем количество зарегистрированных сайтов:

  ```
  sqlite> SELECT COUNT(*) FROM site;
  3
  ```

- MIN(<Поле>) — минимальное значение в указанном поле. Выведем минимальный коэффициент релевантности:

  ```
  sqlite> SELECT MIN(iq) FROM site;
  0
  ```

- MAX(<Поле>) — максимальное значение в указанном поле. Выведем максимальный коэффициент релевантности:

  ```
  sqlite> SELECT MAX(iq) FROM site;
  3000
  ```

- AVG(<Поле>) — средняя величина значений в указанном поле. Выведем среднее значение коэффициента релевантности:

```
sqlite> SELECT AVG(iq) FROM site;
1333.33333333333
```

- SUM(<Поле>) — сумма значений в указанном поле. Выведем сумму значений коэффициентов релевантности:

```
sqlite> SELECT SUM(iq) FROM site;
4000
```

Найденные записи можно отсортировать с помощью инструкции ORDER BY. Допустимо производить сортировку сразу по нескольким полям. По умолчанию записи сортируются по возрастанию (значение ASC). Если в конце указано слово DESC, то записи будут отсортированы в обратном порядке. После ключевого слова COLLATE может быть указана функция сравнения (BINARY или NOCASE). Выведем названия рубрик по возрастанию и убыванию:

```
sqlite> SELECT * FROM rubr ORDER BY name_rubr;
2|Музыка
3|Поисковые порталы
1|Программирование
sqlite> SELECT * FROM rubr ORDER BY name_rubr DESC;
1|Программирование
3|Поисковые порталы
2|Музыка
```

Если требуется, чтобы при поиске выводились не все найденные записи, а лишь их часть, то следует использовать инструкцию LIMIT. Например, если таблица site содержит много описаний сайтов, то вместо того чтобы выводить все сайты за один раз, можно выводить их частями, скажем, по 10 сайтов за раз. Инструкция имеет следующие форматы:

```
LIMIT <Количество записей>
LIMIT <Начальная позиция>, <Количество записей>
LIMIT <Количество записей> OFFSET <Начальная позиция>
```

Первый формат задает количество записей от начальной позиции. Обратите внимание на то, что начальная позиция имеет индекс 0. Второй и третий форматы позволяют явно указать начальную позицию и количество записей. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (id INTEGER);
sqlite> INSERT INTO tmp3 VALUES(1);
sqlite> INSERT INTO tmp3 VALUES(2);
sqlite> INSERT INTO tmp3 VALUES(3);
sqlite> INSERT INTO tmp3 VALUES(4);
sqlite> INSERT INTO tmp3 VALUES(5);
sqlite> SELECT * FROM tmp3 LIMIT 3; -- Эквивалентно LIMIT 0, 3
1
2
3
sqlite> SELECT * FROM tmp3 LIMIT 2, 3;
3
4
5
sqlite> SELECT * FROM tmp3 LIMIT 3 OFFSET 2;
3
4
5
sqlite> DROP TABLE tmp3;
```

16.7. Выбор записей из нескольких таблиц

SQL-команда SELECT позволяет выбирать записи сразу из нескольких таблиц одновременно. Для этого используются следующие форматы инструкции FROM:

```
FROM <Название таблицы1> [AS <Псевдоним>]
, | [NATURAL] [LEFT] [OUTER | INNER | CROSS] JOIN
<Название таблицы2> [AS <Псевдоним>]
[ON <Выражение>] [USING ( <Поле> )]
```

В первом формате таблицы перечисляются через запятую в инструкции FROM, а в инструкции WHERE через запятую указываются пары полей, являющиеся связуемыми для таблиц. Причем в условии и перечислении полей вначале указывается название таблицы (или псевдоним), а затем через точку название поля. В качестве примера выведем сайты из таблицы site, но вместо индекса пользователя укажем его e-mail, а вместо индекса рубрики ее название:

```
sqlite> SELECT site.url, rubr.name_rubr, users.email
   ...> FROM rubr, users, site
   ...> WHERE site.id_rubr=rubr.id_rubr AND
   ...> site.id_user=users.id_user;
http://wwwadmin.ru|Программирование|unicross@mail.ru
http://python.org|Программирование|unicross@mail.ru
http://google.ru|Поисковые порталы|unicross@mail.ru
```

Вместо названия таблиц можно использовать псевдоним. Кроме того, если поля в таблицах имеют разные названия, то название таблицы можно не указывать:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr AS r, users AS u, site AS s
   ...> WHERE s.id_rubr=r.id_rubr AND
   ...> s.id_user=u.id_user;
```

Объединить таблицы позволяет также оператор JOIN, который имеет два синонима: CROSS JOIN и INNER JOIN. Переделаем наш предыдущий пример и используем оператор JOIN:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr JOIN users JOIN site
   ...> WHERE site.id_rubr=rubr.id_rubr AND
   ...> site.id_user=users.id_user;
```

Инструкцию WHERE можно заменить инструкцией ON, а в инструкции WHERE указать дополнительное условие. В качестве примера выведем сайты, зарегистрированные в рубрике с идентификатором 1:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr JOIN users JOIN site
   ...> ON site.id_rubr=rubr.id_rubr AND
   ...> site.id_user=users.id_user
   ...> WHERE site.id_rubr=1;
```

Если названия связующих полей в таблицах являются одинаковыми, то вместо инструкции ON можно использовать инструкцию USING:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr JOIN site USING (id_rubr) JOIN users USING (id_user);
```

Оператор JOIN объединяет все записи, которые существуют во всех связующих полях. Например, если попробовать вывести количество сайтов в каждой рубрике, то мы не получим рубрики без зарегистрированных сайтов:

```
sqlite> SELECT name_rubr, COUNT(id_site)
   ...> FROM rubr JOIN site USING (id_rubr)
   ...> GROUP BY rubr.id_rubr;
Программирование|2
Поисковые порталы|1
```

В этом примере мы не получили количество сайтов в рубрике "Музыка", т. к. в этой рубрике нет сайтов. Чтобы получить количество сайтов во всех рубриках, необходимо использовать левостороннее объединение. Формат левостороннего объединения:

```
<Таблица1> LEFT [OUTER] JOIN <Таблица2>
ON <Таблица1>.<Поле1>=<Таблица2>.<Поле2> | USING (<Поле>)
```

При левостороннем объединении возвращаются записи, соответствующие условию, а также записи из таблицы <Таблица1>, которым нет соответствия в таблице <Таблица2> (при этом поля из таблицы <Таблица2> будут иметь значение NULL). Выведем количество сайтов в рубриках и отсортируем по названию рубрики:

```
sqlite> SELECT name_rubr, COUNT(id_site)
   ...> FROM rubr LEFT JOIN site USING (id_rubr)
   ...> GROUP BY rubr.id_rubr
   ...> ORDER BY rubr.name_rubr;
Музыка|0
Поисковые порталы|1
Программирование|2
```

16.8. Условия в инструкции *WHERE*

В предыдущих разделах мы оставили без внимания рассмотрение выражений в инструкциях WHERE и HAVING. Эти инструкции позволяют ограничить набор выводимых, изменяемых или удаляемых записей с помощью некоторого условия. Внутри условий можно использовать следующие операторы сравнения:

- = или == — проверка на равенство. Пример:

  ```
  sqlite> SELECT * FROM rubr WHERE id_rubr=1;
  1|Программирование
  sqlite> SELECT 10 = 10, 5 = 10, 10 == 10, 5 == 10;
  1|0|1|0
  ```

  Как видно из примера, выражения можно разместить не только в инструкциях WHERE и HAVING, но и после ключевого слова SELECT. В этом случае результатом операции сравнения являются следующие значения:

  - 0 — ложь;
  - 1 — истина;
  - NULL.

  Результат сравнения двух строк зависит от используемой функции сравнения. Задать функцию можно при создании таблицы с помощью инструкции COLLATE <Функция>.

В параметре <Функция> указывается функция BINARY (значение по умолчанию), NOCASE (без учета регистра) или RTRIM. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1, p2 TEXT COLLATE NOCASE);
sqlite> INSERT INTO tmp3 VALUES ("abcd", "abcd");
sqlite> SELECT p1 = "ABCD" FROM tmp3; -- Не найдено
0
sqlite> SELECT p2 = "ABCD" FROM tmp3; -- Найдено
1
sqlite> DROP TABLE tmp3;
```

Указать функцию сравнения можно также после выражения:

```
sqlite> SELECT 's' = 'S', 's' = 'S' COLLATE NOCASE;
0|1
```

Функция NOCASE не учитывает регистр только латинских букв. При использовании русских букв возможны проблемы с регистром. Пример:

```
sqlite> SELECT 'ы' = 'Ы', 'ы' = 'Ы' COLLATE NOCASE;
0|0
```

- != или <> — не равно:

```
sqlite> SELECT 10 != 10, 5 != 10, 10 <> 10, 5 <> 10;
0|1|0|1
```

- < — меньше;
- > — больше;
- <= — меньше или равно;
- >= — больше или равно;
- IS NOT NULL, NOT NULL или NOTNULL — проверка на наличие значения;
- IS NULL или ISNULL — проверка на отсутствие значения;
- BETWEEN <Начало> AND <Конец> — проверка на вхождение в диапазон значений. Пример:

```
sqlite> SELECT 100 BETWEEN 1 AND 100;
1
sqlite> SELECT 101 BETWEEN 1 AND 100;
0
```

- IN (<Список значений>) — проверка на наличие значения в определенном наборе. Сравнение зависит от регистра букв. Пример:

```
sqlite> SELECT 'один' IN ('один', 'два', 'три');
1
sqlite> SELECT 'Один' IN ('один', 'два', 'три');
0
```

- LIKE <Шаблон> [ESCAPE <Символ>] — проверка на соответствие шаблону. В шаблоне используются следующие специальные символы:
  - % — любое количество символов;
  - _ — любой одиночный символ.

  Специальные символы могут быть расположены в любом месте шаблона. Например, чтобы найти все вхождения, необходимо указать символ % в начале и в конце шаблона:

```
sqlite> SELECT 'test word test' LIKE '%word%';
1
```

Можно установить привязку или только к началу строки, или только к концу:

```
sqlite> SELECT 'test word test' LIKE 'test%';
1
sqlite> SELECT 'test word test' LIKE 'word%';
0
```

Кроме того, шаблон для поиска может иметь очень сложную структуру:

```
sqlite> SELECT 'test word test' LIKE '%es_%wo_d%';
1
sqlite> SELECT 'test word test' LIKE '%wor%d%';
1
```

Обратите внимание на последнюю строку поиска. Этот пример демонстрирует, что специальный символ `%` соответствует не только любому количеству символов, но и полному их отсутствию.

Что же делать, если необходимо найти символы `%` и _? Ведь они являются специальными. В этом случае специальные символы необходимо экранировать с помощью символа, указанного в инструкции `ESCAPE <Символ>`:

```
sqlite> SELECT '10$' LIKE '10%';
1
sqlite> SELECT '10$' LIKE '10\%' ESCAPE '\';
0
sqlite> SELECT '10%' LIKE '10\%' ESCAPE '\';
1
```

Следует учитывать, что сравнение с шаблоном для латинских букв производится без учета регистра символов. Чтобы учитывался регистр, необходимо присвоить значение `true` (или `1`, `yes`, `on`) параметру `case_sensitive_like` в SQL-команде `PRAGMA`. Пример:

```
sqlite> PRAGMA case_sensitive_like = true;
sqlite> SELECT 's' LIKE 'S';
0
sqlite> PRAGMA case_sensitive_like = false;
sqlite> SELECT 's' LIKE 'S';
1
```

Теперь посмотрим, учитывается ли регистр русских букв при поиске по шаблону:

```
sqlite> SELECT 'ы' LIKE 'Ы', 'ы' LIKE 'ы';
1|1
```

Результат выполнения примера показывает, что поиск производится без учета регистра. Однако, это далеко не так. Попробуем сравнить две разные буквы и два разных слова:

```
sqlite> SELECT 'г' LIKE 'Ы', 'слово' LIKE 'текст';
1|1
```

Этот пример показывает, что буква "г" равна букве "Ы", а "слово" равно "текст". Иными словами, производится сравнение длины строк, а не символов в строке. Такой странный

результат был получен при использовании кодировки Windows-1251. Если изменить кодировку на cp866, то результат выполнения примера будет другим:

```
C:\book>chcp 866
Текущая кодовая страница: 866

C:\book>sqlite3.exe testdb.db
SQLite version 3.7.6
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT 'г' LIKE 'Ы', 'слово' LIKE 'текст';
0|0
sqlite> SELECT 'ы' LIKE 'Ы', 'ы' LIKE 'ы';
0|1
```

Результат выполнения становится более логичным. Таким образом, поиск русских букв зависит от кодировки. По умолчанию в SQLite используется кодировка UTF-8. С помощью инструкции `PRAGMA encoding = <Кодировка>` можно указать другую кодировку. Поддерживаются кодировки UTF-8, UTF-16, UTF-16le и UTF-16be. В этот список не входят кодировки cp866 и Windows-1251, поэтому результат сравнения строк может быть некорректным. С кодировкой UTF-8 мы еще поработаем в следующей главе, а на данный момент следует запомнить, что результат сравнения русских букв зависит от регистра символов. Кроме того, если поиск сравнивает только длину строк, то необходимо проверить кодировку данных. В рабочих проектах данные должны быть в кодировке UTF-8.

Результат логического выражения можно изменить на противоположный. Для этого необходимо перед выражением разместить оператор `NOT`. Пример:

```
sqlite> SELECT 's' = 'S', NOT ('s' = 'S');
0|1
sqlite> SELECT NOT 'один' IN ('один', 'два', 'три');
0
```

Кроме того, допустимо проверять сразу несколько условий, указав между выражениями следующие операторы:

- `AND` — логическое И;
- `OR` — логическое ИЛИ.

16.9. Индексы

Все записи в полях таблицы расположены в произвольном порядке. Чтобы найти какие-либо данные, необходимо каждый раз просматривать все записи. Для ускорения выполнения запросов применяются *индексы (ключи)*. Индексированные поля всегда поддерживаются в отсортированном состоянии, что позволяет быстро найти необходимую запись, не просматривая все записи. Надо сразу заметить, что применение индексов приводит к увеличению размера базы данных, а также к затратам времени на поддержание индекса в отсортированном состоянии при каждом добавлении данных. По этой причине индексировать следует поля, которые очень часто используются в запросах типа:

```
SELECT <Список полей> FROM <Таблица> WHERE <Поле>=<Значение>;
```

В SQLite существуют следующие виды индексов:

- первичный ключ;
- уникальный индекс;
- обычный индекс.

Первичный ключ служит для однозначной идентификации каждой записи в таблице. Для создания индекса в инструкции `CREATE TABLE` используется ключевое слово `PRIMARY KEY`. Ключевое слово можно указать после описания поля или после перечисления всех полей. Второй вариант позволяет указать сразу несколько полей в качестве первичного ключа.

Посмотреть, каким образом будет выполняться запрос и какие индексы будут использоваться, позволяет SQL-команда `EXPLAIN`. Формат SQL-команды:

```
EXPLAIN [QUERY PLAN] <SQL-запрос>
```

Если ключевые слова `QUERY PLAN` не указаны, то выводится полный список параметров и их значений. Если ключевые слова указаны, то выводится информация об используемых индексах. В качестве примера попробуем выполнить запрос на извлечение записей из таблицы `site`. В первом случае поиск произведем в поле, являющемся первичным ключом, а во втором случае — в обычном поле:

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_site=1;
0|0|0|SEARCH TABLE site USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|0|SCAN TABLE site (~100000 rows)
```

В первом случае фраза "USING INTEGER PRIMARY KEY" означает, что при поиске будет использован первичный ключ, а во втором случае никакие индексы не используются.

В одной таблице не может быть более одного первичного ключа. А вот обычных и уникальных индексов допускается создать несколько. Для создания индекса применяется SQL-команда `CREATE INDEX`. Формат команды:

```
CREATE [UNIQUE] INDEX [IF NOT EXISTS]
[<Название базы данных>.]<Название индекса>
ON <Название таблицы>
(<Название поля> [ COLLATE <Функция сравнения>] [ ASC | DESC ][, ...])
```

Если между ключевыми словами `CREATE` и `INDEX` указано слово `UNIQUE`, то создается уникальный индекс. В этом случае дублирование данных в поле не допускается. Если слово `UNIQUE` не указано, то создается обычный индекс.

Все сайты в нашем каталоге распределяются по рубрикам. Это означает, что при выводе сайтов, зарегистрированных в определенной рубрике, в инструкции `WHERE` будет постоянно выполняться условие:

```
WHERE id_rubr=<Номер рубрики>
```

Чтобы ускорить выборку сайтов по номеру рубрики, создадим обычный индекс для этого поля и проверим с помощью SQL-команды `EXPLAIN`, задействуется ли этот индекс:

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|0|SCAN TABLE site (~100000 rows)
sqlite> CREATE INDEX index_rubr ON site (id_rubr);
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|0|SEARCH TABLE site USING INDEX index_rubr (id_rubr=?) (~10 rows)
```

Обратите внимание на то, что после создания индекса добавилась фраза "USING INDEX index_rubr". Это означает, что теперь при поиске будет задействован индекс, и поиск будет выполняться быстрее. При выполнении запроса название индекса явным образом указывать нет необходимости. Использовать индекс или нет, SQLite решает самостоятельно. Таким образом, SQL-запрос будет выглядеть обычным образом:

```
sqlite> SELECT * FROM site WHERE id_rubr=1;
1|1|1|http://wwwadmin.ru|Название|Описание|0
2|1|1|http://python.org|Python||1000
```

В некоторых случаях необходимо пересоздать индексы. Для этого применяется SQL-команда REINDEX. Формат команды:

```
REINDEX [<Название базы данных>.]<Название таблицы или индекса>
```

Если указано название таблицы, то пересоздаются все существующие индексы в таблице. При задании названия индекса пересоздается только указанный индекс.

Удалить обычный и уникальный индексы позволяет SQL-команда DROP INDEX. Формат команды:

```
DROP INDEX [IF EXISTS] [<Название базы данных>.]<Название индекса>
```

Удаление индекса приводит к дефрагментации файла с базой данных. Чтобы освободить неиспользуемое свободное пространство, можно воспользоваться SQL-командой VACUUM.

Вся статистическая информация об индексах хранится в специальной таблице sqlite_stat1. В данный момент в таблице нет никакой информации. Чтобы собрать статистическую информацию и поместить ее в эту таблицу, предназначена SQL-команда ANALYZE. Формат команды:

```
ANALYZE [[<Название базы данных>.]<Название таблицы>];
```

Выполним SQL-команду ANALYZE и выведем содержимое таблицы sqlite_stat1:

```
sqlite> SELECT * FROM sqlite_stat1; -- Нет записей
Error: no such table: sqlite_stat1
sqlite> ANALYZE;
sqlite> SELECT * FROM sqlite_stat1;
site|index_rubr|3 2
rubr||3
users||1
```

16.10. Вложенные запросы

Результаты выполнения инструкции SELECT можно использовать в других инструкциях, создавая вложенные запросы. Для создания таблицы с помощью вложенного запроса используется следующий формат:

```
CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS]
[<Название базы данных>.]<Название таблицы> AS <Запрос SELECT>;
```

В качестве примера создадим временную копию таблицы rubr и выведем ее содержимое:

```
sqlite> CREATE TEMP TABLE tmp_rubr AS SELECT * FROM rubr;
sqlite> SELECT * FROM tmp_rubr;
```

```
1|Программирование
2|Музыка
3|Поисковые порталы
```

В результате выполнения вложенного запроса создается таблица с полями, перечисленными после ключевого слова SELECT, и сразу заполняется данными.

Использовать вложенные запросы можно и в инструкции INSERT. Для этого предназначен следующий формат:

```
INSERT [OR <Алгоритм>] INTO [<Название базы данных>.]<Название таблицы>
[(<Поле1>, <Поле2>, ...)] <Запрос SELECT>;
```

Очистим временную таблицу tmp_rubr, а затем опять заполним ее с помощью вложенного запроса:

```
sqlite> DELETE FROM tmp_rubr;
sqlite> INSERT INTO tmp_rubr SELECT * FROM rubr WHERE id_rubr<3;
sqlite> SELECT * FROM tmp_rubr;
1|Программирование
2|Музыка
```

Если производится попытка вставить повторяющееся значение, и не указан <Алгоритм>, то это приведет к ошибке. С помощью алгоритмов ROLLBACK, ABORT, FAIL, IGNORE или REPLACE можно указать, как следует обрабатывать записи с дублированными значениями. При использовании алгоритма IGNORE повторяющиеся записи отбрасываются, а при использовании REPLACE — новые записи заменяют существующие.

Использовать вложенные запросы можно также в инструкции WHERE. В этом случае вложенный запрос размещается в операторе IN. Для примера выведем сайты, зарегистрированные в рубрике с названием "Программирование":

```
sqlite> SELECT * FROM site WHERE id_rubr IN (
   ...> SELECT id_rubr FROM rubr
   ...> WHERE name_rubr='Программирование');
1|1|1|http://wwwadmin.ru|Название|Описание|0
2|1|1|http://python.org|Python||1000
```

16.11. Транзакции

Очень часто несколько инструкций выполняются последовательно. Например, при совершении покупки деньги списываются со счета клиента и сразу добавляются на счет магазина. Если во время добавления денег на счет магазина произойдет ошибка, то деньги будут списаны со счета клиента, но не попадут на счет магазина. Чтобы гарантировать успешное выполнение группы инструкций, предназначены транзакции. После запуска транзакции группа инструкций выполняется как единое целое. Если во время транзакции произойдет ошибка, например отключится компьютер, все операции с начала транзакции будут отменены.

В SQLite каждая инструкция, производящая изменения в базе данных, автоматически запускает транзакцию, если транзакция не была запущена ранее. После завершения выполнения инструкции транзакция автоматически завершается. Для явного запуска транзакции предназначена инструкция BEGIN. Формат инструкции:

```
BEGIN [DEFERRED | IMMEDIATE | EXCLUSIVE] [TRANSACTION];
```

Для нормального завершения транзакции предназначены инструкции COMMIT и END. Эти инструкции сохраняют все изменения и завершают транзакцию. Инструкции имеют следующий формат:

```
COMMIT [TRANSACTION];
END [TRANSACTION];
```

Чтобы отменить изменения, выполненные с начала транзакции, используется инструкция ROLLBACK. Формат инструкции:

```
ROLLBACK [TRANSACTION] [TO [SAVEPOINT] <Название метки>];
```

В качестве примера запустим транзакцию, вставим две записи, а затем отменим все произведенные изменения и выведем содержимое таблицы:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO rubr VALUES (NULL, 'Кино');
sqlite> INSERT INTO rubr VALUES (NULL, 'Разное');
sqlite> ROLLBACK TRANSACTION; -- Отменяем вставку
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
```

Как видно из результата, новые записи не были вставлены в таблицу. Аналогичные действия будут выполнены автоматически, если соединение с базой данных будет закрыто или отключится компьютер.

Если ошибка возникает в одной из инструкций внутри транзакции, то используется алгоритм обработки ошибок, указанный в конструкции ON CONFLICT <Алгоритм> при создании таблицы или в конструкции OR <Алгоритм> при вставке или обновлении записей. По умолчанию используется алгоритм ABORT. Согласно этому алгоритму при возникновении ошибки аннулируются все изменения, произведенные текущей командой, и выводится сообщение об ошибке. Все изменения, сделанные предыдущими командами в транзакции, сохраняются. Запустим транзакцию и попробуем вставить две записи. При вставке второй записи укажем индекс, который уже существует в таблице:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO rubr VALUES (NULL, 'Кино');
sqlite> INSERT INTO rubr VALUES (3, 'Разное'); -- Ошибка
Error: PRIMARY KEY must be unique
sqlite> COMMIT TRANSACTION;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

Как видно из примера, первая запись успешно добавлена в таблицу. Если необходимо отменить все изменения внутри транзакции, то при вставке следует указать алгоритм ROLLBACK. Согласно этому алгоритму при ошибке транзакция завершается с откатом всех измененных ранее записей, дальнейшее выполнение прерывается и выводится сообщение об ошибке. Рассмотрим это на примере:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT OR ROLLBACK INTO rubr VALUES (NULL, 'Мода');
```

```
sqlite> INSERT OR ROLLBACK INTO rubr VALUES (3, 'Разное');
Error: PRIMARY KEY must be unique
sqlite> COMMIT TRANSACTION; -- Транзакция уже завершена!
Error: cannot commit — no transaction is active
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

Вместо запуска транзакции с помощью инструкции `BEGIN` можно создать именованную метку. Метка создается с помощью инструкции `SAVEPOINT`. Формат инструкции:

```
SAVEPOINT <Название метки>;
```

Для нормального завершения транзакции и сохранения всех изменений предназначена инструкция `RELEASE`. Формат инструкции:

```
RELEASE [SAVEPOINT] <Название метки>;
```

Чтобы отменить изменения, выполненные после метки, используется инструкция `ROLLBACK`. В качестве примера запустим транзакцию, вставим две записи, а затем отменим все произведенные изменения и выведем содержимое таблицы:

```
sqlite> SAVEPOINT metka1;
sqlite> INSERT INTO rubr VALUES (NULL, 'Мода');
sqlite> INSERT INTO rubr VALUES (NULL, 'Разное');
sqlite> ROLLBACK TO SAVEPOINT metka1;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

16.12. Удаление таблицы и базы данных

Удалить таблицу позволяет инструкция `DROP TABLE`. Удалить можно как обычную таблицу, так и временную. Все индексы и триггеры, связанные с таблицей, также удаляются. Формат инструкции:

```
DROP TABLE [IF EXISTS] [<Название базы данных>.]<Название таблицы>;
```

Так как SQLite напрямую работает с файлом, не существует инструкции для удаления базы данных. Чтобы удалить базу, достаточно просто удалить файл.

В этой главе мы рассмотрели лишь основные возможности SQLite. Остались не рассмотренными триггеры, представления, виртуальные таблицы, внешние ключи, операторы, встроенные функции и некоторые другие возможности. За подробной информацией обращайтесь к документации по SQLite.

ГЛАВА 17

Доступ к базе данных SQLite из Python

Итак, изучение основ SQLite закончено, и мы возвращаемся к изучению языка Python. В этой главе мы рассмотрим возможности модуля `sqlite3`, позволяющего работать с базой данных SQLite. Модуль `sqlite3` входит в состав стандартной библиотеки Python, начиная с версии 2.5, и в дополнительной установке не нуждается.

Для работы с базами данных в языке Python существует единый интерфейс доступа. Все разработчики модулей, осуществляющих связь базы данных с Python, должны придерживаться спецификации DB-API (DataBase Application Program Interface). Эта спецификация более интересна для разработчиков модулей, чем для прикладных программистов, поэтому мы не будем ее подробно рассматривать. Получить полное описание спецификации DB-API 2.0 можно в документе PEP 249, расположенном по адресу **http://www.python.org/dev/peps/pep-0249**.

Модуль `sqlite3` поддерживает спецификацию DB-API 2.0, а также предоставляет некоторые нестандартные возможности. Поэтому, изучив методы и атрибуты этого модуля, вы получите достаточно полное представление о спецификации DB-API 2.0 и сможете в дальнейшем работать с другой базой данных. Получить номер спецификации, поддерживаемой модулем, можно с помощью атрибута `apilevel`:

```
>>> import sqlite3            # Подключаем модуль
>>> sqlite3.apilevel          # Получаем номер спецификации
'2.0'
```

Получить номер версии используемого модуля `sqlite3` можно с помощью атрибутов `sqlite_version` и `sqlite_version_info`. Атрибут `sqlite_version` возвращает номер версии в виде строки, а атрибут `sqlite_version_info` в виде кортежа из трех чисел. Пример:

```
>>> sqlite3.sqlite_version
'3.7.4'
>>> sqlite3.sqlite_version_info
(3, 7, 4)
```

Согласно спецификации DB-API 2.0 последовательность работы с базой данных выглядит следующим образом:

1. Производится подключение к базе данных с помощью функции `connect()`. Функция возвращает объект соединения, с помощью которого осуществляется дальнейшая работа с базой данных.
2. Создается объект-курсор.

3. Выполняются SQL-запросы и обрабатываются результаты. Перед выполнением первого запроса, который изменяет записи (INSERT, REPLACE, UPDATE и DELETE), автоматически запускается транзакция.
4. Завершается транзакция или отменяются все изменения в рамках транзакции.
5. Закрывается объект-курсор.
6. Закрывается соединение с базой данных.

17.1. Создание и открытие базы данных

Для создания и открытия базы данных используется функция connect(). Функция имеет следующий формат:

```
connect(database[, timeout][, isolation_level][, detect_types]
       [, factory][, check_same_thread][, cached_statements])
```

В параметре database указывается абсолютный или относительный путь к базе данных. Если база данных не существует, то она будет создана и открыта для работы. Если база данных уже существует, то она просто открывается без удаления имеющихся данных. Вместо пути к базе данных можно указать значение :memory:, которое означает, что база данных будет создана в оперативной памяти. После закрытия такой базы все данные будут удалены.

Все остальные параметры являются необязательными и могут быть указаны в произвольном порядке путем присвоения значения названию параметра. Необязательный параметр timeout задает время ожидания снятия блокировки с открываемой базы данных. По умолчанию значение параметра timeout равно пяти секундам. Предназначение остальных параметров мы рассмотрим немного позже.

Функция connect() возвращает объект соединения, с помощью которого осуществляется вся дальнейшая работа с базой данных. Если открыть базу данных не удалось, то возбуждается исключение. Соединение закрывается, когда вызывается метод close() объекта соединения. В качестве примера откроем и сразу закроем базу данных testdb.db, расположенную в текущем рабочем каталоге:

```
>>> import sqlite3                          # Подключаем модуль sqlite3
>>> con = sqlite3.connect("testdb.db")      # Открываем базу данных
>>>                                         # Работаем с базой данных
>>> con.close()                             # Закрываем базу данных
```

17.2. Выполнение запроса

Согласно спецификации DB-API 2.0 после создания объекта соединения необходимо создать объект-курсор. Все дальнейшие запросы должны производиться через этот объект. Создание объекта-курсора производится с помощью метода cursor(). Для выполнения запроса к базе данных предназначены следующие методы объекта-курсора:

- close() — закрывает объект-курсор;
- executescript(<SQL-запросы через точку с запятой>) — выполняет несколько SQL-запросов за один раз. Если в процессе выполнения запросов возникает ошибка, то метод возбуждает исключение. В качестве примера создадим базу данных и три таблицы в ней:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()        # Создаем объект-курсор
sql = """\
CREATE TABLE user (
   id_user INTEGER PRIMARY KEY AUTOINCREMENT,
   email TEXT,
   passw TEXT
);
CREATE TABLE rubr (
   id_rubr INTEGER PRIMARY KEY AUTOINCREMENT,
   name_rubr TEXT
);
CREATE TABLE site (
   id_site INTEGER PRIMARY KEY AUTOINCREMENT,
   id_user INTEGER,
   id_rubr INTEGER,
   url TEXT,
   title TEXT,
   msg TEXT,
   iq INTEGER
);
"""
try:                      # Обрабатываем исключения
    cur.executescript(sql) # Выполняем SQL-запросы
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
input()
```

Сохраняем код в файл, а затем запускаем его с помощью двойного щелчка на значке файла. Обратите внимание на то, что мы работаем с кодировкой UTF-8. Эта кодировка по умолчанию используется в SQLite;

- `execute(<SQL-запрос>[, <Значения>])` — выполняет один SQL-запрос. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Добавим пользователя в таблицу `user`:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()        # Создаем объект-курсор
sql = """\
INSERT INTO user (email, passw)
VALUES ('unicross@mail.ru', 'password1')
"""
```

```
try:
    cur.execute(sql)      # Выполняем SQL-запрос
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()          # Завершаем транзакцию
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
input()
```

В этом примере мы использовали метод `commit()` объекта соединения. Метод `commit()` позволяет завершить транзакцию, которая запускается автоматически. Если метод не вызвать и при этом закрыть соединение с базой данных, то все произведенные изменения будут автоматически отменены. Более подробно управление транзакциями мы рассмотрим далее в этой главе, а сейчас следует запомнить, что запросы, изменяющие записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), нужно завершать вызовом метода `commit()`.

В некоторых случаях в SQL-запрос необходимо подставлять данные, полученные от пользователя. Если данные не обработать и подставить в SQL-запрос, то пользователь получает возможность видоизменить запрос и, например, зайти в закрытый раздел без ввода пароля. Чтобы значения были правильно подставлены, нужно их передавать в виде кортежа или словаря во втором параметре метода `execute()`. В этом случае в SQL-запросе указываются следующие специальные заполнители:

- `?` — при указании значения в виде кортежа;
- `:<Ключ>` — при указании значения в виде словаря.

В качестве примера заполним таблицу с рубриками этими способами:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()               # Создаем объект-курсор
t1 = ("Программирование",)
t2 = (2, "Музыка")
d  = {"id": 3, "name": """Поисковые ' " порталы"""}
sql_t1 = "INSERT INTO rubr (name_rubr) VALUES (?)"
sql_t2 = "INSERT INTO rubr VALUES (?, ?)"
sql_d  = "INSERT INTO rubr VALUES (:id, :name)"
try:
    cur.execute(sql_t1, t1)      # Кортеж из 1-го элемента
    cur.execute(sql_t2, t2)      # Кортеж из 2-х элементов
    cur.execute(sql_d,  d)       # Словарь
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()                 # Завершаем транзакцию
cur.close()                      # Закрываем объект-курсор
con.close()                      # Закрываем соединение
input()
```

Обратите внимание на значение переменной `t1`. Перед закрывающей круглой скобкой запятая указана не по ошибке. Если запятую убрать, то вместо кортежа мы получим строку. Не скобки создают кортеж, а запятые. Поэтому при создании кортежа из одного элемента в конце необходимо добавить запятую. Как показывает практика, новички постоянно забывают указать запятую и при этом получают сообщение об ошибке.

В значении ключа `name` переменной `d` апостроф и двойная кавычка также указаны не случайно. Это значение показывает, что при подстановке все специальные символы экранируются, поэтому никакой ошибки при вставке значения в таблицу не будет.

> ***Внимание!***
>
> Никогда напрямую не передавайте в SQL-запрос данные, полученные от пользователя. Это потенциальная угроза безопасности. Данные следует передавать через второй параметр методов `execute()` и `executemany()`.

- `executemany(<SQL-запрос>, <Последовательность>)` — выполняет SQL-запрос несколько раз, при этом подставляя значения из последовательности. Каждый элемент последовательности должен быть кортежем (при использовании заполнителя `"?"`) или словарем (при использовании заполнителя `":<Ключ>"`). Вместо последовательности можно указать объект-итератор или объект-генератор. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Заполним таблицу `site` с помощью метода `executemany()`:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()              # Создаем объект-курсор
arr = [
         (1, 1, "http://wwwadmin.ru", "Название", "", 100),
         (1, 1, "http://python.org", "Python", "", 1000),
         (1, 3, "http://google.ru", "Гугль", "", 3000)
      ]
sql = """\
INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
VALUES (?, ?, ?, ?, ?, ?)
"""
try:
    cur.executemany(sql, arr)
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()                # Завершаем транзакцию
cur.close()                     # Закрываем объект-курсор
con.close()                     # Закрываем соединение
input()
```

Модуль `sqlite3` содержит также методы `execute()`, `executemany()` и `executescript()` объекта соединения, которые позволяют выполнить запрос без создания объекта-курсора. Эти методы не входят в спецификацию DB-API 2.0. В качестве примера изменим название рубрики с идентификатором `3` (листинг 17.1).

Листинг 17.1. Использование метода execute()

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
try:
    con.execute("""UPDATE rubr SET name_rubr='Поисковые порталы'
                   WHERE id_rubr=3""")
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    con.commit()                  # Завершаем транзакцию
    print("Запрос успешно выполнен")
con.close()                       # Закрываем соединение
input()
```

Объект-курсор поддерживает несколько атрибутов:

- `lastrowid` — индекс последней добавленной записи с помощью инструкции `INSERT` и метода `execute()`. Если индекс не определен, то атрибут будет содержать значение `None`. В качестве примера добавим новую рубрику и выведем ее индекс:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()                # Создаем объект-курсор
try:
    cur.execute("""INSERT INTO rubr (name_rubr)
                   VALUES ('Кино')""")
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    con.commit()                  # Завершаем транзакцию
    print("Запрос успешно выполнен")
    print("Индекс:", cur.lastrowid)
cur.close()                       # Закрываем объект-курсор
con.close()                       # Закрываем соединение
input()
```

- `rowcount` — количество измененных или удаленных записей. Если количество не определено, то атрибут имеет значение `-1`;
- `description` — содержит кортеж кортежей с именами полей в результате выполнения инструкции `SELECT`. Каждый внутренний кортеж состоит из семи элементов. Первый элемент содержит название поля, а остальные элементы всегда имеют значение `None`. Например, если выполнить SQL-запрос `SELECT * FROM rubr`, то атрибут будет содержать следующее значение:

```
(('id_rubr', None, None, None, None, None, None),
 ('name_rubr', None, None, None, None, None, None))
```

17.3. Обработка результата запроса

Для обработки результата запроса применяются следующие методы объекта-курсора:

- `fetchone()` — при каждом вызове возвращает одну запись из результата запроса в виде кортежа, а затем перемещает указатель текущей позиции. Если записей больше нет, метод возвращает значение `None`. Выведем все записи из таблицы `user`:

```
>>> import sqlite3
>>> con = sqlite3.connect("catalog.db")
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.fetchone()
(1, 'unicross@mail.ru', 'password1')
>>> print(cur.fetchone())
None
```

- `__next__()` — при каждом вызове возвращает одну запись из результата запроса в виде кортежа, а затем перемещает указатель текущей позиции. Если записей больше нет, метод возбуждает исключение `StopIteration`. Выведем все записи из таблицы `user` с помощью метода `next()`:

```
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.__next__()
(1, 'unicross@mail.ru', 'password1')
>>> cur.__next__()
... Фрагмент опущен ...
StopIteration
```

  Цикл `for` на каждой итерации вызывает метод `__next__()` автоматически. Поэтому для перебора записей достаточно указать объект-курсор в качестве параметра цикла. Выведем все записи из таблицы `rubr`:

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E2F0>
>>> for id_rubr, name in cur: print("{0}|{1}".format(id_rubr, name))


1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

- `fetchmany([size=cursor.arraysize])` — при каждом вызове возвращает список записей из результата запроса, а затем перемещает указатель текущей позиции. Каждый элемент списка является кортежем. Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра или значения атрибута `arraysize` объекта-курсора. Если количество записей в результате запроса меньше указанного количества элементов списка, то количество элементов списка будет соответствовать оставшемуся количеству записей. Если записей больше нет, метод возвращает пустой список. Пример:

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E3B0>
```

```
>>> cur.arraysize
1
>>> cur.fetchmany()
[(1, 'Программирование')]
>>> cur.fetchmany(2)
[(2, 'Музыка'), (3, 'Поисковые порталы')]
>>> cur.fetchmany(3)
[(4, 'Кино')]
>>> cur.fetchmany()
[]
```

- `fetchall()` — возвращает список всех (или всех оставшихся) записей из результата запроса. Каждый элемент списка является кортежем. Если записей больше нет, метод возвращает пустой список. Пример:

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.fetchall()
[(1, 'Программирование'), (2, 'Музыка'), (3, 'Поисковые порталы'),
 (4, 'Кино')]
>>> cur.fetchall()
[]
>>> con.close()
```

Все рассмотренные методы возвращают запись в виде кортежа. Если необходимо изменить такое поведение и, например, получить записи в виде словаря, то следует воспользоваться атрибутом `row_factory` объекта соединения. В качестве значения атрибут принимает ссылку на функцию обратного вызова, имеющую следующий формат:

```
def <Название функции>(<Объект-курсор>, <Запись>):
    # Обработка записи
    return <Новый объект>
```

В качестве примера выведем записи из таблицы `user` в виде словаря (листинг 17.2).

Листинг 17.2. Атрибут `row_factory`

```
# -*- coding: utf-8 -*-
import sqlite3
def my_factory(c, r):
    d = {}
    for i, name in enumerate(c.description):
        d[name[0]] = r[i] # Ключи в виде названий полей
        d[i] = r[i]       # Ключи в виде индексов полей
    return d

con = sqlite3.connect("catalog.db")
con.row_factory = my_factory
cur = con.cursor()        # Создаем объект-курсор
cur.execute("SELECT * FROM user")
arr = cur.fetchall()
```

```
print(arr)                  # Результат:
"""[{0: 1, 1: 'unicross@mail.ru', 2: 'password1', 'id_user': 1,
'passw': 'password1', 'email': 'unicross@mail.ru'}]"""
print(arr[0][1])          # Доступ по индексу
print(arr[0]["email"])    # Доступ но названию поля
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
input()
```

Функция `my_factory()` будет вызываться для каждой записи. Обратите внимание на то, что название функции в операции присваивания атрибуту `row_factory` указывается без круглых скобок. Если скобки указать, то смысл операции будет совсем иным.

Атрибуту `row_factory` можно присвоить ссылку на объект `Row` из модуля `sqlite3`. Этот объект позволяет получить доступ к значению поля как по индексу, так и по названию поля. Причем название не зависит от регистра символов. Объект `Row` поддерживает итерации, доступ по индексу и метод `keys()`, который возвращает список с названиями полей. Переделаем наш предыдущий пример и используем объект `Row` (листинг 17.3).

Листинг 17.3. Объект `Row`

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
con.row_factory = sqlite3.Row
cur = con.cursor()
cur.execute("SELECT * FROM user")
arr = cur.fetchall()
print(type(arr[0]))       # <class 'sqlite3.Row'>
print(len(arr[0]))        # 3
print(arr[0][1])          # Доступ по индексу
print(arr[0]["email"])    # Доступ по названию поля
print(arr[0]["EMAIL"])    # Не зависит от регистра символов
for elem in arr[0]:
    print(elem)
print(arr[0].keys())      # ['id_user', 'email', 'passw']
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
input()
```

Как видно из результатов предыдущих примеров, все данные, имеющие в SQLite тип `TEXT`, возвращаются в виде строк. В предыдущей главе мы создали базу данных testdb.db и сохраняли данные в полях таблицы в кодировке Windows-1251. Попробуем отобразить записи из таблицы с рубриками:

```
>>> con = sqlite3.connect("testdb.db")
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM rubr")
... Фрагмент опущен ...
sqlite3.OperationalError: Could not decode to UTF-8 column 'name_rubr'
>>> con.close()
```

При осуществлении преобразования предполагается, что строка хранится в кодировке UTF-8. Так как в нашем примере мы используем другую кодировку, то при преобразовании возникает ошибка и возбуждается исключение `OperationalError`. Обойти это исключение позволяет атрибут `text_factory` объекта соединения. В качестве значения атрибута указывается ссылка на функцию, которая будет использоваться для осуществления преобразования значения текстовых полей. Например, чтобы вернуть последовательность байтов, следует указать ссылку на функцию `bytes()` (листинг 17.4).

Листинг 17.4. Атрибут `text_factory`

```
>>> con = sqlite3.connect("testdb.db")
>>> con.text_factory = bytes # Название функции без круглых скобок!
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x014FE380>
>>> cur.fetchone()
(1, b'\xcf\xf0\xee\xe3\xf0\xe0\xec\xec\xe8\xf0\xee\xe2\xe0\xed\xe8\xe5')
```

Если необходимо вернуть строку, то внутри функции обратного вызова следует вызвать функцию `str()` и явно указать кодировку данных. Функция обратного вызова должна принимать один параметр и возвращать преобразованную строку. Выведем текстовые данные в виде строки (листинг 17.5).

Листинг 17.5. Указание пользовательской функции преобразования

```
>>> con.text_factory = lambda s: str(s, "cp1251")
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x014FE380>
>>> cur.fetchone()
(1, 'Программирование')
>>> con.close()
```

17.4. Управление транзакциями

Перед выполнением первого запроса автоматически запускается транзакция. Поэтому все запросы, изменяющие записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), необходимо завершать вызовом метода `commit()` объекта соединения. Если метод не вызвать и при этом закрыть соединение с базой данных, то все произведенные изменения будут отменены. Транзакция может автоматически завершаться при выполнении запросов `CREATE TABLE`, `VACUUM` и некоторых других. После выполнения этих запросов транзакция запускается снова.

Если необходимо отменить изменения, то следует вызвать метод `rollback()` объекта соединения. В качестве примера добавим нового пользователя, а затем отменим транзакцию и выведем содержимое таблицы (листинг 17.6).

Листинг 17.6. Отмена изменений с помощью метода `rollback()`

```
>>> con = sqlite3.connect("catalog.db")
>>> cur = con.cursor()
>>> cur.execute("INSERT INTO user VALUES (NULL, 'user@mail.ru', '')")
```

```
<sqlite3.Cursor object at 0x01508CB0>
>>> con.rollback()                          # Отмена изменений
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x01508CB0>
>>> cur.fetchall()
[(1, 'unicross@mail.ru', 'password1')]
>>> con.close()
```

Управлять транзакцией можно с помощью параметра `isolation_level` в функции `connect()`, а также с помощью атрибута `isolation_level` объекта соединения. Допустимые значения: `"DEFERRED"`, `"IMMEDIATE"`, `"EXCLUSIVE"`, пустая строка и `None`. Первые три значения передаются в инструкцию `BEGIN`. Если в качестве значения указать `None`, то транзакция запускаться не будет. В этом случае нет необходимости вызывать метод `commit()`. Все изменения будут сразу сохраняться в базе данных. Отключим автоматический запуск транзакции с помощью параметра `isolation_level`, добавим нового пользователя, а затем подключимся заново и выведем все записи из таблицы (листинг 17.7).

Листинг 17.7. Управление транзакциями

```
>>> con = sqlite3.connect("catalog.db", isolation_level=None)
>>> cur = con.cursor()
>>> cur.execute("INSERT INTO user VALUES (NULL, 'user@mail.ru', '')")
<sqlite3.Cursor object at 0x01508CE0>
>>> con.close()
>>> con = sqlite3.connect("catalog.db")
>>> con.isolation_level = None # Отключение запуска транзакции
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x01508530>
>>> cur.fetchall()
[(1, 'unicross@mail.ru', 'password1'), (2, 'user@mail.ru', '')]
>>> con.close()
```

17.5. Создание пользовательской сортировки

По умолчанию сортировка с помощью инструкции `ORDER BY` зависит от регистра символов. Например, если сортировать слова "единица1", "Единица2" и "Единый", то в результате мы получим неправильную сортировку ("Единица2", "Единый" и лишь затем "единица1"). Модуль `sqlite3` позволяет создать пользовательскую функцию сортировки и связать ее с названием функции в SQL-запросе. В дальнейшем это название можно указать в инструкции `ORDER BY` после ключевого слова `COLLATE`.

Связать название функции в SQL-запросе с пользовательской функцией в программе позволяет метод `create_collation()` объекта соединения. Формат метода:

```
create_collation(<Название функции в SQL-запросе в виде строки>,
                 <Ссылка на функцию сортировки>)
```

Функция сортировки принимает две строки и должна возвращать:

- `1` — если первая строка больше второй;
- `-1` — если вторая больше первой;
- `0` — если строки равны.

Обратите внимание на то, что функция сортировки будет вызываться только при сравнении текстовых значений. При сравнении чисел функция вызвана не будет.

В качестве примера создадим новую таблицу с одним полем, вставим три записи, а затем произведем сортировку стандартным методом и с помощью пользовательской функции (листинг 17.8).

Листинг 17.8. Сортировка записей

```
# -*- coding: utf-8 -*-
import sqlite3

def myfunc(s1, s2): # Пользовательская функция сортировки
    s1 = s1.lower()
    s2 = s2.lower()
    if s1 == s2:
        return 0
    elif s1 > s2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:", isolation_level=None)
# Связываем имя "myfunc" с функцией myfunc()
con.create_collation("myfunc", myfunc)
cur = con.cursor()
cur.execute("CREATE TABLE words (word TEXT)")
cur.execute("INSERT INTO words VALUES('единица1')")
cur.execute("INSERT INTO words VALUES('Единый')")
cur.execute("INSERT INTO words VALUES('Единица2')")
# Стандартная сортировка
cur.execute("SELECT * FROM words ORDER BY word")
for line in cur:
    print(line[0], end=" ") # Результат: Единица2 Единый единица1
print()
# Пользовательская сортировка
cur.execute("""SELECT * FROM words
               ORDER BY word COLLATE myfunc""")
for line in cur:
    print(line[0], end=" ") # Результат: единица1 Единица2 Единый
cur.close()
con.close()
input()
```

17.6. Поиск без учета регистра символов

Как уже говорилось в предыдущей главе, сравнение строк и поиск с помощью оператора `LIKE` для русских букв производится с учетом регистра символов. Поэтому следующие выражения вернут значение `0`:

```
cur.execute("SELECT 'строка' = 'Строка'")
print(cur.fetchone()[0])                # Результат: 0 (не равно)
cur.execute("SELECT 'строка' LIKE 'Строка'")
print(cur.fetchone()[0])                # Результат: 0 (не найдено)
```

Одним из вариантов решения проблемы является преобразование символов обеих строк к верхнему или нижнему регистру. Но встроенные функции SQLite `UPPER()` и `LOWER()` с русскими буквами опять работают некорректно. Модуль `sqlite3` позволяет создать пользовательскую функцию и связать ее с названием функции в SQL-запросе. Таким образом, можно создать пользовательскую функцию преобразования регистра символов, а затем указать связанное с ней имя в SQL-запросе.

Связать название функции в SQL-запросе с пользовательской функцией в программе позволяет метод `create_function()` объекта соединения. Формат метода:

```
create_function(<Название функции в SQL-запросе в виде строки>,
                <Количество параметров>, <Ссылка на функцию>)
```

В первом параметре указывается название функции в виде строки. Количество параметров, принимаемых функцией, задается во втором параметре. Параметры могут быть любого типа. Если функция принимает строку, то ее типом данных будет `str`. В третьем параметре указывается ссылка на пользовательскую функцию в программе. Для примера произведем поиск рубрики без учета регистра символов (листинг 17.9).

Листинг 17.9. Поиск без учета регистра символов

```
# -*- coding: utf-8 -*-
import sqlite3

# Пользовательская функция изменения регистра
def myfunc(s):
    return s.lower()

con = sqlite3.connect("catalog.db")
# Связываем имя "mylower" с функцией myfunc()
con.create_function("mylower", 1, myfunc)
cur = con.cursor()
string = "%МуЗЫка%"                          # Строка для поиска
# Поиск без учета регистра символов
sql = """SELECT * FROM rubr
         WHERE mylower(name_rubr) LIKE ?"""
cur.execute(sql, (string.lower(),))
print(cur.fetchone()[1])                     # Результат: Музыка
cur.close()
con.close()
input()
```

В этом примере предполагается, что значение переменной `string` получено от пользователя. Обратите внимание на то, что строку для поиска в метод `execute()` мы передаем в нижнем регистре. Если этого не сделать и указать преобразование в SQL-запросе, то будет производиться лишнее преобразование регистра при каждом сравнении.

Метод `create_function()` используется не только для создания функции изменения регистра символов, но и для других целей. Например, в SQLite нет специального типа данных для хранения даты и времени. Дату и время можно хранить разными способами, например, как количество секунд, прошедших с начала эпохи, в числовом поле. Для преобразования количества секунд в другой формат следует создать пользовательскую функцию форматирования (листинг 17.10).

Листинг 17.10. Преобразование даты и времени

```
# -*- coding: utf-8 -*-
import sqlite3
import time

def myfunc(d):
    return time.strftime("%d.%m.%Y", time.localtime(d))

con = sqlite3.connect(":memory:")
# Связываем имя "mytime" с функцией myfunc()
con.create_function("mytime", 1, myfunc)
cur = con.cursor()
cur.execute("SELECT mytime(1303520699)")
print(cur.fetchone()[0]) # Результат: 23.04.2011
cur.close()
con.close()
input()
```

17.7. Создание агрегатных функций

При изучении SQLite мы рассматривали встроенные агрегатные функции `COUNT()`, `MIN()`, `MAX()`, `AVG()` и `SUM()`. Если возможностей этих функций окажется недостаточно, то можно определить пользовательскую агрегатную функцию. Связать название функции в SQL-запросе с пользовательским классом в программе позволяет метод `create_aggregate()` объекта соединения. Формат метода:

```
create_aggregate(<Название функции в SQL-запросе в виде строки>,
                 <Количество параметров>, <Ссылка на класс>)
```

В первом параметре указывается название агрегатной функции в виде строки. В третьем параметре передается ссылка на класс (название класса без круглых скобок). Этот класс должен иметь два метода: `step()` и `finalize()`. Метод `step()` вызывается несколько раз и ему передаются параметры. Количество параметров задается во втором параметре метода `create_aggregate()`. Метод `finalize()` должен возвращать результат выполнения. В качестве примера выведем все названия рубрик в алфавитном порядке через разделитель (листинг 17.11).

Листинг 17.11. Создание агрегатной функции

```
# -*- coding: utf-8 -*-
import sqlite3

class MyClass:
    def __init__(self):
        self.result = []
    def step(self, value):
        self.result.append(value)
    def finalize(self):
        self.result.sort()
        return " — ".join(self.result)

con = sqlite3.connect("catalog.db")
# Связываем имя "myfunc" с классом MyClass
con.create_aggregate("myfunc", 1, MyClass)
cur = con.cursor()
cur.execute("SELECT myfunc(name_rubr) FROM rubr")
print(cur.fetchone()[0])
# Результат: Кино — Музыка — Поисковые порталы — Программирование
cur.close()
con.close()
input()
```

17.8. Преобразование типов данных

SQLite поддерживает пять типов данных. Для каждого типа SQLite в модуле `sqlite3` определено соответствие с типом данных в языке Python:

- `NULL` — значение `NULL`. Значение соответствует типу `None` в Python;
- `INTEGER` — целые числа. Соответствует типу `int`;
- `REAL` — вещественные числа. Соответствует типу `float`;
- `TEXT` — строки. По умолчанию преобразуется в тип `str`. Предполагается, что строка в базе данных хранится в кодировке UTF-8. Соответствие можно изменить с помощью атрибута `text_factory`;
- `BLOB` — бинарные данные. Соответствует типу `bytes`.

Если необходимо сохранить в таблице данные, которые имеют тип, не поддерживаемый SQLite, то следует преобразовать тип самостоятельно. Для этого с помощью функции `register_adapter()` можно зарегистрировать пользовательскую функцию, которая будет вызываться при попытке вставки объекта в SQL-запрос. Функция имеет следующий формат:

```
register_adapter(<Тип данных или класс>, <Ссылка на функцию>)
```

В первом параметре указывается тип данных или ссылка на класс. Во втором параметре задается ссылка на функцию, которая будет вызываться для преобразования типа. Функция принимает один параметр и должна возвращать значение, имеющее тип данных, поддержи-

ваемый SQLite. В качестве примера создадим новую таблицу и сохраним в ней значения атрибутов класса (листинг 17.12).

Листинг 17.12. Сохранение в базе атрибутов класса

```
# -*- coding: utf-8 -*-
import sqlite3

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color

def my_adapter(car):
    return "{0}|{1}".format(car.model, car.color)

# Регистрируем функцию для преобразования типа
sqlite3.register_adapter(Car, my_adapter)
# Создаем экземпляр класса Car
car = Car("ВАЗ-2109", "красный")
con = sqlite3.connect("catalog.db")
cur = con.cursor()
try:
    cur.execute("CREATE TABLE cars1 (model TEXT)")
    cur.execute("INSERT INTO cars1 VALUES (?)", (car,))
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()
cur.close()
con.close()
input()
```

Вместо регистрации функции преобразования типа можно внутри класса определить метод `__conform__()`. Формат метода:

```
__conform__(self, <Протокол>)
```

Параметр `<Протокол>` будет соответствовать `PrepareProtocol`. Более подробно о протоколе можно прочитать в документе PEP 246. Метод должен возвращать значение, имеющее тип данных, поддерживаемый SQLite. Создадим таблицу `cars2` и сохраним в ней значения атрибутов, используя метод `__conform__()` (листинг 17.13).

Листинг 17.13. Использование метода `__conform__()`

```
# -*- coding: utf-8 -*-
import sqlite3

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color
```

```
    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "{0}|{1}".format(car.model, car.color)

# Создаем экземпляр класса Car
car = Car("Москвич-412", "синий")
con = sqlite3.connect("catalog.db")
cur = con.cursor()
try:
    cur.execute("CREATE TABLE cars2 (model mycar)")
    cur.execute("INSERT INTO cars2 VALUES (?)", (car,))
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()
cur.close()
con.close()
input()
```

Чтобы восстановить объект, следует зарегистрировать функцию преобразования типа данных SQLite в тип данных Python с помощью функции `register_converter()`. Функция имеет следующий формат:

```
register_converter(<Тип данных>, <Ссылка на функцию>)
```

В первом параметре указывается преобразуемый тип данных в виде строки, а во втором параметре задается ссылка на функцию, которая будет использоваться для преобразования типа данных. Функция должна принимать один параметр и возвращать преобразованное значение.

Чтобы интерпретатор смог определить, какую функцию необходимо вызвать для преобразования типа данных, следует явно указать местоположение метки с помощью параметра `detect_types` функции `connect()`. Параметр может принимать следующие значения (или их комбинацию через оператор `|`):

- `sqlite3.PARSE_COLNAMES` — тип данных указывается в SQL-запросе в псевдониме поля внутри квадратных скобок. Пример указания типа `mycar` для поля `model`:

  ```
  SELECT model as "c [mycar]" FROM cars1
  ```

- `sqlite3.PARSE_DECLTYPES` — тип данных определяется по значению, указанному после названия поля в инструкции `CREATE TABLE`. Пример указания типа `mycar` для поля `model`:

  ```
  CREATE TABLE cars2 (model mycar)
  ```

Выведем сохраненное значение из таблицы `cars1` (листинг 17.14).

Листинг 17.14. Использование значения `sqlite3.PARSE_COLNAMES`

```
# -*- coding: utf-8 -*-
import sqlite3, sys

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color
```

```
    def __repr__(self):
        s = "Модель: {0}, цвет: {1}".format(self.model, self.color)
        return s

def my_converter(value):
    value = str(value, "utf-8")
    model, color = value.split("|")
    return Car(model, color)

# Регистрируем функцию для преобразования типа
sqlite3.register_converter("mycar", my_converter)
con = sqlite3.connect("catalog.db",
                      detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("""SELECT model as "c [mycar]" FROM cars1""")
print(cur.fetchone()[0])
# Результат: Модель: ВАЗ-2109, цвет: красный
con.close()
input()
```

Теперь выведем значение из таблицы `cars2` (листинг 17.15).

Листинг 17.15. Использование значения `sqlite3.PARSE_DECLTYPES`

```
# -*- coding: utf-8 -*-
import sqlite3, sys

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __repr__(self):
        s = "Модель: {0}, цвет: {1}".format(self.model, self.color)
        return s

def my_converter(value):
    value = str(value, "utf-8")
    model, color = value.split("|")
    return Car(model, color)

# Регистрируем функцию для преобразования типа
sqlite3.register_converter("mycar", my_converter)
con = sqlite3.connect("catalog.db",
                      detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("SELECT model FROM cars2")
print(cur.fetchone()[0])
# Результат: Модель: Москвич-412, цвет: синий
con.close()
input()
```

17.9. Сохранение в таблице даты и времени

В SQLite нет специальных типов данных для представления даты и времени. Поэтому обычно дату преобразовывают в строку или число (количество секунд, прошедших с начала эпохи) и сохраняют в соответствующих полях. При выводе данные необходимо опять преобразовывать. Используя знания, полученные в предыдущем разделе, можно зарегистрировать две функции преобразования (листинг 17.16).

Листинг 17.16. Сохранение в таблице даты и времени

```
# -*- coding: utf-8 -*-
import sqlite3, datetime, time

# Преобразование даты в число
def my_adapter(t):
    return time.mktime(t.timetuple())

# Преобразование в дату
def my_converter(t):
    return datetime.datetime.fromtimestamp(float(t))

# Регистрируем обработчики
sqlite3.register_adapter(datetime.datetime, my_adapter)
sqlite3.register_converter("mytime", my_converter)
# Получаем текущую дату и время
dt = datetime.datetime.today()
con = sqlite3.connect(":memory:", isolation_level=None,
                      detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("CREATE TABLE times (time)")
cur.execute("INSERT INTO times VALUES (?)", (dt,))
cur.execute("""SELECT time as "t [mytime]" FROM times""")
print(cur.fetchone()[0]) # 2011-05-01 14:38:14
con.close()
input()
```

Модуль `sqlite3` для типов `date` и `datetime` из модуля `datetime` содержит встроенные функции для преобразования типов. Для `datetime.date` зарегистрирован тип `date`, а для `datetime.datetime` — тип `timestamp`. Таким образом, создавать пользовательские функции преобразования не нужно. Пример сохранения в таблице даты и времени приведен в листинге 17.17.

Листинг 17.17. Встроенные функции для преобразования типов

```
# -*- coding: utf-8 -*-
import sqlite3, datetime
# Получаем текущую дату и время
d = datetime.date.today()
dt = datetime.datetime.today()
```

```
con = sqlite3.connect(":memory:", isolation_level=None,
                      detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("CREATE TABLE times (d date, dt timestamp)")
cur.execute("INSERT INTO times VALUES (?, ?)", (d, dt))
cur.execute("SELECT d, dt FROM times")
res = cur.fetchone()
print(res[0]) # 2011-05-01
print(res[1]) # 2011-05-01 14:40:28.562000
con.close()
input()
```

17.10. Обработка исключений

Модуль `sqlite3` поддерживает следующую иерархию исключений:

```
Exception
    Warning
    Error
        InterfaceError
        DatabaseError
            DataError
            OperationalError
            IntegrityError
            InternalError
            ProgrammingError
            NotSupportedError
```

Базовым классом самого верхнего уровня является класс `Exception`. Все остальные исключения определены в модуле `sqlite3`. Поэтому при указании исключения в инструкции `except` следует предварительно указать название модуля (например, `sqlite3.DatabaseError`). Исключения возбуждаются в следующих случаях:

- `Warning` — при наличии важных предупреждений;
- `Error` — базовый класс для всех остальных исключений, возбуждаемых в случае ошибки. Если указать этот класс в инструкции `except`, то будут перехватываться все ошибки;
- `InterfaceError` — при ошибках, которые связаны с интерфейсом базы данных, а не с самой базой данных;
- `DatabaseError` — базовый класс для исключений, которые связаны с базой данных;
- `DataError` — при ошибках, возникающих при обработке данных;
- `OperationalError` — вызывается при ошибках, которые связаны с операциями в базе данных, например, при синтаксической ошибке в SQL-запросе, несоответствии количества полей в инструкции `INSERT`, отсутствии поля с указанным именем и т. д. Иногда не зависит от правильности SQL-запроса;
- `IntegrityError` — при наличии проблем с внешними ключами или индексами;
- `InternalError` — при внутренней ошибке в базе данных;

- `ProgrammingError` — возникает при ошибках программирования. Например, количество переменных, указанных во втором параметре метода `execute()`, не совпадает с количеством специальных символов в SQL-запросе;
- `NotSupportedError` — при использовании методов, не поддерживаемых базой данных.

В качестве примера обработки исключений напишем программу, которая позволяет пользователям вводить название базы данных и SQL-команды в консоли (листинг 17.18).

Листинг 17.18. Выполнение SQL-команд, введенных в консоли

```
# -*- coding: utf-8 -*-
import sqlite3, sys, re

def db_connect(db_name):
    try:
        db = sqlite3.connect(db_name, isolation_level=None)
    except (sqlite3.Error, sqlite3.Warning) as err:
        print("Не удалось подключиться к БД")
        input()
        sys.exit(0)
    return db

print("Введите название базы данных:", end=" ")
db_name = input()
db_name = db_name.rstrip("\r")  # Для версии 3.2.0 (см. разд. 1.7)
con = db_connect(db_name)       # Подключаемся к базе
cur = con.cursor()
sql = ""
print("Чтобы закончить выполнение программы, введите <Q>+<Enter>")
while True:
    tmp = input()
    tmp = tmp.rstrip("\r")      # Для версии 3.2.0 (см. разд. 1.7)
    if tmp in ["q", "Q"]:
        break
    if tmp.strip() == "":
        continue
    sql = "{0} {1}".format(sql, tmp)
    if sqlite3.complete_statement(sql):
        try:
            sql = sql.strip()
            cur.execute(sql)
            if re.match("SELECT ", sql, re.I):
                print(cur.fetchall())
        except (sqlite3.Error, sqlite3.Warning) as err:
            print("Ошибка:", err)
        else:
            print("Запрос успешно выполнен")
        sql = ""
cur.close()
con.close()
```

Чтобы SQL-запрос можно было разместить на нескольких строках, мы выполняем проверку завершенности запроса с помощью функции `complete_statement(<SQL-запрос>)`. Функция

возвращает `True`, если параметр содержит один или более полных SQL-запросов. Признаком завершенности запроса является точка с запятой. Никакой проверки правильности SQL-запроса не производится. Пример использования функции:

```
>>> sql = "SELECT 10 > 5;"
>>> sqlite3.complete_statement(sql)
True
>>> sql = "SELECT 10 > 5"
>>> sqlite3.complete_statement(sql)
False
>>> sql = "SELECT 10 > 5; SELECT 20 + 2;"
>>> sqlite3.complete_statement(sql)
True
```

Начиная с версии 2.6, язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует выполнение завершающих действий вне зависимости от того, произошло исключение внутри блока кода или нет. В модуле `sqlite3` объект соединения поддерживает этот протокол. Если внутри блока `with` не произошло исключение, то автоматически вызывается метод `commit()`. В противном случае все изменения отменяются с помощью метода `rollback()`. Для примера добавим три рубрики в таблицу `rubr`. В первом случае запрос будет без ошибок, а во втором случае выполним два запроса, последний из которых будет добавлять рубрику с уже существующим идентификатором (листинг 17.19).

Листинг 17.19. Инструкция `with...as`

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect(r"C:\book\catalog.db")
try:
    with con:
        # Добавление новой рубрики
        con.execute("""INSERT INTO rubr VALUES (NULL, 'Мода')""")
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
try:
    with con:
        # Добавление новой рубрики
        con.execute("""INSERT INTO rubr VALUES (NULL, 'Спорт')""")
        # Рубрика с идентификатором 1 уже существует!
        con.execute("""INSERT INTO rubr VALUES (1, 'Казино')""")
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
con.close()
input()
```

Итак, в первом случае запрос не содержит ошибок и рубрика "Мода" будет успешно добавлена в таблицу. Во втором случае возникнет исключение `IntegrityError`. Поэтому ни рубрика "Спорт", ни рубрика "Казино" в таблицу добавлены не будут, т. к. все изменения автоматически отменяются с помощью вызова метода `rollback()`.

ГЛАВА 18

Взаимодействие с Интернетом

Интернет прочно вошел в нашу жизнь. Очень часто необходимо передать данные на Web-сервер или, наоборот, получить данные. Например, нужно получить котировки валют или прогноз погоды, проверить наличие писем в почтовом ящике и т. д. В состав стандартной библиотеки Python входит множество модулей, позволяющих работать практически со всеми протоколами Интернета. В этой главе мы рассмотрим только наиболее часто встречающиеся задачи: разбор URL-адреса и строки запроса на составляющие, преобразование гиперссылок, разбор HTML-эквивалентов, определение кодировки документа, а также обмен данными по протоколу HTTP с помощью модулей `http.client` и `urllib.request`.

18.1. Разбор URL-адреса

С помощью модуля `urllib.parse` можно манипулировать URL-адресом. Например, разобрать его на составляющие или получить абсолютный URL-адрес, указав базовый адрес и относительный. URL-адрес состоит из следующих элементов:

```
<Протокол>://<Домен>:<Порт>/<Путь>;<Параметры>?<Запрос>#<Якорь>
```

Схема URL-адреса для протокола FTP выглядит по-другому:

```
<Протокол>://<Пользователь>:<Пароль>@<Домен>
```

Разобрать URL-адрес на составляющие позволяет функция `urlparse()`:

```
urlparse(<URL-адрес>[, <Схема>[, <Разбор якоря>]])
```

Функция возвращает объект `ParseResult` с результатами разбора URL-адреса. Получить значения можно с помощью атрибутов или индексов. Объект можно преобразовать в кортеж из следующих элементов: `(scheme, netloc, path, params, query, fragment)`. Элементы соответствуют схеме URL-адреса:

```
<scheme>://<netloc>/<path>;<params>?<query>#<fragment>.
```

Обратите внимание на то, что название домена будет содержать номер порта. Кроме того, не ко всем атрибутам объекта можно получить доступ с помощью индексов. Результат разбора URL-адреса приведен в листинге 18.1.

Листинг 18.1. Разбор URL-адреса с помощью функции `urlparse()`

```
>>> from urllib.parse import urlparse
>>> url = urlparse("http://wwwadmin.ru:80/test.php;st?var=5#metka")
>>> url
```

```
ParseResult(scheme='http', netloc='wwwadmin.ru:80', path='/test.php',
params='st', query='var=5', fragment='metka')
>>> tuple(url) # Преобразование в кортеж
('http', 'wwwadmin.ru:80', '/test.php', 'st', 'var=5', 'metka')
```

Во втором параметре функции `urlparse()` можно указать название протокола, которое будет использоваться, если протокола нет в составе URL-адреса. По умолчанию используется пустая строка. Пример:

```
>>> urlparse("//wwwadmin.ru/test.php")
ParseResult(scheme='', netloc='wwwadmin.ru', path='/test.php',
params='', query='', fragment='')
>>> urlparse("//wwwadmin.ru/test.php", "http")
ParseResult(scheme='http', netloc='wwwadmin.ru', path='/test.php',
params='', query='', fragment='')
```

Объект `ParseResult`, возвращаемый функцией `urlparse()`, содержит следующие атрибуты:

- `scheme` — название протокола. Значение доступно также по индексу 0. По умолчанию пустая строка. Пример:

  ```
  >>> url.scheme, url[0]
  ('http', 'http')
  ```

- `netloc` — название домена вместе с номером порта. Значение доступно также по индексу 1. По умолчанию пустая строка. Пример:

  ```
  >>> url.netloc, url[1]
  ('wwwadmin.ru:80', 'wwwadmin.ru:80')
  ```

- `hostname` — название домена в нижнем регистре. Значение по умолчанию: `None`;
- `port` — номер порта. Значение по умолчанию: `None`. Пример:

  ```
  >>> url.hostname, url.port
  ('wwwadmin.ru', 80)
  ```

- `path` — путь. Значение доступно также по индексу 2. По умолчанию пустая строка. Пример:

  ```
  >>> url.path, url[2]
  ('/test.php', '/test.php')
  ```

- `params` — параметры. Значение доступно также по индексу 3. По умолчанию пустая строка. Пример:

  ```
  >>> url.params, url[3]
  ('st', 'st')
  ```

- `query` — строка запроса. Значение доступно также по индексу 4. По умолчанию пустая строка. Пример:

  ```
  >>> url.query, url[4]
  ('var=5', 'var=5')
  ```

- `fragment` — якорь. Значение доступно также по индексу 5. По умолчанию пустая строка. Пример:

  ```
  >>> url.fragment, url[5]
  ('metka', 'metka')
  ```

Если третий параметр в функции `urlparse()` имеет значение `False`, то якорь будет входить в состав значения других атрибутов, а не `fragment`. По умолчанию параметр имеет значение `True`. Пример:

```
>>> u = urlparse("http://site.ru/add.php?v=5#metka")
>>> u.query, u.fragment
('v=5', 'metka')
>>> u = urlparse("http://site.ru/add.php?v=5#metka","",False)
>>> u.query, u.fragment
('v=5#metka', '')
```

- `username` — имя пользователя. Значение по умолчанию: `None`;
- `password` — пароль. Значение по умолчанию: `None`. Пример:

```
>>> ftp = urlparse("ftp://user:123456@mysite.ru")
>>> ftp.scheme, ftp.hostname, ftp.username, ftp.password
('ftp', 'mysite.ru', 'user', '123456')
```

- `geturl()` — метод возвращает URL-адрес. Пример:

```
>>> url.geturl()
'http://wwwadmin.ru:80/test.php;st?var=5#metka'
```

Выполнить обратную операцию (собрать URL-адрес из отдельных значений) позволяет функция `urlunparse(<Последовательность>)` (листинг 18.2).

Листинг 18.2. Использование функции `urlunparse()`

```
>>> from urllib.parse import urlunparse
>>> t = ('http', 'wwwadmin.ru:80', '/test.php', '', 'var=5', 'metka')
>>> urlunparse(t)
'http://wwwadmin.ru:80/test.php?var=5#metka'
>>> l = ['http', 'wwwadmin.ru:80', '/test.php', '', 'var=5', 'metka']
>>> urlunparse(l)
'http://wwwadmin.ru:80/test.php?var=5#metka'
```

Вместо функции `urlparse()` можно воспользоваться функцией `urlsplit(<URL-адрес>[, <Схема>[, <Разбор якоря>]])`. Функция возвращает объект `SplitResult` с результатами разбора URL-адреса. Объект можно преобразовать в кортеж из следующих элементов: `(scheme, netloc, path, query, fragment)`. Обратиться к значениям можно как по индексу, так и названию атрибутов. Пример использования функции `urlsplit()` приведен в листинге 18.3.

Листинг 18.3. Разбор URL-адреса с помощью функции `urlsplit()`

```
>>> from urllib.parse import urlsplit
>>> url = urlsplit("http://wwwadmin.ru:80/test.php;st?var=5#metka")
>>> url
SplitResult(scheme='http', netloc='wwwadmin.ru:80',
path='/test.php;st', query='var=5', fragment='metka')
>>> url[0], url[1], url[2], url[3], url[4]
('http', 'wwwadmin.ru:80', '/test.php;st', 'var=5', 'metka')
```

```
>>> url.scheme, url.netloc, url.hostname, url.port
('http', 'wwwadmin.ru:80', 'wwwadmin.ru', 80)
>>> url.path, url.query, url.fragment
('/test.php;st', 'var=5', 'metka')
>>> ftp = urlsplit("ftp://user:123456@mysite.ru")
>>> ftp.scheme, ftp.hostname, ftp.username, ftp.password
('ftp', 'mysite.ru', 'user', '123456')
```

Выполнить обратную операцию (собрать URL-адрес из отдельных значений) позволяет функция `urlunsplit(<Последовательность>)` (листинг 18.4).

Листинг 18.4. Использование функции `urlunsplit()`

```
>>> from urllib.parse import urlunsplit
>>> t = ('http', 'wwwadmin.ru:80', '/test.php;st', 'var=5', 'metka')
>>> urlunsplit(t)
'http://wwwadmin.ru:80/test.php;st?var=5#metka'
```

18.2. Кодирование и декодирование строки запроса

В предыдущем разделе мы научились разбирать URL-адрес на составляющие. Обратите внимание на то, что значение параметра `<Запрос>` возвращается в виде строки. Строка запроса является составной конструкцией, содержащей пары `параметр=значение`. Все специальные символы внутри названия параметра и значения кодируются последовательностями `%nn`. Например, для параметра `str`, имеющего значение "Строка" в кодировке Windows-1251, строка запроса будет выглядеть так:

```
str=%D1%F2%F0%EE%EA%E0
```

Если строка запроса содержит несколько пар `параметр=значение`, то они разделяются символом `&`. Добавим параметр `v` со значением 10:

```
str=%D1%F2%F0%EE%EA%E0&v=10
```

В строке запроса может быть несколько параметров с одним названием, но разными значениями. Например, если передаются значения нескольких выбранных пунктов в списке с множественным выбором:

```
str=%D1%F2%F0%EE%EA%E0&v=10&v=20
```

Разобрать строку запроса на составляющие и декодировать данные позволяют следующие функции из модуля `urllib.parse`:

- `parse_qs()` — разбирает строку запроса и возвращает словарь с ключами, содержащими названия параметров, и списком значений. Формат функции:

  ```
  parse_qs(<Строка запроса>[, keep_blank_values=False][,
     strict_parsing=False][, encoding='utf-8'][, errors='replace'])
  ```

  Если в параметре `keep_blank_values` указано значение `True`, то параметры, не имеющие значений внутри строки запроса, также будут добавлены в результат. По умолчанию пустые параметры игнорируются. Если в параметре `strict_parsing` указано значение

`True`, то при наличии ошибки возбуждается исключение `ValueError`. По умолчанию ошибки игнорируются. Параметр `encoding` позволяет указать кодировку данных, а параметр `errors` — уровень обработки ошибок. Пример разбора строки запроса:

```
>>> from urllib.parse import parse_qs
>>> s = "str=%D1%F2%F0%EE%EA%E0&v=10&v=20&t="
>>> parse_qs(s, encoding="cp1251")
{'str': ['Строка'], 'v': ['10', '20']}
>>> parse_qs(s, keep_blank_values=True, encoding="cp1251")
{'t': [''], 'str': ['Строка'], 'v': ['10', '20']}
```

- `parse_qsl()` — функция аналогична `parse_qs()`, но возвращает не словарь, а список кортежей из двух элементов. Первый элемент кортежа содержит название параметра, а второй элемент его значение. Если строка запроса содержит несколько параметров с одинаковым значением, то они будут расположены в разных кортежах. Формат функции:

```
parse_qsl(<Строка запроса>[, keep_blank_values=False][,
   strict_parsing=False][, encoding='utf-8'][, errors='replace'])
```

Пример разбора строки запроса:

```
>>> from urllib.parse import parse_qsl
>>> s = "str=%D1%F2%F0%EE%EA%E0&v=10&v=20&t="
>>> parse_qsl(s, encoding="cp1251")
[('str', 'Строка'), ('v', '10'), ('v', '20')]
>>> parse_qsl(s, keep_blank_values=True, encoding="cp1251")
[('str', 'Строка'), ('v', '10'), ('v', '20'), ('t', '')]
```

Выполнить обратную операцию, преобразовать отдельные составляющие в строку запроса позволяет функция `urlencode()` из модуля `urllib.parse`. Формат функции:

```
urlencode(<Объект>[, doseq=False][, safe=''][, encoding=None]
          [, errors=None])
```

В качестве первого параметра можно указать словарь с данными или последовательность, каждый элемент которой содержит кортеж из двух элементов. Первый элемент кортежа становится параметром, а второй элемент его значением. Параметры и значения автоматически обрабатываются с помощью функции `quote_plus()` из модуля `urllib.parse`. В случае указания последовательности, параметры внутри строки будут идти в том же порядке, что и внутри последовательности. Пример указания словаря и последовательности приведен в листинге 18.5.

Листинг 18.5. Функция `urlencode()`

```
>>> from urllib.parse import urlencode
>>> params = {"str": "Строка 2", "var": 20}       # Словарь
>>> urlencode(params, encoding="cp1251")
'var=20&str=%D1%F2%F0%EE%EA%E0+2'
>>> params = [ ("str", "Строка 2"), ("var", 20) ] # Список
>>> urlencode(params, encoding="cp1251")
'str=%D1%F2%F0%EE%EA%E0+2&var=20'
```

Если необязательный параметр `doseq` в функции `urlencode()` имеет значение `True`, то можно указать последовательность из нескольких значений во втором параметре кортежа. В этом

случае в строку запроса добавляются несколько параметров со значениями из этой последовательности. Значение параметра `doseq` по умолчанию — `False`. В качестве примера укажем список из двух элементов (листинг 18.6).

Листинг 18.6. Составление строки запроса из элементов последовательности

```
>>> params = [ ("var", [10, 20]) ]
>>> urlencode(params, doseq=False, encoding="cp1251")
'var=%5B10%2C+20%5D'
>>> urlencode(params, doseq=True, encoding="cp1251")
'var=10&var=20'
```

Последовательность можно также указать в качестве значения в словаре:

```
>>> params = { "var": [10, 20] }
>>> urlencode(params, doseq=True, encoding="cp1251")
'var=10&var=20'
```

Выполнить кодирование и декодирование отдельных элементов строки запроса позволяют следующие функции из модуля `urllib.parse`:

- `quote()` — заменяет все специальные символы последовательностями `%nn`. Цифры, английские буквы и символы подчеркивания (_), точки (.) и дефиса (-) не кодируются. Пробелы преобразуются в последовательность `%20`. Формат функции:

  ```
  quote(<Строка>[, safe='/'][, encoding=None][, errors=None])
  ```

  В параметре `safe` можно указать символы, которые преобразовывать нельзя. По умолчанию параметр имеет значение `/`. Параметр `encoding` позволяет указать кодировку данных, а параметр `errors` — уровень обработки ошибок. Пример:

  ```
  >>> from urllib.parse import quote
  >>> quote("Строка", encoding="cp1251")  # Кодировка Windows-1251
  '%D1%F2%F0%EE%EA%E0'
  >>> quote("Строка", encoding="utf-8")   # Кодировка UTF-8
  '%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0'
  >>> quote("/~nik/"), quote("/~nik/", safe="")
  ('/%7Enik/', '%2F%7Enik%2F')
  >>> quote("/~nik/", safe="/~")
  '/~nik/'
  ```

- `quote_plus()` — функция аналогична `quote()`, но пробелы заменяются символом `+`, а не преобразуются в последовательность `%20`. Кроме того, по умолчанию символ `/` преобразуется в последовательность `%2F`. Формат функции:

  ```
  quote_plus(<Строка>[, safe=''][, encoding=None][, errors=None])
  ```

  Пример:

  ```
  >>> from urllib.parse import quote, quote_plus
  >>> quote("Строка 2", encoding="cp1251")
  '%D1%F2%F0%EE%EA%E0%202'
  >>> quote_plus("Строка 2", encoding="cp1251")
  '%D1%F2%F0%EE%EA%E0+2'
  >>> quote_plus("/~nik/")
  ```

```
'%2F%7Enik%2F'
>>> quote_plus("/~nik/", safe="/~")
'/~nik/'
```

- `quote_from_bytes()` — функция аналогична `quote()`, но в качестве первого параметра принимает последовательность байтов, а не строку. Формат функции:

  ```
  quote_from_bytes(<Последовательность байтов>[, safe='/'])
  ```

  Пример:

  ```
  >>> from urllib.parse import quote_from_bytes
  >>> quote_from_bytes(bytes("Строка 2", encoding="cp1251"))
  '%D1%F2%F0%EE%EA%E0%202'
  ```

- `unquote()` — заменяет последовательности `%nn` соответствующими символами. Символ `+` пробелом не заменяется. Формат функции:

  ```
  unquote(<Строка>[, encoding='utf-8'][, errors='replace'])
  ```

  Пример:

  ```
  >>> from urllib.parse import unquote
  >>> unquote("%D1%F2%F0%EE%EA%E0", encoding="cp1251")
  'Строка'
  >>> s = "%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0"
  >>> unquote(s, encoding="utf-8")
  'Строка'
  >>> unquote('%D1%F2%F0%EE%EA%E0+2', encoding="cp1251")
  'Строка+2'
  ```

- `unquote_plus()` — функция аналогична `unquote()`, но дополнительно заменяет символ `+` пробелом. Формат функции:

  ```
  unquote_plus(<Строка>[, encoding='utf-8'][, errors='replace'])
  ```

  Пример:

  ```
  >>> from urllib.parse import unquote_plus
  >>> unquote_plus("%D1%F2%F0%EE%EA%E0+2", encoding="cp1251")
  'Строка 2'
  >>> unquote_plus("%D1%F2%F0%EE%EA%E0%202", encoding="cp1251")
  'Строка 2'
  ```

- `unquote_to_bytes()` — функция аналогична `unquote()`, но в качестве первого параметра принимает строку или последовательность байтов и возвращает последовательность байтов. Формат функции:

  ```
  unquote_to_bytes(<Строка или последовательность байтов>)
  ```

  Пример:

  ```
  >>> from urllib.parse import unquote_to_bytes
  >>> unquote_to_bytes("%D1%F2%F0%EE%EA%E0%202")
  b'\xd1\xf2\xf0\xee\xea\xe0 2'
  >>> unquote_to_bytes(b"%D1%F2%F0%EE%EA%E0%202")
  b'\xd1\xf2\xf0\xee\xea\xe0 2'
  >>> unquote_to_bytes("%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0")
  b'\xd0\xa1\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
  >>> str(_, "utf-8")
  'Строка'
  ```

18.3. Преобразование относительной ссылки в абсолютную

Очень часто в HTML-документах указываются не абсолютные ссылки, а относительные. При относительном URL-адресе путь определяется с учетом местоположения Web-страницы, на которой находится ссылка, или значения параметра `href` тега `<base>`. Преобразовать относительную ссылку в абсолютный URL-адрес позволяет функция `urljoin()` из модуля `urllib.parse`. Формат функции:

```
urljoin(<Базовый URL-адрес>, <Относительный или абсолютный URL-адрес>
        [, <Разбор якоря>])
```

В качестве примера рассмотрим преобразование различных относительных ссылок (листинг 18.7).

Листинг 18.7. Варианты преобразования относительных ссылок

```
>>> from urllib.parse import urljoin
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', 'file.html')
'http://wwwadmin.ru/f1/f2/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', 'f3/file.html')
'http://wwwadmin.ru/f1/f2/f3/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '/file.html')
'http://wwwadmin.ru/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', './file.html')
'http://wwwadmin.ru/f1/f2/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '../file.html')
'http://wwwadmin.ru/f1/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '../../file.html')
'http://wwwadmin.ru/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '../../../file.html')
'http://wwwadmin.ru/../file.html'
```

В последнем случае мы специально указали уровень относительности больше, чем нужно. Как видно из результата, в данном случае возникает ошибка.

18.4. Разбор HTML-эквивалентов

В HTML-документе некоторые символы являются специальными. Например, знак "меньше" (<) и знак "больше" (>), кавычки и др. Для отображения специальных символов используются HTML-эквиваленты. Например, знак "меньше" заменяется последовательностью `<`, а знак "больше" — `>`. Манипулировать HTML-эквивалентами позволяют следующие функции из модуля `xml.sax.saxutils`:

- `escape(<Строка>[, <Словарь>])` — заменяет символы <, > и & соответствующими HTML-эквивалентами. Необязательный параметр `<Словарь>` позволяет указать словарь с дополнительными символами в качестве ключей и их HTML-эквивалентами в качестве значений. Пример:

```
>>> from xml.sax.saxutils import escape
>>> s = """&<>" """
```

```
>>> escape(s)
'&amp;&lt;&gt;" '
>>> escape(s, { '"': "&quot;", " ": " " } )
'&amp;&lt;&gt;&quot; '
```

- `quoteattr(<Строка>[, <Словарь>])` — функция аналогична `escape()`, но дополнительно заключает строку в кавычки или апострофы. Если внутри строки встречаются только двойные кавычки, то строка заключается в апострофы. Если внутри строки встречаются и кавычки, и апострофы, то двойные кавычки заменяются HTML-эквивалентом, а строка заключается в двойные кавычки. Если кавычки и апострофы не входят в строку, то строка заключается в двойные кавычки. Пример:

```
>>> from xml.sax.saxutils import quoteattr
>>> print(quoteattr("""&<>" """))
'&amp;&lt;&gt;" '
>>> print(quoteattr("""&<>"'"""))
"&amp;&lt;&gt;&quot;'"
>>> print(quoteattr("""&<>" """, { '"': "&quot;" } ))
"&amp;&lt;&gt;&quot; "
```

- `unescape(<Строка>[, <Словарь>])` — заменяет HTML-эквиваленты `&`, `<` и `>` обычными символами. Необязательный параметр `<Словарь>` позволяет указать словарь с дополнительными HTML-эквивалентами в качестве ключей и обычными символами в качестве значений. Пример:

```
>>> from xml.sax.saxutils import unescape
>>> s = '&amp;&lt;&gt;&quot; '
>>> unescape(s)
'&<>&quot; '
>>> unescape(s, { "&quot;": '"', " ": " " } )
'&<>" '
```

Для замены символов <, > и & HTML-эквивалентами можно также воспользоваться функцией `escape(<Строка>[, <Флаг>])` из модуля `cgi`. Если во втором параметре указано значение `True`, то двойные кавычки также будут заменяться HTML-эквивалентом (листинг 18.8). В Python 3.2 эта функция помечена устаревшей. Вместо нее следует использовать функцию `escape(<Строка>[, quote=True])` из модуля `html`.

Листинг 18.8. Замена спецсимволов HTML-эквивалентами

```
>>> import cgi, html
>>> cgi.escape("""&<>"' """), html.escape("""&<>"' """, False)
('&amp;&lt;&gt;"\' ', '&amp;&lt;&gt;"\' ')
>>> cgi.escape("""&<>"' """, True), html.escape("""&<>"' """, True)
("&amp;&lt;&gt;&quot;' ", '&amp;&lt;&gt;&quot;&#x27; ')
```

18.5. Обмен данными по протоколу HTTP

Модуль `http.client` позволяет получить информацию из Интернета по протоколам HTTP и HTTPS. Отправить запрос можно методами `GET`, `POST` и `HEAD`. Для создания объекта соединения, использующего протокол HTTP, предназначен класс `HTTPConnection`.

Конструктор класса имеет следующий формат:

```
HTTPConnection(<Домен>[, <Порт>[, strict[, timeout[, source_address]]]])
```

В первом параметре указывается название домена без протокола. Во втором параметре задается номер порта. Если порт не указан, то используется порт 80. Номер порта можно также задать после названия домена через двоеточие. Пример создания объекта соединения:

```
>>> from http.client import HTTPConnection
>>> con = HTTPConnection("test1.ru")
>>> con2 = HTTPConnection("test1.ru", 80)
>>> con3 = HTTPConnection("test1.ru:80")
```

После создания объекта соединения необходимо отправить параметры запроса с помощью метода `request()`. Формат метода:

```
request(<Метод>, <Путь>[, body=None][, headers=<Заголовки>])
```

В первом параметре указывается метод передачи данных (`GET`, `POST` или `HEAD`). Второй параметр задает путь от корня сайта. Если для передачи данных используется метод `GET`, то после вопросительного знака можно указать передаваемые данные. В необязательном третьем параметре задаются данные, которые передаются методом `POST`. Допустимо указать строку или файловый объект. Четвертый параметр задает HTTP-заголовки, отправляемые на сервер. Заголовки указываются в виде словаря.

Получить объект результата запроса позволяет метод `getresponse()`. Прочитать ответ сервера (без заголовков) можно с помощью метода `read([<Количество байт>])`. Если параметр не указан, то метод `read()` возвращает все данные, а при наличии значения — только указанное количество байтов при каждом вызове. Если данных больше нет, метод возвращает пустую строку. Прежде чем выполнять другой запрос, данные должны быть получены полностью. Метод `read()` возвращает последовательность байтов, а не строку. Закрыть объект соединения позволяет метод `close()`. В качестве примера отправим запрос методом `GET` и прочитаем результат (листинг 18.9).

Листинг 18.9. Отправка данных методом `GET`

```
>>> from http.client import HTTPConnection
>>> from urllib.parse import urlencode
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Referer": "/index.php" }
>>> con = HTTPConnection("test1.ru")
>>> con.request("GET", "/testrobots.php?%s" % data, headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print(result.read().decode("cp1251")) # Читаем данные
... Фрагмент опущен ...
>>> con.close()                 # Закрываем объект соединения
```

Теперь отправим данные методом `POST`. В этом случае в первом параметре метода `request()` задается значение "POST", а данные передаются через третий параметр. Размер строки за-

проса автоматически указывается в заголовке Content-Length. Пример отправки данных методом POST приведен в листинге 18.10.

Листинг 18.10. Отправка данных методом POST

```
>>> from http.client import HTTPConnection
>>> from urllib.parse import urlencode
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Content-Type": "application/x-www-form-urlencoded",
                "Referer": "/index.php" }
>>> con = HTTPConnection("test1.ru")
>>> con.request("POST", "/testrobots.php", data, headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print(result.read().decode("cp1251"))
... Фрагмент опущен ...
>>> con.close()
```

Обратите внимание на заголовок Content-Type. Если в этом заголовке указано значение application/x-www-form-urlencoded, то это означает, что отправлены данные формы. При наличии этого заголовка некоторые языки программирования автоматически производят разбор строки запроса. Например, в PHP переданные данные будут доступны через глобальный массив $_POST. Если заголовок не указать, то данные через массив $_POST доступны не будут.

Объект результата предоставляет следующие методы и атрибуты:

- getheader(<Заголовок>[, <Значение по умолчанию>]) — возвращает значение указанного заголовка. Если заголовок не найден, возвращается значение None или значение из второго параметра. Пример:

```
>>> result.getheader("Content-Type")
'text/plain; charset=windows-1251'
>>> print(result.getheader("Content-Types"))
None
>>> result.getheader("Content-Types", 10)
10
```

- getheaders() — возвращает все заголовки ответа сервера в виде списка кортежей. Каждый кортеж состоит из двух элементов: (<Заголовок>, <Значение>). Пример получения заголовков ответа сервера:

```
>>> result.getheaders()
[('Date', 'Mon, 02 May 2011 11:09:37 GMT'), ('Server', 'Apache/2.2.4
 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'), ('X-Powered-By',
 'PHP/5.2.4'), ('Content-Length', '422'), ('Content-Type',
 'text/plain; charset=windows-1251')]
```

С помощью функции `dict()` такой список можно преобразовать в словарь:

```
>>> dict(result.getheaders())
{'Date': 'Mon, 02 May 2011 11:09:37 GMT', 'Content-Length': '422',
 'X-Powered-By': 'PHP/5.2.4', 'Content-Type': 'text/plain;
 charset=windows-1251', 'Server': 'Apache/2.2.4 (Win32)
 mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'}
```

- `status` — код возврата в виде числа. Успешными считаются коды от 200 до 299 и код 304, означающий, что документ не был изменен со времени последнего посещения. Коды 301 и 302 задают перенаправление. Код 401 означает необходимость авторизации, 403 — доступ закрыт, 404 — документ не найден, а код 500 и коды выше информируют об ошибке сервера. Пример:

```
>>> result.status
200
```

- `reason` — текстовый статус возврата. Пример:

```
>>> result.reason           # При коде 200
'OK'
>>> result.reason           # При коде 302
'Moved Temporarily'
```

- `version` — версия протокола в виде числа. Число 10 для протокола HTTP/1.0 и число 11 для протокола HTTP/1.1. Пример:

```
>>> result.version          # Протокол HTTP/1.1
11
```

- `msg` — объект `http.client.HTTPMessage`. С его помощью можно получить дополнительную информацию о заголовках ответа сервера. Если объект передать функции `print()`, то получим все заголовки ответа сервера:

```
>>> print(result.msg)
Date: Mon, 02 May 2011 11:09:37 GMT
Server: Apache/2.2.4 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4
X-Powered-By: PHP/5.2.4
Content-Length: 422
Content-Type: text/plain; charset=windows-1251
```

Рассмотрим основные методы и атрибуты объекта `http.client.HTTPMessage`:

- `as_string([unixfrom=False][, maxheaderlen=0])` — возвращает все заголовки ответа сервера в виде строки. Пример:

```
>>> result.msg.as_string()
'Date: Mon, 02 May 2011 11:09:37 GMT\nServer: Apache/2.2.4 (Win32)
 mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4\nX-Powered-By:
 PHP/5.2.4\nContent-Length: 422\nContent-Type: text/plain;
 charset=windows-1251\n\n'
```

- `items()` — список всех заголовков ответа сервера. Пример:

```
>>> result.msg.items()
[('Date', 'Mon, 02 May 2011 11:09:37 GMT'), ('Server', 'Apache/2.2.4
 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'), ('X-Powered-By',
```

```
 'PHP/5.2.4'), ('Content-Length', '422'), ('Content-Type',
 'text/plain; charset=windows-1251')]
```

- `keys()` — список ключей в заголовках ответа сервера. Пример:

```
>>> result.msg.keys()
['Date', 'Server', 'X-Powered-By', 'Content-Length', 'Content-Type']
```

- `values()` — список значений в заголовках ответа сервера. Пример:

```
>>> result.msg.values()
['Mon, 02 May 2011 01:09:37 GMT', 'Apache/2.2.4 (Win32) mod_ssl/2.2.4
 OpenSSL/0.9.8d PHP/5.2.4', 'PHP/5.2.4', '422', 'text/plain;
 charset=windows-1251']
```

- `get(<Заголовок>[, failobj=None])` — возвращает значение указанного заголовка в виде строки. Если заголовок не найден, возвращается значение `None` или значение из второго параметра. Пример:

```
>>> result.msg.get("X-Powered-By")
'PHP/5.2.4'
>>> print(result.msg.get("X-Powered-By2"))
None
>>> result.msg.get("X-Powered-By2", failobj=10)
10
```

- `get_all(<Заголовок>[, failobj=None])` — возвращает список всех значений указанного заголовка. Если заголовок не найден, возвращается значение `None` или значение из второго параметра. Пример:

```
>>> result.msg.get_all("X-Powered-By")
['PHP/5.2.4']
```

- `get_content_type()` — возвращает MIME-тип документа из заголовка `Content-Type`:

```
>>> result.msg.get_content_type()
'text/plain'
```

- `get_content_maintype()` — возвращает первую составляющую MIME-типа:

```
>>> result.msg.get_content_maintype()
'text'
```

- `get_content_subtype()` — возвращает вторую составляющую MIME-типа:

```
>>> result.msg.get_content_subtype()
'plain'
```

- `get_content_charset([failobj=None])` — позволяет получить кодировку из заголовка `Content-Type`. Если кодировка не найдена, возвращается значение `None` или значение из параметра `failobj`. Получим кодировку документа:

```
>>> result.msg.get_content_charset()
'windows-1251'
```

В качестве примера отправим запрос методом `HEAD` и выведем заголовки ответа сервера (листинг 18.11).

Листинг 18.11. Отправка запроса методом `HEAD`

```
>>> from http.client import HTTPConnection
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Referer": "/index.php" }
>>> con = HTTPConnection("test1.ru")
>>> con.request("HEAD", "/testrobots.php", headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print(result.msg)
Date: Mon, 02 May 2011 12:33:46 GMT
Server: Apache/2.2.4 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4
X-Powered-By: PHP/5.2.4
Content-Type: text/plain; charset=windows-1251

>>> result.read() # Данные не передаются, только заголовки!
b''
>>> con.close()
```

Перечислим основные HTTP-заголовки и их предназначение:

- `GET` — заголовок запроса при передаче данных методом `GET`;
- `POST` — заголовок запроса при передаче данных методом `POST`;
- `Host` — название домена;
- `Accept` — MIME-типы, поддерживаемые Web-браузером;
- `Accept-Language` — список поддерживаемых языков в порядке предпочтения;
- `Accept-Charset` — список поддерживаемых кодировок;
- `Accept-Encoding` — список поддерживаемых методов сжатия;
- `Content-Type` — тип передаваемых данных;
- `Content-Length` — длина передаваемых данных при методе `POST`;
- `Cookie` — информация об установленных cookies;
- `Last-Modified` — дата последней модификации файла;
- `Location` — перенаправление на другой URL-адрес;
- `Pragma` — заголовок, запрещающий кэширование документа в протоколе HTTP/1.0;
- `Cache-Control` — заголовок, управляющий кэшированием документа в протоколе HTTP/1.1;
- `Referer` — URL-адрес, с которого пользователь перешел на наш сайт;
- `Server` — название и версия программного обеспечения сервера;
- `User-Agent` — информация об используемом Web-браузере.

Получить полное описание заголовков можно в спецификации RFC 2616, расположенной по адресу **http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html**. Чтобы "подсмотреть"

заголовки, отправляемые Web-браузером и сервером, можно воспользоваться модулем Firebug для Firefox. Для этого на вкладке **Сеть** следует щелкнуть мышью на строке запроса. Кроме того, можно установить панель ieHTTPHeaders в Web-браузере Internet Explorer.

18.6. Обмен данными с помощью модуля *urllib.request*

Модуль `urllib.request` предоставляет расширенные возможности для получения информации из Интернета. Поддерживаются автоматические перенаправления при получении заголовка `Location`, возможность аутентификации, обработка cookies и др.

Для выполнения запроса предназначена функция `urlopen()`. Формат функции:

```
urlopen(<URL-адрес или объект запроса>[, <Данные>][, <Тайм-аут>]
       [, cafile=None][, capath=None])
```

В первом параметре задается полный URL-адрес или объект, возвращаемый конструктором класса `Request`. Запрос выполняется методом `GET`, если данные не указаны во втором параметре, и методом `POST` в противном случае. При передаче данных методом `POST` автоматически добавляется заголовок `Content-Type` со значением `application/x-www-form-urlencoded`. Третий параметр задает максимальное время выполнения запроса в секундах. Объект, возвращаемый функцией `urlopen()`, содержит следующие методы и атрибуты:

- `read([<Количество байтов>])` — считывает данные. Если параметр не указан, то возвращается содержимое результата от текущей позиции указателя до конца. Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество байтов. Когда достигается конец, метод возвращает пустую строку. Пример:

```
>>> from urllib.request import urlopen
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> print(res.read(34).decode("cp1251"))
Название робота: Python-urllib/3.2
>>> print(res.read().decode("cp1251"))
... Фрагмент опущен ...
>>> res.read()
b''
```

- `readline([<Количество байтов>])` — считывает одну строку при каждом вызове. При достижении конца возвращается пустая строка. Если в необязательном параметре указано число, то считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца или не будет прочитано указанное количество байтов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество байтов. Если количество байтов в строке больше, то возвращается указанное количество байтов. Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> print(res.readline().decode("cp1251"))
Название робота: Python-urllib/3.2
```

- `readlines([<Количество байтов>])` — считывает весь результат в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Если задан параметр, то считывается указанное количество байтов плюс фрагмент до конца строки. При достижении конца возвращается пустой список.

Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> res.readlines(3)
[b'\xcd\xe0\xe7\xe2\xe0\xed\xe8\xe5 \xf0\xee\xe1\xee\xf2\xe0:
 Python-urllib/3.2\n']
>>> res.readlines()
... Фрагмент опущен ...
>>> res.readlines()
[]
```

- `__next__()` — считывает одну строку при каждом вызове. При достижении конца результата возбуждается исключение `StopIteration`. Благодаря методу `__next__()` можно перебирать результат построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `__next__()`. Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> for line in res: print(line)
```

- `close()` — закрывает объект результата;
- `geturl()` — возвращает URL-адрес полученного документа. Так как все перенаправления автоматически обрабатываются, URL-адрес полученного документа может не совпадать с URL-адресом, заданным первоначально;
- `info()` — возвращает объект, с помощью которого можно получить информацию о заголовках ответа сервера. Основные методы и атрибуты этого объекта мы рассматривали при изучении модуля `http.client` (см. значение атрибута `msg` объекта результата). Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> info = res.info()
>>> info.items()
[('Date', 'Mon, 02 May 2011 12:01:44 GMT'), ('Server', 'Apache/2.2.4
 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'), ('X-Powered-By',
 'PHP/5.2.4'), ('Content-Length', '288'), ('Connection', 'close'),
 ('Content-Type', 'text/plain; charset=windows-1251')]
>>> info.get("Content-Type")
'text/plain; charset=windows-1251'
>>> info.get_content_type(), info.get_content_charset()
('text/plain', 'windows-1251')
>>> info.get_content_maintype(), info.get_content_subtype()
('text', 'plain')
```

- `code` — содержит код возврата в виде числа;
- `msg` — содержит текстовый статус возврата. Пример:

```
>>> res.code, res.msg
(200, 'OK')
```

В качестве примера выполним запросы методами `GET` и `POST` (листинг 18.12).

Листинг 18.12. Отправка данных методами GET и POST

```
>>> from urllib.request import urlopen
>>> from urllib.parse import urlencode
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
```

```
>>> # Отправка данных методом GET
>>> url = "http://test1.ru/testrobots.php?" + data
>>> res = urlopen(url)
>>> print(res.read(34).decode("cp1251"))
Название робота: Python-urllib/3.2
>>> res.close()
>>> # Отправка данных методом POST
>>> url = " http://test1.ru/testrobots.php"
>>> res = urlopen(url, data.encode("cp1251"))
>>> print(res.read().decode("cp1251"))
... Фрагмент опущен ...
>>> res.close()
```

Как видно из результата, по умолчанию название робота — "Python-urllib/<Версия Python>". Если необходимо задать свое название робота и передать дополнительные заголовки, то следует создать экземпляр класса `Request` и передать его в функцию `urlopen()` вместо URL-адреса. Конструктор класса `Request` имеет следующий формат:

```
Request(<URL-адрес>[, data=None][, headers={}][, origin_req_host=None]
                   [, unverifiable=False])
```

В первом параметре указывается URL-адрес. Запрос выполняется методом `GET`, если данные не указаны во втором параметре, и методом `POST` в противном случае. При передаче данных методом `POST` автоматически добавляется заголовок `Content-Type` со значением `application/x-www-form-urlencoded`. Третий параметр задает заголовки запроса в виде словаря. Четвертый и пятый параметр используются для обработки cookies. За дополнительной информацией по этим параметрам обращайтесь к документации. В качестве примера выполним запросы методами `GET` и `POST` (листинг 18.13).

Листинг 18.13. Использование класса `Request`

```
>>> from urllib.request import urlopen, Request
>>> from urllib.parse import urlencode
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Referer": "/index.php" }
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
>>> # Отправка данных методом GET
>>> url = "http://test1.ru/testrobots.php?" + data
>>> request = Request(url, headers=headers)
>>> res = urlopen(request)
>>> print(res.read(29).decode("cp1251"))
Название робота: MySpider/1.0
>>> res.close()
>>> # Отправка данных методом POST
>>> url = "http://test1.ru/testrobots.php"
>>> request = Request(url, data.encode("cp1251"), headers=headers)
>>> res = urlopen(request)
```

```
>>> print(res.read().decode("cp1251"))
... Фрагмент опущен ...
>>> res.close()
```

Как видно из результата, название нашего робота теперь "MySpider/1.0".

18.7. Определение кодировки

Документы в Интернете могут быть в различных кодировках. Чтобы документ был правильно обработан, необходимо знать его кодировку. Определить кодировку можно по заголовку `Content-Type` в заголовках ответа сервера:

```
Content-Type: text/html; charset=utf-8
```

Кодировку HTML-документа можно также определить по значению параметра `content` тега `<meta>`, расположенного в разделе `HEAD`:

```
<meta http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
```

Очень часто встречается ситуация, когда кодировка в ответе сервера не совпадает с кодировкой, указанной в теге `<meta>`, или кодировка вообще не указана. Определить кодировку документа в этом случае позволяет библиотека `chardet`. Для установки библиотеки со страницы **http://chardet.feedparser.org/download/** скачиваем архив python3-chardet-2.0.1.tgz, а затем распаковываем его в текущую папку, например, с помощью архиватора WinRAR. Запускаем командную строку и переходим в папку с библиотекой (в моем случае библиотека разархивирована в папку C:\book), выполнив команду:

```
cd C:\book\python3-chardet-2.0.1
```

Затем запускаем программу установки, выполнив команду:

```
C:\Python32\python.exe setup.py install
```

Для проверки установки запускаем редактор IDLE и в окне **Python Shell** выполняем следующий код:

```
>>> import chardet
>>> chardet.__version__
'2.0.1'
```

Определить кодировку строки позволяет функция `detect(<Последовательность байтов>)`. В качестве значения функция возвращает словарь с двумя элементами. Ключ `encoding` содержит название кодировки, а ключ `confidence` — коэффициент точности определения (вещественное число от `0` до `1`). Пример определения кодировки приведен в листинге 18.14.

Листинг 18.14. Пример определения кодировки

```
>>> import chardet
>>> chardet.detect(bytes("Строка", "cp1251"))
{'confidence': 0.99, 'encoding': 'windows-1251'}
>>> chardet.detect(bytes("Строка", "koi8-r"))
{'confidence': 0.99, 'encoding': 'KOI8-R'}
>>> chardet.detect(bytes("Строка", "utf-8"))
{'confidence': 0.99, 'encoding': 'utf-8'}
```

Если файл имеет большой размер, то вместо считывания всего файла в строку и использования функции `detect()` можно воспользоваться классом `UniversalDetector`. В этом случае можно читать файл построчно и передавать текущую строку методу `feed()`. Если определение кодировки прошло успешно, атрибут `done` будет иметь значение `True`. Это условие можно использовать для выхода из цикла. После окончания проверки следует вызвать метод `close()`. Получить результат определения кодировки позволяет атрибут `result`. Очистить результат и подготовить объект к дальнейшему определению кодировки можно с помощью метода `reset()`. Пример использования класса `UniversalDetector` приведен в листинге 18.15.

Листинг 18.15. Пример использования класса `UniversalDetector`

```
# -*- coding: utf-8 -*-
from chardet.universaldetector import UniversalDetector
ud = UniversalDetector()       # Создаем объект
for line in open("file.txt", "rb"):
    ud.feed(line)              # Передаем текущую строку
    if ud.done: break          # Прерываем цикл, если done == True
ud.close()                     # Закрываем объект
print(ud.result)               # Выводим результат
input()
```

Как показали мои тесты, при использовании кодировки Windows-1251 файл просматривается полностью. Причем определение кодировки файла, содержащего 6500 строк, занимает почти секунду. Если сменить кодировку файла на UTF-8 без BOM, то время определения увеличивается до 5 секунд. Использовать класс `UniversalDetector` или нет — решать вам.

ЧАСТЬ II

Создание оконных приложений

Глава 19. Знакомство с PyQt

Глава 20. Управление окном приложения

Глава 21. Обработка сигналов и событий

Глава 22. Размещение нескольких компонентов в окне

Глава 23. Основные компоненты

Глава 24. Списки и таблицы

Глава 25. Работа с графикой

Глава 26. Графическая сцена

Глава 27. Диалоговые окна

Глава 28. Создание SDI- и MDI-приложений

ГЛАВА 19

Знакомство с PyQt

Итак, изучение основ языка Python закончено, и мы можем перейти к рассмотрению библиотеки PyQt, позволяющей разрабатывать приложения с графическим интерфейсом. Первые три главы во второй части книги являются основными, так как в них описываются базовые возможности библиотеки и методы, которые наследуют все компоненты. Материал этих глав нужно знать обязательно. Остальные главы содержат справочный материал по основным компонентам. Все листинги (более 750 шт.) для этих глав доступны отдельно. Во-первых, большие листинги в книге не смотрятся, а во-вторых, это позволило уменьшить объем всей книги, т. к. объем листингов превышает объем второй части книги почти в два раза. Информация о способе получения дополнительных листингов доступна на обложке книги.

19.1. Установка PyQt

Библиотека PyQt не входит в состав стандартной библиотеки Python. Прежде чем начать изучение основ, необходимо установить PyQt на компьютер.

1. Скачиваем программу установки PyQt-Py3.2-x86-gpl-4.8.3-1.exe со страницы **http://www.riverbankcomputing.co.uk/software/pyqt/download** и запускаем ее с помощью двойного щелчка на значке файла.
2. В открывшемся окне (рис. 19.1) нажимаем кнопку **Next**.
3. Далее (рис. 19.2) принимаем лицензионное соглашение, нажимая кнопку **I Agree**.
4. Теперь (рис. 19.3) можно выбрать компоненты, которые следует установить. Оставляем выбранными все компоненты и нажимаем кнопку **Next.**
5. На следующем шаге (рис. 19.4) задается путь к каталогу, в котором расположен интерпретатор Python (C:\Python32\). Нажимаем кнопку **Install** для запуска процесса установки PyQt.
6. После завершения установки будет выведено окно, изображенное на рис. 19.5. Нажимаем кнопку **Finish** для выхода из программы установки.

В результате установки все необходимые файлы будут скопированы в папку C:\Python32\Lib\site-packages\PyQt4\, а в начало системной переменной **PATH** добавлен путь к папке C:\Python32\Lib\site-packages\PyQt4\bin. В папке bin расположены программы Designer, Linguist и Assistant, а также библиотеки динамической компоновки (например, QtCore4.dll, QtGui4.dll), необходимые для нормального функционирования программы, написанной на

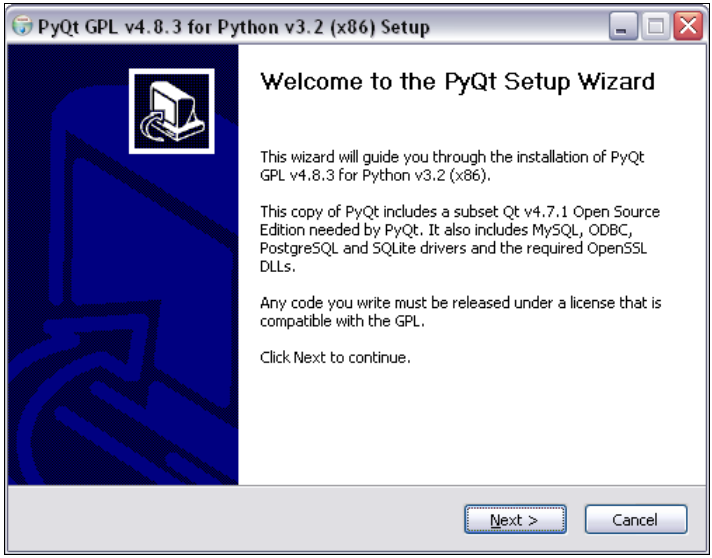

Рис. 19.1. Установка PyQt. Шаг 1

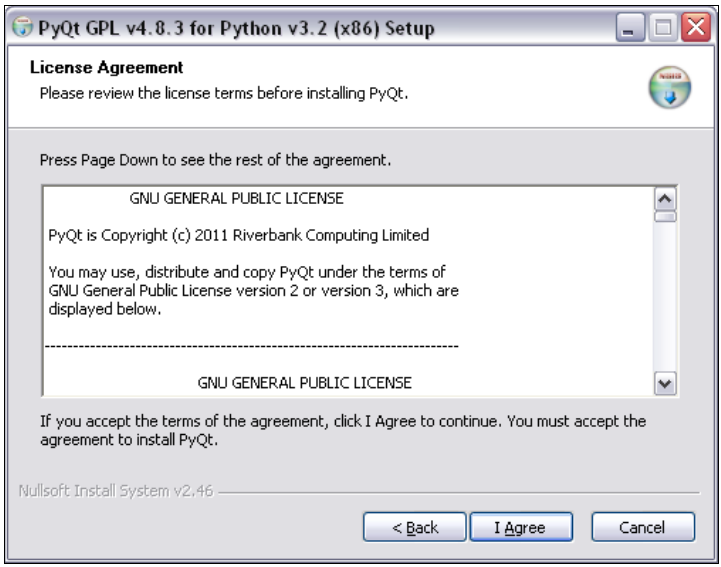

Рис. 19.2. Установка PyQt. Шаг 2

PyQt. Кроме того, в папке bin находится библиотека libmySQL.dll, предназначенная для доступа к базе данных MySQL. Так как путь к папке C:\Python32\Lib\site-packages\PyQt4\bin добавляется в самое начало переменной **PATH**, библиотека libmySQL.dll будет всегда подгружаться из этой папки во всех программах. Если вы занимаетесь Web-программированием и подключаетесь к MySQL из PHP версии 5.2, то возможны проблемы с несоответствием версий библиотеки libmySQL.dll. Если проблема возникает, то следует переместить путь к папке bin, например, в самый конец переменной **PATH**.

Чтобы проверить правильность установки, выведем версии PyQt и Qt (листинг 19.1).

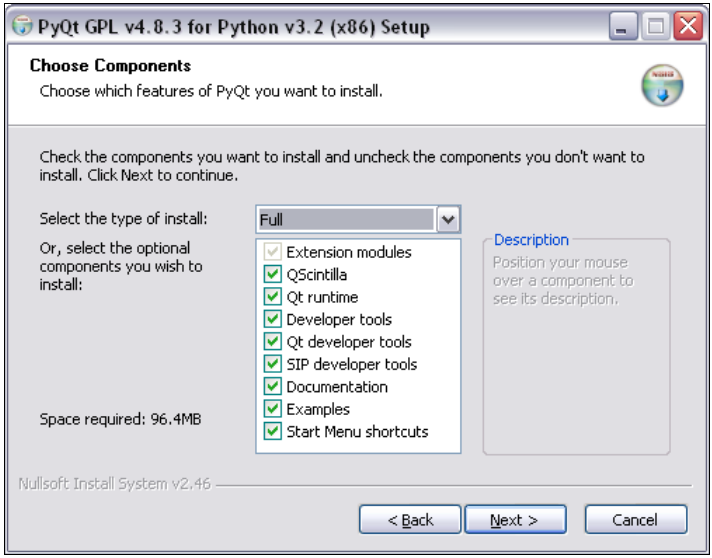

Рис. 19.3. Установка PyQt. Шаг 3

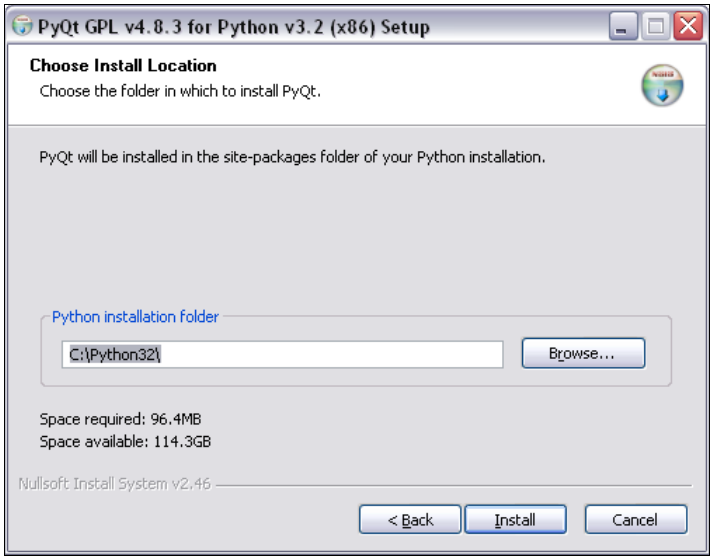

Рис. 19.4. Установка PyQt. Шаг 4

Листинг 19.1. Проверка правильности установки PyQt

```
>>> from PyQt4 import QtCore
>>> QtCore.PYQT_VERSION_STR
'4.8.3'
>>> QtCore.QT_VERSION_STR
'4.7.1'
```

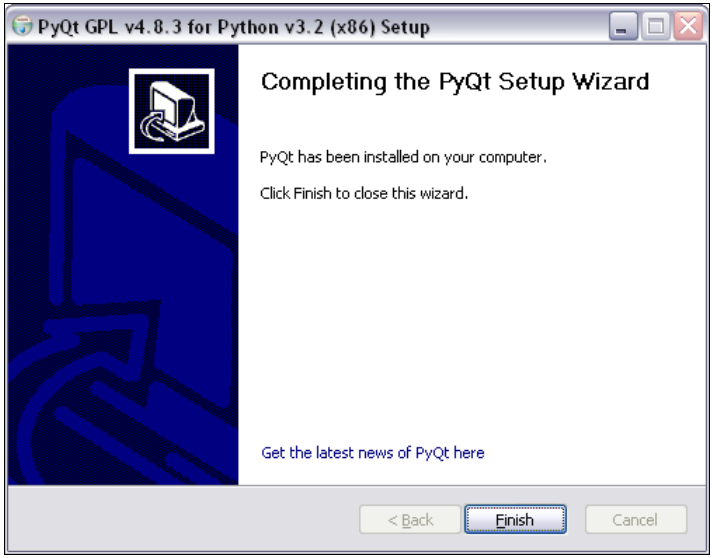

Рис. 19.5. Установка PyQt. Шаг 5

19.2. Первая программа

При изучении языков и технологий принято начинать с программы, выводящей надпись "Привет, мир!". Не будем нарушать традицию и создадим окно с приветствием и кнопкой для закрытия окна (листинг 19.2).

Листинг 19.2. Первая программа на PyQt

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Первая программа на PyQt")
window.resize(300, 70)
label = QtGui.QLabel("<center>Привет, мир!</center>")
btnQuit = QtGui.QPushButton("&Закрыть окно")
vbox = QtGui.QVBoxLayout()
vbox.addWidget(label)
vbox.addWidget(btnQuit)
window.setLayout(vbox)
QtCore.QObject.connect(btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
window.show()
sys.exit(app.exec_())
```

Для создания файла с программой можно по-прежнему пользоваться редактором IDLE. Однако в IDLE не работает автодополнение кода для PyQt, поэтому названия классов и ме-

тодов придется набирать вручную. Многим программистам это не нравится. Кроме того, запуск оконного приложения из IDLE (нажатием клавиши <F5>) приводит к очень неприятным ошибкам и даже аварийному завершению работы редактора. Поэтому запускать оконные приложения следует двойным щелчком на значке файла.

Вместо редактора IDLE для редактирования и запуска программ на PyQt советую воспользоваться редактором Eclipse и модулем PyDev. В этом случае при использовании точечной нотации будет автоматически выводиться список классов, методов и атрибутов. Кроме того, при выделении метода в списке можно посмотреть, какие параметры принимает метод и что он возвращает. Запуск программы из Eclipse выполняется очень просто. Достаточно нажать кнопку на панели инструментов. Рассмотрение возможностей Eclipse выходит за рамки этой книги, поэтому изучать редактор вам придется самостоятельно.

До сих пор мы создавали файлы с расширением py и все результаты выполнения программы выводили в окно консоли. Оконное приложение также можно сохранить с расширением py, но при запуске помимо основного окна будет дополнительно выводиться окно консоли. На этапе отладки в окно консоли можно выводить отладочную информацию (этим способом мы будем пользоваться в дальнейших примерах). Чтобы избавиться от окна консоли, следует сохранять файл с расширением pyw. Попробуйте создать два файла с различным расширением и запустить их с помощью двойного щелчка на значке. Результат выполнения листинга 19.2 показан на рис. 19.6.

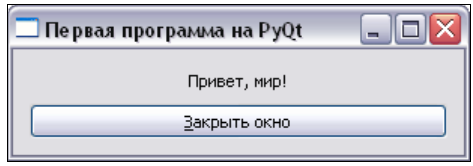

Рис. 19.6. Результат выполнения листинга 19.2

19.3. Структура программы

Запускать программу мы научились, теперь рассмотрим код из листинга 19.2 построчно. В первой строке указывается кодировка файла. Так как кодировка UTF-8 является в Python 3 кодировкой модулей по умолчанию, эту строку можно и не указывать. Во второй строке подключаются модули `QtCore` и `QtGui`. Модуль `QtCore` содержит классы не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули PyQt. Модуль `QtGui` содержит классы, реализующие компоненты графического интерфейса, например надписи, кнопки, текстовые поля и др. В третьей строке производится подключение модуля `sys`, из которого нам потребуется список параметров, переданных в командной строке (`argv`), а также функция `exit()`, позволяющая завершить выполнение программы.

Инструкция

```
app = QtGui.QApplication(sys.argv)
```

создает объект приложения с помощью класса `QApplication`. Конструктор этого класса принимает список параметров, переданных в командной строке. Следует помнить, что в программе всегда должен быть объект приложения, причем обязательно только один. Может показаться, что после создания объекта он больше нигде не используется в программе, однако с помощью этого объекта осуществляется управление приложением незаметно для нас. Получить доступ к этому объекту из любого места в программе можно через атрибут `qApp`

из модуля QtGui. Например, вывести список параметров, переданных в командной строке, можно так:

```
print(QtGui.qApp.argv())
```

Следующая инструкция

```
window = QtGui.QWidget()
```

создает объект окна с помощью класса QWidget. Этот класс наследуют практически все классы, реализующие компоненты графического интерфейса. Поэтому любой компонент, не имеющий родителя, обладает своим собственным окном.

Инструкция

```
window.setWindowTitle("Первая программа на PyQt")
```

задает текст, который будет выводиться в заголовке окна. Следующая инструкция

```
window.resize(300, 70)
```

задает минимальные размеры окна. В первом параметре метода resize() указывается ширина окна, а во втором параметре — высота окна. Следует учитывать, что эти размеры не включают высоту заголовка окна и ширину границ, а также являются рекомендацией, т. е. если компоненты не помещаются, размеры окна будут увеличены.

Инструкция

```
label = QtGui.QLabel("<center>Привет, мир!</center>")
```

создает объект надписи. Текст надписи задается в качестве параметра в конструкторе класса QLabel. Обратите внимание на то, что внутри строки мы указали HTML-теги. В данном примере с помощью тега произвели выравнивание текста по центру компонента. Возможность использования HTML-тегов и CSS-атрибутов является отличительной чертой библиотеки PyQt. Например, внутри надписи можно вывести таблицу или отобразить изображение. Это очень удобно.

Следующая инструкция

```
btnQuit = QtGui.QPushButton("&Закрыть окно")
```

создает объект кнопки. Текст, который будет отображен на кнопке, задается в качестве параметра в конструкторе класса QPushButton. Обратите внимание на символ & перед буквой "З". Таким образом задаются клавиши быстрого доступа. Если нажать одновременно клавишу <Alt> и клавишу с буквой, перед которой в строке указан символ &, то кнопка будет нажата.

Инструкция

```
vbox = QtGui.QVBoxLayout()
```

создает вертикальный контейнер. Все компоненты, добавляемые в этот контейнер, будут располагаться друг под другом в порядке добавления. Внутри контейнера автоматически производится подгонка размеров добавляемых компонентов под размеры контейнера. При изменении размеров контейнера будет произведено изменение размеров всех компонентов. В следующих двух инструкциях

```
vbox.addWidget(label)
vbox.addWidget(btnQuit)
```

с помощью метода addWidget() производится добавление объектов надписи и кнопки в вертикальный контейнер. Так как объект надписи добавляется первым, он будет расположен

над кнопкой. При добавлении компонентов в контейнер, они автоматически становятся потомками контейнера.

Следующая инструкция

```
window.setLayout(vbox)
```

добавляет контейнер в основное окно с помощью метода `setLayout()`. Таким образом, контейнер становится потомком основного окна.

Инструкция

```
QtCore.QObject.connect(btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
```

назначает обработчик сигнала `clicked()`, который генерируется при нажатии кнопки. В первом параметре статического метода `connect()` указывается объект, генерирующий сигнал, а во втором параметре — название сигнала. В третьем параметре указывается объект, принимающий сигнал, а в четвертом параметре — метод этого объекта, который будет вызван при наступлении события. Этот метод принято называть *слотом*. В нашем примере получателем сигнала является объект приложения, доступный через атрибут `qApp`. При наступлении события будет вызван метод `quit()`, который завершит работу всего приложения.

Следующая инструкция

```
window.show()
```

отображает окно и все компоненты, которые мы ранее добавили. И, наконец, инструкция

```
sys.exit(app.exec_())
```

запускает бесконечный цикл обработки событий. Инструкции, расположенные после вызова метода `exec_()`, будут выполнены только после завершения работы приложения. Так как результат выполнения метода `exec_()` мы передаем функции `exit()`, дальнейшее выполнение программы будет прекращено, а код возврата передан операционной системе.

19.4. ООП-стиль создания окна

Библиотека PyQt написана в объектно-ориентированном стиле (ООП-стиле) и содержит более 600 классов. Иерархия наследования всех классов имеет слишком большой размер, поэтому приводить ее в книге нет возможности. Тем не менее, чтобы показать зависимости, при описании компонентов иерархия наследования конкретного класса будет показываться. В качестве примера выведем базовые классы класса `QWidget`:

```
>>> from PyQt4 import QtGui
>>> QtGui.QWidget.__bases__
(<class 'PyQt4.QtCore.QObject'>, <class 'PyQt4.QtGui.QPaintDevice'>)
```

Как видно из примера, класс `QWidget` наследует два класса — `QObject` и `QPaintDevice`. Класс `QObject` является классом верхнего уровня, его наследуют большинство классов в PyQt. В свою очередь класс `QWidget` является базовым классом для всех визуальных компонентов. Таким образом, класс `QWidget` наследует все атрибуты и методы базовых классов. Это обстоятельство следует учитывать при изучении документации, т. к. в ней описываются атрибуты и методы конкретного класса, а на унаследованные атрибуты и методы даются только ссылки.

В своих программах вы можете наследовать стандартные классы и добавлять новую функциональность. В качестве примера переделаем код из листинга 19.2 и создадим окно в ООП-стиле (листинг 19.3).

Листинг 19.3. ООП-стиль создания окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Привет, мир!")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.btnQuit = QtGui.QPushButton("&Закрыть окно")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnQuit)
        self.setLayout(self.vbox)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()   # Создаем экземпляр класса
    window.setWindowTitle("ООП-стиль создания окна")
    window.resize(300, 70)
    window.show()         # Отображаем окно
    sys.exit(app.exec_()) # Запускаем цикл обработки событий
```

В первых двух строках как обычно указывается кодировка файла и подключаются модули `QtCore` и `QtGui`. Далее мы определяем класс `MyWindow`, который наследует класс `QWidget`:

```
class MyWindow(QtGui.QWidget):
```

Помимо класса `QWidget` можно наследовать и другие классы, которые являются наследниками класса `QWidget`, например класс `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При наследовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь только две кнопки в заголовке окна — **Справка** и **Закрыть**. Кроме того, можно наследовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Наследование класса `QMainWindow` имеет свои отличия, которые мы будем рассматривать в отдельной главе.

Следующая инструкция

```
def __init__(self, parent=None):
```

создает конструктор класса. В качестве параметров конструктор принимает ссылку на экземпляр класса (`self`) и ссылку на родительский компонент (`parent`). Родительский компонент может отсутствовать, поэтому в определении конструктора параметру присваивается

значение по умолчанию (`None`). Внутри метода `__init__()` вызывается конструктор базового класса и ему передается ссылка на родительский компонент:

```
QtGui.QWidget.__init__(self, parent)
```

Следующие инструкции внутри конструктора создают объекты надписи, кнопки и контейнера, затем добавляют компоненты в контейнер, а сам контейнер в основное окно. Следует обратить внимание на то, что объекты надписи и кнопки сохраняются в атрибутах экземпляра класса. В дальнейшем из методов класса можно управлять этими объектами, например, изменить текст надписи. Если объекты не сохранить, то получить к ним доступ будет не так просто.

Далее производится назначение обработчика нажатия кнопки:

```
self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
             QtGui.qApp.quit)
```

В отличие от предыдущего примера (листинг 19.2), метод `connect()` вызывается не через класс `QObject`, а через экземпляр нашего класса. Это возможно, т. к. мы наследовали класс `QWidget`, который в свою очередь является наследником класса `QObject` и наследует метод `connect()`. Еще одно отличие состоит в количестве параметров, которые принимает метод `connect()`. В первом параметре как обычно указывается объект, генерирующий сигнал, а во втором — название обрабатываемого сигнала. В третьем параметре указывается ссылка на метод, который будет вызван при наступлении события. Это возможно, т. к. в языке Python функция является обычным объектом, на который можно получить ссылку, указав название функции без круглых скобок.

В предыдущем примере (листинг 19.2) мы выравнивали надпись с помощью HTML-разметки. Произвести аналогичную операцию позволяет также метод `setAlignment()`, которому следует передать атрибут `AlignCenter`:

```
self.label.setAlignment(QtCore.Qt.AlignCenter)
```

Создание объекта приложения и экземпляра класса `MyWindow` производится внутри условия:

```
if __name__ == "__main__":
```

Если вы внимательно читали первую часть книги, то сможете вспомнить, что атрибут модуля `__name__` будет содержать значение `"__main__"` только в случае запуска модуля как главной программы. Если модуль импортировать, то атрибут будет содержать другое значение. Поэтому весь последующий код создания объекта приложения и объекта окна выполняется только при запуске программы с помощью двойного щелчка на значке файла. Может возникнуть вопрос, зачем это нужно? Одним из преимуществ ООП-стиля программирования является повторное использование кода. Следовательно, можно импортировать модуль и использовать класс `MyWindow` в другом приложении. Рассмотрим эту возможность на примере. Сохраняем код из листинга 19.3 в файле с названием MyWindow.py, а затем создаем в той же папке еще один файл (например, с названием test.pyw) и вставляем в него код из листинга 19.4.

Листинг 19.4. Повторное использование кода при ООП-стиле

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import MyWindow
```

```
class MyDialog(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QDialog.__init__(self, parent)
        self.myWidget = MyWindow.MyWindow()
        self.myWidget.vbox.setMargin(0)
        self.button = QtGui.QPushButton("&Изменить надпись")
        mainBox = QtGui.QVBoxLayout()
        mainBox.addWidget(self.myWidget)
        mainBox.addWidget(self.button)
        self.setLayout(mainBox)
        self.connect(self.button, QtCore.SIGNAL("clicked()"),
                     self.on_clicked)

    def on_clicked(self):
        self.myWidget.label.setText("Новая надпись")
        self.button.setDisabled(True)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyDialog()
    window.setWindowTitle("Преимущество ООП-стиля")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

Теперь запустим файл test.pyw с помощью двойного щелчка на значке файла. В результате будет отображено окно с надписью и двумя кнопками (рис. 19.7). При нажатии кнопки **Изменить надпись** производится изменение текста надписи и кнопка делается неактивной. Нажатие кнопки **Закрыть окно** будет по-прежнему завершать выполнение приложения.

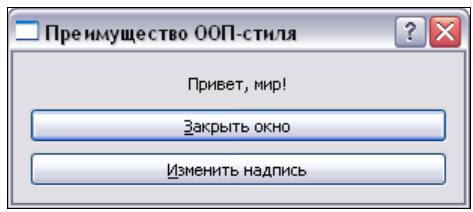

Рис. 19.7. Результат выполнения листинга 19.4

В этом примере мы создали класс `MyDialog`, который наследует класс `QDialog`. Поэтому при выводе окно автоматически выравнивается по центру экрана. Кроме того, в заголовке окна выводятся только две кнопки — **Справка** и **Закрыть**. Внутри конструктора мы создаем экземпляр класса `MyWindow` и сохраняем его в атрибуте `myWidget`:

```
self.myWidget = MyWindow.MyWindow()
```

С помощью этого атрибута можно получить доступ ко всем атрибутам класса `MyWindow`. Например, в следующей строке производится изменение отступа:

```
self.myWidget.vbox.setMargin(0)
```

В следующих инструкциях внутри конструктора производится создание объекта кнопки и контейнера, затем экземпляр класса `MyWindow` и объект кнопки добавляются в контейнер, а далее контейнер добавляется в основное окно.

Инструкция

```
self.connect(self.button, QtCore.SIGNAL("clicked()"),
             self.on_clicked)
```

назначает обработчик нажатия кнопки. В третьем параметре указывается ссылка на метод `on_clicked()`, внутри которого производится изменение текста надписи (с помощью метода `setText()`) и кнопка делается неактивной (с помощью метода `setDisabled()`). Внутри метода `on_clicked()` доступен указатель `self`, через который можно получить доступ как к атрибутам класса `MyDialog`, так и к атрибутам класса `MyWindow`.

Таким образом производится повторное использование ранее написанного кода. Мы создаем класс и сохраняем его внутри отдельного модуля. Чтобы протестировать модуль или использовать его как отдельное приложение, размещаем код создания объекта приложения и объекта окна внутри условия:

```
if __name__ == "__main__":
```

При запуске с помощью двойного щелчка на значке файла производится выполнение кода как отдельного приложения. Если модуль импортируется, то создание объекта приложения не производится, и мы можем использовать класс в других приложениях. Например, т. к. это было сделано в листинге 19.4 или путем наследования класса и добавления или переопределения методов.

В некоторых случаях использование ООП-стиля является обязательным. Например, чтобы обработать нажатие клавиши на клавиатуре, необходимо наследовать какой-либо класс и переопределить в нем метод с предопределенным названием. Какие методы необходимо переопределять, мы рассмотрим при изучении обработки событий.

19.5. Создание окна с помощью программы Qt Designer

Если вы ранее пользовались Visual Studio или Delphi, то вспомните, что размещение компонентов на форме производили с помощью мыши. Щелкали левой кнопкой мыши на соответствующей кнопке на панели инструментов и перетаскивали компонент на форму. Далее с помощью инспектора свойств производили настройку значений некоторых свойств, а остальные свойства получали значения по умолчанию. При этом весь код генерировался автоматически. Произвести аналогичную операцию в PyQt позволяет программа Qt Designer, которая входит в состав установленных компонентов.

19.5.1. Создание формы

Для запуска программы Qt Designer в меню **Пуск** выбираем пункт **Программы | PyQt GPL v4.8.3 for Python v3.2 (x86) | Designer**. В результате откроется окно, изображенное на рис. 19.8. В окне **Новая форма** выделяем пункт **Widget** и нажимаем кнопку **Создать**. В результате откроется окно с пустой формой, на которую можно перетаскивать компоненты с панели **Панель виджетов** с помощью мыши.

В качестве примера добавим на форму надпись и кнопку. Для этого на панели **Панель виджетов** в группе **Display Widgets** щелкаем левой кнопкой мыши на пункте **Label** и, не

отпуская кнопку мыши, перетаскиваем компонент на форму. Далее проделываем аналогичную операцию с компонентом **Push Button** и размещаем его ниже надписи. Теперь выделяем одновременно надпись и кнопку, а затем щелкаем правой кнопкой мыши над любым компонентом и из контекстного меню выбираем пункт **Компоновка | Скомпоновать по вертикали**. Чтобы компоненты занимали всю область формы, щелкаем правой кнопкой мыши на свободном месте формы и из контекстного меню выбираем пункт **Компоновка | Скомпоновать по горизонтали**.

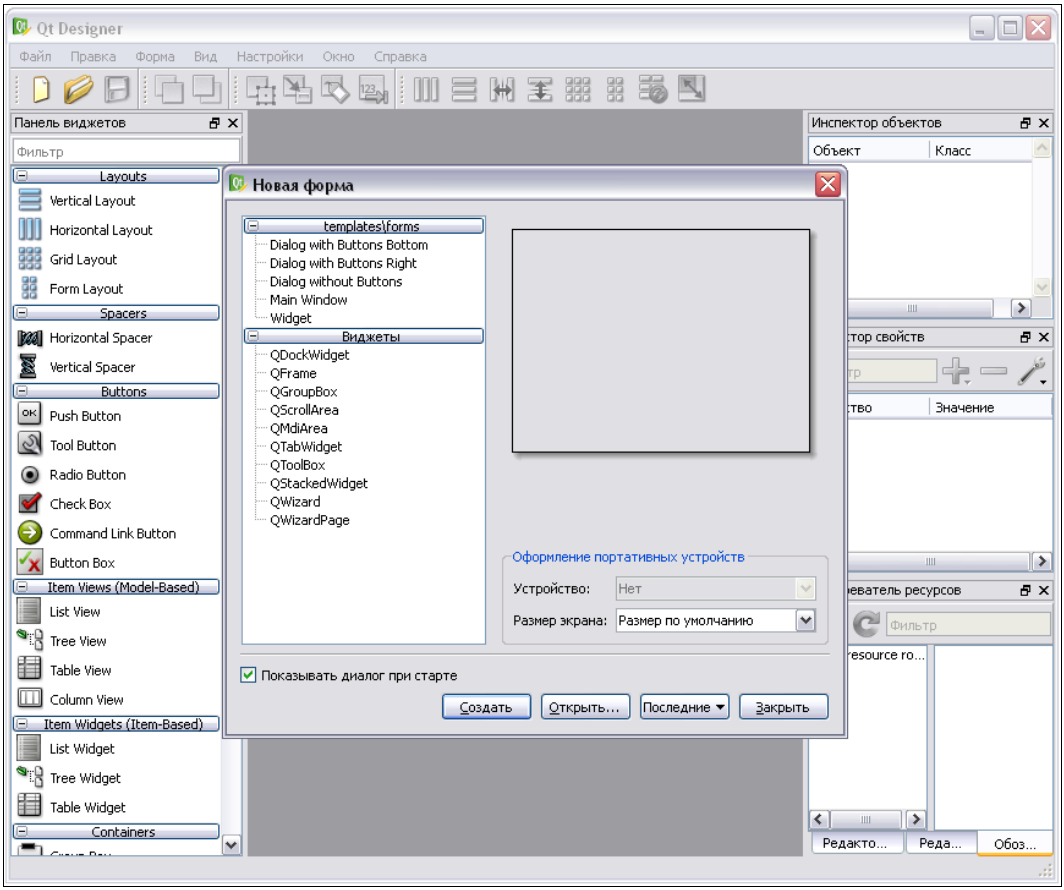

Рис. 19.8. Программа Qt Designer

Теперь изменим некоторые свойства окна. Для этого в окне **Инспектор объектов** (рис. 19.9) выделяем первый пункт и переходим в окно **Редактор свойств**. Находим свойство **objectName** и справа от свойства вводим значение `MyForm`. Далее находим свойство **geometry** и щелкаем мышью на плюсике слева, чтобы отобразить скрытые свойства. Задаем ширину равной `300`, а высоту равной `70` (рис. 19.10). Размеры формы автоматически изменятся. Указать текст, который будет отображаться в заголовке окна, позволяет свойство **windowTitle**.

Чтобы изменить свойства надписи, следует вначале выделить компонент с помощью мыши или выделить соответствующий пункт в окне **Инспектор объектов**. Изменяем значение свойства **objectName** на `label`, в свойстве **text** указываем текст надписи, а в свойстве

alignment задаем значение `AlignHCenter`. Теперь выделяем кнопку и изменяем значение свойства **objectName** на `btnQuit`, а в свойстве **text** указываем текст надписи. Также можно изменить текст надписи, дважды щелкнув мышью на компоненте.

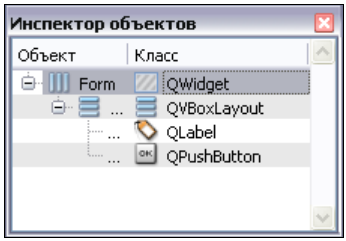

Рис. 19.9. Окно **Инспектор объектов**

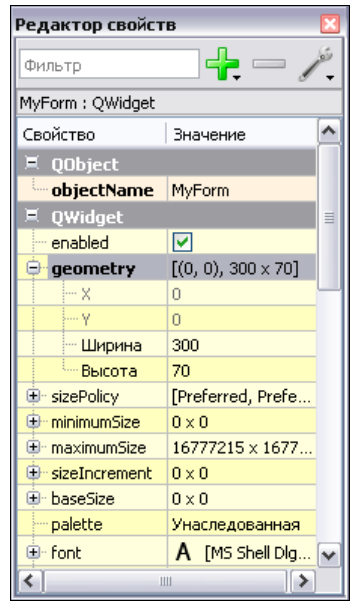

Рис. 19.10. Окно **Редактор свойств**

После окончания настройки формы и компонентов сохраняем форму в файл. Для этого в меню **Файл** выбираем пункт **Сохранить** и сохраняем файл под названием MyForm.ui. При необходимости внести какие-либо изменения этот файл можно открыть в программе Qt Designer, выбрав в меню **Файл** пункт **Открыть**.

19.5.2. Загрузка ui-файла в программе

Как вы можете убедиться, внутри ui-файла содержится текст в XML-формате, а не программа на языке Python. Следовательно, подключить файл с помощью инструкции `import` не получится. Чтобы использовать ui-файл внутри программы, следует воспользоваться модулем `uic`, который входит в состав библиотеки PyQt. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
from PyQt4 import uic
```

Для загрузки ui-файла предназначена функция `loadUi()`. Формат функции:

```
loadUi(<ui-файл>[, <Экземпляр класса>])
```

Если второй параметр не указан, то функция возвращает ссылку на объект формы. С помощью этой ссылки можно получить доступ к компонентам формы и, например, назначить обработчики сигналов (листинг 19.5). Названия компонентов задаются в программе Qt Designer в свойстве **objectName**.

Листинг 19.5. Использование функции `loadUi()`. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui, uic
import sys
app = QtGui.QApplication(sys.argv)
window = uic.loadUi("MyForm.ui")
QtCore.QObject.connect(window.btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp.quit)
window.show()
sys.exit(app.exec_())
```

Если во втором параметре указать ссылку на экземпляр класса, то все компоненты формы будут доступны через указатель `self` (листинг 19.6).

Листинг 19.6. Использование функции `loadUi()`. Вариант 2

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui, uic

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        uic.loadUi("MyForm.ui", self)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Загрузить ui-файл позволяет также функция `loadUiType()`. Функция возвращает кортеж из двух элементов: ссылки на класс формы и ссылки на базовый класс. Так как функция возвращает ссылку на класс, а не на экземпляр класса, мы можем создать множество экземпляров класса. После создания экземпляра класса формы необходимо вызвать метод `setupUi()` и передать ему указатель `self` (листинг 19.7).

Листинг 19.7. Использование функции `loadUiType()`. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui, uic

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        Form, Base = uic.loadUiType("MyForm.ui")
```

```
        self.ui = Form()
        self.ui.setupUi(self)
        self.connect(self.ui.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Загрузить ui-файл можно вне класса, а затем указать класс формы во втором параметре в списке наследования. В этом случае наш класс наследует все методы класса формы. В качестве иллюстрации изменим определение класса `MyWindow` из предыдущего примера (листинг 19.8).

Листинг 19.8. Использование функции `loadUiType()`. Вариант 2

```
Form, Base = uic.loadUiType("MyForm.ui")
class MyWindow(QtGui.QWidget, Form):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)
```

19.5.3. Преобразование ui-файла в py-файл

Вместо подключения ui-файла можно автоматически сгенерировать код на языке Python. Для преобразования ui-файла в py-файл предназначена утилита pyuic4.bat, расположенная в каталоге C:\Python32\Lib\site-packages\PyQt4\bin. Запускаем командную строку и переходим в каталог, в котором находится ui-файл. Для генерации программы выполняем команду:

```
pyuic4.bat MyForm.ui -o ui_MyForm.py
```

В результате будет создан файл ui_MyForm.py, который мы можем подключить с помощью инструкции `import`. Внутри файла находится класс `Ui_MyForm` с двумя методами: `setupUi()` и `retranslateUi()`. При использовании процедурного стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему ссылку на экземпляр окна (листинг 19.9).

Листинг 19.9. Использование класса формы. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys, ui_MyForm

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
```

```
ui = ui_MyForm.Ui_MyForm()
ui.setupUi(window)
QtCore.QObject.connect(ui.btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp.quit)
window.show()
sys.exit(app.exec_())
```

При использовании ООП-стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему указатель `self` (листинг 19.10).

Листинг 19.10. Использование класса формы. Вариант 2

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import ui_MyForm

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.ui = ui_MyForm.Ui_MyForm()
        self.ui.setupUi(self)
        self.connect(self.ui.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Класс формы можно указать во втором параметре в списке наследования. В этом случае наш класс наследует все методы класса формы. В качестве иллюстрации изменим определение класса `MyWindow` из предыдущего примера (листинг 19.11).

Листинг 19.11. Использование класса формы. Вариант 3

```
class MyWindow(QtGui.QWidget, ui_MyForm.Ui_MyForm):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)
```

Как видите, и в PyQt можно создавать формы, размещать компоненты с помощью мыши, а затем непосредственно подключать ui-файл в программе или автоматически генерировать код с помощью утилиты pyuic4.bat. Все это очень удобно. Тем не менее, чтобы полностью владеть технологией, необходимо уметь создавать код вручную. Поэтому в оставшейся части книги мы больше не будем рассматривать программу Qt Designer. Научиться "мышкотворчеству" вы сможете и самостоятельно без моей помощи.

19.6. Модули PyQt

В состав библиотеки PyQt входят следующие модули, объединенные в пакет `PyQt4`:

- `QtCore` — содержит классы, не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули;
- `QtGui` — содержит классы, реализующие компоненты графического интерфейса, например, надписи, кнопки, текстовые поля и др.;
- `QtSql` — включает поддержку баз данных SQLite, MySQL и др.;
- `QtSvg` — позволяет работать с векторной графикой (SVG);
- `QtOpenGL` — обеспечивает поддержку OpenGL;
- `QtNetwork` — содержит классы, предназначенные для работы с сетью;
- `QtWebKit` — предназначен для отображения HTML-документов;
- `phonon` и `QtMultimedia` — содержат поддержку мультимедиа;
- `QtXml` и `QtXmlPatterns` — предназначены для обработки XML;
- `QtScript` и `QtScriptTools` — содержат поддержку языка сценариев Qt Script;
- `Qt` — включает классы из всех модулей сразу.

Помимо перечисленных модулей в состав пакета `PyQt4` входят также модули `QtDeclarative`, `QAxContainer`, `QtTest`, `QtHelp` и `QtDesigner`.

Для подключения модулей используется следующий синтаксис:

```
from PyQt4 import <Названия модулей через запятую>
```

Пример подключения модулей `QtCore` и `QtGui`:

```
from PyQt4 import QtCore, QtGui
```

В этой книге мы будем рассматривать только модули `QtCore` и `QtGui`. Чтобы получить информацию по другим модулям, обращайтесь к документации.

19.7. Типы данных в PyQt

Библиотека PyQt является надстройкой над библиотекой Qt, написанной на языке C++. Библиотека Qt содержит множество классов, которые расширяют стандартные типы данных языка C++. Они реализуют динамические массивы, ассоциативные массивы, множества и др. Все эти классы очень помогают при программировании на языке C++, но для языка Python они не представляют особого интереса, т. к. весь этот функционал содержат стандартные типы данных. Тем не менее при чтении документации вы столкнетесь с этими типами, поэтому сейчас мы кратко рассмотрим основные типы:

- `QString` — Unicode-строка. В API версии 2 (используется по умолчанию) автоматически преобразуется в тип `str` и обратно. Класс доступен, только если используется API версии 1. Чтобы указать версию, следует в самом начале программы импортировать модуль `sip`, а затем вызвать функцию `setapi()`:

```
>>> import sip
>>> sip.setapi('QString', 1)
>>> from PyQt4 import QtCore
```

```
>>> s = QtCore.QString("строка")
>>> s
PyQt4.QtCore.QString('строка')
>>> str(s)
'строка'
```

- `QChar` — Unicode-символ. В API версии 2 (используется по умолчанию) автоматически преобразуется в тип `str`. Класс доступен, только если используется API версии 1. Пример:

```
>>> import sip
>>> sip.setapi('QString', 1)
>>> from PyQt4 import QtCore
>>> QtCore.QChar("a")
PyQt4.QtCore.QChar(0x0061)
```

- `QStringList` — массив Unicode-строк. В API версии 2 (используется по умолчанию) автоматически преобразуется в список. Класс доступен, только если используется API версии 1. Пример:

```
>>> import sip
>>> sip.setapi('QString', 1)
>>> from PyQt4 import QtCore
>>> L = QtCore.QStringList(["s1", "s2"])
>>> L
<PyQt4.QtCore.QStringList object at 0x00FD8618>
>>> list(L)
[PyQt4.QtCore.QString('s1'), PyQt4.QtCore.QString('s2')]
>>> [str(s) for s in list(L)]
['s1', 's2']
```

- `QByteArray` — массив байтов. Преобразуется в тип `bytes`. Пример:

```
>>> from PyQt4 import QtCore
>>> arr = QtCore.QByteArray(bytes("str", "cp1251"))
>>> arr
PyQt4.QtCore.QByteArray(b'str')
>>> bytes(arr)
b'str'
```

- `QVariant` — может хранить данные любого типа. Функциональность класса зависит от версии API. Если явно указана версия 1, то конструктор класса может принимать данные любого типа. Чтобы преобразовать данные, хранящиеся в экземпляре класса `QVariant`, в тип данных Python нужно вызвать метод `toPyObject()`. Чтобы создать пустой объект, следует после названия класса указать пустые скобки. Пример:

```
>>> import sip
>>> sip.setapi('QVariant', 1)
>>> from PyQt4 import QtCore
>>> n = QtCore.QVariant(10)
>>> n
<PyQt4.QtCore.QVariant object at 0x00FD8D50>
>>> n.toPyObject()
10
```

```
>>> QtCore.QVariant() # Пустой объект
<PyQt4.QtCore.QVariant object at 0x00FD8420>
```

В API версии 2 (версия используется по умолчанию) все преобразования производятся автоматически. Обратите внимание на то, что создать экземпляр класса `QVariant` в API версии 2 нельзя. Попытка создания экземпляра класса приведет к исключению `TypeError`:

```
>>> from PyQt4 import QtCore
>>> QtCore.QVariant(10) # Создать экземпляр класса нельзя
... Фрагмент опущен ...
TypeError: PyQt4.QtCore.QVariant represents a mapped type
and cannot be instantiated
```

Если метод ожидает данные типа `QVariant`, то можно передать данные любого типа. При получении данных производится автоматическое преобразование объекта в тип, поддерживаемый языком Python. Чтобы создать пустой объект, следует воспользоваться классом `QPyNullVariant`. Конструктор класса принимает название типа из библиотеки Qt в виде строки (например, `"QString"`) или тип данных в языке Python (например, `str`). Класс содержит метод `isNull()`, который всегда возвращает значение `True`. Получить название типа данных в виде строки позволяет метод `typeName()`. Пример:

```
>>> from PyQt4 import QtCore
>>> nullVariant = QtCore.QPyNullVariant(int)
>>> nullVariant.isNull(), nullVariant.typeName()
(True, 'int')
>>> nullVariant = QtCore.QPyNullVariant("QString")
>>> nullVariant.isNull(), nullVariant.typeName()
(True, 'QString')
```

От версии API зависят также классы `QDate` (представление даты), `QTime` (представление времени), `QDateTime` (представление даты и времени), `QTextStream` (текстовый поток) и `QUrl` (URL-адрес).

19.8. Управление основным циклом приложения

Для взаимодействия с системой и запуска обработчиков сигналов предназначен основной цикл приложения. После вызова метода `exec_()` программа переходит в бесконечный цикл. Инструкции, расположенные после вызова этого метода, будут выполнены только после завершения работы всего приложения. Чтобы завершить работу приложения, необходимо вызвать слот `quit()` или метод `exit([returnCode=0])`. Так как программа находится внутри цикла, то вызвать эти методы можно только при наступлении какого-либо события, например при нажатии пользователем кнопки. Цикл автоматически прерывается при нажатии пользователем кнопки **Закрыть** в заголовке последнего окна. С помощью статического метода `setQuitOnLastWindowClosed()` из класса `QApplication` это поведение можно изменить.

После наступления события цикл прерывается и управление передается в обработчик. После завершения работы обработчика управление опять передается основному циклу приложения. Если внутри обработчика выполняется длительная операция, то программа перестает реагировать на события. В качестве примера изобразим длительный процесс с помощью функции `sleep()` из модуля `time` (листинг 19.12).

Листинг 19.12. Выполнение длительной операции

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys, time

def on_clicked():
    time.sleep(10) # "Засыпаем" на 10 секунд

app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Запустить процесс")
button.resize(200, 40)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере при нажатии кнопки **Запустить процесс** вызывается функция `on_clicked()`, внутри которой мы "засыпаем" на десять секунд и тем самым прерываем основной цикл приложения. Попробуйте нажать кнопку, перекрыть окно другим окном, а затем заново его отобразить. В результате окно примет вид, изображенный на рис. 19.11. Таким образом, окно перестает реагировать на любые события и после перекрытия другим окном приложение не может выполнить перерисовку окна, пока не закончится выполнение процесса. Если в этот момент нажать кнопку **Закрыть** в заголовке окна, то будет выведено окно **Завершение программы**, с помощью которого система сообщает, что программа не отвечает и предлагает завершить ее.

Рис. 19.11. Вид окна при выполнении длительной операции

При выполнении длительной операции ее можно разбить на несколько этапов и периодически запускать оборот основного цикла с помощью статического метода `processEvents([flags=AllEvents])` из класса `QCoreApplication`. Переделаем предыдущую программу и с помощью цикла инсценируем длительную операцию, которая выполняется 20 секунд (листинг 19.13).

Листинг 19.13. Использование метода `processEvents()`

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys, time

def on_clicked():
    button.setDisabled(True)        # Делаем кнопку неактивной
    for i in range(1, 21):
        QtGui.qApp.processEvents()  # Запускаем оборот цикла
```

```
        time.sleep(1)               # "Засыпаем" на 1 секунду
        print("step -", i)
    button.setDisabled(False)       # Делаем кнопку активной

app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Запустить процесс")
button.resize(200, 40)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере длительная операция разбита на одинаковые этапы. После выполнения каждого этапа запускается оборот основного цикла приложения. Теперь при перекрытии окна и повторном его отображении окно будет перерисовано. Таким образом, приложение по-прежнему будет взаимодействовать с системой, хотя и с некоторой задержкой.

19.9. Многопоточные приложения

При обработке больших объемов данных не всегда можно равномерно разбить операцию на небольшие по времени этапы. Поэтому при использовании метода `processEvents()` возможны проблемы. Вместо использования метода `processEvents()` лучше вынести длительную операцию в отдельный поток. В этом случае длительная операция будет выполняться параллельно с основным циклом приложения и не будет его блокировать. В одном процессе можно запустить сразу несколько независимых потоков. Если процессор является многоядерным, то потоки будут равномерно распределены по ядрам. За счет этого можно не только избежать блокировки GUI-потока, но и значительно увеличить эффективность выполнения программы. Завершение основного цикла приложения приводит к завершению работы всех потоков.

19.9.1. Класс *QThread*. Создание потока

Для создания потока в PyQt предназначен класс `QThread`, который наследует класс `QObject`. Конструктор класса `QThread` имеет следующий формат:

```
<Объект> = QThread([parent=None])
```

Чтобы использовать потоки, следует создать класс, который будет наследником класса `QThread`, и определить в нем метод `run()`. Код, расположенный в методе `run()`, будет выполняться в отдельном потоке. После завершения выполнения метода `run()` поток прекратит свое существование. Далее нужно создать экземпляр класса и вызвать метод `start()`, который после запуска потока вызовет метод `run()`. Обратите внимание на то, что если напрямую вызвать метод `run()`, то код будет выполняться в GUI-потоке, а не в отдельном потоке. Поэтому, чтобы запустить поток, необходимо вызвать именно метод `start()`, а не метод `run()`. Метод `start()` имеет следующий формат:

```
start([priority=QThread.InheritPriority])
```

Параметр `priority` задает приоритет выполнения потока по отношению к другим потокам. Следует учитывать, что при наличии потока с самым высоким приоритетом поток с самым низким приоритетом может быть просто проигнорирован в некоторых операционных сис-

темах. Перечислим допустимые значения параметра (в порядке увеличения приоритета) и соответствующие им атрибуты из класса `QThread`:

- `0` — `IdlePriority` — самый низкий приоритет;
- `1` — `LowestPriority`;
- `2` — `LowPriority`;
- `3` — `NormalPriority`;
- `4` — `HighPriority`;
- `5` — `HighestPriority`;
- `6` — `TimeCriticalPriority` — самый высокий приоритет;
- `7` — `InheritPriority` — автоматический выбор приоритета (значение используется по умолчанию).

Задать приоритет потока позволяет также метод `setPriority(<Приоритет>)`. Узнать какой приоритет использует запущенный поток можно с помощью метода `priority()`.

После запуска потока генерируется сигнал `started()`, а после завершения — сигнал `finished()`. Назначив обработчики этим сигналам, можно контролировать статус потока из основного цикла приложения. Если необходимо узнать текущий статус, то следует воспользоваться методами `isRunning()` и `isFinished()`. Метод `isRunning()` возвращает значение `True`, если поток выполняется, а метод `isFinished()` возвращает значение `True`, если поток закончил выполнение.

Потоки выполняются внутри одного процесса и имеют доступ ко всем глобальным переменным. Однако следует учитывать, что из потока нельзя изменять что-либо в GUI-потоке, например, выводить текст на надпись. Для изменения данных в GUI-потоке нужно использовать сигналы. Внутри потока с помощью метода `emit()` производим генерацию сигнала и передаем данные. В первом параметре метод принимает объект сигнала, а в остальных параметрах данные, которые будут переданы. Внутри основного потока назначаем обработчик этого сигнала и в обработчике изменяем что-либо в GUI-потоке.

Рассмотрим использование класса `QThread` на примере (листинг 19.14).

Листинг 19.14. Использование класса `QThread`

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyThread(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        for i in range(1, 21):
            self.sleep(3)              # "Засыпаем" на 3 секунды
            # Передача данных из потока через сигнал
            self.emit(QtCore.SIGNAL("mysignal(QString)"), "i = %s" % i)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
```

```
        self.label = QtGui.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button = QtGui.QPushButton("Запустить процесс")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.mythread = MyThread()      # Создаем экземпляр класса
        self.connect(self.button, QtCore.SIGNAL("clicked()"),
                     self.on_clicked)
        self.connect(self.mythread, QtCore.SIGNAL("started()"),
                     self.on_started)
        self.connect(self.mythread, QtCore.SIGNAL("finished()"),
                     self.on_finished)
        self.connect(self.mythread, QtCore.SIGNAL("mysignal(QString)"),
                     self.on_change, QtCore.Qt.QueuedConnection)
    def on_clicked(self):
        self.button.setDisabled(True)  # Делаем кнопку неактивной
        self.mythread.start()          # Запускаем поток
    def on_started(self):              # Вызывается при запуске потока
        self.label.setText("Вызван метод on_started()")
    def on_finished(self):          # Вызывается при завершении потока
        self.label.setText("Вызван метод on_finished()")
        self.button.setDisabled(False) # Делаем кнопку активной
    def on_change(self, s):
        self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QThread")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())
```

Итак, мы создали класс `MyThread`, который является наследником класса `QThread`, и определили метод `run()`. Внутри метода `run()` производится имитация процесса с помощью цикла `for` и метода `sleep()`. Каждые три секунды выполняется генерация сигнала `mysignal(QString)` и передача текущего значения переменной `i`. Внутри конструктора класса `MyWindow` мы назначили обработчик этого сигнала с помощью инструкции:

```
self.connect(self.mythread, QtCore.SIGNAL("mysignal(QString)"),
             self.on_change, QtCore.Qt.QueuedConnection)
```

В первом параметре указана ссылка на экземпляр класса `MyThread` (экземпляр создается в конструкторе класса `MyWindow` и сохраняется в атрибуте `mythread`), а во втором параметре — объект сигнала. Обратите внимание на то, что название сигнала в методе `connect()` полностью совпадает с названием сигнала в методе `emit()`. В третьем параметре указана ссылка на метод, который будет вызван при наступлении события. Этот метод принимает один па-

раметр и производит изменение текста надписи с помощью метода `setText()`. В четвертом параметре метода `connect()` с помощью атрибута `QueuedConnection` указывается, что сигнал помещается в очередь обработки событий и обработчик должен выполняться в потоке приемника сигнала, т. е. в GUI-потоке. Из GUI-потока мы можем смело изменять свойства компонентов интерфейса.

Внутри конструктора класса `MyWindow` производится создание надписи и кнопки, а затем размещение их внутри вертикального контейнера. Далее выполняется создание экземпляра класса `MyThread` и сохранение его в атрибуте `mythread`. С помощью этого атрибута мы можем управлять потоком и назначить обработчики сигналов `started()`, `finished()` и `mysignal(QString)`. Запуск потока производится с помощью метода `start()` внутри обработчика нажатия кнопки. Чтобы исключить повторный запуск потока, мы делаем кнопку неактивной с помощью метода `setDisabled()`. После окончания работы потока внутри обработчика сигнала `finished()` делаем кнопку опять активной.

Обратите внимание на то, что для имитации длительного процесса мы использовали статический метод `sleep()` из класса `QThread`, а не функцию `sleep()` из модуля `time`. Временно прервать выполнение потока позволяют следующие статические методы из класса `QThread`:

- `sleep()` — продолжительность задается в секундах:

```
QtCore.QThread.sleep(3)         # "Засыпаем" на 3 секунды
```

- `msleep()` — продолжительность задается в миллисекундах:

```
QtCore.QThread.msleep(3000)     # "Засыпаем" на 3 секунды
```

- `usleep()` — продолжительность задается в микросекундах:

```
QtCore.QThread.usleep(3000000) # "Засыпаем" на 3 секунды
```

19.9.2. Управление циклом внутри потока

Очень часто внутри потока одни и те же инструкции выполняются многократно. Например, при осуществлении мониторинга серверов в Интернете на каждой итерации цикла посылается запрос к одному серверу. В этом случае используется бесконечный цикл внутри метода `run()`. Выход из цикла производится после окончания опроса всех серверов. В некоторых случаях этот цикл необходимо прервать преждевременно при нажатии кнопки пользователем. Чтобы это стало возможным, в классе, реализующем поток, следует создать атрибут, который будет содержать флаг текущего состояния. Далее на каждой итерации цикла проверяем состояние флага. При изменении состояния прерываем выполнение цикла. Чтобы изменить значение атрибута, создаем обработчик и связываем его с сигналом `clicked()` соответствующей кнопки. При нажатии кнопки внутри обработчика производим изменение значения атрибута. Пример запуска и остановки потока с помощью кнопок приведен в листинге 19.15.

Листинг 19.15. Запуск и остановка потока

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyThread(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
```

```
        self.running = False   # Флаг выполнения
        self.count = 0
    def run(self):
        self.running = True
        while self.running:    # Проверяем значение флага
            self.count += 1
            self.emit(QtCore.SIGNAL("mysignal(QString)"),
                      "count = %s" % self.count)
            self.sleep(1)      # Имитируем процесс

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.btnStart = QtGui.QPushButton("Запустить поток")
        self.btnStop = QtGui.QPushButton("Остановить поток")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnStart)
        self.vbox.addWidget(self.btnStop)
        self.setLayout(self.vbox)
        self.mythread = MyThread()
        self.connect(self.btnStart, QtCore.SIGNAL("clicked()"),
                     self.on_start)
        self.connect(self.btnStop, QtCore.SIGNAL("clicked()"),
                     self.on_stop)
        self.connect(self.mythread, QtCore.SIGNAL("mysignal(QString)"),
                     self.on_change, QtCore.Qt.QueuedConnection)
    def on_start(self):
        if not self.mythread.isRunning():
            self.mythread.start()      # Запускаем поток
    def on_stop(self):
        self.mythread.running = False  # Изменяем флаг выполнения
    def on_change(self, s):
        self.label.setText(s)
    def closeEvent(self, event):       # Вызывается при закрытии окна
        self.hide()                    # Скрываем окно
        self.mythread.running = False  # Изменяем флаг выполнения
        self.mythread.wait(5000)       # Даем время, чтобы закончить
        event.accept()                 # Закрываем окно

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Запуск и остановка потока")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

В этом примере внутри конструктора класса MyThread создается атрибут running и ему присваивается значение False. При запуске потока внутри метода run() значение атрибута изменяется на True. Затем запускается цикл, в котором атрибут указывается в качестве условия. Как только значение атрибута станет равным значению False, цикл будет остановлен.

Внутри конструктора класса MyWindow производится создание надписи, двух кнопок и экземпляра класса MyThread. Далее назначаются обработчики сигналов. При нажатии кнопки **Запустить поток** запустится метод on_start(), внутри которого с помощью метода isRunning() производится проверка текущего статуса потока. Если поток не запущен, то выполняется запуск потока с помощью метода start(). При нажатии кнопки **Остановить поток** запустится метод on_stop(), внутри которого атрибуту running присваивается значение False. Это значение является условием выхода из цикла внутри метода run().

Путем изменения значения атрибута можно прервать выполнение цикла только в том случае, если закончится выполнение очередной итерации. Если поток длительное время ожидает какое-либо событие, например ответ сервера, то можно так и не дождаться завершения потока. Чтобы прервать выполнение потока в любом случае, следует воспользоваться методом terminate(). Этим методом можно пользоваться только в крайнем случае, т. к. прерывание производится в любой части кода. При этом блокировки автоматически не снимаются. Кроме того, можно повредить данные, над которыми производились операции в момент прерывания. После вызова метода terminate() следует вызвать метод wait().

При закрытии окна приложение завершает работу. Завершение основного цикла приложения приводит к завершению всех потоков. Чтобы предотвратить повреждение данных, мы перехватываем событие закрытия окна и дожидаемся окончания выполнения. Чтобы перехватить событие, необходимо внутри класса создать метод с предопределенным названием, в нашем случае с названием closeEvent(). Этот метод будет автоматически вызван при попытке закрыть окно. В качестве параметра метод принимает объект события event. Через этот объект можно получить дополнительную информацию о событии. Чтобы закрыть окно внутри метода closeEvent(), следует вызвать метод accept() объекта события. Если необходимо предотвратить закрытие окна, то следует вызвать метод ignore().

Внутри метода closeEvent() мы присваиваем атрибуту running значение False. Далее с помощью метода wait() даем возможность потоку нормально завершить работу. В качестве параметра метод wait() принимает количество миллисекунд, по истечении которых управление будет передано следующей инструкции. Необходимо учитывать, что это максимальное время. Если поток закончит работу раньше, то и метод закончит выполнение раньше. Метод wait() возвращает значение True, если поток успешно завершил работу, и False — в противном случае. Ожидание завершения потока занимает некоторое время, в течение которого окно будет по-прежнему видимым. Чтобы не вводить пользователя в заблуждение в самом начале метода closeEvent(), мы скрываем окно с помощью метода hide().

Каждый поток может иметь собственный цикл обработки сигналов, который запускается с помощью метода exec_(). В этом случае потоки могут обмениваться сигналами между собой. Чтобы прервать цикл, следует вызвать слот quit() или метод exit([returnCode=0]). Рассмотрим обмен сигналами между потоками на примере (листинг 19.16).

Листинг 19.16. Обмен сигналами между потоками

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
```

```
class Thread1(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.count = 0
    def run(self):
        self.exec_()          # Запускаем цикл обработки сигналов
    def on_start(self):
        self.count += 1
        self.emit(QtCore.SIGNAL("s1(int)"), self.count)

class Thread2(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        self.exec_()          # Запускаем цикл обработки сигналов
    def on_change(self, i):
        i += 10
        self.emit(QtCore.SIGNAL("s2(const QString&)"), "%d" % i)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Нажмите кнопку")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button = QtGui.QPushButton("Сгенерировать сигнал")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.thread1 = Thread1()
        self.thread2 = Thread2()
        self.thread1.start()
        self.thread2.start()
        self.connect(self.button, QtCore.SIGNAL("clicked()"),
                     self.thread1.on_start)
        self.connect(self.thread1, QtCore.SIGNAL("s1(int)"),
                     self.thread2.on_change)
        self.connect(self.thread2, QtCore.SIGNAL("s2(const QString&)"),
                     self.label, QtCore.SLOT("setText(const QString&)"))

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Обмен сигналами между потоками")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())
```

В этом примере мы создали классы `Thread1`, `Thread2` и `MyWindow`. Первые два класса предназначены для создания потоков. Внутри этих классов в методе `run()` вызывается метод `exec_()`, который запускает цикл обработки событий. Внутри конструктора класса `MyWindow` производится создание надписи, кнопки и экземпляров классов `Thread1` и `Thread2`. Далее выполняется запуск сразу двух потоков.

В следующей инструкции сигнал нажатия кнопки соединяется с методом `on_start()` первого потока. Внутри этого метода производится какая-либо операция (в нашем случае увеличение значения атрибута `count`), а затем с помощью метода `emit()` генерируется сигнал `s1(int)` и во втором параметре передается результат выполнения метода. Сигнал `s1(int)` соединен с методом `on_change()` второго потока. Внутри этого метода также производится какая-либо операция, а затем генерируется сигнал `s2(const QString&)` и передается результат выполнения метода. В свою очередь сигнал `s2(const QString&)` соединен со слотом `setText(const QString&)` объекта надписи. Таким образом, при нажатии кнопки **Сгенерировать сигнал** вначале будет вызван метод `on_start()` из класса `Thread1`, затем метод `on_change()` из класса `Thread2`, а потом результат выполнения отобразится внутри надписи в окне.

19.9.3. Модуль *queue*. Создание очереди заданий

В предыдущем разделе мы рассмотрели возможность обмена сигналами между потоками. Теперь предположим, что запущены десять потоков, которые ожидают задания в бесконечном цикле. Как передать задание одному потоку, а не всем сразу? И как определить, какому потоку передать задание? Можно, конечно, создать список в глобальном пространстве имен и добавлять задания в этот список. Но в этом случае придется решать вопрос о совместном использовании одного ресурса сразу десятью потоками. Ведь если потоки будут получать задания одновременно, то одно задание могут получить сразу несколько потоков, а какому-либо потоку не хватит заданий и возникнет исключительная ситуация. Говоря простым языком, возникает ситуация, когда вы пытаетесь сесть на стул, а другой человек одновременно пытается вытащить этот стул из-под вас. Думаете, что успеете сесть?

Модуль `queue`, входящий в состав стандартной библиотеки Python, позволяет решить эту проблему. Модуль содержит несколько классов, которые реализуют разного рода потокобезопасные очереди. Перечислим эти классы:

- `Queue` — очередь (первым пришел, первым вышел). Формат конструктора:

  ```
  <Объект> = Queue([maxsize=0])
  ```

  Пример:

  ```
  >>> import queue
  >>> q = queue.Queue()
  >>> q.put_nowait("elem1")
  >>> q.put_nowait("elem2")
  >>> q.get_nowait()
  'elem1'
  >>> q.get_nowait()
  'elem2'
  ```

- `LifoQueue` — стек (последним пришел, первым вышел). Формат конструктора:

  ```
  <Объект> = LifoQueue([maxsize=0])
  ```

Пример:

```
>>> q = queue.LifoQueue()
>>> q.put_nowait("elem1")
>>> q.put_nowait("elem2")
>>> q.get_nowait()
'elem2'
>>> q.get_nowait()
'elem1'
```

- `PriorityQueue` — очередь с приоритетами. При получении значения возвращается элемент с наивысшим приоритетом (наименьшим значением в первом параметре кортежа). Элементы очереди должны быть кортежами, в которых первым элементом является число, означающее приоритет, а вторым — значение элемента. Формат конструктора класса:

```
<Объект> = PriorityQueue([maxsize=0])
```

Пример:

```
>>> q = queue.PriorityQueue()
>>> q.put_nowait((10, "elem1"))
>>> q.put_nowait((3, "elem2"))
>>> q.put_nowait((12, "elem3"))
>>> q.get_nowait()
(3, 'elem2')
>>> q.get_nowait()
(10, 'elem1')
>>> q.get_nowait()
(12, 'elem3')
```

Параметр `maxsize` задает максимальное количество элементов, которое может содержать очередь. Если параметр равен нулю (значение по умолчанию) или отрицательному значению, то размер очереди неограничен.

Экземпляры этих классов содержат следующие методы:

- `put(<Элемент>[, block=True][, timeout=None])` — добавляет элемент в очередь. Если в параметре `block` указано значение `True`, то поток будет ожидать возможности добавления элемента. В параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось добавить, то возбуждается исключение `queue.Full`;
- `put_nowait(<Элемент>)` — добавление элемента без ожидания. Эквивалентно:

```
put(<Элемент>, False)
```

- `get([block=True][, timeout=None])` — удаляет и возвращает элемент из очереди. Если в параметре `block` указано значение `True`, то поток будет ожидать возможности получения элемента. В параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось получить, то возбуждается исключение `queue.Empty`;
- `get_nowait()` — эквивалентно инструкции `get(False)`;
- `join()` — блокирует поток, пока не будут обработаны все задания в очереди. Другие потоки после обработки текущего задания должны вызывать метод `task_done()`. Как только все задания будут обработаны, поток будет разблокирован;
- `task_done()` — этот метод должны вызывать потоки после обработки задания;

- `qsize()` — возвращает приблизительное количество элементов в очереди. Так как доступ к очереди имеют сразу несколько потоков, доверять этому значению не следует. В любой момент времени количество элементов может измениться;
- `empty()` — возвращает `True`, если очередь пуста, и `False` — в противном случае;
- `full()` — возвращает `True`, если очередь содержит элементы, и `False` — в противном случае.

Рассмотрим использование очереди в многопоточном приложении на примере (листинг 19.17).

Листинг 19.17. Использование модуля queue

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import queue

class MyThread(QtCore.QThread):
    def __init__(self, id, queue, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
        self.queue = queue
    def run(self):
        while True:
            task = self.queue.get()       # Получаем задание
            self.sleep(5)                 # Имитируем обработку
            self.emit(QtCore.SIGNAL("taskDone(int, int)"),
                      task, self.id)      # Передаем данные обратно
            self.queue.task_done()

class MyWindow(QtGui.QPushButton):
    def __init__(self):
        QtGui.QPushButton.__init__(self)
        self.setText("Раздать задания")
        self.queue = queue.Queue()        # Создаем очередь
        self.threads = []
        for i in range(1, 3):             # Создаем потоки и запускаем
            thread = MyThread(i, self.queue)
            self.threads.append(thread)
            self.connect(thread, QtCore.SIGNAL("taskDone(int, int)"),
                         self.on_task_done, QtCore.Qt.QueuedConnection)
            thread.start()
        self.connect(self, QtCore.SIGNAL("clicked()"),
                     self.on_add_task)
    def on_add_task(self):
        for i in range(0, 11):
            self.queue.put(i)             # Добавляем задания в очередь
    def on_task_done(self, data, id):
        print(data, "- id =", id)         # Выводим обработанные данные
```

```
if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование модуля queue")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())
```

В этом примере конструктор класса `MyThread` принимает уникальный идентификатор (`id`) и ссылку на очередь (`queue`), которые сохраняются в одноименных атрибутах класса. В методе `run()` внутри бесконечного цикла производится получение элемента из очереди с помощью метода `get()`. Если очередь пуста, то поток будет ожидать, пока не появится хотя бы один элемент. Далее производится обработка задания (в нашем случае просто задержка), а затем обработанные данные передаются главному потоку через сигнал `taskDone(int, int)`. В следующей инструкции с помощью метода `task_done()` указывается, что задание было обработано.

Главный поток реализуется с помощью класса `MyWindow`. Обратите внимание на то, что наследуется класс `QPushButton` (класс кнопки), а не класс `QWidget`. Все визуальные компоненты являются наследниками класса `QWidget`, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В нашем случае используется только кнопка, поэтому можно сразу наследовать класс `QPushButton`.

Внутри конструктора класса `MyWindow` с помощью метода `setText()` задается текст надписи на кнопке. Далее создается экземпляр класса `Queue` и сохраняется в атрибуте `queue`. В следующей инструкции производится создание списка, в котором будут храниться ссылки на объекты потоков. Сами объекты потоков создаются внутри цикла (в нашем случае создаются два объекта) и добавляются в список. Внутри цикла производится также назначение обработчика сигнала `taskDone(int, int)` и запуск потока с помощью метода `start()`. Далее назначается обработчик нажатия кнопки.

При нажатии кнопки **Раздать задания** вызывается метод `on_add_task()`, внутри которого производится добавление заданий в конец очереди. После добавления заданий потоки выходят из цикла ожидания и каждый из них получает одно уникальное задание. После окончания обработки потоки генерируют сигнал `taskDone(int, int)` и вызывают метод `task_done()`, информирующий об окончании обработки задания. Главный поток получает сигнал и вызывает метод `on_task_done()`, внутри которого через параметры будут доступны обработанные данные. Так как метод расположен в GUI-потоке, мы можем изменять свойства компонентов и, например, добавить результат в список или таблицу. В нашем примере результат просто выводится в окно консоли. Чтобы увидеть сообщения, следует сохранить файл с расширением py, а не pyw. После окончания обработки задания потоки снова получают задания. Если очередь будет пуста, то потоки перейдут в режим ожидания заданий.

19.9.4. Классы *QMutex* и *QMutexLocker*

Как вы уже знаете, совместное использование одного ресурса сразу несколькими потоками может привести к непредсказуемому поведению программы. Следовательно, внутри программы необходимо предусмотреть возможность блокировки ресурса одним потоком и ожидание разблокировки другим потоком. В один момент времени только один поток должен иметь доступ к ресурсу.

Реализовать блокировку ресурса в PyQt позволяют классы `QMutex` и `QMutexLocker` из модуля `QtCore`. Конструктор класса `QMutex` имеет следующий формат:

```
<Объект> = QMutex([mode=QtCore.QMutex.NonRecursive])
```

Необязательный параметр `mode` может принимать значения `NonRecursive` (поток может запросить блокировку только единожды; после снятия блокировка может быть запрошена снова; значение по умолчанию) и `Recursive` (поток может запросить блокировку несколько раз; чтобы полностью снять блокировку, следует вызвать метод `unlock()` соответствующее количество раз). Конструктор возвращает объект, через который доступны следующие методы:

- `lock()` — устанавливает блокировку. Если ресурс был заблокирован другим потоком, то работа текущего потока приостанавливается до снятия блокировки;
- `tryLock([<Время ожидания>])` — устанавливает блокировку. Если блокировка была успешно установлена, то метод возвращает значение `True`. Если ресурс заблокирован другим потоком, то метод возвращает значение `False` без ожидания возможности установить блокировку. Максимальное время ожидания в миллисекундах можно указать в качестве необязательного параметра. Если в параметре указано отрицательное значение, то метод `tryLock()` аналогичен методу `lock()`;
- `unlock()` — снимает блокировку.

Рассмотрим использование класса `QMutex` на примере (листинг 19.18).

Листинг 19.18. Использование класса `QMutex`

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyThread(QtCore.QThread):
    x = 10                                  # Атрибут класса
    mutex = QtCore.QMutex()                 # Мьютекс
    def __init__(self, id, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
    def run(self):
        self.change_x()
    def change_x(self):
        MyThread.mutex.lock()               # Блокируем
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        MyThread.mutex.unlock()             # Снимаем блокировку

class MyWindow(QtGui.QPushButton):
    def __init__(self):
        QtGui.QPushButton.__init__(self)
        self.setText("Запустить")
```

```
        self.thread1 = MyThread(1)
        self.thread2 = MyThread(2)
        self.connect(self, QtCore.SIGNAL("clicked()"), self.on_start)
    def on_start(self):
        if not self.thread1.isRunning(): self.thread1.start()
        if not self.thread2.isRunning(): self.thread2.start()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QMutex")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())
```

В этом примере внутри класса `MyThread` мы создали атрибут `x`, который доступен всем экземплярам класса. Изменение значения атрибута в одном потоке повлечет изменение значения и в другом потоке. Если потоки будут изменять значение одновременно, то предсказать текущее значение атрибута становится невозможно. Следовательно, изменять значение можно только после установки блокировки.

Чтобы обеспечить блокировку, внутри класса `MyThread` создается экземпляр класса `QMutex` и сохраняется в атрибуте `mutex`. Обратите внимание на то, что сохранение производится в атрибуте класса, а не в атрибуте экземпляра класса. Чтобы блокировка сработала, необходимо, чтобы защищаемый атрибут и мьютекс находились в одной области видимости. Далее все содержимое метода `change_x()`, в котором производится изменение атрибута `x`, указывается между вызовами методов `lock()` и `unlock()`. Таким образом гарантируется, что все инструкции внутри метода `change_x()` будут выполнены сначала одним потоком и только потом другим потоком.

Внутри конструктора класса `MyWindow` производится создание двух объектов класса `MyThread` и назначение обработчика нажатия кнопки. После нажатия кнопки **Запустить** будет вызван метод `on_start()`, внутри которого производится запуск сразу двух потоков одновременно, при условии, что потоки не были запущены ранее. В результате мы получим следующий результат в окне консоли:

```
x = 10 id = 1
x = 15 id = 1
x = 49 id = 1
x = 49 id = 2
x = 54 id = 2
x = 88 id = 2
```

Как видно из результата, сначала изменение атрибута производил поток с идентификатором `1`, а лишь затем поток с идентификатором `2`. Если блокировку не указать, то результат будет другим:

```
x = 10 id = 1
x = 15 id = 2
x = 20 id = 1
x = 54 id = 1
```

```
x = 54 id = 2
x = 88 id = 2
```

В этом случае поток с идентификатором 2 изменил значение атрибута x до окончания выполнения метода change_x() в потоке с идентификатором 1.

При возникновении исключения внутри метода change_x() ресурс останется заблокированным, т. к. поток управления не дойдет до вызова метода unlock(). Кроме того, можно забыть вызвать метод unlock() по случайности, что также приведет к вечной блокировке. Исключить подобную ситуацию позволяет класс QMutexLocker. Конструктор этого класса принимает объект мьютекса и устанавливает блокировку. После выхода из области видимости будет вызван деструктор класса, внутри которого блокировка автоматически снимается. Следовательно, если создать экземпляр класса QMutexLocker в начале метода, то после выхода из метода блокировка автоматически снимется. Переделаем метод change_x() из класса MyThread и используем класс QMutexLocker (листинг 19.19).

Листинг 19.19. Использование класса QMutexLocker

```
    def change_x(self):
        ml = QtCore.QMutexLocker(MyThread.mutex)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Блокировка автоматически снимется
```

При использовании класса QMutexLocker следует помнить о разнице между областями видимости в языках C++ и Python. В языке C++ область видимости ограничена блоком. Блоком может быть как функция, так и просто область, ограниченная фигурными скобками. Таким образом, если переменная объявлена внутри блока условного оператора, например, if, то при выходе из этого блока переменная уже не будет видна:

```
if (условие) {
    int x = 10; // Объявляем переменную
    // ...
}
// Здесь переменная x уже не видна!
```

В языке Python область видимости гораздо шире. Если мы объявим переменную внутри условного оператора, то она будет видна и после выхода из этого блока:

```
if условие:
    x = 10      # Объявляем переменную
    # ...
# Здесь переменная x видна
```

Таким образом, область видимости локальной переменной в языке Python ограничена функцией, а не любым блоком. Класс QMutexLocker в PyQt поддерживает протокол менеджеров контекста, который позволяет ограничить область видимости блоком инструкции with...as. Этот протокол гарантирует снятие блокировки, даже при наличии исключения

внутри инструкции `with...as`. Переделаем метод `change_x()` из класса `MyThread` еще раз и используем инструкцию `with...as` (листинг 19.20).

Листинг 19.20. Использование инструкции `with...as`

```
def change_x(self):
    with QtCore.QMutexLocker(MyThread.mutex):
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Блокировка автоматически снимется
```

Теперь, когда вы уже знаете о возможности блокировки ресурса, следует сделать несколько замечаний. Установка и снятие блокировки занимают некоторый промежуток времени, тем самым снижая эффективность всей программы. Поэтому встроенные типы данных не обеспечивают безопасную работу в многопоточном приложении. Прежде чем использовать блокировки, подумайте, может быть в вашем приложении они и не нужны. Второе замечание относится к доступу к защищенному ресурсу из GUI-потока. Ожидание снятия блокировки может заблокировать GUI-поток, и приложение перестанет реагировать на события. Поэтому в данном случае следует использовать сигналы, а не прямой доступ. И последнее замечание относится к *взаимной блокировке*. Если первый поток, владея ресурсом `A`, захочет получить доступ к ресурсу `B`, а второй поток, владея ресурсом `B`, захочет получить доступ к ресурсу `A`, то оба потока будут ждать снятия блокировки вечно. В этой ситуации следует предусмотреть возможность временного освобождения ресурсов одним из потоков после превышения периода ожидания.

Примечание

Для синхронизации и координации потоков предназначены также классы `QSemaphore` и `QWaitCondition`. За подробной информацией по этим классам обращайтесь к документации PyQt, а также к примерам, расположенным в папке C:\Python32\Lib\site-packages\PyQt4\examples\threads. Следует также помнить, что в стандартную библиотеку языка Python входят модули `multiprocessing` и `threading`, которые позволяют работать с потоками в любом приложении. Однако при использовании PyQt нужно отдать предпочтение классу `QThread`, т. к. он позволяет работать с сигналами.

19.10. Вывод заставки

В больших приложениях загрузка начальных данных может занимать продолжительное время. На это время принято выводить окно-заставку, внутри которого отображается процесс загрузки. По окончании загрузки окно-заставка скрывается и отображается главное окно приложения. Для отображения окна-заставки в PyQt предназначен класс `QSplashScreen` из модуля `QtGui`. Конструктор класса имеет следующие форматы:

```
<Объект> = QSplashScreen([<Изображение>][, flags=<Тип окна>])
<Объект> = QSplashScreen(<Родитель>[, <Изображение>][, flags=<Тип окна>])
```

Параметр `<Родитель>` позволяет указать ссылку на родительский компонент. В параметре `<Изображение>` указывается ссылка на изображение (экземпляр класса `QPixmap`), которое бу-

дет отображаться на заставке. Конструктору класса `QPixmap` можно передать путь к файлу с изображением. Параметр `flags` предназначен для указания типа окна, например, чтобы заставка отображалась поверх всех остальных окон, следует передать флаг `WindowStaysOnTopHint`. Экземпляр класса `QSplashScreen` содержит следующие методы:

- `show()` — отображает заставку;
- `finish(<Ссылка на окно>)` — закрывает заставку. В качестве параметра указывается ссылка на главное окно приложения;
- `showMessage(<Сообщение>[, <Выравнивание>[, <Цвет>]])` — выводит сообщение. Во втором параметре указывается местоположение надписи в окне. По умолчанию надпись выводится в левом верхнем углу окна. В качестве значения можно указать комбинацию следующих флагов через оператор `|`: `AlignTop` (по верху), `AlignCenter` (по центру вертикали и горизонтали), `AlignBottom` (по низу), `AlignHCenter` (по центру горизонтали), `AlignVCenter` (по центру вертикали), `AlignLeft` (по левой стороне), `AlignRight` (по правой стороне). В третьем параметре указывается цвет текста. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black` (по умолчанию), `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.);
- `clearMessage()` — стирает надпись;
- `setPixmap(<Изображение>)` — позволяет изменить изображение в окне. В качестве параметра указывается экземпляр класса `QPixmap`;
- `pixmap()` — возвращает экземпляр класса `QPixmap`.

Пример вывода заставки показан в листинге 19.21.

Листинг 19.21. Вывод заставки

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
from PyQt4.QtCore import Qt, SIGNAL
import time

class MyWindow(QtGui.QPushButton):
    def __init__(self):
        QtGui.QPushButton.__init__(self)
        self.setText("Закрыть окно")
        self.connect(self, SIGNAL("clicked()"), QtGui.qApp.quit)
    def load_data(self, sp):
        for i in range(1, 11):          # Имитируем процесс
            time.sleep(2)               # Что-то загружаем
            sp.showMessage("Загрузка данных... {0}%".format(i * 10),
                           Qt.AlignHCenter | Qt.AlignBottom, Qt.black)
            QtGui.qApp.processEvents()  # Запускаем оборот цикла

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    splash = QtGui.QSplashScreen(QtGui.QPixmap("img.png"))
```

```
splash.showMessage("Загрузка данных... 0%",
                   Qt.AlignHCenter | Qt.AlignBottom, Qt.black)
splash.show()                          # Отображаем заставку
QtGui.qApp.processEvents()             # Запускаем оборот цикла
window = MyWindow()
window.setWindowTitle("Использование класса QSplashScreen")
window.resize(300, 30)
window.load_data(splash)               # Загружаем данные
window.show()
splash.finish(window)                  # Скрываем заставку
sys.exit(app.exec_())
```

19.11. Доступ к документации

Библиотека PyQt содержит свыше 600 классов. Если описывать один класс на одной странице, то объем книги по PyQt будет более 600 страниц. Однако уложиться в одну страницу практически невозможно, следовательно, объем вырастет в два, а то и три раза. Издать книгу такого большого объема не представляется возможным, поэтому в данной книге мы рассмотрим только наиболее часто используемые возможности библиотеки PyQt, те возможности, которые вы будете использовать каждый день в своей практике. Чтобы получить полную информацию, следует обращаться к документации. Где ее найти, мы сейчас и рассмотрим.

Самая последняя версия документации в формате HTML доступна на сайте **http://www.riverbankcomputing.co.uk/**. С этого же сайта можно загрузить последнюю версию библиотеки. Документация входит также в состав дистрибутива и устанавливается в папку C:\Python32\Lib\site-packages\PyQt4\doc\html. Чтобы отобразить содержание, следует открыть в браузере файл index.html. Полный список классов расположен в файле classes.html, а список всех модулей — в файле modules.html. Каждое название класса или модуля является ссылкой на страницу с подробным описанием. Помимо документации в состав дистрибутива входят примеры, которые устанавливаются в папку C:\Python32\Lib\site-packages\PyQt4\examples. Обязательно изучите эти примеры.

ГЛАВА 20

Управление окном приложения

Создать окно и управлять им позволяет класс `QWidget`. Класс `QWidget` наследует два класса — `QObject` и `QPaintDevice`. В свою очередь класс `QWidget` является базовым классом для всех визуальных компонентов, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В этой главе мы рассмотрим методы класса `QWidget` применительно к окну верхнего уровня, однако следует помнить, что те же самые методы можно применять и к любым компонентам. Например, метод, позволяющий управлять размерами окна, можно использовать и для изменения размеров компонента, имеющего родителя. Тем не менее, некоторые методы имеет смысл использовать только для окон верхнего уровня, например, метод, позволяющий изменить текст в заголовке окна, не имеет смысла использовать в обычных компонентах.

Для создания окна верхнего уровня помимо класса `QWidget` можно использовать и другие классы, которые являются наследниками класса `QWidget`, например класс `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При использовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь только две кнопки в заголовке окна — **Справка** и **Закрыть**. Кроме того, можно использовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Использование классов `QDialog` и `QMainWindow` имеет свои отличия, которые мы будем рассматривать в отдельных главах.

20.1. Создание и отображение окна

Самый простой способ создать пустое окно показан в листинге 20.1.

Листинг 20.1. Создание и отображение окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()                    # Создаем окно
window.setWindowTitle("Заголовок окна")     # Указываем заголовок
window.resize(300, 50)                      # Минимальные размеры
window.show()                               # Отображаем окно
sys.exit(app.exec_())
```

Конструктор класса QWidget имеет следующий формат:

```
<Объект> = QWidget([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, то компонент будет обладать своим собственным окном. Если в параметре flags указан тип окна, то компонент, имея родителя, будет обладать своим собственным окном, но будет привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре flags, мы рассмотрим в следующем разделе.

Указать ссылку на родительский компонент уже после создания объекта позволяет метод setParent(). Формат метода:

```
setParent(<Родитель>[, <Тип окна>])
```

Получить ссылку на родительский компонент можно с помощью метода parentWidget(). Если компонент не имеет родителя, то метод возвращает значение None.

Для изменения текста в заголовке окна предназначен метод setWindowTitle(). Формат метода:

```
setWindowTitle(<Текст, отображаемый в заголовке>)
```

После создания окна необходимо вызвать метод show(), чтобы окно и все дочерние компоненты отобразились на экране. Для сокрытия окна предназначен метод hide(). Для отображения и сокрытия компонентов можно также воспользоваться методом setVisible(<Флаг>). Если в параметре указано значение True, то компонент будет отображен, а если значение False, то компонент будет скрыт. Пример отображения окна и всех дочерних компонентов:

```
window.setVisible(True)
```

Проверить, видим компонент в настоящее время или нет, позволяет метод isVisible(). Метод возвращает True, если компонент видим, и False — в противном случае. Кроме того, можно воспользоваться методом isHidden(). Метод возвращает True, если компонент скрыт, и False — в противном случае.

20.2. Указание типа окна

При использовании класса QWidget по умолчанию окно создается с заголовком, в котором расположены: иконка, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Закрыть**. Указать другой тип создаваемого окна позволяет метод setWindowFlags() или параметр flags в конструкторе класса QWidget. Обратите внимание на то, что метод setWindowFlags() должен вызываться перед отображением окна. Формат метода:

```
setWindowFlags(<Тип окна>)
```

В параметре <Тип окна> можно указать следующие атрибуты из класса QtCore.Qt:

- Widget — тип по умолчанию для класса QWidget;
- Window — указывает, что компонент является окном, независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Свернуть**, **Развернуть** и **Закрыть**. По умолчанию размеры окна можно изменять с помощью мыши;

- `Dialog` — диалоговое окно. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Справка** и **Закрыть**. Размеры окна можно изменять с помощью мыши. Пример указания типа для диалогового окна:

```
window.setWindowFlags(QtCore.Qt.Dialog)
```

- `Sheet`;
- `Drawer`;
- `Popup` — указывает, что окно является всплывающим меню. Окно выводится без рамки и заголовка. Кроме того, окно может отбрасывать тень. Изменить размеры окна с помощью мыши нельзя;
- `Tool` — сообщает, что окно является панелью инструментов. Окно выводится с рамкой и заголовком (меньшем по высоте, чем обычное окно), в котором расположена кнопка **Закрыть**. Размеры окна можно изменять с помощью мыши;
- `ToolTip` — указывает, что окно является всплывающей подсказкой. Окно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;
- `SplashScreen` — сообщает, что окно является заставкой. Окно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя. Значение по умолчанию для класса `QSplashScreen`;
- `Desktop` — указывает, что окно является рабочим столом. Окно вообще не отображается на экране;
- `SubWindow` — сообщает, что окно является дочерним компонентом, независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком (меньшем по высоте, чем обычное окно) без кнопок. Изменить размеры окна с помощью мыши нельзя.

Определить тип окна из программы позволяет метод `windowType()`.

Для окон верхнего уровня можно дополнительно указать следующие атрибуты из класса `QtCore.Qt` через оператор `|` (перечислены только наиболее часто используемые атрибуты; полный список смотрите в документации):

- `MSWindowsFixedSizeDialogHint` — запрещает изменение размеров окна. Изменить размеры с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной;
- `FramelessWindowHint` — убирает рамку и заголовок окна. Изменять размеры окна и перемещать его нельзя;
- `CustomizeWindowHint` — убирает рамку и заголовок окна, но добавляет эффект объемности. Размеры окна можно изменять с помощью мыши;
- `WindowTitleHint` — добавляет заголовок окна. Выведем окно фиксированного размера с заголовком, в котором находится только текст:

```
window.setWindowFlags(QtCore.Qt.Window |
                      QtCore.Qt.FramelessWindowHint |
                      QtCore.Qt.WindowTitleHint)
```

- `WindowSystemMenuHint` — добавляет оконное меню и кнопку **Закрыть**;
- `WindowMinimizeButtonHint` — кнопка **Свернуть** в заголовке окна делается активной, а кнопка **Развернуть** — неактивной;
- `WindowMaximizeButtonHint` — кнопка **Развернуть** в заголовке окна делается активной, а кнопка **Свернуть** — неактивной;

- `WindowMinMaxButtonsHint` — кнопки **Свернуть** и **Развернуть** в заголовке окна делаются активными;
- `WindowCloseButtonHint` — добавляет кнопку **Закрыть**;
- `WindowContextHelpButtonHint` — добавляет кнопку **Справка**. Кнопки **Свернуть** и **Развернуть** в заголовке окна не отображаются;
- `WindowStaysOnTopHint` — сообщает системе, что окно всегда должно отображаться поверх всех других окон;
- `WindowStaysOnBottomHint` — сообщает системе, что окно всегда должно расположено позади всех других окон.

Получить все установленные флаги из программы позволяет метод `windowFlags()`.

20.3. Изменение и получение размеров окна

Для изменения размеров окна предназначены следующие методы:

- `resize(<Ширина>, <Высота>)` — изменяет текущий размер окна. Если содержимое окна не помещается в установленный размер, то размер будет выбран так, чтобы компоненты поместились без искажения, при условии, что используются менеджеры геометрии. Следовательно, заданный размер может не соответствовать реальному размеру окна. Если используется абсолютное позиционирование, то компоненты могут оказаться наполовину или полностью за пределами видимой части окна. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.resize(100, 70)
window.resize(QtCore.QSize(100, 70))
```

- `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущий размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. В качестве параметра можно также указать экземпляр класса `QRect`. Пример:

```
window.setGeometry(100, 100, 100, 70)
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

- `setFixedSize(<Ширина>, <Высота>)` — задает фиксированный размер. Изменить размеры окна с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setFixedSize(100, 70)
window.setFixedSize(QtCore.QSize(100, 70))
```

- `setFixedWidth(<Ширина>)` — задает фиксированный размер только по ширине. Изменить ширину окна с помощью мыши нельзя;
- `setFixedHeight(<Высота>)` — задает фиксированный размер только по высоте. Изменить высоту окна с помощью мыши нельзя;
- `setMinimumSize(<Ширина>, <Высота>)` — задает минимальный размер. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setMinimumSize(100, 70)
window.setMinimumSize(QtCore.QSize(100, 70))
```

- `setMinimumWidth(<Ширина>)` и `setMinimumHeight(<Высота>)` — задают минимальный размер только по ширине и высоте соответственно;
- `setMaximumSize(<Ширина>, <Высота>)` — задает максимальный размер. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setMaximumSize(100, 70)
window.setMaximumSize(QtCore.QSize(100, 70))
```

- `setMaximumWidth(<Ширина>)` и `setMaximumHeight(<Высота>)` — задают максимальный размер только по ширине и высоте соответственно;
- `setBaseSize(<Ширина>, <Высота>)` — задает базовые размеры. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setBaseSize(500, 500)
window.setBaseSize(QtCore.QSize(500, 500))
```

- `adjustSize()` — подгоняет размеры компонента под содержимое. При этом учитываются рекомендуемые размеры, возвращаемые методом `sizeHint()`.

Получить размеры позволяют следующие методы:

- `width()` и `height()` — возвращают текущую ширину и высоту соответственно:

```
window.resize(50, 70)
print(window.width(), window.height())                # 50 70
```

- `size()` — возвращает экземпляр класса `QSize`, содержащий текущие размеры:

```
window.resize(50, 70)
print(window.size().width(), window.size().height()) # 50 70
```

- `minimumSize()` — возвращает экземпляр класса `QSize`, содержащий минимальные размеры;
- `minimumWidth()` и `minimumHeight()` — возвращают минимальную ширину и высоту соответственно;
- `maximumSize()` — возвращает экземпляр класса `QSize`, содержащий максимальные размеры;
- `maximumWidth()` и `maximumHeight()` — возвращают максимальную ширину и высоту соответственно;
- `baseSize()` — возвращает экземпляр класса `QSize`, содержащий базовые размеры;
- `sizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого размера;
- `minimumSizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый минимальный размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого минимального размера;
- `rect()` — возвращает экземпляр класса `QRect`, содержащий координаты и размеры прямоугольника, в который вписан компонент. Пример:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.rect()
print(rect.left(), rect.top())     # 0 0
print(rect.width(), rect.height()) # 100 70
```

- `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Пример:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.geometry()
print(rect.left(), rect.top())     # 100 100
print(rect.width(), rect.height()) # 100 70
```

При изменении и получении размеров окна следует учитывать, что:

- размеры не включают высоту заголовка окна и ширину границ;
- размер компонентов может изменяться в зависимости от настроек стиля. Например, на разных компьютерах может быть указан шрифт разного наименования и размера. Поэтому от указания фиксированных размеров лучше отказаться;
- размер окна может изменяться в промежутке между получением значения и действиями, выполняющими обработку этих значений в программе. Например, сразу после получения размера пользователь изменил размеры окна с помощью мыши.

Чтобы получить размеры окна, включая высоту заголовка и ширину границ, следует воспользоваться методом `frameSize()`. Метод возвращает экземпляр класса `QSize`. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. До этого момента размеры совпадают с размерами окна без учета высоты заголовка и ширины границ. Пример получения полного размера окна:

```
window.resize(200, 70)                    # Задаем размеры
# ...
window.show()                             # Отображаем окно
print(window.width(), window.height())    # 200 70
print(window.frameSize().width(),
      window.frameSize().height())        # 208 104
```

Чтобы получить координаты окна с учетом высоты заголовка и ширины границ, следует воспользоваться методом `frameGeometry()`. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. Метод возвращает экземпляр класса `QRect`. Пример:

```
window.setGeometry(100, 100, 200, 70)
# ...
window.show()                             # Отображаем окно
rect = window.geometry()
print(rect.left(), rect.top())            # 100 100
print(rect.width(), rect.height())        # 200 70
rect = window.frameGeometry()
print(rect.left(), rect.top())            # 96 70
print(rect.width(), rect.height())        # 208 104
```

20.4. Местоположение окна на экране

Задать местоположение окна на экране монитора позволяют следующие методы:

- `move(<X>, <Y>)` — изменяет положение компонента относительно родителя. Метод учитывает высоту заголовка и ширину границ. В качестве параметра можно также указать экземпляр класса `QPoint`.

Пример вывода окна в левом верхнем углу экрана:

```
window.move(0, 0)
window.move(QtCore.QPoint(0, 0))
```

- `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущий размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. Обратите внимание на то, что метод не учитывает высоту заголовка и ширину границ. Поэтому если указать координаты `(0, 0)`, то заголовок окна и левая граница будут за пределами экрана. В качестве параметра можно также указать экземпляр класса `QRect`. Пример:

```
window.setGeometry(100, 100, 100, 70)
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

> ***ОБРАТИТЕ ВНИМАНИЕ***
>
> Начало координат расположено в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз.

Получить позицию окна позволяют следующие методы:

- `x()` и `y()` — возвращают координаты левого верхнего угла окна относительно родителя по осям `X` и `Y` соответственно. Методы учитывают высоту заголовка и ширину границ. Пример:

```
window.move(10, 10)
print(window.x(), window.y())              # 10 10
```

- `pos()` — возвращает экземпляр класса `QPoint`, содержащий координаты левого верхнего угла окна относительно родителя. Метод учитывает высоту заголовка и ширину границ. Пример:

```
window.move(10, 10)
print(window.pos().x(), window.pos().y()) # 10 10
```

- `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Обратите внимание на то, что метод не учитывает высоту заголовка и ширину границ. Пример:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.geometry()
print(rect.left(), rect.top())             # 14 40
print(rect.width(), rect.height())         # 300 100
```

- `frameGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты с учетом высоты заголовка и ширины границ. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. Пример:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.frameGeometry()
print(rect.left(), rect.top())             # 10 10
print(rect.width(), rect.height())         # 308 134
```

Для отображения окна по центру экрана или у правой или нижней границы необходимо знать размеры экрана. Для получения размеров экрана вначале следует вызвать статический метод `QApplication.desktop()`, который возвращает ссылку на компонент рабочего стола. Получить размеры экрана позволяют следующие методы этого объекта:

- `width()` — возвращает ширину всего экрана в пикселах;
- `height()` — возвращает высоту всего экрана в пикселах:

```
desktop = QtGui.QApplication.desktop()
print(desktop.width(), desktop.height()) # 1280 1024
```

- `screenGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты всего экрана:

```
desktop = QtGui.QApplication.desktop()
rect = desktop.screenGeometry()
print(rect.left(), rect.top())           # 0 0
print(rect.width(), rect.height())       # 1280 1024
```

- `availableGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты только доступной части экрана (без размера **Панели задач**):

```
desktop = QtGui.QApplication.desktop()
rect = desktop.availableGeometry()
print(rect.left(), rect.top())           # 0 0
print(rect.width(), rect.height())       # 1280 994
```

Пример отображения окна приблизительно по центру экрана показан в листинге 20.2.

Листинг 20.2. Вывод окна приблизительно по центру экрана

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
desktop = QtGui.QApplication.desktop()
x = (desktop.width() - window.width()) // 2
y = (desktop.height() - window.height()) // 2
window.move(x, y)
window.show()
sys.exit(app.exec_())
```

В этом примере мы воспользовались методами `width()` и `height()`, которые не учитывают высоту заголовка и ширину границ. В большинстве случаев этого способа достаточно. Если необходима точность при выравнивании, то для получения размеров окна можно воспользоваться методом `frameSize()`. Однако этот метод возвращает корректные значения только после отображения окна. Если код выравнивания по центру расположить после вызова метода `show()`, то окно отобразится вначале в одном месте экрана, а затем переместится в центр, что вызовет неприятное мелькание. Чтобы исключить мелькание, следует вначале

отобразить окно за пределами экрана, а затем переместить окно в центр экрана (листинг 20.3).

Листинг 20.3. Вывод окна точно по центру экрана

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
window.move(window.width() * -2, 0)
window.show()
desktop = QtGui.QApplication.desktop()
x = (desktop.width() — window.frameSize().width()) // 2
y = (desktop.height() — window.frameSize().height()) // 2
window.move(x, y)
sys.exit(app.exec_())
```

Этот способ можно также использовать для выравнивания окна по правому краю экрана. Например, чтобы расположить окно в правом верхнем углу экрана, необходимо заменить код, выравнивающий окно по центру, из предыдущего примера следующим кодом:

```
desktop = QtGui.QApplication.desktop()
x = desktop.width() — window.frameSize().width()
window.move(x, 0)
```

Если попробовать вывести окно в правом нижнем углу, то может возникнуть проблема, т. к. в операционной системе Windows в нижней части экрана располагается **Панель задач**. В итоге окно будет частично расположено под **Панелью задач**. Чтобы при размещении окна учитывать местоположение **Панели задач**, необходимо использовать метод `availableGeometry()`. Получить высоту **Панели задач**, расположенной в нижней части экрана, можно, например, так:

```
desktop = QtGui.QApplication.desktop()
taskBarHeight = (desktop.screenGeometry().height() -
                 desktop.availableGeometry().height())
```

Следует также заметить, что в некоторых операционных системах **Панель задач** может быть прикреплена к любой стороне экрана. Кроме того, экран может быть разделен на несколько рабочих столов. Все это необходимо учитывать при размещении окна. За подробной информацией обращайтесь к документации по классу `QDesktopWidget`.

20.5. Указание координат и размеров

В двух предыдущих разделах были упомянуты классы `QPoint`, `QSize` и `QRect`. Класс `QPoint` описывает координаты точки, класс `QSize` — размеры, а класс `QRect` — координаты и размеры прямоугольной области. Рассмотрим эти классы более подробно.

Примечание

Классы QPoint, QSize и QRect предназначены для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать классы QPointF, QSizeF и QRectF соответственно.

20.5.1. Класс *QPoint*. Координаты точки

Класс QPoint из модуля QtCore описывает координаты точки. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
<Объект> = QPoint()
<Объект> = QPoint(<X>, <Y>)
<Объект> = QPoint(<QPoint>)
```

Первый конструктор создает экземпляр класса с нулевыми координатами:

```
>>> from PyQt4 import QtCore
>>> p = QtCore.QPoint()
>>> p.x(), p.y()
(0, 0)
```

Второй конструктор позволяет явно указать координаты точки:

```
>>> p = QtCore.QPoint(10, 88)
>>> p.x(), p.y()
(10, 88)
```

Третий конструктор создает новый экземпляр на основе другого экземпляра:

```
>>> p = QtCore.QPoint(QtCore.QPoint(10, 88))
>>> p.x(), p.y()
(10, 88)
```

Через экземпляр класса доступны следующие методы:

- x() и y() — возвращают координаты по осям X и Y соответственно;
- setX(<X>) и setY(<Y>) — задают координаты по осям X и Y соответственно;
- isNull() — возвращает True, если координаты равны нулю, и False — в противном случае:

```
>>> p = QtCore.QPoint()
>>> p.isNull()
True
>>> p.setX(10); p.setY(88)
>>> p.x(), p.y()
(10, 88)
```

- manhattanLength() — возвращает сумму абсолютных значений координат:

```
>>> QtCore.QPoint(10, 88).manhattanLength()
98
```

Над двумя экземплярами класса QPoint определены операции +, +=, – (минус), -=, == и !=. Для смены знака координат можно воспользоваться унарным оператором -. Кроме того, экземпляр класса QPoint можно умножить или разделить на вещественное число (операторы *, *=, / и /=).

Пример:

```
>>> p1 = QtCore.QPoint(10, 20); p2 = QtCore.QPoint(5, 9)
>>> p1 + p2, p1 — p2
(PyQt4.QtCore.QPoint(15, 29), PyQt4.QtCore.QPoint(5, 11))
>>> p1 * 2.5, p1 / 2.0
(PyQt4.QtCore.QPoint(25, 50), PyQt4.QtCore.QPoint(5, 10))
>>> -p1, p1 == p2, p1 != p2
(PyQt4.QtCore.QPoint(-10, -20), False, True)
```

20.5.2. Класс *QSize*. Размеры прямоугольной области

Класс `QSize` из модуля `QtCore` описывает размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
<Объект> = QSize()
<Объект> = QSize(<Ширина>, <Высота>)
<Объект> = QSize(<QSize>)
```

Первый конструктор создает экземпляр класса с отрицательной шириной и высотой. Второй конструктор позволяет явно указать ширину и высоту. Третий конструктор создает новый экземпляр на основе другого экземпляра. Пример:

```
>>> from PyQt4 import QtCore
>>> s1=QtCore.QSize(); s2=QtCore.QSize(10, 55); s3=QtCore.QSize(s2)
>>> s1
PyQt4.QtCore.QSize(-1, -1)
>>> s2, s3
(PyQt4.QtCore.QSize(10, 55), PyQt4.QtCore.QSize(10, 55))
```

Через экземпляр класса доступны следующие методы:

- `width()` и `height()` — возвращают ширину и высоту соответственно;
- `setWidth(<Ширина>)` и `setHeight(<Высота>)` — задают ширину и высоту соответственно:

  ```
  >>> s = QtCore.QSize()
  >>> s.setWidth(10); s.setHeight(55)
  >>> s.width(), s.height()
  (10, 55)
  ```

- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если ширина и высота больше или равны нулю, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если один параметр (ширина или высота) меньше или равен нулю, и `False` — в противном случае;
- `scale()` — производит изменение размеров области в соответствии со значением параметра `<Тип преобразования>`. Метод изменяет текущий объект и ничего не возвращает. Форматы метода:

  ```
  scale(<QSize>, <Тип преобразования>)
  scale(<Ширина>, <Высота>, <Тип преобразования>)
  ```

В параметре <Тип преобразования> могут быть указаны следующие атрибуты из класса QtCore.Qt:

- IgnoreAspectRatio — 0 — непосредственно изменяет размеры без сохранения пропорций сторон;
- KeepAspectRatio — 1 — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
- KeepAspectRatioByExpanding — 2 — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

Если новая ширина или высота имеет значение 0, то размеры изменяются непосредственно без сохранения пропорций, вне зависимости от значения параметра <Тип преобразования>.

> ***Примечание***
>
> Чтобы полностью понять принцип действия атрибутов KeepAspectRatio и KeepAspectRatioByExpanding, откройте изображение qimage-scaling.png, которое расположено в папке C:\Python32\Lib\site-packages\PyQt4\doc\html\images.

Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.IgnoreAspectRatio); s
PyQt4.QtCore.QSize(70, 60)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatio); s
PyQt4.QtCore.QSize(70, 28)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatioByExpanding); s
PyQt4.QtCore.QSize(150, 60)
```

- boundedTo(<QSize>) — возвращает экземпляр класса QSize, который содержит минимальную ширину и высоту из текущих размеров и размеров, указанных в параметре. Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.boundedTo(QtCore.QSize(400, 5))
PyQt4.QtCore.QSize(50, 5)
>>> s.boundedTo(QtCore.QSize(40, 50))
PyQt4.QtCore.QSize(40, 20)
```

- expandedTo(<QSize>) — возвращает экземпляр класса QSize, который содержит максимальную ширину и высоту из текущих размеров и размеров, указанных в параметре. Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.expandedTo(QtCore.QSize(400, 5))
PyQt4.QtCore.QSize(400, 20)
>>> s.expandedTo(QtCore.QSize(40, 50))
PyQt4.QtCore.QSize(50, 50)
```

- transpose() — меняет значения местами. Метод изменяет текущий объект и ничего не возвращает.

Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.transpose(); s
PyQt4.QtCore.QSize(20, 50)
```

Над двумя экземплярами класса `QSize` определены операции +, +=, – (минус), -=, == и !=. Кроме того, экземпляр класса `QSize` можно умножить или разделить на вещественное число (операторы *, *=, / и /=). Пример:

```
>>> s1 = QtCore.QSize(50, 20); s2 = QtCore.QSize(10, 5)
>>> s1 + s2, s1 – s2
(PyQt4.QtCore.QSize(60, 25), PyQt4.QtCore.QSize(40, 15))
>>> s1 * 2.5, s1 / 2
(PyQt4.QtCore.QSize(125, 50), PyQt4.QtCore.QSize(25, 10))
>>> s1 == s2, s1 != s2
(False, True)
```

20.5.3. Класс *QRect*. Координаты и размеры прямоугольной области

Класс `QRect` из модуля `QtCore` описывает координаты и размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструктора:

```
<Объект> = QRect()
<Объект> = QRect(<left>, <top>, <Ширина>, <Высота>)
<Объект> = QRect(<Координаты левого верхнего угла>, <Размеры>)
<Объект> = QRect(<Координаты левого верхнего угла>,
                 <Координаты правого нижнего угла>)
<Объект> = QRect(<QRect>)
```

Первый конструктор создает экземпляр класса со значениями по умолчанию. Второй и третий конструкторы позволяют указать координаты левого верхнего угла и размеры области. Во втором конструкторе значения указываются отдельно. В третьем конструкторе координаты задаются с помощью класса `QPoint`, а размеры — с помощью класса `QSize`. Четвертый конструктор позволяет указать координаты левого верхнего угла и правого нижнего угла. В качестве значений указываются экземпляры класса `QPoint`. Пятый конструктор создает новый экземпляр на основе другого экземпляра. Пример:

```
>>> from PyQt4 import QtCore
>>> r = QtCore.QRect()
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(0, 0, -1, -1, 0, 0)
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QSize(400, 300))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QPoint(409, 314))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
```

```
>>> QtCore.QRect(r)
PyQt4.QtCore.QRect(10, 15, 400, 300)
```

Изменить значения уже после создания экземпляра позволяют следующие методы:

- `setLeft(<X1>)`, `setX(<X1>)`, `setTop(<Y1>)` и `setY(<Y1>)` — задают координаты левого верхнего угла по осям `X` и `Y`. Пример:

```
>>> r = QtCore.QRect()
>>> r.setLeft(10); r.setTop(55); r
PyQt4.QtCore.QRect(10, 55, -10, -55)
>>> r.setX(12); r.setY(81); r
PyQt4.QtCore.QRect(12, 81, -12, -81)
```

- `setRight(<X2>)` и `setBottom(<Y2>)` — задают координаты правого нижнего угла по осям `X` и `Y`. Пример:

```
>>> r = QtCore.QRect()
>>> r.setRight(12); r.setBottom(81); r
PyQt4.QtCore.QRect(0, 0, 13, 82)
```

- `setTopLeft(<QPoint>)` — задает координаты левого верхнего угла;
- `setTopRight(<QPoint>)` — задает координаты правого верхнего угла;
- `setBottomLeft(<QPoint>)` — задает координаты левого нижнего угла;
- `setBottomRight(<QPoint>)` — задает координаты правого нижнего угла:

```
>>> r = QtCore.QRect()
>>> r.setTopLeft(QtCore.QPoint(10, 5))
>>> r.setBottomRight(QtCore.QPoint(39, 19)); r
PyQt4.QtCore.QRect(10, 5, 30, 15)
>>> r.setTopRight(QtCore.QPoint(39, 5))
>>> r.setBottomLeft(QtCore.QPoint(10, 19)); r
PyQt4.QtCore.QRect(10, 5, 30, 15)
```

- `setWidth(<Ширина>)`, `setHeight(<Высота>)` и `setSize(<QSize>)` — задают ширину и высоту области;
- `setRect(<X1>, <Y1>, <Ширина>, <Высота>)` — задает координаты левого верхнего угла и размеры области;
- `setCoords(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты левого верхнего угла и правого нижнего угла. Пример:

```
>>> r = QtCore.QRect()
>>> r.setRect(10, 10, 100, 500); r
PyQt4.QtCore.QRect(10, 10, 100, 500)
>>> r.setCoords(10, 10, 109, 509); r
PyQt4.QtCore.QRect(10, 10, 100, 500)
```

Переместить область при изменении координат позволяют следующие методы:

- `moveTo(<X1>, <Y1>)`, `moveTo(<QPoint>)`, `moveLeft(<X1>)` и `moveTop(<Y1>)` — перемещают координаты левого верхнего угла:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTo(0, 0); r
PyQt4.QtCore.QRect(0, 0, 400, 300)
```

```
>>> r.moveTo(QtCore.QPoint(10, 10)); r
PyQt4.QtCore.QRect(10, 10, 400, 300)
>>> r.moveLeft(5); r.moveTop(0); r
PyQt4.QtCore.QRect(5, 0, 400, 300)
```

- `moveRight(<X2>)` и `moveBottom(<Y2>)` — перемещают координаты правого нижнего угла;
- `moveTopLeft(<QPoint>)` — перемещает координаты левого верхнего угла;
- `moveTopRight(<QPoint>)` — перемещает координаты правого верхнего угла;
- `moveBottomLeft(<QPoint>)` — перемещает координаты левого нижнего угла;
- `moveBottomRight(<QPoint>)` — перемещает координаты правого нижнего угла. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTopLeft(QtCore.QPoint(0, 0)); r
PyQt4.QtCore.QRect(0, 0, 400, 300)
>>> r.moveBottomRight(QtCore.QPoint(599, 499)); r
PyQt4.QtCore.QRect(200, 200, 400, 300)
```

- `moveCenter(<QPoint>)` — перемещает координаты центра;
- `translate(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translate(<QPoint>)` — перемещают координаты левого верхнего угла относительно текущего значения координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.translate(20, 15); r
PyQt4.QtCore.QRect(20, 15, 400, 300)
>>> r.translate(QtCore.QPoint(10, 5)); r
PyQt4.QtCore.QRect(30, 20, 400, 300)
```

- `translated(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translated(<QPoint>)` — метод аналогичен методу `translate()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий;
- `adjust(<X1>, <Y1>, <X2>, <Y2>)` — сдвигает координаты левого верхнего угла и правого нижнего угла относительно текущих значений координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.adjust(10, 5, 10, 5); r
PyQt4.QtCore.QRect(10, 5, 400, 300)
```

- `adjusted(<X1>, <Y1>, <X2>, <Y2>)` — метод аналогичен методу `adjust()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий.

Для получения значений предназначены следующие методы:

- `left()` и `x()` — возвращают координату левого верхнего угла по оси `X`;
- `top()` и `y()` — возвращают координату левого верхнего угла по оси `Y`;
- `right()` и `bottom()` — возвращают координаты правого нижнего угла по осям `X` и `Y` соответственно;
- `width()` и `height()` — возвращают ширину и высоту соответственно;
- `size()` — возвращает размеры в виде экземпляра класса `QSize`. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.x(), r.y(), r.right(), r.bottom()
(10, 15, 10, 15, 409, 314)
```

```
>>> r.width(), r.height(), r.size()
(400, 300, PyQt4.QtCore.QSize(400, 300))
```

- `topLeft()` — возвращает координаты левого верхнего угла;
- `topRight()` — возвращает координаты правого верхнего угла;
- `bottomLeft()` — возвращает координаты левого нижнего угла;
- `bottomRight()` — возвращает координаты правого нижнего угла. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.topLeft(), r.topRight()
(PyQt4.QtCore.QPoint(10, 15), PyQt4.QtCore.QPoint(409, 15))
>>> r.bottomLeft(), r.bottomRight()
(PyQt4.QtCore.QPoint(10, 314), PyQt4.QtCore.QPoint(409, 314))
```

- `center()` — возвращает координаты центра области. Например, вывести окно по центру доступной области экрана можно так:

```
desktop = QtGui.QApplication.desktop()
window.move(desktop.availableGeometry().center() -
            window.rect().center())
```

- `getRect()` — возвращает кортеж с координатами левого верхнего угла и размерами области;
- `getCoords()` — возвращает кортеж с координатами левого верхнего угла и правого нижнего угла. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.getRect(), r.getCoords()
((10, 15, 400, 300), (10, 15, 409, 314))
```

Прочие методы:

- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если `left() < right()` и `top() < bottom()`, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если `left() > right()` или `top() > bottom()`, и `False` — в противном случае;
- `normalized()` — исправляет ситуацию, при которой `left() > right()` или `top() > bottom()`, и возвращает новый экземпляр класса `QRect`. Пример:

```
>>> r = QtCore.QRect(QtCore.QPoint(409, 314),
                     QtCore.QPoint(10, 15))
>>> r
PyQt4.QtCore.QRect(409, 314, -398, -298)
>>> r.normalized()
PyQt4.QtCore.QRect(10, 15, 400, 300)
```

- `contains(<QPoint>[, <Флаг>])` и `contains(<X>, <Y>[, <Флаг>])` — возвращает `True`, если точка с указанными координатами расположена внутри области или на ее границе, и `False` — в противном случае. Если во втором параметре указано значение `True`, то точка должна быть расположена только внутри области, а не на ее границе. Значение параметра по умолчанию — `False`.

Пример:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(0, 10), r.contains(0, 10, True)
(True, False)
```

- `contains(<QRect>[, <Флаг>])` — возвращает `True`, если указанная область расположена внутри текущей области или на ее краю, и `False` — в противном случае. Если во втором параметре указано значение `True`, то указанная область должна быть расположена только внутри текущей области, а не на ее краю. Значение параметра по умолчанию — `False`. Пример:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(QtCore.QRect(0, 0, 20, 5))
True
>>> r.contains(QtCore.QRect(0, 0, 20, 5), True)
False
```

- `intersects(<QRect>)` — возвращает `True`, если указанная область пересекается с текущей областью, и `False` — в противном случае;
- `intersect(<QRect>)` и `intersected(<QRect>)` — возвращают область, которая расположена на пересечении текущей и указанной областей. Пример:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.intersects(QtCore.QRect(10, 10, 20, 20))
True
>>> r.intersect(QtCore.QRect(10, 10, 20, 20))
PyQt4.QtCore.QRect(10, 10, 10, 10)
>>> r.intersected(QtCore.QRect(10, 10, 20, 20))
PyQt4.QtCore.QRect(10, 10, 10, 10)
```

- `unite(<QRect>)` и `united(<QRect>)` — возвращают область, которая охватывает текущую и указанную области. Пример:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.unite(QtCore.QRect(30, 30, 20, 20))
PyQt4.QtCore.QRect(0, 0, 50, 50)
>>> r.united(QtCore.QRect(30, 30, 20, 20))
PyQt4.QtCore.QRect(0, 0, 50, 50)
```

Над двумя экземплярами класса `QRect` определены операции `&` и `&=` (пересечение), `|` и `|=` (объединение), `in` (проверка на вхождение), `==` и `!=`. Пример:

```
>>> r1, r2 = QtCore.QRect(0, 0, 20, 20), QtCore.QRect(10, 10, 20, 20)
>>> r1 & r2, r1 | r2
(PyQt4.QtCore.QRect(10, 10, 10, 10), PyQt4.QtCore.QRect(0, 0, 30, 30))
>>> r1 in r2, r1 in QtCore.QRect(0, 0, 30, 30)
(False, True)
>>> r1 == r2, r1 != r2
(False, True)
```

20.6. Разворачивание и сворачивание окна

В заголовке окна расположены кнопки **Свернуть** и **Развернуть**, с помощью которых можно свернуть окно в значок на **Панели задач** или максимально развернуть окно. Выполнить подобные действия из программы позволяют следующие методы класса `QWidget`:

- `showMinimized()` — сворачивает окно на **Панель задач**. Эквивалентно нажатию кнопки **Свернуть** в заголовке окна;
- `showMaximized()` — разворачивает окно до максимального размера. Эквивалентно нажатию кнопки **Развернуть** в заголовке окна;
- `showFullScreen()` — включает полноэкранный режим отображения окна. Окно отображается без заголовка и границ;
- `showNormal()` — отменяет сворачивание, максимальный размер и полноэкранный режим;
- `activateWindow()` — делает окно активным (т. е. имеющим фокус ввода). В Windows, если окно было ранее свернуто в значок на **Панель задач**, то оно автоматически не будет отображено на экране. В этом случае станет активным только значок на **Панели задач**;
- `setWindowState(<Флаги>)` — изменяет статус окна в зависимости от переданных флагов. В качестве параметра указывается комбинация следующих атрибутов из класса `QtCore.Qt` через побитовые операторы:
  - `WindowNoState` — нормальное состояние окна;
  - `WindowMinimized` — окно свернуто;
  - `WindowMaximized` — окно максимально развернуто;
  - `WindowFullScreen` — полноэкранный режим;
  - `WindowActive` — окно имеет фокус ввода, т. е. является активным.

  Например, включить полноэкранный режим можно так:

```
window.setWindowState((window.windowState() &
    ~(QtCore.Qt.WindowMinimized | QtCore.Qt.WindowMaximized))
    | QtCore.Qt.WindowFullScreen)
```

Проверить текущий статус окна позволяют следующие методы:

- `isMinimized()` — возвращает `True`, если окно свернуто, и `False` — в противном случае;
- `isMaximized()` — возвращает `True`, если окно раскрыто до максимальных размеров, и `False` — в противном случае;
- `isFullScreen()` — возвращает `True`, если включен полноэкранный режим, и `False` — в противном случае;
- `isActiveWindow()` — возвращает `True`, если окно имеет фокус ввода, и `False` — в противном случае;
- `windowState()` — возвращает комбинацию флагов, обозначающих текущий статус окна. Пример проверки использования полноэкранного режима:

```
if window.windowState() & QtCore.Qt.WindowFullScreen:
    print("Полноэкранный режим")
```

Пример разворачивания и сворачивания окна приведен в листинге 20.4.

Листинг 20.4. Разворачивание и сворачивание окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.btnMin = QtGui.QPushButton("Свернуть")
        self.btnMax = QtGui.QPushButton("Развернуть")
        self.btnFull = QtGui.QPushButton("Полный экран")
        self.btnNormal = QtGui.QPushButton("Нормальный размер")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.btnMin)
        vbox.addWidget(self.btnMax)
        vbox.addWidget(self.btnFull)
        vbox.addWidget(self.btnNormal)
        self.setLayout(vbox)
        self.connect(self.btnMin, QtCore.SIGNAL("clicked()"),
                     self.on_min)
        self.connect(self.btnMax, QtCore.SIGNAL("clicked()"),
                     self.on_max)
        self.connect(self.btnFull, QtCore.SIGNAL("clicked()"),
                     self.on_full)
        self.connect(self.btnNormal, QtCore.SIGNAL("clicked()"),
                     self.on_normal)
    def on_min(self):
        self.showMinimized()
    def on_max(self):
        self.showMaximized()
    def on_full(self):
        self.showFullScreen()
    def on_normal(self):
        self.showNormal()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Разворачивание и сворачивание окна")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

20.7. Управление прозрачностью окна

Сделать окно полупрозрачным позволяет метод `setWindowOpacity()` из класса `QWidget`. Формат метода:

```
setWindowOpacity(<Вещественное число от 0.0 до 1.0>)
```

В качестве параметра указывается вещественное число от 0.0 до 1.0. Число 0.0 соответствует полностью прозрачному окну, а число 1.0 — отсутствию прозрачности. Для получения степени прозрачности окна из программы предназначен метод windowOpacity(), который возвращает вещественное число от 0.0 до 1.0. Выведем окно со степенью прозрачности 0.5 (листинг 20.5).

Листинг 20.5. Полупрозрачное окно

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Полупрозрачное окно")
window.resize(300, 100)
window.setWindowOpacity(0.5)
window.show()
print(window.windowOpacity()) # Выведет: 0.4980392156862745
sys.exit(app.exec_())
```

20.8. Модальные окна

Модальным называется окно, которое не позволяет взаимодействовать с другими окнами в том же приложении. Пока модальное окно не будет закрыто, сделать активным другое окно нельзя. Например, если в программе Microsoft Word выбрать пункт меню **Файл | Сохранить как**, то откроется модальное диалоговое окно, позволяющее выбрать путь и название файла. Пока это окно не будет закрыто, вы не сможете взаимодействовать с главным окном приложения.

Указать, что окно является модальным, позволяет метод setWindowModality(<Флаг>) из класса QWidget. В качестве параметра могут быть указаны следующие атрибуты из класса QtCore.Qt:

- NonModal — 0 — окно не является модальным;
- WindowModal — 1 — окно блокирует только родительские окна в пределах иерархии;
- ApplicationModal — 2 — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод setWindowModality() должен быть вызван до отображения окна.

Получить текущее значение позволяет метод windowModality(). Проверить, является ли окно модальным, можно с помощью метода isModal(). Метод возвращает True, если окно является модальным, и False — в противном случае.

Создадим два независимых окна. В первом окне разместим кнопку, при нажатии которой откроем модальное окно. Это модальное окно будет блокировать только первое окно, но не второе. При открытии модального окна отобразим его примерно по центру родительского окна (листинг 20.6).

Листинг 20.6. Модальные окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

def show_modal_window():
    global modalWindow
    modalWindow = QtGui.QWidget(window1, QtCore.Qt.Window)
    modalWindow.setWindowTitle("Модальное окно")
    modalWindow.resize(200, 50)
    modalWindow.setWindowModality(QtCore.Qt.WindowModal)
    modalWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
    modalWindow.move(window1.geometry().center() -
         modalWindow.rect().center() — QtCore.QPoint(4, 30))
    modalWindow.show()

app = QtGui.QApplication(sys.argv)
window1 = QtGui.QWidget()
window1.setWindowTitle("Обычное окно")
window1.resize(300, 100)
button = QtGui.QPushButton("Открыть модальное окно")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       show_modal_window)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(button)
window1.setLayout(vbox)
window1.show()

window2 = QtGui.QWidget()
window2.setWindowTitle("Это окно не будет блокировано при WindowModal")
window2.resize(500, 100)
window2.show()

sys.exit(app.exec_())
```

Если запустить приложение и нажать кнопку **Открыть модальное окно**, то откроется окно, выровненное примерно по центру родительского окна (произвести точное выравнивание вы сможете самостоятельно). При этом получить доступ к родительскому окну можно только при закрытии модального окна, а второе окно блокировано не будет. Если заменить атрибут `WindowModal` атрибутом `ApplicationModal`, то оба окна будут блокированы.

Обратите внимание на то, что в конструктор модального окна мы передали ссылку на первое окно и атрибут `Window`. Если ссылку не указать, то окно блокировано не будет, а если атрибут не указать, то окно вообще не откроется. Кроме того, мы объявили переменную `modalWindow` глобальной, иначе при достижении конца функции переменная выйдет из области видимости и окно будет автоматически удалено. Чтобы объект окна автоматически удалялся при закрытии окна, атрибуту `WA_DeleteOnClose` в методе `setAttribute()` было присвоено значение `True`.

Модальные окна в большинстве случаев являются диалоговыми. Для работы с диалоговыми окнами в PyQt предназначен класс `QDialog`, который автоматически выравнивает окно по

центру экрана или по центру родительского окна. Кроме того, этот класс предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и многое другое. Подробно класс `QDialog` мы будем изучать в отдельной главе.

20.9. Смена иконки в заголовке окна

По умолчанию в левом верхнем углу окна отображается стандартная иконка. Отобразить другую иконку в заголовке окна позволяет метод `setWindowIcon()` из класса `QWidget`. В качестве параметра метод принимает экземпляр класса `QIcon` (см. *разд. 25.3.4*). Чтобы загрузить иконку из файла, следует передать путь к файлу конструктору класса. Если указан относительный путь, то поиск файла будет производиться относительно текущего рабочего каталога. Получить список поддерживаемых форматов файлов можно с помощью статического метода `supportedImageFormats()` из класса `QImageReader`. Метод возвращает список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов:

```
from PyQt4 import QtGui
for i in QtGui.QImageReader.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Результат выполнения на моем компьютере:

```
BMP GIF ICO JPEG JPG MNG PBM PGM PNG PPM SVG SVGZ TIF TIFF XBM XPM
```

Если для окна не задана иконка, то будет использоваться иконка приложения, установленная с помощью метода `setWindowIcon()` из класса `QApplication`. В качестве параметра метод принимает экземпляр класса `QIcon`.

Вместо загрузки иконки из файла можно воспользоваться одной из встроенных иконок. Загрузить стандартную иконку позволяет следующий код:

```
ico = window.style().standardIcon(QtGui.QStyle.SP_MessageBoxCritical)
window.setWindowIcon(ico)
```

Посмотреть список всех встроенных иконок можно в документации к классу `QStyle` (C:/Python32/Lib/site-packages/PyQt4/doc/html/qstyle.html#StandardPixmap-enum).

В качестве примера создаем иконку в формате PNG размером 16 на 16 пикселов и сохраняем ее в одной папке с программой, а далее устанавливаем эту иконку для окна и для всего приложения (листинг 20.7).

Листинг 20.7. Смена иконки в заголовке окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Смена иконки в заголовке окна")
window.resize(300, 100)
window.setWindowIcon(QtGui.QIcon("icon.png")) # Иконка для окна
app.setWindowIcon(QtGui.QIcon("icon.png"))    # Иконка приложения
window.show()
sys.exit(app.exec_())
```

20.10. Изменение цвета фона окна

Чтобы изменить цвет фона окна (или компонента), следует установить палитру с настроенной ролью `Window` (или `Background`). Цветовая палитра содержит цвета для каждой роли и состояния компонента. Указать состояние компонента позволяют следующие атрибуты из класса `QPalette`:

- `Active` и `Normal` — `0` — компонент активен (окно находится в фокусе ввода);
- `Disabled` — `1` — компонент недоступен;
- `Inactive` — `2` — компонент неактивен (окно находится вне фокуса ввода).

Получить текущую палитру компонента позволяет метод `palette()`. Чтобы изменить цвет для какой-либо роли и состояния, следует воспользоваться методом `setColor()` из класса `QPalette`. Формат метода:

```
setColor([<Состояние>, ]<Роль>, <Цвет>)
```

В параметре `<Роль>` указывается для какого элемента изменяется цвет. Например, атрибут `Window` (или `Background`) изменяет цвет фона, а `WindowText` (или `Foreground`) — цвет текста. Полный список атрибутов смотрите в документации по классу `QPalette`.

В параметре `<Цвет>` указывается цвет элемента. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.).

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `True` методу `setAutoFillBackground()`.

Изменить цвет фона можно также с помощью CSS-атрибута `background-color`. Для этого следует передать таблицу стилей в метод `setStyleSheet()`. Таблицы стилей могут быть внешними (подключение через командную строку), установленными на уровне приложения (с помощью метода `setStyleSheet()` из класса `QApplication`) или установленными на уровне компонента (с помощью метода `setStyleSheet()` из класса `QWidget`). Атрибуты, установленные последними, обычно перекрывают значения аналогичных атрибутов, указанных ранее. Если вы занимались Web-программированием, то CSS вам уже знаком, а если нет, то придется дополнительно изучить HTML и CSS.

Создадим окно с надписью. Для активного окна установим зеленый цвет, а для неактивного — красный. Цвет фона надписи сделаем белым. Для изменения фона окна будем устанавливать палитру, а для изменения цвета фона надписи — CSS-атрибут `background-color` (листинг 20.8).

Листинг 20.8. Изменение цвета фона окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Изменение цвета фона окна")
```

```
window.resize(300, 100)
pal = window.palette()
pal.setColor(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QColor("#008800"))
pal.setColor(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
             QtGui.QColor("#ff0000"))
window.setPalette(pal)
label = QtGui.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignCenter)
label.setStyleSheet("background-color: #ffffff;")
label.setAutoFillBackground(True)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

20.11. Использование изображения в качестве фона

В качестве фона окна (или компонента) можно использовать изображение. Для этого необходимо получить текущую палитру компонента с помощью метода `palette()`, а затем вызвать метод `setBrush()` из класса `QPalette`. Формат метода:

```
setBrush([<Состояние>, ]<Роль>, <QBrush>)
```

Первые два параметра аналогичны соответствующим параметрам в методе `setColor()`, который мы рассматривали в предыдущем разделе. В третьем параметре указывается экземпляр класса `QBrush`. Форматы конструктора класса:

```
<Объект> = QBrush(<Стиль кисти>)
<Объект> = QBrush(<Цвет>[, <Стиль кисти>=SolidPattern])
<Объект> = QBrush(<Цвет>, <QPixmap>)
<Объект> = QBrush(<QPixmap>)
<Объект> = QBrush(<QImage>)
```

В параметре `<Стиль кисти>` указываются атрибуты из класса `QtCore.Qt`, задающие стиль кисти, например, `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `CrossPattern` и др. С помощью этого параметра можно сделать цвет сплошным (`SolidPattern`) или имеющим текстуру (например, атрибут `CrossPattern` задает текстуру в виде сетки).

В параметре `<Цвет>` указывается цвет кисти. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Например, установка сплошного цвета фона окна выглядит так:

```
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
    QtGui.QBrush(QtGui.QColor("#008800"), QtCore.Qt.SolidPattern))
window.setPalette(pal)
```

Параметры <QPixmap> и <QImage> позволяют передать объекты изображений. Конструкторы этих классов принимают путь к файлу, который может быть как абсолютным, так и относительным.

После настройки палитры необходимо вызвать метод setPalette() и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение True в метод setAutoFillBackground().

Указать, что изображение используется в качестве фона, можно также с помощью CSS-атрибутов background и background-image. С помощью CSS-атрибута background-repeat можно дополнительно указать режим повтора фонового рисунка. Он может принимать значения repeat, repeat-x (повтор только по горизонтали), repeat-y (повтор только по вертикали) и no-repeat (не повторяется).

Создадим окно с надписью. Для активного окна установим одно изображение (с помощью изменения палитры), а для надписи другое изображение (с помощью CSS-атрибута background-image) (листинг 20.9).

Листинг 20.9. Использование изображения в качестве фона

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Изображение в качестве фона")
window.resize(300, 100)
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QBrush(QtGui.QPixmap("img1.png")))
window.setPalette(pal)
label = QtGui.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignCenter)
label.setAutoFillBackground(True)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

20.12. Создание окна произвольной формы

Чтобы создать окно произвольной формы, нужно выполнить следующие шаги:

1. Создать изображение нужной формы с прозрачным фоном и сохранить его, например, в формате PNG.
2. Создать экземпляр класса QPixmap, передав конструктору класса абсолютный или относительный путь к изображению.
3. Установить изображение в качестве фона окна с помощью палитры.

4. Отделить альфа-канал с помощью метода `mask()` из класса `QPixmap`.
5. Передать маску в метод `setMask()` объекта окна.
6. Убрать рамку окна, например, передав комбинацию следующих флагов:

```
QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint
```

Если для создания окна используется класс `QLabel`, то вместо установки палитры можно передать экземпляр класса `QPixmap` в метод `setPixmap()`, а маску в метод `setMask()`.

В качестве примера создадим круглое окно с кнопкой, с помощью которой можно закрыть окно. Окно выведем без заголовка и границ (листинг 20.10).

Листинг 20.10. Создание окна произвольной формы

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowFlags(QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint)
window.setWindowTitle("Создание окна произвольной формы")
window.resize(300, 300)
pixmap = QtGui.QPixmap("fon.png")
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QBrush(pixmap))
pal.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
             QtGui.QBrush(pixmap))
window.setPalette(pal)
window.setMask(pixmap.mask())
button = QtGui.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 135)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
window.show()
sys.exit(app.exec_())
```

20.13. Всплывающие подсказки

При работе с программой у пользователя могут возникать вопросы о предназначении того или иного компонента. Обычно для информирования пользователя используются надписи, расположенные над компонентом или перед ним. Но часто место в окне либо ограничено, либо вывод таких надписей испортит весь дизайн окна. В таких случаях принято выводить текст подсказки в отдельном окне без рамки при наведении указателя мыши на компонент. После выведения указателя окно должно автоматически закрываться.

В PyQt нет необходимости создавать окно с подсказкой самому и следить за перемещениями указателя мыши. Весь процесс автоматизирован и максимально упрощен. Чтобы создать

всплывающие подсказки для окна или любого другого компонента и управлять ими, нужно воспользоваться следующими методами из класса `QWidget`:

- `setToolTip(<Текст>)` — задает текст всплывающей подсказки. В качестве параметра можно указать простой текст или текст в формате HTML. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст справки. Обычно этот метод используется для вывода информации большего объема, чем во всплывающей подсказке. Чтобы отобразить текст справки, необходимо сделать компонент активным и нажать комбинацию клавиш <Shift>+<F1>. У диалоговых окон в заголовке окна есть кнопка **Справка**. После нажатия этой кнопки вид курсора изменится на стрелку со знаком вопроса. Чтобы в этом случае отобразить текст справки, следует щелкнуть мышью на компоненте. В качестве параметра можно указать простой текст или текст в формате HTML. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- `whatsThis()` — возвращает текст справки.

Создадим окно с кнопкой и зададим для них текст всплывающих подсказок и текст справки (листинг 20.11).

Листинг 20.11. Всплывающие подсказки

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Всплывающие подсказки")
window.resize(300, 70)
button = QtGui.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.setToolTip("Это всплывающая подсказка для кнопки")
window.setToolTip("Это всплывающая подсказка для окна")
button.setWhatsThis("Это справка для кнопки")
window.setWhatsThis("Это справка для окна")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
window.show()
sys.exit(app.exec_())
```

20.14. Закрытие окна из программы

В предыдущих разделах для закрытия окна мы использовали слот `quit()` и метод `exit([returnCode=0])` объекта приложения. Однако эти методы не только закрывают текущее окно, но и завершают выполнение всего приложения. Чтобы закрыть только текущее окно, следует воспользоваться методом `close()` из класса `QWidget`. Метод возвращает значение `True`, если окно успешно закрыто, и `False` — в противном случае. Закрыть сразу все окна приложения позволяет слот `closeAllWindows()` из класса `QApplication`.

Если для окна атрибут `WA_DeleteOnClose` из класса `QtCore.Qt` установлен в истинное значение, то после закрытия окна объект окна будет автоматически удален. Если атрибут имеет ложное значение, то окно просто скрывается. Значение атрибута можно изменить с помощью метода `setAttribute()`:

```
window.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
```

После вызова метода `close()` или нажатия кнопки **Закрыть** в заголовке окна генерируется событие `QEvent.Close`. Если внутри класса определить метод с предопределенным названием `closeEvent()`, то это событие можно перехватить и обработать. В качестве параметра метод принимает объект класса `QCloseEvent`, который содержит методы `accept()` (позволяет закрыть окно) и `ignore()` (запрещает закрытие окна). Вызывая эти методы, можно контролировать процесс закрытия окна.

В качестве примера закроем окно при нажатии кнопки (листинг 20.12).

Листинг 20.12. Закрытие окна из программы

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Закрытие окна из программы")
window.resize(300, 70)
button = QtGui.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       window, QtCore.SLOT("close()"))
window.show()
sys.exit(app.exec_())
```

ГЛАВА 21

Обработка сигналов и событий

При взаимодействии пользователя с окном происходят события. В ответ на события система генерирует определенные сигналы. *Сигналы* — это своего рода извещения системы о том, что пользователь выполнил какое-либо действие или в самой системе возникло некоторое условие. Сигналы являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик сигнала, как удалить обработчик, а также уметь правильно обработать событие. Какие сигналы генерирует тот или иной компонент, мы будем рассматривать при изучении конкретного компонента.

21.1. Назначение обработчиков сигналов

Чтобы обработать какой-либо сигнал, необходимо сопоставить ему функцию или метод класса, которые будут вызваны при наступлении события. Назначить обработчик позволяет статический метод `connect()` из класса `QObject`. Форматы метода:

```
connect(<Объект>, <Сигнал>, <Обработчик>[, <ConnectionType>])
connect(<Объект1>, <Сигнал>, <Объект2>, <Слот>[, <ConnectionType>])
connect(<Объект1>, <Сигнал>, <Объект2>, <Сигнал>[, <ConnectionType>])
```

Кроме того, существует обычный (не статический) метод `connect()`:

```
<Объект2>.connect(<Объект1>, <Сигнал>, <Слот>[, <ConnectionType>])
```

Первый формат позволяет назначить обработчик сигнала `<Сигнал>`, возникшего при изменении статуса объекта `<Объект>`. Если обработчик успешно назначен, то метод возвращает значение `True`. Для одного сигнала можно назначить несколько обработчиков, которые будут вызываться в порядке назначения в программе. В параметре `<Сигнал>` указывается результат выполнения функции `SIGNAL()`. Формат функции:

```
QtCore.SIGNAL("<Название сигнала>([Тип параметров])")
```

Каждый компонент имеет определенный набор сигналов, например, при щелчке на кнопке генерируется сигнал `clicked(bool=0)`. Внутри круглых скобок могут быть указаны типы параметров, которые передаются в обработчик. Если параметров нет, то указываются только круглые скобки. Пример указания сигнала без параметров:

```
QtCore.SIGNAL("clicked()")
```

В этом случае обработчик не принимает никаких параметров. Указание сигнала с параметром выглядит следующим образом:

```
QtCore.SIGNAL("clicked(bool)")
```

В этом случае обработчик должен принимать один параметр, значение которого всегда будет равно `0` (`False`), т. к. это значение по умолчанию для сигнала `clicked()`.

В параметре `<Обработчик>` можно указать:

- ссылку на пользовательскую функцию;
- ссылку на метод класса;
- ссылку на экземпляр класса. В этом случае внутри класса должен существовать метод `__call__()`.

Пример обработки щелчка на кнопке приведен в листинге 21.1.

Листинг 21.1. Варианты назначения пользовательского обработчика

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

def on_clicked():
    print("Кнопка нажата. Функция on_clicked()")

class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x =", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Нажми меня")
# Назначаем обработчиком функцию
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), on_clicked)
# Назначаем обработчиком метод класса
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       obj.on_clicked)
# Передача параметра в обработчик
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), MyClass(10))
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), MyClass(5))
button.show()
sys.exit(app.exec_())
```

Результат выполнения в окне консоли при одном щелчке на кнопке:

```
Кнопка нажата. Функция on_clicked()
Кнопка нажата. Метод MyClass.on_clicked()
Кнопка нажата. Метод MyClass.__call__()
x = 10
Кнопка нажата. Метод MyClass.__call__()
x = 5
```

Второй формат метода `connect()` назначает в качестве обработчика метод Qt-объекта `<Объект2>`. Обычно используется для назначения стандартного метода из класса, входящего в состав библиотеки Qt. В качестве примера при щелчке на кнопке завершим работу приложения (листинг 21.2).

Листинг 21.2. Завершение работы приложения при щелчке на кнопке

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Завершить работу")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       app,    QtCore.SLOT("quit()"))
button.show()
sys.exit(app.exec_())
```

Как видно из примера, в третьем параметре метода `connect()` указывается объект приложения, а в четвертом параметре в функцию `SLOT()` передается название метода `quit()` в виде строки. Благодаря гибкости языка Python данное назначение обработчика можно записать иначе:

```
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), app.quit)
```

Третий формат метода `connect()` позволяет передать сигнал другому объекту. Рассмотрим передачу сигнала на примере (листинг 21.3).

Листинг 21.3. Передача сигнала другому объекту

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.button1 = QtGui.QPushButton("Кнопка 1. Нажми меня")
        self.button2 = QtGui.QPushButton("Кнопка 2")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.resize(300, 100)
        # Передача сигнала от кнопки 1 к кнопке 2
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.button2, QtCore.SIGNAL('clicked()'))
        # Способ 1 (4 параметра)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self, QtCore.SLOT("on_clicked_button2()"))
```

```
        # Способ 2 (3 параметра)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button2()"))
    @QtCore.pyqtSlot()
    def on_clicked_button2(self):
        print("Сигнал получен кнопкой 2")

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

В этом примере мы создали класс `MyWindow`, который наследует класс `QWidget`. В методе инициализации `__init__()` вначале вызывается конструктор базового класса и создаются две кнопки. Далее создается вертикальный контейнер и в него добавляются объекты кнопок с помощью метода `addWidget()`. С помощью метода `setLayout()` вертикальный контейнер добавляется в основное окно. Затем назначаются обработчики событий для кнопок. Обратите внимание на то, что метод `connect()` вызывается как метод нашего класса. Это возможно потому, что большинство PyQt-классов наследуют класс `QObject`, в котором определен метод `connect()`. Обработка нажатия кнопки производится с помощью метода `on_clicked_button2()`, который превращен декоратором `@QtCore.pyqtSlot()` в одноименный слот.

При нажатии первой кнопки производится вызов первого обработчика, который перенаправляет сигнал на вторую кнопку. Назначение перенаправления, соответствующее третьему формату метода `connect()`, выглядит так:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             self.button2, QtCore.SIGNAL('clicked()'))
```

После перенаправления сигнала вызывается обработчик второй кнопки. Для второй кнопки мы назначили обработчик двумя способами. Первый способ соответствует второму формату метода `connect()`:

```
self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button2()"))
```

Второй способ соответствует четвертому формату метода `connect()`:

```
self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             QtCore.SLOT("on_clicked_button2()"))
```

Необязательный параметр `<ConnectionType>` определяет тип соединения между сигналом и обработчиком. На этот параметр следует обратить особое внимание при использовании нескольких потоков в приложении, т. к. изменять GUI-поток из другого потока нельзя. В параметре можно указать один из следующих атрибутов из класса `QtCore.Qt`:

- `AutoConnection` — `0` — значение по умолчанию. Если источник сигнала и обработчик находятся в одном потоке, то эквивалентно значению `DirectConnection`, а если в разных потоках — то `QueuedConnection`;
- `DirectConnection` — `1` — обработчик вызывается сразу после генерации сигнала. Обработчик выполняется в потоке источника сигнала;

- QueuedConnection — 2 — сигнал помещается в очередь обработки событий. Обработчик выполняется в потоке приемника сигнала;
- BlockingQueuedConnection — 4 — аналогично значению QueuedConnection, но пока сигнал не обработан, поток будет заблокирован. Обратите внимание на то, что источник сигнала и обработчик должны быть обязательно расположены в разных потоках;
- UniqueConnection — 0x80 — аналогично значению AutoConnection, но обработчик можно назначить только если он не был назначен ранее. Например, если изменить способы назначения обработчика из предыдущего примера для кнопки button2 следующим образом, то второй обработчик назначен не будет:

```
st = self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button2()"),
             QtCore.Qt.UniqueConnection)
print(st)
st = self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button2()"),
             QtCore.Qt.UniqueConnection)
print(st)
```

Результат:

```
True
False
```

- AutoCompatConnection — 3 — значение использовалось по умолчанию в Qt 3.

21.2. Блокировка и удаление обработчика

Для блокировки и удаления обработчиков предназначены следующие методы из класса QObject:

- blockSignals(<Флаг>) — временно блокирует прием сигналов, если параметр имеет значение True, и снимает блокировку, если параметр имеет значение False. Метод возвращает логическое представление предыдущего состояния соединения;
- signalsBlocked() — метод возвращает значение True, если блокировка установлена, и False — в противном случае;
- disconnect() — удаляет обработчик. Метод является статическим и доступен без создания экземпляра класса. Форматы метода:

```
disconnect(<Объект>, <Сигнал>, <Обработчик>)
disconnect(<Объект1>, <Сигнал>, <Объект2>, <Слот>)
```

Первый формат метода disconnect() позволяет удалить <Обработчик> сигнала. В параметре <Обработчик> можно указать ссылку на пользовательскую функцию или метод класса. Второй формат позволяет удалить слот, связанный с объектом. Если обработчик успешно удален, то метод disconnect() возвращает значение True. Значения, указанные в методе disconnect(), должны совпадать со значениями, используемыми при назначении обработчика. Например, если обработчик назначался таким образом:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button1()"))
```

или таким:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             QtCore.SLOT("on_clicked_button1()"))
```

то удалить его можно следующим образом:

```
self.disconnect(self.button1, QtCore.SIGNAL("clicked()"),
                self, QtCore.SLOT("on_clicked_button1()"))
```

Эти два способа назначения обработчика являются эквивалентными. В первом способе вызывается статический метод `connect()`. В этом случае ссылка на экземпляр класса передается явным образом в третьем параметре. Во втором способе используется метод класса `QObject`. В этом случае ссылка на экземпляр класса передается неявным образом. Метод `disconnect()` является статическим методом, поэтому ссылку на экземпляр класса необходимо передавать явным образом через третий параметр.

Создадим окно с четырьмя кнопками (листинг 21.4). Для кнопки **Нажми меня** назначим обработчик для сигнала `clicked()`. Чтобы информировать о нажатии кнопки, выведем сообщение в окно консоли. Для кнопок **Блокировать**, **Разблокировать** и **Удалить обработчик** создадим обработчики, которые будут изменять статус обработчика для кнопки **Нажми меня.**

Листинг 21.4. Блокировка и удаление обработчика

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Блокировка и удаление обработчика")
        self.resize(300, 150)
        self.button1 = QtGui.QPushButton("Нажми меня")
        self.button2 = QtGui.QPushButton("Блокировать")
        self.button3 = QtGui.QPushButton("Разблокировать")
        self.button4 = QtGui.QPushButton("Удалить обработчик")
        self.button3.setEnabled(False)
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        vbox.addWidget(self.button4)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button1()"))
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button2()"))
        self.connect(self.button3, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button3()"))
        self.connect(self.button4, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button4()"))
```

```
    @QtCore.pyqtSlot()
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
    @QtCore.pyqtSlot()
    def on_clicked_button2(self):
        self.button1.blockSignals(True)
        self.button2.setEnabled(False)
        self.button3.setEnabled(True)
    @QtCore.pyqtSlot()
    def on_clicked_button3(self):
        self.button1.blockSignals(False)
        self.button2.setEnabled(True)
        self.button3.setEnabled(False)
    @QtCore.pyqtSlot()
    def on_clicked_button4(self):
        self.disconnect(self.button1,
                        QtCore.SIGNAL("clicked()"), self,
                        QtCore.SLOT("on_clicked_button1()"))
        self.button2.setEnabled(False)
        self.button3.setEnabled(False)
        self.button4.setEnabled(False)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Если после отображения окна нажать кнопку **Нажми меня**, то в окно консоли будет выведена строка "Нажата кнопка button1". Нажатие кнопки **Блокировать** производит блокировку обработчика. Теперь при нажатии кнопки **Нажми меня** никаких сообщений в окно консоли не выводится. Отменить блокировку можно с помощью кнопки **Разблокировать**. Нажатие кнопки **Удалить обработчик** производит полное удаление обработчика. В этом случае, чтобы обрабатывать нажатие кнопки **Нажми меня**, необходимо заново назначить обработчик.

Также можно отключить генерацию сигнала, сделав компонент недоступным с помощью следующих методов из класса `QWidget`:

- `setEnabled(<Флаг>)` — если в параметре указано значение `False`, то компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `True`;
- `setDisabled(<Флаг>)` — если в параметре указано значение `True`, то компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `False`.

Проверить, доступен компонент или нет, позволяет метод `isEnabled()`. Метод возвращает значение `True`, если компонент доступен, и `False` — в противном случае.

21.3. Генерация сигнала из программы

В некоторых случаях необходимо вызвать сигнал из программы. Например, при заполнении последнего текстового поля и нажатии клавиши <Enter> можно имитировать нажатие кнопки и тем самым запустить обработчик этого сигнала. Выполнить генерацию сигнала из программы позволяет метод `emit()` из класса `QObject`. Формат метода:

```
<Объект>.emit(<Сигнал>[, <Данные через запятую>])
```

В параметре `<Сигнал>` указывается результат выполнения функции `SIGNAL()`. Формат функции:

```
QtCore.SIGNAL("<Название сигнала>([Тип параметров])")
```

Через второй необязательный параметр можно передать дополнительные данные через запятую. Метод всегда вызывается через объект, которому посылается сигнал.

В качестве примера создадим окно с двумя кнопками (листинг 21.5). Для этих кнопок назначим обработчики сигнала `clicked()` (нажатие кнопки). Внутри обработчика щелчка на первой кнопке сгенерируем два сигнала. Первый сигнал будет имитировать нажатие второй кнопки, а второй сигнал будет пользовательским, привязанным ко второй кнопке. Внутри обработчиков выведем сообщения в окно консоли.

Листинг 21.5. Генерация сигнала из программы

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Генерация сигнала из программы")
        self.resize(300, 100)
        self.button1 = QtGui.QPushButton("Нажми меня")
        self.button2 = QtGui.QPushButton("Кнопка 2")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button1)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button2)
        self.connect(self.button2, QtCore.SIGNAL("mysignal"),
                     self.on_mysignal)
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
        # Генерируем сигналы
        self.button2.emit(QtCore.SIGNAL("clicked(bool)"), False)
        self.button2.emit(QtCore.SIGNAL("mysignal"), 10, 20)
    def on_clicked_button2(self):
        print("Нажата кнопка button2")
```

```
    def on_mysignal(self, x, y):
        print("Обработан пользовательский сигнал mysignal()")
        print("x =", x, "y =", y)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Результат выполнения после нажатия первой кнопки:

```
Нажата кнопка button1
Нажата кнопка button2
Обработан пользовательский сигнал mysignal()
x = 10 y = 20
```

Назначение обработчиков нажатия кнопки производится обычным образом, а вот назначение обработчика пользовательского сигнала имеет свои отличия. Как видно из примера, после названия пользовательского сигнала нет круглых скобок. В этом случае при генерации сигнала мы можем передать в обработчик произвольное количество значений любого типа. В нашем примере мы передаем два числа:

```
self.button2.emit(QtCore.SIGNAL("mysignal"), 10, 20)
```

При использовании пользовательских сигналов допускается также указывать тип передаваемых данных внутри круглых скобок. Пример назначения обработчика с двумя параметрами:

```
self.connect(self.button2, QtCore.SIGNAL("mysignal(int, int)"),
             self.on_mysignal)
```

Генерация сигнала выглядит так:

```
self.button2.emit(QtCore.SIGNAL("mysignal(int, int)"), 10, 20)
```

Вместо конкретного типа можно указать тип `PyQt_PyObject`. В этом случае при генерации сигнала допускается передавать данные любого типа. Пример назначения обработчика с одним параметром:

```
self.connect(self.button2, QtCore.SIGNAL("mysignal(PyQt_PyObject)"),
             self.on_mysignal)
```

Пример генерации сигнала:

```
self.button2.emit(QtCore.SIGNAL("mysignal(PyQt_PyObject)"), 20)
self.button2.emit(QtCore.SIGNAL("mysignal(PyQt_PyObject)"), [1, "2"])
```

Сгенерировать сигнал можно не только с помощью метода `emit()`. Некоторые компоненты предоставляют методы, которые посылают сигнал. Например, у кнопок существует метод `click()`. Используя этот метод, инструкцию:

```
self.button2.emit(QtCore.SIGNAL("clicked(bool)"), False)
```

можно записать следующим образом:

```
self.button2.click()
```

Более подробно такие методы мы будем рассматривать при изучении конкретных компонентов.

21.4. Новый стиль назначения и удаления обработчиков

В PyQt существует альтернативный способ назначения и удаления обработчиков, а также генерации сигнала. Для назначения обработчика используются следующие форматы метода `connect()`:

```
<Компонент>.<Сигнал>.connect(<Обработчик>[, type=<ConnectionType>])
<Компонент>.<Сигнал>[<Тип>].connect(<Обработчик>[,
                                    type=<ConnectionType>])
```

В параметре `<Компонент>` указывается компонент, для которого назначается обработчик сигнала `<Сигнал>`. Название сигнала указывается без круглых скобок. Если одноименные сигналы отличаются типом параметров, то внутри квадратных скобок можно указать для какого типа назначается обработчик. В параметре `<Обработчик>` передается ссылка на функцию или метод класса, а в именованном параметре `type` можно дополнительно указать тип соединения. По умолчанию параметр `type` имеет значение `AutoConnection`. Пример:

```
self.button1.clicked.connect(self.on_clicked_button1)
self.button2.clicked["bool"].connect(self.on_clicked_button2)
```

Для удаления обработчика используются следующие форматы метода `disconnect()`:

```
<Компонент>.<Сигнал>.disconnect([<Обработчик>])
<Компонент>.<Сигнал>[<Тип>].disconnect([<Обработчик>])
```

Если параметр `<Обработчик>` не указан, то удаляются все обработчики, назначенные ранее, в противном случае удаляется только указанный обработчик. Пример:

```
self.button1.clicked.disconnect()
self.button2.clicked["bool"].disconnect(self.on_clicked_button2)
```

Для генерации сигнала используются следующие форматы метода `emit()`:

```
<Компонент>.<Сигнал>.emit([<Данные>])
<Компонент>.<Сигнал>[<Тип>].emit([<Данные>])
```

Пример генерации сигнала:

```
self.button2.clicked.emit(False)
self.button2.clicked["bool"].emit(False)
```

Для регистрации пользовательских сигналов используется функция `pyqtSignal()` из модуля `QtCore`. Формат функции:

```
<Объект сигнала> = pyqtSignal(*types[, name])
```

В параметре `types` через запятую задаются названия типов данных в Python, например `bool` или `int`, которые принимает метод:

```
mysignal = QtCore.pyqtSignal(int, name="mysignal")
```

При указании типа данных C++ его название необходимо указать в виде строки:

```
mysignal = QtCore.pyqtSignal("QString", name="mysignal")
```

Если метод не принимает параметров, то параметр types не указывается. Сигнал может иметь несколько перегруженных версий. В этом случае типы параметров указываются внутри квадратных скобок. Пример сигнала, передающего данные типа int или str:

```
mysignal = QtCore.pyqtSignal([int], [str], name="mysignal")
```

В именованном параметре name можно передать название сигнала в виде строки. Это название будет использоваться вместо названия метода. Если параметр name не задан, то название сигнала будет совпадать с названием метода. Метод возвращает объект сигнала.

Создадим окно с тремя кнопками и назначим обработчики разными способами (листинг 21.6). При нажатии первой кнопки имитируем нажатие второй кнопки. При нажатии третьей кнопки удалим все обработчики и сделаем кнопку недоступной.

Листинг 21.6. Альтернативный способ назначения и удаления обработчиков

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    mysignal = QtCore.pyqtSignal([int], [str], name="mysignal")
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Обработка сигналов")
        self.resize(300, 100)
        self.button1 = QtGui.QPushButton("Нажми меня")
        self.button2 = QtGui.QPushButton("Кнопка 2")
        self.button3 = QtGui.QPushButton("Удалить обработчики")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        self.setLayout(vbox)
        # Назначение обработчиков
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked["bool"].connect(self.on_clicked_button2)
        self.button3.clicked.connect(self.on_clicked_button3)
        self.mysignal[int].connect(self.on_mysignal)
        self.mysignal[str].connect(self.on_mysignal)
    def on_clicked_button1(self, status):
        print("Нажата кнопка button1", status)
        # Генерация сигнала
        self.button2.clicked["bool"].emit(False)
        self.mysignal[int].emit(10)
        self.mysignal[str].emit("строка")
    def on_clicked_button2(self, status):
        print("Нажата кнопка button2", status)
    def on_clicked_button3(self):
        print("Нажата кнопка button3")
        # Удаление обработчиков
        self.button1.clicked.disconnect()
```

```
        self.button2.clicked["bool"].disconnect(
                                    self.on_clicked_button2)
        self.mysignal[int].disconnect()
        self.mysignal[str].disconnect()
        self.button3.setEnabled(False)
    def on_mysignal(self, n):
        print("Обработан пользовательский сигнал\n n =", n)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Чтобы пользовательский метод сделать слотом, необходимо перед реализацией метода добавить декоратор `@pyqtSlot()`. Обратите внимание на то, что класс, в котором находится метод, должен быть наследником класса `QObject` или наследником класса, который в свою очередь наследует класс `QObject`. Формат декоратора:

```
@QtCore.pyqtSlot(*types, name=None, result=None)
```

В параметре `types` через запятую задаются названия типов данных в Python, например `bool` или `int`, которые принимает метод. При указании типа данных C++ его название необходимо указать в виде строки. Если метод не принимает параметров, то параметр `types` не указывается. В именованном параметре `name` можно передать название слота в виде строки. Это название будет использоваться вместо названия метода. Если параметр `name` не задан, то название слота будет совпадать с названием метода. Именованный параметр `result` предназначен для указания типа данных, возвращаемых методом. Если параметр не задан, то метод ничего не возвращает. Чтобы создать перегруженную версию слота, декоратор указывается последовательно несколько раз с разными типами данных. Пример использования декоратора `@pyqtSlot()` приведен в листинге 21.7.

Листинг 21.7. Использование декоратора `@pyqtSlot()`

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

class MyClass(QtCore.QObject):
    def __init__(self):
        QtCore.QObject.__init__(self)
    @QtCore.pyqtSlot()
    def on_clicked(self):
        print("Кнопка нажата. Слот on_clicked()")
    @QtCore.pyqtSlot(bool, name="myslot")
    def on_clicked2(self, status):
        print("Кнопка нажата. Слот myslot(bool)", status)

obj = MyClass()
app = QtGui.QApplication(sys.argv)
```

```
button = QtGui.QPushButton("Нажми меня")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       obj, QtCore.SLOT("on_clicked()"))
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked(bool)"),
                       obj, QtCore.SLOT("myslot(bool)"))
button.show()
sys.exit(app.exec_())
```

21.5. Передача данных в обработчик

При назначении обработчика в метод `connect()` передается ссылка на функцию или метод. Если после названия функции указать внутри круглых скобок какой-либо параметр, то это приведет к вызову функции и вместо ссылки будет передан результат выполнения функции, что приведет к ошибке. Передать данные в обработчик можно следующими способами:

- указать lambda-функцию, внутри которой вызывается обработчик с параметром. Пример передачи числа `10`:

  ```
  self.connect(self.button1, QtCore.SIGNAL("clicked()"),
               lambda : self.on_clicked_button1(10))
  ```

  Если значение вычисляется динамически и передается с помощью переменной, то переменную следует указывать как значение по умолчанию в lambda-функции, иначе внутри lambda-функции сохраняется ссылка на переменную, а не ее значение. Пример:

  ```
  y = 10
  self.connect(self.button1, QtCore.SIGNAL("clicked()"),
               lambda x=y: self.on_clicked_button1(x))
  ```

- передать ссылку на экземпляр класса, внутри которого определен метод `__call__()`. Значение указывается при создании экземпляра класса. Пример класса:

  ```
  class MyClass():
      def __init__(self, x=0):
          self.x = x
      def __call__(self):
          print("x =", self.x)
  ```

  Пример передачи параметра в обработчик:

  ```
  self.connect(self.button1, QtCore.SIGNAL("clicked()"),
               MyClass(10))
  ```

- передать ссылку на обработчик и данные в функцию `partial()` из модуля `functools`. Формат функции:

  ```
  from functools import partial
  partial(func[, *args][, **keywords])
  ```

  Пример передачи параметра в обработчик:

  ```
  self.connect(self.button1, QtCore.SIGNAL("clicked()"),
               partial(self.on_clicked_button1, 10))
  ```

  Если при генерации сигнала передается предопределенное значение, то оно будет доступно в обработчике после остальных параметров. Назначим обработчик сигнала `clicked(bool)` и передадим число:

```
self.connect(self.button1, QtCore.SIGNAL("clicked(bool)"),
             partial(self.on_clicked_button1, 10))
```

Обработчик будет иметь следующий вид:

```
def on_clicked_button1(self, x, status):
    print("Нажата кнопка button1", x, status)
```

Результат выполнения:

```
Нажата кнопка button1 10 False
```

21.6. Использование таймеров

Таймеры позволяют через определенный интервал времени выполнять метод с предопределенным названием `timerEvent()`. Для назначения таймера используется метод `startTimer()` из класса `QObject`. Формат метода:

```
<Id> = <Объект>.startTimer(<Интервал>)
```

Метод `startTimer()` возвращает идентификатор таймера, с помощью которого можно остановить таймер. Параметр `<Интервал>` задает промежуток времени в миллисекундах, по истечении которого выполняется метод `timerEvent()`. Формат метода `timerEvent()`:

```
timerEvent(self, <Объект класса QTimerEvent>)
```

Внутри метода `timerEvent()` можно получить идентификатор таймера с помощью метода `timerId()` объекта класса `QTimerEvent`. Формат метода:

```
<Id> = <Объект класса QTimerEvent>.timerId()
```

Минимальное значение интервала зависит от операционной системы. Если в параметре `<Интервал>` указать значение `0`, то таймер будет срабатывать много раз при отсутствии других необработанных событий.

Чтобы остановить таймер, необходимо воспользоваться методом `killTimer()` из класса `QObject`. Формат метода:

```
<Объект>.killTimer(<Id>)
```

В качестве параметра указывается идентификатор, возвращаемый методом `startTimer()`.

Создадим часы в окне, которые будут отображать текущее системное время с точностью до секунды, и добавим возможность запуска и остановки часов с помощью кнопок (листинг 21.8).

Листинг 21.8. Вывод времени в окне с точностью до секунды

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import time

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Часы в окне")
        self.resize(200, 100)
```

```
        self.timer_id = 0
        self.label = QtGui.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button1 = QtGui.QPushButton("Запустить")
        self.button2 = QtGui.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button1)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button2)
    def on_clicked_button1(self):
        self.timer_id = self.startTimer(1000) # 1 секунда
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        if self.timer_id:
            self.killTimer(self.timer_id)
            self.timer_id = 0
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def timerEvent(self, event):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Вместо методов `startTimer()` и `killTimer()` можно воспользоваться классом `QTimer` из модуля `QtCore`. Конструктор класса имеет следующий формат:

```
<Объект> = QTimer([parent=None])
```

Методы класса:

- `setInterval(<Интервал>)` — задает промежуток времени в миллисекундах, по истечении которого генерируется сигнал `timeout()`. Минимальное значение интервала зависит от операционной системы. Если в параметре `<Интервал>` указать значение `0`, то таймер будет срабатывать много раз при отсутствии других необработанных сигналов;
- `start([<Интервал>])` — запускает таймер. В необязательном параметре можно указать промежуток времени в миллисекундах. Если параметр не указан, то используется значение, возвращаемое методом `interval()`;
- `stop()` — останавливает таймер;

- `setSingleShot(<Флаг>)` — если в параметре указано значение `True`, то таймер будет срабатывать только один раз, в противном случае — многократно;
- `interval()` — возвращает установленный интервал;
- `timerId()` — возвращает идентификатор таймера или значение `-1`;
- `isSingleShot()` — возвращает значение `True`, если таймер будет срабатывать только один раз, и `False` — в противном случае;
- `isActive()` — возвращает значение `True`, если таймер генерирует сигналы, и `False` — в противном случае.

Переделаем предыдущий пример и используем класс `QTimer` вместо методов `startTimer()` и `killTimer()` (листинг 21.9).

Листинг 21.9. Использование класса `QTimer`

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import time

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Использование класса QTimer")
        self.resize(200, 100)
        self.label = QtGui.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button1 = QtGui.QPushButton("Запустить")
        self.button2 = QtGui.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button1)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button2)
        self.timer = QtCore.QTimer()
        self.connect(self.timer, QtCore.SIGNAL("timeout()"),
                     self.on_timeout);
    def on_clicked_button1(self):
        self.timer.start(1000) # 1 секунда
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        self.timer.stop()
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
```

```
    def on_timeout(self):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Кроме перечисленных методов в классе `QTimer` определен статический метод `singleShot()`, предназначенный для вызова указанной функции, метода или слота через определенный промежуток времени. Таймер срабатывает только один раз. Форматы метода:

```
QtCore.QTimer.singleShot(<Интервал>, <Обработчик>)
QtCore.QTimer.singleShot(<Интервал>, <Объект>, <Слот>)
```

Примеры использования статического метода `singleShot()`:

```
QtCore.QTimer.singleShot(1000, self.on_timeout)
QtCore.QTimer.singleShot(1000, QtGui.qApp, QtCore.SLOT("quit()"))
```

21.7. Перехват всех событий

В предыдущих разделах мы рассмотрели обработку сигналов, которые позволяют обмениваться сообщениями между компонентами. Обработка внешних событий, например нажатий клавиш, осуществляется несколько иначе. Чтобы обработать событие, необходимо наследовать класс и переопределить в нем метод со специальным названием, например, чтобы обработать нажатие клавиши, следует переопределить метод `keyPressEvent()`. Специальные методы принимают объект, содержащий детальную информацию о событии, например код нажатой клавиши. Все эти объекты являются наследниками класса `QEvent` и наследуют следующие методы:

- `accept()` — устанавливает флаг, который является признаком согласия с дальнейшей обработкой события, например, если в методе `closeEvent()` вызывать метод `accept()` через объект события, то окно будет закрыто. Флаг обычно устанавливается по умолчанию;
- `ignore()` — сбрасывает флаг, например, если в методе `closeEvent()` вызывать метод `ignore()` через объект события, то окно закрыто не будет;
- `setAccepted(<Флаг>)` — если в качестве параметра указано значение `True`, то флаг будет установлен (аналогично вызову метода `accept()`), а если `False` — то сброшен (аналогично вызову метода `ignore()`);
- `isAccepted()` — возвращает текущее состояние флага;
- `registerEventType([<Число>])` — позволяет зарегистрировать пользовательский тип события. Метод возвращает идентификатор зарегистрированного события. В качестве параметра можно указать значение в пределах от `QEvent.User` (`1000`) до `QEvent.MaxUser` (`65535`). Метод является статическим;
- `spontaneous()` — возвращает `True`, если событие сгенерировано системой, и `False`, если событие сгенерировано внутри программы искусственно;

- `type()` — возвращает тип события. Перечислим основные типы событий (полный список смотрите в документации по классу `QEvent`):
  - `0` — нет события;
  - `1` — `Timer` — событие таймера;
  - `2` — `MouseButtonPress` — нажата кнопка мыши;
  - `3` — `MouseButtonRelease` — отпущена кнопка мыши;
  - `4` — `MouseButtonDblClick` — двойной щелчок мышью;
  - `5` — `MouseMove` — перемещение мыши;
  - `6` — `KeyPress` — клавиша на клавиатуре нажата;
  - `7` — `KeyRelease` — клавиша на клавиатуре отпущена;
  - `8` — `FocusIn` — получен фокус ввода с клавиатуры;
  - `9` — `FocusOut` — потерян фокус ввода с клавиатуры;
  - `10` — `Enter` — указатель мыши входит в область компонента;
  - `11` — `Leave` — указатель мыши покидает область компонента;
  - `12` — `Paint` — перерисовка компонента;
  - `13` — `Move` — позиция компонента изменилась;
  - `14` — `Resize` — изменился размер компонента;
  - `17` — `Show` — компонент отображен;
  - `18` — `Hide` — компонент скрыт;
  - `19` — `Close` — окно закрыто;
  - `24` — `WindowActivate` — окно стало активным;
  - `25` — `WindowDeactivate` — окно стало неактивным;
  - `26` — `ShowToParent` — дочерний компонент отображен;
  - `27` — `HideToParent` — дочерний компонент скрыт;
  - `31` — `Wheel` — прокручено колесико мыши;
  - `40` — `Clipboard` — содержимое буфера обмена изменено;
  - `60` — `DragEnter` — указатель мыши входит в область компонента при операции перетаскивания;
  - `61` — `DragMove` — производится операция перетаскивания;
  - `62` — `DragLeave` — указатель мыши покидает область компонента при операции перетаскивания;
  - `63` — `Drop` — операция перетаскивания завершена;
  - `68` — `ChildAdded` — добавлен дочерний компонент;
  - `69` — `ChildPolished` — производится настройка дочернего компонента;
  - `71` — `ChildRemoved` — удален дочерний компонент;
  - `74` — `PolishRequest` — компонент настроен;
  - `75` — `Polish` — производится настройка компонента;

- 82 — ContextMenu — событие контекстного меню;
- 99 — ActivationChange — изменился статус активности окна верхнего уровня;
- 103 — WindowBlocked — окно блокировано модальным окном;
- 104 — WindowUnblocked — текущее окно разблокировано после закрытия модального окна;
- 105 — WindowStateChange — статус окна изменился;
- 121 — ApplicationActivate — приложение стало доступно пользователю;
- 122 — ApplicationDeactivate — приложение стало недоступно пользователю;
- 1000 — User — пользовательское событие;
- 65535 — MaxUser — максимальный идентификатор пользовательского события.

Перехват всех событий осуществляется с помощью метода с предопределенным названием event(self, <event>). Через параметр <event> доступен объект с дополнительной информацией о событии. Этот объект отличается для разных типов событий, например, для события MouseButtonPress объект будет экземпляром класса QMouseEvent, а для события KeyPress — экземпляром класса QKeyEvent. Какие методы содержат эти классы, мы рассмотрим в следующих разделах.

Внутри метода event() следует вернуть значение True, если событие принято, и False — в противном случае. Если возвращается значение True, то родительский компонент не получит событие. Чтобы продолжить распространение события, необходимо вызвать метод event() базового класса и передать ему текущий объект события. Обычно это делается так:

```
return QtGui.QWidget.event(self, e)
```

В этом случае пользовательский класс является наследником класса QWidget и переопределяет метод event(). Если вы наследуете другой класс, то следует вызывать метод именно этого класса. Например, при наследовании класса QLabel инструкция будет выглядеть так:

```
return QtGui.QLabel.event(self, e)
```

Пример перехвата нажатия клавиши, щелчка мышью и закрытия окна показан в листинге 21.10.

Листинг 21.10. Перехват всех событий

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def event(self, e):
        if e.type() == QtCore.QEvent.KeyPress:
            print("Нажата клавиша на клавиатуре")
            print("Код:", e.key(), ", текст:", e.text())
        elif e.type() == QtCore.QEvent.Close:
            print("Окно закрыто")
```

```
        elif e.type() == QtCore.QEvent.MouseButtonPress:
            print("Щелчок мышью. Координаты:", e.x(), e.y())
        return QtGui.QWidget.event(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

21.8. События окна

Перехватывать все события следует только в самом крайнем случае. В обычных ситуациях нужно использовать методы, предназначенные для обработки определенного события, например, чтобы обработать закрытие окна, достаточно переопределить метод `closeEvent()`. Какие методы требуется переопределять для обработки событий окна, мы сейчас и рассмотрим.

21.8.1. Изменение состояния окна

Отследить изменение состояния окна (сворачивание, разворачивание, сокрытие и отображение) позволяют следующие методы:

- `changeEvent(self, <event>)` — вызывается при изменении состояния окна, приложения или компонента. Иными словами, метод вызывается не только при изменении статуса окна, но и при изменении заголовка окна, палитры, статуса активности окна верхнего уровня, языка, локали и др. (полный список смотрите в документации). При обработке события `WindowStateChange` через параметр `<event>` доступен экземпляр класса `QWindowStateChangeEvent`. Этот класс содержит только метод `oldState()`, с помощью которого можно получить предыдущий статус окна;
- `showEvent(self, <event>)` — вызывается при отображении компонента. Через параметр `<event>` доступен экземпляр класса `QShowEvent`;
- `hideEvent(self, <event>)` — вызывается при сокрытии компонента. Через параметр `<event>` доступен экземпляр класса `QHideEvent`.

Для примера выведем текущее состояние окна в консоль при сворачивании, разворачивании, сокрытии и отображении окна (листинг 21.11).

Листинг 21.11. Отслеживание состояния окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
```

```
    def changeEvent(self, e):
        if e.type() == QtCore.QEvent.WindowStateChange:
            if self.isMinimized():
                print("Окно свернуто")
            elif self.isMaximized():
                print("Окно раскрыто до максимальных размеров")
            elif self.isFullScreen():
                print("Полноэкранный режим")
            elif self.isActiveWindow():
                print("Окно находится в фокусе ввода")
        QtGui.QWidget.changeEvent(self, e) # Отправляем дальше
    def showEvent(self, e):
        print("Окно отображено")
        QtGui.QWidget.showEvent(self, e)   # Отправляем дальше
    def hideEvent(self, e):
        print("Окно скрыто")
        QtGui.QWidget.hideEvent(self, e)   # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

21.8.2. Изменение положения окна и его размеров

При перемещении окна и изменении размеров вызываются следующие методы:

- `moveEvent(self, <event>)` — вызывается непрерывно при перемещении окна. Через параметр `<event>` доступен экземпляр класса `QMoveEvent`. Получить координаты окна позволяют следующие методы из класса `QMoveEvent`:
  - `pos()` — возвращает экземпляр класса `QPoint` с текущими координатами;
  - `oldPos()` — возвращает экземпляр класса `QPoint` с предыдущими координатами;
- `resizeEvent(self, <event>)` — вызывается непрерывно при изменении размеров окна. Через параметр `<event>` доступен экземпляр класса `QResizeEvent`. Получить размеры окна позволяют следующие методы из класса `QResizeEvent`:
  - `size()` — возвращает экземпляр класса `QSize` с текущими размерами;
  - `oldSize()` — возвращает экземпляр класса `QSize` с предыдущими размерами.

Пример обработки изменения положения окна и его размера показан в листинге 21.12.

Листинг 21.12. Изменение положения окна и его размера

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
```

```
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def moveEvent(self, e):
        print("x = {0}; y = {1}".format(e.pos().x(), e.pos().y()))
        QtGui.QWidget.moveEvent(self, e)   # Отправляем дальше
    def resizeEvent(self, e):
        print("w = {0}; h = {1}".format(e.size().width(),
                                        e.size().height()))
        QtGui.QWidget.resizeEvent(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

21.8.3. Перерисовка окна или его части

Когда компонент (или его часть) становится видимым, требуется выполнить перерисовку компонента или только его части. В этом случае вызывается метод с названием `paintEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QPaintEvent`, который содержит следующие методы:

- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, которую требуется перерисовать;
- `region()` — возвращает экземпляр класса `QRegion` с регионом, требующим перерисовки.

С помощью этих методов можно получать координаты области, которая, например, была ранее перекрыта другим окном и теперь оказалась в зоне видимости. Перерисовывая только область, а не весь компонент можно достичь более эффективного расходования ресурсов компьютера. Следует также заметить, что в целях эффективности, последовательность событий перерисовки может быть объединена в одно событие с общей областью перерисовки.

В некоторых случаях перерисовку окна необходимо выполнить вне зависимости от внешних действий системы или пользователя, например, при изменении каких-либо значений нужно обновить график. Вызвать событие перерисовки компонента позволяют следующие методы из класса `QWidget`:

- `repaint()` — незамедлительно вызывает метод `paintEvent()` для перерисовки компонента, при условии, что компонент не скрыт и обновление не запрещено с помощью метода `setUpdatesEnabled()`. Форматы метода:

  `repaint()`

  `repaint(<X>, <Y>, <Ширина>, <Высота>)`

  `repaint(<QRect>)`

  `repaint(<QRegion>)`

- `update()` — посылает сообщение о необходимости перерисовки компонента, при условии, что компонент не скрыт и обновление не запрещено. Событие будет обработано на следующей итерации основного цикла приложения. Если посылаются сразу несколько сообщений, то они объединяются в одно сообщение. Благодаря этому можно избежать

неприятного мерцания. Метод `update()` предпочтительнее использовать вместо метода `repaint()`. Форматы метода:

```
update()
update(<X>, <Y>, <Ширина>, <Высота>)
update(<QRect>)
update(<QRegion>)
```

21.8.4. Предотвращение закрытия окна

При нажатии кнопки **Закрыть** в заголовке окна или при вызове метода `close()` вызывается метод `closeEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QCloseEvent`. Чтобы предотвратить закрытие окна, необходимо вызвать метод `ignore()` через объект события, в противном случае — метод `accept()`.

В качестве примера при нажатии кнопки **Закрыть** в заголовке окна выведем стандартное диалоговое окно с запросом подтверждения закрытия окна (листинг 21.13). Если пользователь нажимает кнопку **Yes**, то закроем окно, а если кнопку **No** или просто закрывает диалоговое окно, то прервем закрытие окна.

Листинг 21.13. Обработка закрытия окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def closeEvent(self, e):
        result = QtGui.QMessageBox.question(self,
                      "Подтверждение закрытия окна",
                      "Вы действительно хотите закрыть окно?",
                      QtGui.QMessageBox.Yes | QtGui.QMessageBox.No,
                      QtGui.QMessageBox.No)
        if result == QtGui.QMessageBox.Yes:
            e.accept()
            QtGui.QWidget.closeEvent(self, e)
        else:
            e.ignore()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

21.9. События клавиатуры

События клавиатуры очень часто обрабатываются внутри приложений. Например, при нажатии клавиши <F1> можно вывести справочную информацию, при нажатии клавиши <Enter> в однострочном текстовом поле перенести фокус ввода на другой компонент и т. д. Рассмотрим события клавиатуры подробно.

21.9.1. Установка фокуса ввода

В один момент времени только один компонент (или вообще не одного) может иметь фокус ввода. Для управления фокусом ввода предназначены следующие методы из класса `QWidget`:

- `setFocus([<Причина>])` — устанавливает фокус ввода, если компонент находится в активном окне. В параметре `<Причина>` можно указать причину изменения фокуса ввода. Указываются следующие атрибуты из класса `QtCore.Qt`:
  - `MouseFocusReason` — `0` — фокус изменен с помощью мыши;
  - `TabFocusReason` — `1` — нажата клавиша <Tab>;
  - `BacktabFocusReason` — `2` — нажата комбинация клавиш <Shift>+<Tab>;
  - `ActiveWindowFocusReason` — `3` — окно стало активным или неактивным;
  - `PopupFocusReason` — `4` — открыто или закрыто всплывающее окно;
  - `ShortcutFocusReason` — `5` — нажата комбинация клавиш быстрого доступа;
  - `MenuBarFocusReason` — `6` — фокус изменился из-за меню;
  - `OtherFocusReason` — `7` — другая причина;
- `clearFocus()` — убирает фокус ввода с компонента;
- `hasFocus()` — возвращает значение `True`, если компонент находится в фокусе ввода, и `False` — в противном случае;
- `focusWidget()` — возвращает ссылку на последний компонент, для которого вызывался метод `setFocus()`. Для компонентов верхнего уровня возвращается ссылка на компонент, который получит фокус после того, как окно станет активным;
- `setFocusProxy(<QWidget>)` — позволяет указать ссылку на компонент, который будет получать фокус ввода вместо текущего компонента;
- `focusProxy()` — возвращает ссылку на компонент, который обрабатывает фокус ввода вместо текущего компонента. Если компонента нет, то метод возвращает значение `None`;
- `focusNextChild()` — находит следующий компонент, которому можно передать фокус и передает фокус. Аналогично нажатию клавиши <Tab>. Метод возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- `focusPreviousChild()` — находит предыдущий компонент, которому можно передать фокус, и передает фокус. Аналогично нажатию комбинации клавиш <Shift>+<Tab>. Метод возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- `focusNextPrevChild(<Флаг>)` — если в параметре указано значение `True`, то метод аналогичен методу `focusNextChild()`. Если в параметре указано значение `False`, то метод аналогичен методу `focusPreviousChild()`. Метод возвращает значение `True`, если компонент найден, и `False` — в противном случае;

- `setTabOrder(<Компонент1>, <Компонент2>)` — позволяет задать последовательность смены фокуса при нажатии клавиши <Tab>. Метод является статическим. В параметре `<Компонент2>` указывается ссылка на компонент, на который переместится фокус с компонента `<Компонент1>`. Если компонентов много, то метод вызывается несколько раз. Пример указания цепочки перехода `widget1 -> widget2 -> widget3 -> widget4`:

```
QtGui.QWidget.setTabOrder(widget1, widget2)
QtGui.QWidget.setTabOrder(widget2, widget3)
QtGui.QWidget.setTabOrder(widget3, widget4)
```

- `setFocusPolicy(<Способ>)` — задает способ получения фокуса компонентом. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt`:
  - `NoFocus` — `0` — компонент не может получать фокус;
  - `TabFocus` — `1` — получает фокус с помощью клавиши <Tab>;
  - `ClickFocus` — `2` — получает фокус с помощью щелчка мышью;
  - `StrongFocus` — `11` — получает фокус с помощью клавиши <Tab> и щелчка мышью;
  - `WheelFocus` — `15` — получает фокус с помощью клавиши <Tab>, щелчка мышью и колесика мыши;
- `focusPolicy()` — возвращает текущий способ получения фокуса;
- `grabKeyboard()` — захватывает ввод с клавиатуры. Другие компоненты не будут получать события клавиатуры, пока не будет вызван метод `releaseKeyboard()`;
- `releaseKeyboard()` — освобождает захваченный ранее ввод с клавиатуры.

Получить ссылку на компонент, находящийся в фокусе ввода, позволяет статический метод `focusWidget()` из класса `QApplication`. Если ни один компонент не имеет фокуса ввода, то метод возвращает значение `None`. Не путайте этот метод с одноименным методом из класса `QWidget`.

Обработать получение и потерю фокуса ввода позволяют следующие методы:

- `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода;
- `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода.

Через параметр `<event>` доступен экземпляр класса `QFocusEvent`, который содержит следующие методы:

- `gotFocus()` — возвращает значение `True`, если тип события `QEvent.FocusIn`, и `False` — в противном случае;
- `lostFocus()` — возвращает значение `True`, если тип события `QEvent.FocusOut`, и `False` — в противном случае;
- `reason()` — возвращает причину установки фокуса. Значение аналогично значению параметра `<Причина>` в методе `setFocus()`.

Создадим окно с кнопкой и двумя однострочными полями ввода (листинг 21.14). Для полей ввода обработаем получение и потерю фокуса ввода, а при нажатии кнопки установим фокус ввода на второе поле. Кроме того, зададим последовательность перехода при нажатии клавиши <Tab>.

Листинг 21.14. Установка фокуса ввода

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui

class MyLineEdit(QtGui.QLineEdit):
    def __init__(self, id, parent=None):
        QtGui.QLineEdit.__init__(self, parent)
        self.id = id
    def focusInEvent(self, e):
        print("Получен фокус полем", self.id)
        QtGui.QLineEdit.focusInEvent(self, e)
    def focusOutEvent(self, e):
        print("Потерян фокус полем", self.id)
        QtGui.QLineEdit.focusOutEvent(self, e)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.button = QtGui.QPushButton("Установить фокус на поле 2")
        self.line1 = MyLineEdit(1)
        self.line2 = MyLineEdit(2)
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.button)
        self.vbox.addWidget(self.line1)
        self.vbox.addWidget(self.line2)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
        # Задаем порядок обхода с помощью клавиши <Tab>
        QtGui.QWidget.setTabOrder(self.line1, self.line2)
        QtGui.QWidget.setTabOrder(self.line2, self.button)
    def on_clicked(self):
        self.line2.setFocus()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

21.9.2. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют "горячими" клавишами) позволяют установить фокус ввода с помощью нажатия специальной клавиши (например, <Alt> или <Ctrl>) и какой-либо дополнительной клавиши. Если после нажатия клавиш быстрого доступа в фокусе окажется кнопка (или пункт меню), то она будет нажата.

Чтобы задать клавиши быстрого доступа, следует в тексте надписи указать символ & перед буквой. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент окажется в фокусе ввода. Некоторые компоненты, например текстовое поле, не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов, необходимо отдельно создать надпись и связать ее с компонентом с помощью метода setBuddy(<Компонент>) из класса QLabel. Если же создание надписи не представляется возможным, то можно воспользоваться следующими методами из класса QWidget:

- grabShortcut(<Клавиши>[, <Контекст>]) — регистрирует клавиши быстрого доступа и возвращает идентификатор, с помощью которого можно управлять ими в дальнейшем. В параметре <Клавиши> указывается экземпляр класса QKeySequence. Создать экземпляр этого класса для комбинации клавиш <Alt>+<E> можно, например, так:

  ```
  QtGui.QKeySequence.mnemonic("&e")
  QtGui.QKeySequence("Alt+e")
  QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E)
  ```

  В параметре <Контекст> можно указать атрибуты WidgetShortcut, WidgetWithChildrenShortcut, WindowShortcut (значение по умолчанию) и ApplicationShortcut из класса QtCore.Qt;

- releaseShortcut(<ID>) — удаляет комбинацию с идентификатором <ID>;

- setShortcutEnabled(<ID>[, <Флаг>]) — если в качестве параметра <Флаг> указано значение True (значение по умолчанию), то клавиши быстрого доступа с идентификатором <ID> разрешены. Значение False запрещает использование клавиш быстрого доступа.

При нажатии клавиш быстрого доступа генерируется событие QEvent.Shortcut, которое можно обработать в методе event(self, <event>). Через параметр <event> доступен экземпляр класса QShortcutEvent, который содержит следующие методы:

- shortcutId() — возвращает идентификатор комбинации клавиш;

- isAmbiguous() — возвращает значение True, если событие отправлено сразу нескольким компонентам, и False — в противном случае;

- key() — возвращает экземпляр класса QKeySequence.

Создадим окно с надписью, двумя однострочными текстовыми полями и кнопкой (листинг 21.15). Для первого текстового поля назначим комбинацию клавиш (<Alt>+<B>) через надпись, а для второго поля (<Alt>+<E>) — с помощью метода grabShortcut(). Для кнопки назначим комбинацию клавиш (<Alt>+<У>) обычным образом через надпись на кнопке.

Листинг 21.15. Назначение клавиш быстрого доступа

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyLineEdit(QtGui.QLineEdit):
    def __init__(self, parent=None):
        QtGui.QLineEdit.__init__(self, parent)
        self.id = None
    def event(self, e):
        if e.type() == QtCore.QEvent.Shortcut:
```

```
            if self.id == e.shortcutId():
                self.setFocus(QtCore.Qt.ShortcutFocusReason)
                return True
        return QtGui.QLineEdit.event(self, e)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.label = QtGui.QLabel("Устано&вить фокус на поле 1")
        self.lineEdit1 = QtGui.QLineEdit()
        self.label.setBuddy(self.lineEdit1)
        self.lineEdit2 = MyLineEdit()
        self.lineEdit2.id = self.lineEdit2.grabShortcut(
                            QtGui.QKeySequence.mnemonic("&e"))
        self.button = QtGui.QPushButton("&Убрать фокус с поля 1")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.lineEdit1)
        self.vbox.addWidget(self.lineEdit2)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
    def on_clicked(self):
        self.lineEdit1.clearFocus()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Помимо рассмотренных способов для назначения клавиш быстрого доступа можно воспользоваться классом `QShortcut`. В этом случае назначение клавиш для второго текстового поля будет выглядеть так:

```
self.lineEdit2 = QtGui.QLineEdit()
self.shc = QtGui.QShortcut(QtGui.QKeySequence.mnemonic("&e"), self)
self.shc.setContext(QtCore.Qt.WindowShortcut)
self.shc.activated.connect(self.lineEdit2.setFocus)
```

Назначить комбинацию быстрых клавиш позволяет также класс `QAction`. Назначение клавиш для второго текстового поля выглядит следующим образом:

```
self.lineEdit2 = QtGui.QLineEdit()
self.act = QtGui.QAction(self)
self.act.setShortcut(QtGui.QKeySequence.mnemonic("&e"))
self.act.triggered.connect(self.lineEdit2.setFocus)
self.addAction(self.act)
```

21.9.3. Нажатие и отпускание клавиши на клавиатуре

При нажатии и отпускании клавиши вызываются следующие методы:

- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши на клавиатуре. Если клавишу удерживать нажатой, то событие генерируется постоянно, пока клавиша не будет отпущена;
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой клавиши.

Через параметр `<event>` доступен экземпляр класса `QKeyEvent`, который позволяет получить дополнительную информацию о событии. Класс `QKeyEvent` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QKeyEvent`):

- `key()` — возвращает код нажатой клавиши. Пример определения клавиши:

```
if e.key() == QtCore.Qt.Key_B:
    print("Нажата клавиша <B>")
```

- `text()` — возвращает текстовое представление символа в кодировке Unicode. Если клавиша является специальной, то возвращается пустая строка;
- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с клавишей. Может содержать значения следующих атрибутов из класса `QtCore.Qt` (или комбинацию значений):
  - `NoModifier` — модификаторы не нажаты;
  - `ShiftModifier` — нажата клавиша <Shift>;
  - `ControlModifier` — нажата клавиша <Ctrl>;
  - `AltModifier` — нажата клавиша <Alt>;
  - `MetaModifier` — нажата клавиша <Meta>;
  - `KeypadModifier` — нажата любая клавиша на дополнительной клавиатуре;
  - `GroupSwitchModifier` — нажата клавиша <Mode_switch> (только в X11).

  Пример определения модификатора <Shift>:

```
if e.modifiers() & QtCore.Qt.ShiftModifier:
    print("Нажат модификатор <Shift>")
```

- `isAutoRepeat()` — возвращает значение `True`, если событие вызвано повторно удержанием клавиши нажатой, и `False` — в противном случае;
- `matches(<QKeySequence.StandardKey>)` — возвращает значение `True`, если нажата специальная комбинация клавиш, соответствующая указанному значению, и `False` — в противном случае. В качестве значения указываются атрибуты из класса `QKeySequence`, например, `QKeySequence.Copy` для комбинации клавиш <Ctrl>+<C> (копировать). Полный список атрибутов смотрите в документации по классу `QKeySequence`. Пример:

```
if e.matches(QtGui.QKeySequence.Copy):
    print("Нажата комбинация <Ctrl>+<C>")
```

При обработке нажатия клавиш следует учитывать, что:

- компонент должен иметь возможность принимать фокус ввода. Некоторые компоненты не могут принимать фокус ввода по умолчанию, например надпись. Чтобы изменить

способ получения фокуса, следует воспользоваться методом `setFocusPolicy(<Способ>)`, который мы рассматривали в *разд. 21.9.1*;

- чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `releaseKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему компоненту соответственно. Перехватить нажатие этих клавиш можно только в методе `event(self, <event>)`;
- если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

21.10. События мыши

События мыши обрабатываются не реже, чем события клавиатуры. С помощью специальных методов можно обработать нажатие и отпускание кнопки мыши, перемещение указателя, а также вхождение указателя в область компонента и выхода из этой области. В зависимости от ситуации можно изменить вид указателя, например, при выполнении длительной операции отобразить указатель в виде песочных часов. В этом разделе мы рассмотрим изменение вида указателя мыши как для отдельного компонента, так и для всего приложения.

21.10.1. Нажатие и отпускание кнопки мыши

При нажатии и отпускании кнопки мыши вызываются следующие методы:

- `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши;
- `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши;
- `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью в области компонента. Следует учитывать, что двойному щелчку предшествуют другие события. Последовательность событий при двойном щелчке выглядит так:

```
Событие MouseButtonPress
Событие MouseButtonRelease
Событие MouseButtonDblClick
Событие MouseButtonPress
Событие MouseButtonRelease
```

Задать интервал двойного щелчка позволяет метод `setDoubleClickInterval()` из класса `QApplication`. Получить текущее значение интервала можно с помощью метода `doubleClickInterval()`.

Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, который позволяет получить дополнительную информацию о событии. Класс `QMouseEvent` содержит следующие методы:

- `x()` и `y()` — возвращают координаты по осям `X` и `Y` соответственно в пределах области компонента;
- `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами в пределах области компонента;

- `posF()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах области компонента;
- `globalX()` и `globalY()` — возвращают координаты по осям `X` и `Y` соответственно в пределах экрана;
- `globalPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана;
- `button()` — позволяет определить, какая кнопка мыши вызвала событие. Возвращает значение одного из следующих атрибутов из класса `QtCore.Qt`:
  - `NoButton` — `0` — кнопки не нажаты. Это значение возвращается методом `button()` при перемещении указателя мыши;
  - `LeftButton` — `1` — нажата левая кнопка мыши;
  - `RightButton` — `2` — нажата правая кнопка мыши;
  - `MidButton` и `MiddleButton` — `4` — нажата средняя кнопка мыши;
  - `XButton1` — `8`;
  - `XButton2` — `16`;
- `buttons()` — позволяет определить все кнопки, которые нажаты одновременно. Возвращает комбинацию значений атрибутов `LeftButton`, `RightButton` и `MidButton`. Пример определения кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 21.9.3*.

Если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

Если для компонента атрибут `WA_NoMousePropagation` из класса `QtCore.Qt` установлен в истинное значение, то событие мыши не будет передаваться родительскому компоненту. Значение атрибута можно изменить с помощью метода `setAttribute()`:

```
self.setAttribute(QtCore.Qt.WA_NoMousePropagation, True)
```

По умолчанию событие мыши перехватывает компонент, над которым произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне компонента, следует захватить мышь с помощью метода `grabMouse()`. Освободить захваченную ранее мышь позволяет метод `releaseMouse()`.

21.10.2. Перемещение указателя

Чтобы обработать перемещение указателя мыши, необходимо переопределить метод `mouseMoveEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, который позволяет получить дополнительную информацию о событии. Методы этого класса мы уже рассматривали в предыдущем разделе. Следует учитывать, что метод `button()` при перемещении мыши возвращает значение атрибута `QtCore.Qt.NoButton`.

По умолчанию метод `mouseMoveEvent()` вызывается только в том случае, если при перемещении удерживается нажатой какая-либо кнопка мыши. Это сделано специально, чтобы не

создавать лишних событий при обычном перемещении указателя мыши. Если необходимо обрабатывать любые перемещения указателя в пределах компонента, то следует вызвать метод `setMouseTracking()` из класса `QWidget` и передать ему значение `True`. Чтобы обработать все перемещения внутри окна, нужно дополнительно захватить мышь с помощью метода `grabMouse()`.

Метод `pos()` объекта события возвращает позицию точки в системе координат текущего компонента. Чтобы преобразовать координаты точки в систему координат родительского компонента или в глобальную систему координат, нужно воспользоваться следующими методами из класса `QWidget`:

- `mapToGlobal(<QPoint>)` — преобразует координаты точки из системы координат компонента в глобальную систему координат. Метод возвращает экземпляр класса `QPoint`;
- `mapFromGlobal(<QPoint>)` — преобразует координаты точки из глобальной системы координат в систему координат компонента. Метод возвращает экземпляр класса `QPoint`;
- `mapToParent(<QPoint>)` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента. Если компонент не имеет родителя, то метод аналогичен методу `mapToGlobal()`. Метод возвращает экземпляр класса `QPoint`;
- `mapFromParent(<QPoint>)` — преобразует координаты точки из системы координат родительского компонента в систему координат данного компонента. Если компонент не имеет родителя, то метод аналогичен методу `mapFromGlobal()`. Метод возвращает экземпляр класса `QPoint`;
- `mapTo(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента `<QWidget>`. Метод возвращает экземпляр класса `QPoint`;
- `mapFrom(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат родительского компонента `<QWidget>` в систему координат данного компонента. Метод возвращает экземпляр класса `QPoint`.

21.10.3. Наведение и выведение указателя

Обработать наведение указателя мыши на компонент и выведение указателя позволяют следующие методы:

- `enterEvent(self, <event>)` — вызывается при наведении указателя мыши на область компонента;
- `leaveEvent(self, <event>)` — вызывается, когда указатель мыши покидает область компонента.

Через параметр `<event>` доступен экземпляр класса `QEvent`. Этот параметр не несет никакой дополнительной информации. Вполне достаточно знать, что указатель попал в область компонента или покинул ее.

21.10.4. Прокрутка колесика мыши

Некоторые мыши комплектуются колесиком, обычно используемым для управления прокруткой отдельной области. Обработать поворот этого колесика позволяет метод `wheelEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QWheelEvent`, который позволяет получить дополнительную информацию о событии.

Класс QWheelEvent содержит следующие методы:

- delta() — возвращает расстояние поворота колесика. Значение может быть положительным или отрицательным в зависимости от направления поворота;
- orientation() — возвращает ориентацию, в виде значения одного из следующих атрибутов из класса QtCore.Qt:
  - Horizontal — 1 — по горизонтали;
  - Vertical — 2 — по вертикали;
- x() и y() — возвращают координаты указателя в момент события по осям X и Y соответственно в пределах области компонента;
- pos() — возвращает экземпляр класса QPoint с целочисленными координатами указателя в момент события в пределах области компонента;
- globalX() и globalY() — возвращают координаты указателя в момент события по осям X и Y соответственно в пределах экрана;
- globalPos() — возвращает экземпляр класса QPoint с координатами указателя в момент события в пределах экрана;
- buttons() — позволяет определить кнопки, которые нажаты одновременно с поворотом колесика. Возвращает комбинацию значений атрибутов LeftButton, RightButton и MidButton. Пример определения кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- modifiers() — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты одновременно с поворотом колесика. Возможные значения мы уже рассматривали в *разд. 21.9.3*.

Если событие обработано, то нужно вызвать метод accept() через объект события. Чтобы родительский компонент мог получить событие, вместо метода accept() необходимо вызвать метод ignore().

21.10.5. Изменение внешнего вида указателя мыши

Для изменения внешнего вида указателя мыши при вхождении указателя в область компонента предназначены следующие методы из класса QWidget:

- setCursor(<Курсор>) — задает внешний вид указателя мыши для компонента. В качестве параметра указывается экземпляр класса QCursor или следующие атрибуты из класса QtCore.Qt: ArrowCursor (стандартная стрелка), UpArrowCursor (стрелка, направленная вверх), CrossCursor (крестообразный указатель), WaitCursor (песочные часы), IBeamCursor (I-образный указатель), SizeVerCursor (стрелки, направленные вверх и вниз), SizeHorCursor (стрелки, направленные влево и вправо), SizeBDiagCursor (стрелки, направленные в правый верхний угол и левый нижний угол), SizeFDiagCursor (стрелки, направленные в левый верхний угол и правый нижний угол), SizeAllCursor (стрелки, направленные вверх, вниз, влево и вправо), BlankCursor (пустой указатель), SplitVCursor (указатель изменения высоты), SplitHCursor (указатель изменения ширины), PointingHandCursor (указатель в виде руки), ForbiddenCursor (перечеркнутый круг), OpenHandCursor (разжатая рука), ClosedHandCursor (сжатая рука), WhatsThisCursor (стрелка с вопросительным знаком) и BusyCursor (стрелка с песочными часами).

Пример:

```
self.setCursor(QtCore.Qt.WaitCursor)
```

- `unsetCursor()` — отменяет установку указателя для компонента. В результате внешний вид указателя мыши будет наследоваться от родительского компонента;
- `cursor()` — возвращает экземпляр класса `QCursor` с текущим курсором.

Управлять текущим видом курсора для всего приложения сразу можно с помощью следующих статических методов из класса `QApplication`:

- `setOverrideCursor(<Курсор>)` — задает внешний вид указателя мыши для всего приложения. В качестве параметра указывается экземпляр класса `QCursor` или специальные атрибуты из класса `QtCore.Qt`. Для отмены установки необходимо вызвать метод `restoreOverrideCursor()`;
- `restoreOverrideCursor()` — отменяет изменение внешнего вида курсора для всего приложения. Пример:

```
QtGui.QApplication.setOverrideCursor(QtCore.Qt.WaitCursor)
# Выполняем длительную операцию
QtGui.QApplication.restoreOverrideCursor()
```

- `changeOverrideCursor(<Курсор>)` — изменяет внешний вид указателя мыши для всего приложения. Если до вызова этого метода не вызывался метод `setOverrideCursor()`, то указанное значение игнорируется. В качестве параметра указывается экземпляр класса `QCursor` или специальные атрибуты из класса `QtCore.Qt`;
- `overrideCursor()` — возвращает экземпляр класса `QCursor` с текущим курсором или значение `None`.

Изменять внешний вид указателя мыши для всего приложения принято на небольшой промежуток времени, обычно на время выполнения какой-либо операции, в процессе которой приложение не может нормально реагировать на действия пользователя. Чтобы информировать об этом пользователя, указатель принято выводить в виде песочных часов (атрибут `WaitCursor`).

Метод `setOverrideCursor()` может быть вызван несколько раз. В этом случае курсоры помещаются в стек. Каждый вызов метода `restoreOverrideCursor()` удаляет последний курсор, добавленный в стек. Для нормальной работы приложения необходимо вызывать методы `setOverrideCursor()` и `restoreOverrideCursor()` одинаковое количество раз.

Класс `QCursor` позволяет создать объект курсора с изображением любой формы. Чтобы загрузить изображение, следует передать путь к файлу конструктору класса `QPixmap`. Чтобы создать объект курсора, необходимо передать конструктору класса `QCursor` в первом параметре экземпляр класса `QPixmap`, а во втором и третьем параметрах — координаты "горячей" точки. Пример создания и установки пользовательского курсора:

```
self.setCursor(QtGui.QCursor(QtGui.QPixmap("cursor.png"), 0, 0))
```

Класс `QCursor` содержит также два статических метода:

- `pos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши относительно экрана. Пример:

```
print(QtGui.QCursor.pos().x(), QtGui.QCursor.pos().y())
```

- `setPos()` — позволяет задать позицию указателя мыши. Метод имеет два формата: `setPos(<X>, <Y>)` и `setPos(<QPoint>)`.

21.11. Технология drag & drop

Технология drag & drop позволяет обмениваться данными различных типов между компонентами как одного приложения, так и разных приложений, путем перетаскивания и сбрасывания объектов с помощью мыши. Типичным примером использования технологии служит перемещение файлов в программе Проводник в Windows. Чтобы переместить файл в другой каталог, достаточно нажать левую кнопку мыши над значком файла и, не отпуская кнопку, перетащить файл на значок каталога, а затем отпустить кнопку мыши. Если необходимо скопировать файл, а не переместить, то следует дополнительно удерживать нажатой клавишу <Ctrl>.

21.11.1. Запуск перетаскивания

Операция перетаскивания состоит из двух частей. Первая часть запускает процесс, а вторая часть обрабатывает момент сброса объекта. Обе части могут обрабатываться как одним приложением, так и двумя разными приложениями. Запуск перетаскивания осуществляется следующим образом:

1. Внутри метода `mousePressEvent()` запоминаются координаты щелчка левой кнопкой мыши.
2. Внутри метода `mouseMoveEvent()` вычисляется пройденное расстояние или измеряется время операции. Это необходимо сделать, чтобы предотвратить случайное перетаскивание. Управлять задержкой позволяют следующие статические методы класса `QApplication`:
   - `startDragDistance()` — возвращает минимальное расстояние, после прохождения которого можно запускать операцию перетаскивания;
   - `setStartDragDistance(<Дистанция>)` — задает расстояние;
   - `startDragTime()` — возвращает время задержки в миллисекундах перед запуском операции перетаскивания;
   - `setStartDragTime(<Время>)` — задает время задержки.
3. Если пройдено минимальное расстояние или истек минимальный промежуток времени, то создается экземпляр класса `QDrag` и вызывается метод `exec_()`, который после завершения операции возвращает действие, выполненное с данными (например, данные скопированы или перемещены).

Создать экземпляр класса `QDrag` можно так:

```
<Объект> = QtGui.QDrag(<Ссылка на компонент>)
```

Класс `QDrag` содержит следующие методы:

- `exec_()` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. Метод имеет два формата:

```
exec_([<Действия>=MoveAction])
exec_(<Действия>, <Действие по умолчанию>)
```

  В параметре `<Действия>` указывается комбинация допустимых действий, а в параметре `<Действие по умолчанию>` — действие, которое используется, если не нажаты клавиши-модификаторы. Возможные действия могут быть заданы следующими атрибутами из класса `QtCore.Qt`: `CopyAction` (`1`; копирование), `MoveAction` (`2`; перемещение), `LinkAction` (`4`; ссылка), `IgnoreAction` (`0`; действие игнорировано), `TargetMoveAction` (`32770`).

Пример:

```
act = drag.exec_(QtCore.Qt.MoveAction | QtCore.Qt.CopyAction,
                 QtCore.Qt.MoveAction)
```

- `start([<Действия>=CopyAction])` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. В качестве параметра указывается комбинация допустимых действий;
- `setMimeData(<QMimeData>)` — позволяет задать перемещаемые данные. В качестве значения указывается экземпляр класса `QMimeData`. Пример передачи текста:

```
data = QtCore.QMimeData()
data.setText("Перетаскиваемый текст")
drag = QtGui.QDrag(self)
drag.setMimeData(data)
```

- `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными;
- `setPixmap(<QPixmap>)` — задает изображение, которое будет перемещаться вместе с указателем мыши. В качестве параметра указывается экземпляр класса `QPixmap`. Пример:

```
drag.setPixmap(QtGui.QPixmap("pixmap.png"))
```

- `pixmap()` — возвращает экземпляр класса `QPixmap` с изображением, которое перемещается вместе с указателем;
- `setHotSpot(<QPoint>)` — задает координаты "горячей" точки на перемещаемом изображении. В качестве параметра указывается экземпляр класса `QPoint`. Пример:

```
drag.setHotSpot(QtCore.QPoint(20, 20))
```

- `hotSpot()` — возвращает экземпляр класса `QPoint` с координатами "горячей" точки на перемещаемом изображении;
- `setDragCursor(<QPixmap>, <Действие>)` — позволяет изменить внешний вид указателя мыши для действия, указанного во втором параметре. В первом параметре указывается экземпляр класса `QPixmap`, а во втором параметре — действия `CopyAction`, `MoveAction` или `LinkAction`. Если в первом параметре указан пустой объект класса `QPixmap`, то это позволит удалить ранее установленный вид указателя для действия. Пример изменения указателя для перемещения:

```
drag.setDragCursor(QtGui.QPixmap("cursor.png"),
                   QtCore.Qt.MoveAction)
```

- `source()` — возвращает ссылку на компонент-источник;
- `target()` — возвращает ссылку на компонент-приемник или значение `None`, если компонент находится в другом приложении.

Класс `QDrag` поддерживает два сигнала:

- `actionChanged(Qt::DropAction)` — генерируется при изменении действия;
- `targetChanged(QWidget *)` — генерируется при изменении принимающего компонента.

Пример назначения обработчиков сигналов:

```
self.connect(drag, QtCore.SIGNAL("actionChanged(Qt::DropAction)"),
             self.on_action_changed)
self.connect(drag, QtCore.SIGNAL("targetChanged(QWidget *)"),
             self.on_target_changed)
```

21.11.2. Класс *QMimeData*

Перемещаемые данные и сведения о MIME-типе должны быть представлены классом `QMimeData`. Экземпляр этого класса необходимо передать в метод `setMimeData()` класса `QDrag`. Создание экземпляра класса `QMimeData` выглядит так:

```
data = QtCore.QMimeData()
```

Класс `QMimeData` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QMimeData`):

- `setText(<Текст>)` — устанавливает текстовые данные (MIME-тип `text/plain`). Пример указания значения:

  ```
  data.setText("Перетаскиваемый текст")
  ```

- `text()` — возвращает текстовые данные (MIME-тип `text/plain`);
- `hasText()` — возвращает значение `True`, если объект содержит текстовые данные (MIME-тип `text/plain`), и `False` — в противном случае;
- `setHtml(<HTML-текст>)` — устанавливает текстовые данные в формате HTML (MIME-тип `text/html`). Пример указания значения:

  ```
  data.setHtml("<b>Перетаскиваемый HTML-текст</b>")
  ```

- `html()` — возвращает текстовые данные в формате HTML (MIME-тип `text/html`);
- `hasHtml()` — возвращает значение `True`, если объект содержит текстовые данные в формате HTML (MIME-тип `text/html`), и `False` — в противном случае;
- `setUrls(<Список URI-адресов>)` — устанавливает список URI-адресов (MIME-тип `text/uri-list`). В качестве значения указывается список с экземплярами класса `QUrl`. С помощью этого MIME-типа можно обработать перетаскивание файлов. Пример указания значения:

  ```
  data.setUrls([QtCore.QUrl("http://google.ru/")])
  ```

- `urls()` — возвращает список URI-адресов (MIME-тип `text/uri-list`). Пример получения первого URI-адреса:

  ```
  uri = e.mimeData().urls()[0].toString()
  ```

- `hasUrls()` — возвращает значение `True`, если объект содержит список URI-адресов (MIME-тип `text/uri-list`), и `False` — в противном случае;
- `setImageData(<Объект изображения>)` — устанавливает изображение (MIME-тип `application/x-qt-image`). В качестве значения можно указать, например, экземпляр класса `QImage` или `QPixmap`. Пример указания значения:

  ```
  data.setImageData(QtGui.QImage("pixmap.png"))
  data.setImageData(QtGui.QPixmap("pixmap.png"))
  ```

- `imageData()` — возвращает объект изображения (тип возвращаемого объекта зависит от типа объекта, указанного в методе `setImageData()`);
- `hasImage()` — возвращает значение `True`, если объект содержит изображение (MIME-тип `application/x-qt-image`), и `False` — в противном случае;
- `setData(<MIME-тип>, <Данные>)` — позволяет установить данные пользовательского MIME-типа. В первом параметре указывается MIME-тип в виде строки, а во втором па-

раметре — экземпляр класса QByteArray с данными. Метод можно вызвать несколько раз с различными MIME-типами. Пример передачи текстовых данных:

```
data.setData("text/plain",
             QtCore.QByteArray(bytes("Данные", "utf-8")))
```

- data(<MIME-тип>) — возвращает экземпляр класса QByteArray с данными, соответствующими указанному MIME-типу;
- hasFormat(<MIME-тип>) — возвращает значение True, если объект содержит данные в указанном MIME-типе, и False — в противном случае;
- formats() — возвращает список с поддерживаемыми объектом MIME-типами;
- removeFormat(<MIME-тип>) — удаляет данные, соответствующие указанному MIME-типу;
- clear() — удаляет все данные и информацию о MIME-типе.

Если необходимо перетаскивать данные какого-либо специфического типа, то нужно наследовать класс QMimeData и переопределить методы retrieveData() и formats(). За подробной информацией по этому вопросу обращайтесь к документации.

21.11.3. Обработка сброса

Прежде чем обрабатывать перетаскивание и сбрасывание объекта, необходимо сообщить системе, что компонент может обрабатывать эти события. Для этого внутри конструктора компонента следует вызвать метод setAcceptDrops() из класса QWidget и передать ему значение True:

```
self.setAcceptDrops(True)
```

Обработка перетаскивания и сброса объекта выполняется следующим образом:

1. Внутри метода dragEnterEvent() проверяется MIME-тип перетаскиваемых данных и действие. Если компонент способен обработать сброс этих данных и соглашается с предложенным действием, то необходимо вызвать метод acceptProposedAction() через объект события. Если нужно изменить действие, то методу setDropAction() передается новое действие, а затем вызывается метод accept(), а не метод acceptProposedAction().
2. Если необходимо ограничить область сброса некоторым участком компонента, то можно дополнительно определить метод dragMoveEvent(). Этот метод будет постоянно вызываться при перетаскивании внутри области компонента. При согласии со сбрасыванием следует вызвать метод accept(), которому можно передать экземпляр класса QRect с координатами и размером участка. Если параметр указан, то при перетаскивании внутри участка метод dragMoveEvent() повторно вызываться не будет.
3. Внутри метода dropEvent() производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- dragEnterEvent(self, <event>) — вызывается, когда перетаскиваемый объект входит в область компонента. Через параметр <event> доступен экземпляр класса QDragEnterEvent;
- dragLeaveEvent(self, <event>) — вызывается, когда перетаскиваемый объект покидает область компонента. Через параметр <event> доступен экземпляр класса QDragLeaveEvent;
- dragMoveEvent(self, <event>) — вызывается при перетаскивании объекта внутри области компонента. Через параметр <event> доступен экземпляр класса QDragMoveEvent;

- dropEvent(self, <event>) — вызывается при сбрасывании объекта в области компонента. Через параметр <event> доступен экземпляр класса QDropEvent.

Класс QDragLeaveEvent наследует класс QEvent и не несет никакой дополнительной информации. Достаточно просто знать, что перетаскиваемый объект покинул область компонента. Цепочка наследования остальных классов выглядит так:

(QEvent, QMimeSource) — QDropEvent — QDragMoveEvent — QDragEnterEvent

Класс QMimeSource признан устаревшим и используется только в целях совместимости с предыдущими версиями. Вместо класса QMimeSource следует использовать класс QMimeData. Класс QDragEnterEvent не содержит собственных методов, но наследует все методы классов QDropEvent и QDragMoveEvent.

Класс QDropEvent содержит следующие методы:

- mimeData() — возвращает экземпляр класса QMimeData с перемещаемыми данными и информацией о MIME-типе;
- pos() — возвращает экземпляр класса QPoint с координатами сбрасывания объекта;
- possibleActions() — возвращает комбинацию возможных действий при сбрасывании. Пример определения значений:

```
if e.possibleActions() & QtCore.Qt.MoveAction:
    print("MoveAction")
if e.possibleActions() & QtCore.Qt.CopyAction:
    print("CopyAction")
```

- proposedAction() — возвращает действие по умолчанию при сбрасывании;
- acceptProposedAction() — устанавливает флаг готовности принять перемещаемые данные и согласия с действием, возвращаемым методом proposedAction(). Метод acceptProposedAction() (или метод accept()) необходимо вызвать внутри метода dragEnterEvent(), иначе метод dropEvent() вызван не будет;
- setDropAction(<Действие>) — позволяет изменить действие при сбрасывании. После изменения действия следует вызвать метод accept(), а не acceptProposedAction();
- dropAction() — возвращает действие, которое должно быть выполнено при сбрасывании. Возвращаемое значение может не совпадать со значением, возвращаемым методом proposedAction(), если действие было изменено с помощью метода setDropAction();
- keyboardModifiers() — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 21.9.3*;
- mouseButtons() — позволяет определить кнопки мыши, которые нажаты;
- source() — возвращает ссылку на компонент внутри приложения, являющийся источником события, или значение None.

Теперь рассмотрим методы класса QDragMoveEvent:

- accept([<QRect>]) — устанавливает флаг, который является признаком согласия с дальнейшей обработкой события. В качестве параметра можно указать экземпляр класса QRect с координатами и размерами прямоугольной области, в которой сбрасывание будет принято;
- ignore([<QRect>]) — сбрасывает флаг, что является запретом на дальнейшую обработку события. В качестве параметра можно указать экземпляр класса QRect с координатами и размерами прямоугольной области, в которой сбрасывание выполнить нельзя;

- `answerRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой произойдет сбрасывание, если событие будет принято.

Некоторые компоненты в PyQt по умолчанию поддерживают технологию drag & drop, например, в однострочное текстовое поле можно перетащить текст из другого приложения. Поэтому, прежде чем изобретать свой "велосипед", убедитесь, что поддержка технологии в компоненте не реализована.

21.12. Работа с буфером обмена

Помимо технологии drag & drop для обмена данными между приложениями используется буфер обмена. Одно приложение помещает данные в буфер обмена, а второе приложение (или то же самое) может извлечь их из буфера. Получить ссылку на глобальный объект буфера обмена позволяет статический метод `clipboard()` из класса `QApplication`:

```
clipboard = QtGui.QApplication.clipboard()
```

Класс `QClipboard` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QClipboard`):

- `setText(<Текст>[, <Режим>])` — копирует текст в буфер обмена;
- `text([<Режим>])` — возвращает текст из буфера обмена или пустую строку;
- `text(<Тип>[, <Режим>])` — возвращает кортеж из двух строк. Первый элемент кортежа содержит текст из буфера, а второй элемент — название типа. В параметре `<Тип>` могут быть указаны значения `"plain"` (простой текст), `"html"` (текст в формате HTML) или пустая строка;
- `setMimeData(<QMimeData>[, <Режим>])` — позволяет сохранить в буфере данные любого типа. В качестве первого параметра указывается экземпляр класса `QMimeData`. Этот класс мы уже рассматривали при изучении технологии drag & drop (см. *разд. 21.11.2*);
- `mimeData([<Режим>])` — возвращает экземпляр класса `QMimeData`;
- `clear([<Режим>])` — очищает буфер обмена.

В необязательном параметре `<Режим>` могут быть указаны атрибуты `Clipboard` (используется по умолчанию), `Selection` или `FindBuffer` из класса `QClipboard`.

Отследить изменение состояния буфера обмена позволяет сигнал `dataChanged()`. Назначить обработчик этого сигнала можно так:

```
self.connect(QtGui.qApp.clipboard(), QtCore.SIGNAL("dataChanged()"),
             self.on_change_clipboard)
```

21.13. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы компоненту. Для этого необходимо создать класс, который является наследником класса `QObject`, и переопределить метод `eventFilter(self, <Объект>, <event>)`. Через параметр `<Объект>` доступна ссылка на компонент, а через параметр `<event>` — объект с дополнительной информацией о событии. Этот объект отличается для разных типов событий, например, для события `MouseButtonPress` объект будет экземпляром класса `QMouseEvent`, а для события `KeyPress` — экземпляром класса `QKeyEvent`. Внутри метода `eventFilter()` следует вернуть значение

True, если событие не должно быть передано дальше, и False — в противном случае. Пример фильтра, перехватывающего нажатие клавиши <B>:

```
class MyFilter(QtCore.QObject):
    def __init__(self, parent=None):
        QtCore.QObject.__init__(self, parent)
    def eventFilter(self, obj, e):
        if e.type() == QtCore.QEvent.KeyPress:
            if e.key() == QtCore.Qt.Key_B:
                print("Событие от клавиши <B> не дойдет до компонента")
                return True
        return QtCore.QObject.eventFilter(self, obj, e)
```

Далее следует создать экземпляр этого класса, передав в конструктор ссылку на компонент, а затем вызвать метод installEventFilter(), указав в качестве параметра ссылку на объект фильтра. Пример установки фильтра для надписи:

```
self.label.installEventFilter(MyFilter(self.label))
```

Метод installEventFilter() можно вызвать несколько раз, передавая ссылку на разные объекты фильтров. В этом случае первым будет вызван фильтр, который был добавлен последним. Кроме того, один фильтр можно установить сразу в нескольких компонентах. Ссылка на компонент, который является источником события, доступна через второй параметр метода eventFilter().

Удалить фильтр позволяет метод removeEventFilter(<Фильтр>). Если фильтр не был установлен, то ничего не происходит.

21.14. Искусственные события

Для создания искусственных событий применяются следующие статические методы из класса QCoreApplication:

- sendEvent(<QObject>, <QEvent>) — немедленно посылает событие компоненту и возвращает результат выполнения обработчика;
- postEvent(<QObject>, <QEvent>) — добавляет событие в очередь. Этот метод является потокобезопасным, следовательно, его можно использовать в многопоточных приложениях для обмена событиями между потоками.

В параметре <QObject> указывается ссылка на объект, которому посылается событие, а в параметре <QEvent> — объект события. Объект события может быть как экземпляром стандартного класса (например, экземпляром класса QMouseEvent), так и экземпляром пользовательского класса, который является наследником класса QEvent. Пример отправки события QEvent.MouseButtonPress компоненту label:

```
e = QtGui.QMouseEvent(QtCore.QEvent.MouseButtonPress,
                      QtCore.QPoint(5, 5), QtCore.Qt.LeftButton,
                      QtCore.Qt.LeftButton, QtCore.Qt.NoModifier)
QtCore.QCoreApplication.sendEvent(self.label, e)
```

Для отправки пользовательского события необходимо создать класс, который наследует класс QEvent. Внутри класса следует зарегистрировать пользовательское событие с помощью статического метода registerEventType() и сохранить идентификатор события в атрибуте класса.

Пример:

```
class MyEvent(QtCore.QEvent):
    idType = QtCore.QEvent.registerEventType()
    def __init__(self, data):
        QtCore.QEvent.__init__(self, MyEvent.idType)
        self.data = data
    def get_data(self):
        return self.data
```

Пример отправки события класса `MyEvent` компоненту `label`:

```
QtCore.QCoreApplication.sendEvent(self.label, MyEvent("512"))
```

Обработать пользовательское событие можно с помощью метода `event(self, <event>)` или `customEvent(self, <event>)`. Пример:

```
def customEvent(self, e):
    if e.type() == MyEvent.idType:
        self.setText("Получены данные: {0}".format(e.get_data()))
```

ГЛАВА 22

Размещение нескольких компонентов в окне

При размещении нескольких компонентов в окне обычно возникает вопрос взаимного расположения компонентов, а также их минимальных размеров. Следует помнить, что по умолчанию размеры окна можно изменить, взявшись мышью за границу окна, а значит, необходимо перехватывать событие изменения размеров и производить пересчет позиции и размера каждого компонента. Библиотека PyQt избавляет нас от лишних проблем и предоставляет множество компонентов-контейнеров, которые производят перерасчет автоматически. Все, что от нас требуется — это выбрать нужный контейнер, добавить туда компоненты в определенном порядке, а затем поместить контейнер в окно или в другой контейнер.

22.1. Абсолютное позиционирование

Прежде чем изучать компоненты-контейнеры, рассмотрим возможность абсолютного позиционирования компонентов в окне. Итак, если при создании компонента указана ссылка на родительский компонент, то он выводится в позицию с координатами `(0, 0)`. Иными словами, если мы добавим несколько компонентов, то все они отобразятся на одной и той же позиции. Последний добавленный компонент будет на вершине этой кучи, а остальные компоненты будут видны лишь частично или вообще не видны. Размеры добавляемых компонентов будут соответствовать их содержимому.

Для перемещения компонента можно воспользоваться методом `move()`, а для изменения размеров — методом `resize()`. Выполнить одновременное изменение позиции и размеров позволяет метод `setGeometry()`. Все эти методы, а также множество других методов, позволяющих изменять позицию и размеры, мы уже рассматривали в *разд. 20.3* и *20.4*. Если компонент не имеет родителя, то эти методы изменяют характеристики окна, а если при создании компонента указан родительский компонент, то методы изменяют характеристики только самого компонента.

Для примера выведем внутри окна надпись и кнопку, указав позицию и размеры для каждого компонента (листинг 22.1).

Листинг 22.1. Абсолютное позиционирование

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys
```

```
app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Абсолютное позиционирование")
window.resize(300, 120)
label = QtGui.QLabel("Текст надписи", window)
button = QtGui.QPushButton("Текст на кнопке", window)
label.setGeometry(10, 10, 280, 60)
button.resize(280, 30)
button.move(10, 80)
window.show()
sys.exit(app.exec_())
```

Абсолютное позиционирование имеет следующие недостатки:

- при изменении размеров окна необходимо пересчитывать и изменять характеристики всех компонентов вручную;
- при указании фиксированных размеров надписи на компонентах могут выходить за пределы компонента. Помните, что в разных операционных системах используются разные стили оформления, в том числе и характеристики шрифта. Подогнав размеры в одной операционной системе, можно прийти в ужас при виде приложения в другой операционной системе, где размер шрифта в два раза больше. Поэтому лучше вообще отказаться от указания фиксированных размеров или задавать размер и название шрифта для каждого компонента. Кроме того, приложение может поддерживать несколько языков интерфейса. Длина слов в разных языках отличается, что также станет причиной искажения компонентов.

22.2. Горизонтальное и вертикальное выравнивание

Компоненты-контейнеры (их еще называют *менеджерами компоновки*, *менеджерами геометрии*) лишены недостатков абсолютного позиционирования. При изменении размеров окна производится автоматическое изменение характеристик всех компонентов, добавленных в контейнер. Настройки шрифта при этом также учитываются, поэтому изменение размеров шрифта в два раза приведет только к увеличению компонентов и размеров окна.

Для автоматического выравнивания компонентов используются два класса:

- `QHBoxLayout` — выстраивает все добавляемые компоненты по горизонтали (по умолчанию слева направо). Конструктор класса имеет следующий формат:

  ```
  <Объект> = QHBoxLayout([<Родитель>])
  ```

- `QVBoxLayout` — выстраивает все добавляемые компоненты по вертикали (по умолчанию сверху вниз). Формат конструктора класса:

  ```
  <Объект> = QVBoxLayout([<Родитель>])
  ```

Иерархия наследования для классов `QHBoxLayout` и `QVBoxLayout` выглядит так:

```
(QObject, QLayoutItem) — QLayout — QBoxLayout — QHBoxLayout
(QObject, QLayoutItem) — QLayout — QBoxLayout — QVBoxLayout
```

Обратите внимание на то, что классы не являются наследниками класса `QWidget`, следовательно, они не обладают собственным окном и не могут использоваться отдельно. Поэтому

контейнеры обязательно должны быть привязаны к родительскому компоненту. Передать ссылку на родительский компонент можно через конструктор классов `QHBoxLayout` и `QVBoxLayout`. Кроме того, можно передать ссылку на контейнер в метод `setLayout()` родительского компонента. После этого все компоненты, добавленные в контейнер, автоматически привязываются к родительскому компоненту. Типичный пример использования класса `QHBoxLayout` выглядит следующим образом:

```
window = QtGui.QWidget()              # Родительский компонент
button1 = QtGui.QPushButton("1")
button2 = QtGui.QPushButton("2")
hbox = QtGui.QHBoxLayout()            # Создаем контейнер
hbox.addWidget(button1)               # Добавляем компоненты
hbox.addWidget(button2)
window.setLayout(hbox)                # Передаем ссылку родителю
```

Добавить компоненты в контейнер и удалить их позволяют следующие методы:

- `addWidget()` — добавляет компонент в конец контейнера. Формат метода:

  ```
  addWidget(<Компонент>[, stretch=0][, alignment=0])
  ```

  В первом параметре указывается ссылка на компонент. Необязательный параметр `stretch` задает фактор растяжения для ячейки, а параметр `alignment` — выравнивание компонента внутри ячейки. Два последних параметра можно указывать в порядке следования или по именам в произвольном порядке:

  ```
  hbox.addWidget(button1, 10, QtCore.Qt.AlignRight)
  hbox.addWidget(button2, stretch=10)
  hbox.addWidget(button3, alignment=QtCore.Qt.AlignRight)
  ```

- `insertWidget()` — добавляет компонент в указанную позицию контейнера. Формат метода:

  ```
  insertWidget(<Индекс>, <Компонент>[, stretch=0][, alignment=0])
  ```

  Если в первом параметре указано значение `0`, то компонент будет добавлен в начало контейнера. Если указано отрицательное значение, то компонент добавляется в конец контейнера. Другое значение указывает определенную позицию. Остальные параметры аналогичны параметрам метода `addWidget()`. Пример:

  ```
  hbox.addWidget(button1)
  hbox.insertWidget(-1, button2) # Добавление в конец
  hbox.insertWidget(0, button3)  # Добавление в начало
  ```

- `removeWidget(<Компонент>)` — удаляет компонент из контейнера;
- `addLayout()` — добавляет другой контейнер в конец текущего контейнера. С помощью этого метода можно вкладывать один контейнер в другой, создавая таким образом структуру любой сложности. Формат метода:

  ```
  addLayout(<Контейнер>[, stretch=0])
  ```

- `insertLayout()` — добавляет другой контейнер в указанную позицию текущего контейнера. Если в первом параметре указано отрицательное значение, то контейнер добавляется в конец. Формат метода:

  ```
  insertLayout(<Индекс>, <Контейнер>[, stretch=0])
  ```

- `addSpacing(<Размер>)` — добавляет пустое пространство указанного размера в конец контейнера. Пример:

  ```
  hbox.addSpacing(100)
  ```

- `insertSpacing(<Индекс>, <Размер>)` — добавляет пустое пространство указанного размера в определенную позицию. Если в первом параметре указано отрицательное значение, то пространство добавляется в конец;
- `addStretch([stretch=0])` — добавляет пустое растягиваемое пространство с нулевым минимальным размером и фактором растяжения `stretch` в конец контейнера. Это пространство можно сравнить с пружиной, вставленной между компонентами, а параметр `stretch` с жесткостью пружины;
- `insertStretch(<Индекс>[, stretch=0])` — метод аналогичен методу `addStretch()`, но добавляет растягиваемое пространство в указанную позицию. Если в первом параметре указано отрицательное значение, то пространство добавляется в конец контейнера.

Параметр `alignment` в методах `addWidget()` и `insertWidget()` задает выравнивание компонента внутри ячейки. В этом параметре можно указать следующие атрибуты из класса `QtCore.Qt`:

- `AlignLeft` — `1` — горизонтальное выравнивание по левому краю;
- `AlignRight` — `2` — горизонтальное выравнивание по правому краю;
- `AlignHCenter` — `4` — горизонтальное выравнивание по центру;
- `AlignJustify` — `8` — заполнение всего пространства;
- `AlignTop` — `32` — вертикальное выравнивание по верхнему краю;
- `AlignBottom` — `64` — вертикальное выравнивание по нижнему краю;
- `AlignVCenter` — `128` — вертикальное выравнивание по центру;
- `AlignCenter` — `AlignVCenter | AlignHCenter` — горизонтальное и вертикальное выравнивание по центру;
- `AlignAbsolute` — `16` — если в методе `setLayoutDirection()` из класса `QWidget` указан атрибут `QtCore.Qt.RightToLeft`, то атрибут `AlignLeft` задает выравнивание по правому краю, а атрибут `AlignRight` — по левому краю. Чтобы атрибут `AlignLeft` всегда соответствовал именно левому краю, необходимо указать комбинацию `AlignAbsolute | AlignLeft`. Аналогично следует поступить с атрибутом `AlignRight`.

Можно задавать комбинацию атрибутов. В комбинации допускается указывать только один атрибут горизонтального выравнивания и только один атрибут вертикального выравнивания. Например, комбинация `AlignLeft | AlignTop` задает выравнивание по левому и верхнему краю. Противоречивые значения приводят к непредсказуемым результатам.

Помимо рассмотренных методов, контейнеры поддерживают следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setDirection(<Направление>)` — задает направление вывода компонентов. В параметре можно указать следующие атрибуты из класса `QBoxLayout`:
  - `LeftToRight` — `0` — слева направо (значение по умолчанию для горизонтального контейнера);
  - `RightToLeft` — `1` — справа налево;

- `TopToBottom` — 2 — сверху вниз (значение по умолчанию для вертикального контейнера);
- `BottomToTop` — 3 — снизу вверх;

- `setMargin(<Отступ>)` — задает величину отступа от границ контейнера до компонентов;
- `setSpacing(<Расстояние>)` — задает расстояние между компонентами.

22.3. Выравнивание по сетке

Помимо выравнивания компонентов по горизонтали и вертикали существует возможность размещения компонентов внутри ячеек сетки. Для выравнивания компонентов по сетке предназначен класс `QGridLayout`. Иерархия наследования:

```
(QObject, QLayoutItem) — QLayout — QGridLayout
```

Создать экземпляр класса `QGridLayout` можно следующим образом:

```
<Объект> = QGridLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент. Если параметр не указан, то необходимо передать ссылку на сетку в метод `setLayout()` родительского компонента. Типичный пример использования класса `QGridLayout` выглядит так:

```
window = QtGui.QWidget()                # Родительский компонент
button1 = QtGui.QPushButton("1")
button2 = QtGui.QPushButton("2")
button3 = QtGui.QPushButton("3")
button4 = QtGui.QPushButton("4")
grid = QtGui.QGridLayout()              # Создаем сетку
grid.addWidget(button1, 0, 0)           # Добавляем компоненты
grid.addWidget(button2, 0, 1)
grid.addWidget(button3, 1, 0)
grid.addWidget(button4, 1, 1)
window.setLayout(grid)                  # Передаем ссылку родителю
```

Добавить компоненты и удалить их позволяют следующие методы:

- `addWidget()` — добавляет компонент в указанную ячейку сетки. Метод имеет следующие форматы:

  ```
  addWidget(<Компонент>, <Строка>, <Столбец>[, alignment=0])
  addWidget(<Компонент>, <Строка>, <Столбец>, <Количество строк>,
            <Количество столбцов>[, alignment=0])
  ```

  В первом параметре указывается ссылка на компонент, во втором параметре передается индекс строки, а в третьем — индекс столбца. Нумерация строк и столбцов начинается с нуля. Параметр `<Количество строк>` задает количество объединенных ячеек по вертикали, а параметр `<Количество столбцов>` — по горизонтали. Параметр `alignment` задает выравнивание компонента внутри ячейки. Значения, которые можно указать в этом параметре, мы рассматривали в предыдущем разделе. Пример:

  ```
  grid = QtGui.QGridLayout()
  grid.addWidget(button1, 0, 0, alignment=QtCore.Qt.AlignLeft)
  grid.addWidget(button2, 0, 1, QtCore.Qt.AlignRight)
  grid.addWidget(button3, 1, 0, 1, 2)
  ```

- `addLayout()` — добавляет контейнер в указанную ячейку сетки. Метод имеет следующие форматы:

```
addLayout(<Контейнер>, <Строка>, <Столбец>[, alignment=0])
addLayout(<Контейнер>, <Строка>, <Столбец>, <Количество строк>,
          <Количество столбцов>[, alignment=0])
```

  В первом параметре указывается ссылка на контейнер. Остальные параметры аналогичны параметрам метода `addWidget()`.

Класс `QGridLayout` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowMinimumHeight(<Индекс>, <Высота>)` — задает минимальную высоту строки с индексом `<Индекс>`;
- `setColumnMinimumWidth(<Индекс>, <Ширина>)` — задает минимальную ширину столбца с индексом `<Индекс>`;
- `setRowStretch(<Индекс>, <Фактор растяжения>)` — задает фактор растяжения для строки с индексом `<Индекс>`;
- `setColumnStretch(<Индекс>, <Фактор растяжения>)` — задает фактор растяжения для столбца с индексом `<Индекс>`;
- `setMargin(<Отступ>)` — задает величину отступа от границ сетки до компонентов;
- `setSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали и вертикали;
- `setHorizontalSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали;
- `setVerticalSpacing(<Значение>)` — задает расстояние между компонентами по вертикали;
- `rowCount()` — возвращает количество строк сетки;
- `columnCount()` — возвращает количество столбцов сетки.

22.4. Выравнивание компонентов формы

Класс `QFormLayout` позволяет выравнивать компоненты формы. Контейнер по умолчанию состоит из двух столбцов. Первый столбец предназначен для вывода надписи, а второй столбец — для вывода компонента, например, текстового поля. При этом надпись связывается с компонентом, что позволяет назначать клавиши быстрого доступа, указав символ `&` перед буквой внутри текста надписи. После нажатия комбинации клавиш быстрого доступа (комбинация <Alt>+буква) в фокусе окажется компонент, расположенный справа от надписи. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QFormLayout
```

Создать экземпляр класса `QFormLayout` можно следующим образом:

```
<Объект> = QFormLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент. Если параметр не указан, то необходимо передать ссылку на контейнер в метод `setLayout()` родительского компонента.

Типичный пример использования класса `QFormLayout` выглядит так:

```
window = QtGui.QWidget()
lineEdit = QtGui.QLineEdit()
textEdit = QtGui.QTextEdit()
button1 = QtGui.QPushButton("О&тправить")
button2 = QtGui.QPushButton("О&чистить")
hbox = QtGui.QHBoxLayout()
hbox.addWidget(button1)
hbox.addWidget(button2)
form = QtGui.QFormLayout()
form.addRow("&Название:", lineEdit)
form.addRow("&Описание:", textEdit)
form.addRow(hbox)
window.setLayout(form)
```

Результат выполнения этого кода показан на рис. 22.1.

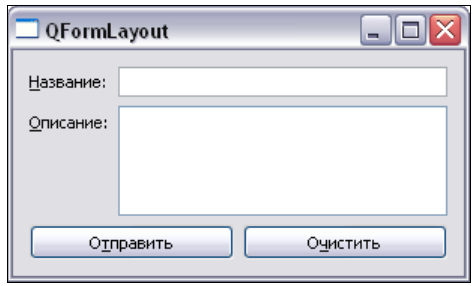

Рис. 22.1. Пример использования класса `QFormLayout`

Класс `QFormLayout` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addRow()` — добавляет строку в конец контейнера. Форматы метода:

  ```
  addRow(<Текст надписи>, <Компонент>)
  addRow(<Текст надписи>, <Контейнер>)
  addRow(<Компонент1>, <Компонент2>)
  addRow(<Компонент>, <Контейнер>)
  addRow(<Компонент>)
  addRow(<Контейнер>)
  ```

  В параметре `<Текст надписи>` можно указать текст, внутри которого перед какой-либо буквой указан символ `&`. В этом случае надпись связывается с компонентом, указанном во втором параметре. После нажатия комбинации клавиш быстрого доступа (комбинация <Alt>+буква) этот компонент окажется в фокусе ввода. Если в первом параметре указан экземпляр класса `QLabel`, то связь с компонентом необходимо устанавливать вручную, передав ссылку на компонент в метод `setBuddy()`. Если указан только один параметр, то компонент (или контейнер) займет сразу два столбца;

- `insertRow()` — добавляет строку в указанную позицию контейнера. Если указано отрицательное значение в первом параметре, то компонент добавляется в конец контейнера.

Форматы метода:

```
insertRow(<Индекс>, <Текст надписи>, <Компонент>)
insertRow(<Индекс>, <Текст надписи>, <Контейнер>)
insertRow(<Индекс>, <Компонент1>, <Компонент2>)
insertRow(<Индекс>, <Компонент>, <Контейнер>)
insertRow(<Индекс>, <Компонент>)
insertRow(<Индекс>, <Контейнер>)
```

- `setFormAlignment(<Режим>)` — задает режим выравнивания формы. Допустимые значения мы рассматривали в *разд. 22.2*. Пример:

```
form.setFormAlignment(
    QtCore.Qt.AlignRight | QtCore.Qt.AlignBottom)
```

- `setLabelAlignment(<Режим>)` — задает режим выравнивания надписи. Допустимые значения мы рассматривали в *разд. 22.2*. Пример выравнивания по правому краю:

```
form.setLabelAlignment(QtCore.Qt.AlignRight)
```

- `setRowWrapPolicy(<Режим>)` — задает местоположение надписи. В качестве параметра указываются следующие атрибуты из класса `QFormLayout`:
  - `DontWrapRows` — `0` — надписи расположены слева от компонентов;
  - `WrapLongRows` — `1` — длинные надписи могут находиться выше компонентов, а короткие надписи — слева от компонентов;
  - `WrapAllRows` — `2` — надписи расположены выше компонентов;
- `setFieldGrowthPolicy(<Режим>)` — задает режим управления размерами компонентов. В качестве параметра указываются следующие атрибуты из класса `QFormLayout`:
  - `FieldsStayAtSizeHint` — `0` — размеры компонентов будут соответствовать рекомендуемым (возвращаемым методом `sizeHint()`);
  - `ExpandingFieldsGrow` — `1` — компоненты, для которых установлена политика изменения размеров `QSizePolicy.Expanding` или `QSizePolicy.MinimumExpanding`, будут занимать всю доступную ширину. Размеры остальных компонентов будут соответствовать рекомендуемым;
  - `AllNonFixedFieldsGrow` — `2` — все компоненты (если это возможно) будут занимать всю доступную ширину;
- `setMargin(<Отступ>)` — задает величину отступа от границ формы до компонентов;
- `setSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали и вертикали;
- `setHorizontalSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали;
- `setVerticalSpacing(<Значение>)` — задает расстояние между компонентами по вертикали.

22.5. Классы *QStackedLayout* и *QStackedWidget*

Класс `QStackedLayout` реализует стек компонентов. В один момент времени показывается только один компонент. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QStackedLayout
```

Создать экземпляр класса QStackedLayout можно следующим образом:

```
<Объект> = QStackedLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент или контейнер. Если параметр не указан, то необходимо передать ссылку на контейнер в метод setLayout() родительского компонента.

Класс QStackedLayout содержит следующие методы:

- setStackingMode(<Режим>) — задает режим отображения компонентов. В параметре могут быть указаны следующие атрибуты из класса QStackedLayout:
  - StackOne — 0 — только один компонент видим (значение по умолчанию);
  - StackAll — 1 — видны все компоненты;
- stackingMode() — возвращает режим отображения компонентов;
- addWidget(<Компонент>) — добавляет компонент в конец контейнера. Метод возвращает индекс добавленного компонента;
- insertWidget(<Индекс>, <Компонент>) — добавляет компонент в указанную позицию контейнера. Метод возвращает индекс добавленного компонента;
- removeWidget(<Компонент>) — удаляет компонент из контейнера;
- count() — возвращает количество компонентов внутри контейнера;
- currentIndex() — возвращает индекс видимого компонента;
- currentWidget() — возвращает ссылку на видимый компонент;
- widget(<Индекс>) — возвращает ссылку на компонент, который расположен по указанному индексу, или значение None;
- setCurrentIndex(int) — делает видимым компонент с указанным в параметре индексом. Метод является слотом;
- setCurrentWidget(QWidget *) — делает видимым компонент, ссылка на который указана в параметре. Метод является слотом.

Класс QStackedLayout содержит следующие сигналы:

- currentChanged(int) — генерируется при изменении видимого компонента. Через параметр внутри обработчика доступен индекс нового компонента;
- widgetRemoved(int) — генерируется при удалении компонента из контейнера. Через параметр внутри обработчика доступен индекс компонента.

Класс QStackedWidget также реализует стек компонентов, но создает новый компонент, а не контейнер. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QStackedWidget
```

Создать экземпляр класса QStackedWidget можно следующим образом:

```
<Объект> = QStackedWidget([<Родитель>])
```

Класс QStackedWidget содержит методы addWidget(), insertWidget(), removeWidget(), count(), currentIndex(), currentWidget(), widget(), setCurrentIndex() и setCurrentWidget(), которые выполняют аналогичные действия, что и одноименные методы в классе QStackedLayout. Кроме того, класс QStackedWidget наследует все методы из базовых классов и содержит два дополнительных метода:

- `indexOf(<Компонент>)` — возвращает индекс компонента, ссылка на который указана в параметре;
- `__len__()` — возвращает количество компонентов. Метод вызывается при использовании функции `len()`, а также для проверки объекта на логическое значение.

Чтобы отследить изменения внутри компонента, следует назначить обработчики сигналов `currentChanged(int)` и `widgetRemoved(int)`.

22.6. Класс *QSizePolicy*

Если в вертикальный контейнер большой высоты добавить надпись и кнопку, то под надпись будет выделено максимальное пространство, а кнопка займет пространство, достаточное для рекомендуемых размеров, которые возвращает метод `sizeHint()`. Управление размерами компонентов внутри контейнера определяется правилами, установленными с помощью класса `QSizePolicy`. Установить правила для компонента можно с помощью метода `setSizePolicy(<QSizePolicy>)` из класса `QWidget`, а получить значение с помощью метода `sizePolicy()`.

Создать экземпляр класса `QSizePolicy` можно следующим способом:

```
<Объект> = QSizePolicy([<Правило для горизонтали>,
                        <Правило для вертикали>[, <Тип компонента>]])
```

Если параметры не заданы, то размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`. В первом и втором параметрах указываются следующие атрибуты из класса `QSizePolicy`:

- `Fixed` — размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`;
- `Minimum` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Размер может быть увеличен компоновщиком;
- `Maximum` — размер, возвращаемый методом `sizeHint()`, является максимальным для компонента. Размер может быть уменьшен компоновщиком;
- `Preferred` — размер, возвращаемый методом `sizeHint()`, является предпочтительным, но может быть как увеличен, так и уменьшен;
- `Expanding` — размер, возвращаемый методом `sizeHint()`, может быть как увеличен, так и уменьшен. Компоновщик должен предоставить компоненту столько пространства, сколько возможно;
- `MinimumExpanding` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Компоновщик должен предоставить компоненту столько пространства, сколько возможно;
- `Ignored` — размер, возвращаемый методом `sizeHint()`, игнорируется. Компонент получит столько пространства, сколько возможно.

Изменить значения уже после создания экземпляра класса `QSizePolicy` позволяют методы `setHorizontalPolicy(<Правило для горизонтали>)` и `setVerticalPolicy(<Правило для вертикали>)`.

С помощью методов `setHorizontalStretch(<Фактор для горизонтали>)` и `setVerticalStretch(<Фактор для вертикали>)` можно указать фактор растяжения. Чем

больше указанное значение относительно значения, заданного в других компонентах, тем больше места будет выделяться под компонент. Этот параметр можно сравнить с жесткостью пружины.

Можно указать, что минимальная высота компонента зависит от его ширины. Для этого необходимо передать значение `True` в метод `setHeightForWidth(<Флаг>)`. Кроме того, следует переопределить метод `heightForWidth(<Ширина>)` в классе компонента. Метод должен возвращать высоту компонента в соответствии с указанной в параметре шириной.

22.7. Объединение компонентов в группу

Состояние некоторых компонентов может зависеть от состояния других компонентов, например, из нескольких переключателей можно выбрать только один. В этом случае компоненты объединяют в группу. Группа компонентов отображается внутри рамки, на границе которой выводится текст подсказки. Реализовать группу позволяет класс `QGroupBox`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QGroupBox
```

Создать экземпляр класса `QGroupBox` можно следующим образом:

```
<Объект> = QGroupBox([<Родитель>])
<Объект> = QGroupBox(<Текст>[, <Родитель>])
```

В необязательном параметре `<Родитель>` можно указать ссылку на родительский компонент. Параметр `<Текст>` задает текст подсказки, которая отобразится на верхней границе рамки. Внутри текста подсказки символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы первый компонент внутри группы окажется в фокусе ввода.

После создания экземпляра класса `QGroupBox` следует добавить компоненты в какой-либо контейнер, а затем передать ссылку на контейнер в метод `setLayout()`. Типичный пример использования класса `QGroupBox` выглядит так:

```
window = QtGui.QWidget()
mainbox = QtGui.QVBoxLayout()
radio1 = QtGui.QRadioButton("&Да")
radio2 = QtGui.QRadioButton("&Нет")
box = QtGui.QGroupBox("&Вы знаете язык Python?") # Объект группы
hbox = QtGui.QHBoxLayout() # Контейнер для группы
hbox.addWidget(radio1)     # Добавляем компоненты
hbox.addWidget(radio2)
box.setLayout(hbox)        # Передаем ссылку на контейнер
mainbox.addWidget(box)     # Добавляем группу в главный контейнер
window.setLayout(mainbox)  # Передаем ссылку на главный контейнер в окно
radio1.setChecked(True)    # Выбираем первый переключатель
```

Результат выполнения этого кода показан на рис. 22.2.

Класс `QGroupBox` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setTitle(<Текст>)` — задает текст подсказки;
- `setAlignment(<Выравнивание>)` — задает горизонтальное местоположение текста подсказки. В параметре указываются следующие атрибуты из класса `QtCore.Qt`: `AlignLeft`, `AlignHCenter` или `AlignRight`. Пример:

```
box.setAlignment(QtCore.Qt.AlignRight)
```

- `setCheckable(<Флаг>)` — если в параметре указать значение `True`, то перед текстом подсказки будет отображен флажок. Если флажок установлен, то группа будет активной, а если флажок снят, то все компоненты внутри группы станут неактивными. По умолчанию флажок не отображается;

Рис. 22.2. Пример использования класса `QGroupBox`

- `isCheckable()` — возвращает значение `True`, если флажок выводится перед надписью, и `False` — в противном случае;
- `setChecked(<Флаг>)` — если в параметре указать значение `True`, то флажок, отображаемый перед текстом подсказки, будет установлен. Значение `False` сбрасывает флажок;
- `isChecked()` — возвращает значение `True`, если флажок, отображаемый перед текстом подсказки, установлен, и `False` — в противном случае;
- `setFlat(<Флаг>)` — если в параметре указано значение `True`, то отображается только верхняя граница рамки, а если `False` — то все границы рамки;
- `isFlat()` — возвращает значение `True`, если отображается только верхняя граница рамки, и `False` — если все границы рамки.

Класс `QGroupBox` содержит следующие сигналы:

- `clicked(bool=0)` — генерируется при щелчке мышью на флажке, выводимом перед текстом подсказки. Если состояние флажка изменяется с помощью метода `setChecked()`, то сигнал не генерируется. Через параметр внутри обработчика доступно значение `True`, если флажок установлен, и `False` — если сброшен;
- `toggled(bool)` — генерируется при изменении статуса флажка, выводимого перед текстом подсказки. Через параметр внутри обработчика доступно значение `True`, если флажок установлен, и `False` — если сброшен.

22.8. Панель с рамкой

Класс `QFrame` расширяет возможности класса `QWidget` за счет добавления рамки различного стиля вокруг компонента. Этот класс наследуют в свою очередь некоторые компоненты, например надписи, многострочные текстовые поля и др. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame
```

Конструктор класса QFrame имеет следующий формат:

```
<Объект> = QFrame([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, то компонент будет обладать своим собственным окном. Если в параметре flags указан тип окна, то компонент, имея родителя, будет обладать своим собственным окном, но будет привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре flags, мы уже рассматривали в *разд. 20.2*.

Класс QFrame содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setFrameShape(<Форма>) — задает форму рамки. Могут быть указаны следующие атрибуты из класса QFrame:
  - NoFrame — 0 — нет рамки;
  - Box — 1 — прямоугольная рамка;
  - Panel — 2 — панель, которая может быть выпуклой или вогнутой;
  - WinPanel — 3 — панель со стилем, принятым в Windows. Ширина границы 2 пиксела. Панель может быть выпуклой или вогнутой;
  - HLine — 4 — горизонтальная линия. Используется как разделитель;
  - VLine — 5 — вертикальная линия без содержимого;
  - StyledPanel — 6 — панель, внешний вид которой зависит от текущего стиля. Панель может быть выпуклой или вогнутой;
- setFrameShadow(<Тень>) — задает стиль тени. Могут быть указаны следующие атрибуты из класса QFrame:
  - Plain — 16 — нет эффектов;
  - Raised — 32 — панель отображается выпуклой;
  - Sunken — 48 — панель отображается вогнутой;
- setFrameStyle(<Стиль>) — задает форму рамки и стиль тени одновременно. В качестве значения указывается комбинация атрибутов из класса QFrame через оператор |. Пример:

  ```
  frame.setFrameStyle(QtGui.QFrame.Panel | QtGui.QFrame.Raised)
  ```

- setLineWidth(<Ширина>) — задает ширину линии рамки;
- setMidLineWidth(<Ширина>) — задает ширину средней линии рамки. Средняя линия используется для создания эффекта выпуклости и вогнутости и доступна только для форм рамки Box, HLine и VLine.

22.9. Панель с вкладками

Для создания панели с вкладками предназначен класс QTabWidget. Панель состоит из области заголовка с ярлыками и набора вкладок с различными компонентами. В один момент времени показывается содержимое только одной вкладки. Щелчок мышью на ярлыке в области заголовка приводит к отображению содержимого соответствующей вкладки.

Иерархия наследования для класса QTabWidget выглядит так:

```
(QObject, QPaintDevice) — QWidget — QTabWidget
```

Конструктор класса QTabWidget имеет следующий формат:

```
<Объект> = QTabWidget([<Родитель>])
```

В параметре <Родитель> указывается ссылка на родительский компонент. Если параметр не указан, то компонент будет обладать своим собственным окном. Типичный пример использования класса QTabWidget выглядит так:

```
window = QtGui.QWidget()
tab = QtGui.QTabWidget()
tab.addTab(QtGui.QLabel("Содержимое вкладки 1"), "Вкладка &1")
tab.addTab(QtGui.QLabel("Содержимое вкладки 2"), "Вкладка &2")
tab.addTab(QtGui.QLabel("Содержимое вкладки 3"), "Вкладка &3")
tab.setCurrentIndex(0)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(tab)
window.setLayout(vbox)
window.show()
```

Результат выполнения этого кода показан на рис. 22.3.

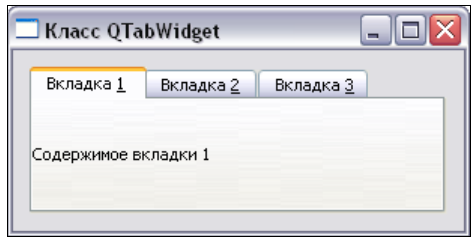

Рис. 22.3. Пример использования класса QTabWidget

Класс QTabWidget содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- addTab() — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  addTab(<Компонент>, <Текст заголовка>)
  addTab(<Компонент>, <QIcon>, <Текст заголовка>)
  ```

  В параметре <Компонент> указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр <Текст заголовка> задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ &, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы соответствующая вкладка будет отображена. Параметр <QIcon> позволяет указать иконку (экземпляр класса QIcon), которая отобразится перед текстом в области заголовка.

Пример указания стандартной иконки:

```
style = window.style()
icon = style.standardIcon(QtGui.QStyle.SP_DriveNetIcon)
tab.addTab(QtGui.QLabel("Содержимое вкладки 1"), icon,
           "Вкладка &1")
```

Пример загрузки иконки из файла:

```
icon = QtGui.QIcon("icon.png")
tab.addTab(QtGui.QLabel("Содержимое вкладки 1"), icon,
           "Вкладка &1")
```

- `insertTab()` — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  insertTab(<Индекс>, <Компонент>, <Текст заголовка>)
  insertTab(<Индекс>, <Компонент>, <QIcon>, <Текст заголовка>)
  ```

- `removeTab(<Индекс>)` — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;
- `clear()` — удаляет все вкладки, при этом компоненты, которые отображались на вкладках, не удаляются;
- `setTabText(<Индекс>, <Текст заголовка>)` — задает текст заголовка для вкладки с указанным индексом;
- `setElideMode(<Режим>)` — задает режим обрезки текста в названии вкладки, если он не помещается в отведенную область. В месте пропуска выводится троеточие. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `ElideLeft` — `0` — текст обрезается слева;
  - `ElideRight` — `1` — текст обрезается справа;
  - `ElideMiddle` — `2` — текст обрезается посередине;
  - `ElideNone` — `3` — текст не обрезается;
- `tabText(<Индекс>)` — возвращает текст заголовка вкладки с указанным индексом;
- `setTabIcon(<Индекс>, <QIcon>)` — устанавливает иконку перед текстом в заголовке вкладки с указанным индексом. Во втором параметре указывается экземпляр класса `QIcon`;
- `setTabPosition(<Позиция>)` — задает позицию области заголовка. Могут быть указаны следующие атрибуты из класса `QTabWidget`:
  - `North` — `0` — сверху;
  - `South` — `1` — снизу;
  - `West` — `2` — слева;
  - `East` — `3` — справа.

  Пример указания значения:

  ```
  tab.setTabPosition(QtGui.QTabWidget.South)
  ```

- `setTabShape(<Форма>)` — задает форму углов ярлыка вкладки в области заголовка. Могут быть указаны следующие атрибуты из класса `QTabWidget`:

- `Rounded` — `0` — скругленные углы (значение по умолчанию);
- `Triangular` — `1` — треугольная форма;

- `setTabsClosable(<Флаг>)` — если в качестве параметра указано значение `True`, то после текста заголовка будет отображена кнопка закрытия вкладки. При нажатии этой кнопки генерируется сигнал `tabCloseRequested(int)`;
- `setMovable(<Флаг>)` — если в качестве параметра указано значение `True`, то ярлыки вкладок можно перемещать с помощью мыши;
- `setDocumentMode(<Флаг>)` — если в качестве параметра указано значение `True`, то область компонента не будет отображаться как панель;
- `setUsesScrollButtons(<Флаг>)` — если в качестве параметра указано значение `True`, то когда все ярлыки вкладок не помещаются в область заголовка, появляются две кнопки, с помощью которых можно прокручивать область заголовка, тем самым отображая только часть ярлыков. Значение `False` запрещает сокрытие ярлыков;
- `setTabToolTip(<Индекс>, <Текст>)` — задает текст всплывающей подсказки для ярлыка вкладки с указанным индексом;
- `setTabWhatsThis(<Индекс>, <Текст>)` — задает текст справки для ярлыка вкладки с указанным индексом;
- `setTabEnabled(<Индекс>, <Флаг>)` — если во втором параметре указано значение `False`, то вкладка с указанным в первом параметре индексом станет недоступной. Значение `True` делает вкладку доступной;
- `isTabEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;
- `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

  ```
  print(tab.count(), len(tab))
  ```

- `currentIndex()` — возвращает индекс видимой вкладки;
- `currentWidget()` — возвращает ссылку на компонент, расположенный на видимой вкладке;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен компонент `<Компонент>`. Если компонент не найден, возвращается значение `-1`;
- `setCurrentIndex(int)` — делает видимой вкладку с указанным в параметре индексом. Метод является слотом;
- `setCurrentWidget(QWidget *)` — делает видимым компонент, ссылка на который указана в параметре. Метод является слотом.

Класс `QTabWidget` содержит следующие сигналы:

- `currentChanged(int)` — генерируется при изменении вкладки. Через параметр внутри обработчика доступен индекс новой вкладки;
- `tabCloseRequested(int)` — генерируется при нажатии кнопки закрытия вкладки. Через параметр внутри обработчика доступен индекс вкладки.

22.10. Компонент "аккордеон"

Класс QToolBox позволяет создать компонент с несколькими вкладками. Изначально отображается содержимое только одной вкладки, а у остальных доступны только заголовки. После щелчка мышью на заголовке вкладки она открывается, а остальные сворачиваются. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QToolBox
```

Конструктор класса QToolBox имеет следующий формат:

```
<Объект> = QToolBox([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, то компонент будет обладать своим собственным окном. В параметре flags может быть указан тип окна. Пример использования класса QToolBox:

```
window = QtGui.QWidget()
toolBox = QtGui.QToolBox()
toolBox.addItem(QtGui.QLabel("Содержимое вкладки 1"), "Вкладка &1")
toolBox.addItem(QtGui.QLabel("Содержимое вкладки 2"), "Вкладка &2")
toolBox.addItem(QtGui.QLabel("Содержимое вкладки 3"), "Вкладка &3")
toolBox.setCurrentIndex(0)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(toolBox)
window.setLayout(vbox)
window.show()
```

Класс QToolBox содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- addItem() — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  addItem(<Компонент>, <Текст заголовка>)
  addItem(<Компонент>, <QIcon>, <Текст заголовка>)
  ```

  В параметре <Компонент> указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр <Текст заголовка> задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ &, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы соответствующая вкладка будет отображена. Параметр <QIcon> позволяет указать иконку (экземпляр класса QIcon), которая отобразится перед текстом в области заголовка;

- insertItem() — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  insertItem(<Индекс>, <Компонент>, <Текст заголовка>)
  insertItem(<Индекс>, <Компонент>, <QIcon>, <Текст заголовка>)
  ```

- removeItem(<Индекс>) — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;

- `setItemText(<Индекс>, <Текст заголовка>)` — задает текст заголовка для вкладки с указанным индексом;
- `itemText(<Индекс>)` — возвращает текст заголовка вкладки с указанным индексом;
- `setItemIcon(<Индекс>, <QIcon>)` — устанавливает иконку перед текстом в заголовке вкладки с указанным индексом. Во втором параметре указывается экземпляр класса `QIcon`;
- `setItemToolTip(<Индекс>, <Текст>)` — задает текст всплывающей подсказки для ярлыка вкладки с указанным индексом;
- `setItemEnabled(<Индекс>, <Флаг>)` — если во втором параметре указано значение `False`, то вкладка с указанным в первом параметре индексом станет недоступной. Значение `True` делает вкладку доступной;
- `isItemEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;
- `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

```
print(toolBox.count(), len(toolBox))
```

- `currentIndex()` — возвращает индекс видимой вкладки;
- `currentWidget()` — возвращает ссылку на компонент, который расположен на видимой вкладке;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен компонент `<Компонент>`. Если компонент не найден, возвращается значение `-1`;
- `setCurrentIndex(int)` — делает видимой вкладку с указанным в параметре индексом. Метод является слотом;
- `setCurrentWidget(QWidget *)` — делает видимым компонент, ссылка на который указана в параметре. Метод является слотом.

При изменении вкладки генерируется сигнал `currentChanged(int)`. Через параметр внутри обработчика доступен индекс новой вкладки.

22.11. Панели с изменяемым размером

Класс `QSplitter` позволяет изменять размеры добавленных компонентов с помощью мыши, взявшись за границу между компонентами. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QSplitter
```

Конструктор класса `QSplitter` имеет два формата:

```
<Объект> = QSplitter([parent=<Родитель>])
<Объект> = QSplitter(<Ориентация>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Параметр `<Ориентация>` задает ориентацию размещения компонентов. Могут быть заданы

атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) из класса `QtCore.Qt`. Если параметр не указан, то компоненты размещаются по горизонтали. Пример использования класса `QSplitter`:

```
window = QtGui.QWidget()
splitter = QtGui.QSplitter(QtCore.Qt.Vertical)
label1 = QtGui.QLabel("Содержимое компонента 1")
label2 = QtGui.QLabel("Содержимое компонента 2")
label1.setFrameStyle(QtGui.QFrame.Box | QtGui.QFrame.Plain)
label2.setFrameStyle(QtGui.QFrame.Box | QtGui.QFrame.Plain)
splitter.addWidget(label1)
splitter.addWidget(label2)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(splitter)
window.setLayout(vbox)
window.show()
```

Класс `QSplitter` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addWidget(<Компонент>)` — добавляет компонент в конец контейнера;
- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент в указанную позицию. Если компонент был добавлен ранее, то он будет перемещен в новую позицию;
- `setOrientation(<Ориентация>)` — задает ориентацию размещения компонентов. Могут быть заданы атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) из класса `QtCore.Qt`;
- `setHandleWidth(<Ширина>)` — задает ширину компонента-разделителя, взявшись за который мышью можно изменить размер области;
- `saveState()` — возвращает экземпляр класса `QByteArray` с размерами всех областей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода `restoreState(<QByteArray>)`;
- `setChildrenCollapsible(<Флаг>)` — если в параметре указано значение `False`, то пользователь не сможет уменьшить размеры всех компонентов до нуля. По умолчанию размер может быть нулевым, даже если установлены минимальные размеры компонента;
- `setCollapsible(<Индекс>, <Флаг>)` — значение `False` в параметре `<Флаг>` запрещает уменьшение размеров до нуля для компонента с указанным индексом;
- `setOpaqueResize(<Флаг>)` — если в качестве параметра указано значение `False`, то размеры компонентов изменятся только после окончания перемещения границы и отпускания кнопки мыши. В процессе перемещения мыши вместе с ней будет перемещаться специальный компонент в виде линии;
- `setStretchFactor(<Индекс>, <Фактор>)` — задает фактор растяжения для компонента с указанным индексом;
- `setSizes(<Список>)` — задает размеры всех компонентов. Для горизонтального контейнера указывается список со значениями ширины каждого компонента, а для вертикального контейнера — список со значениями высоты каждого компонента;
- `sizes()` — возвращает список с размерами (шириной или высотой). Пример:

```
print(splitter.sizes()) # Результат: [308, 15]
```

- `count()` — возвращает количество компонентов. Получить количество компонентов можно также с помощью функции `len()`:

  ```
  print(splitter.count(), len(splitter))
  ```

- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `indexOf(<Компонент>)` — возвращает индекс области, в которой расположен компонент `<Компонент>`. Если компонент не найден, возвращается значение `-1`.

При изменении размеров генерируется сигнал `splitterMoved(int,int)`. Через первый параметр внутри обработчика доступна новая позиция, а через второй параметр — индекс перемещаемого разделителя.

22.12. Область с полосами прокрутки

Класс `QScrollArea` реализует область с полосами прокрутки. Если компонент не помещается в размеры области, то автоматически отображаются полосы прокрутки. Изменение положения полос прокрутки с помощью мыши автоматически приводит к прокрутке содержимого области. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QScrollArea
```

Конструктор класса `QScrollArea` имеет следующий формат:

```
<Объект> = QScrollArea([<Родитель>])
```

Класс `QScrollArea` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setWidget(<Компонент>)` — добавляет компонент в область прокрутки;
- `setWidgetResizable(<Флаг>)` — если в качестве параметра указано значение `True`, то при изменении размеров области будут изменяться и размеры компонента. Значение `False` запрещает изменение размеров компонента;
- `setAlignment(<Выравнивание>)` — задает местоположение компонента внутри области, когда размеры области больше размеров компонента. Пример:

  ```
  scrollArea.setAlignment(QtCore.Qt.AlignCenter)
  ```

- `ensureVisible(<X>, <Y>[, xMargin=50][, yMargin=50])` — прокручивает область к точке с координатами `(<X>, <Y>)` и полями `xMargin` и `yMargin`;
- `ensureWidgetVisible(<Компонент>[, xMargin=50][, yMargin=50])` — прокручивает область таким образом, чтобы `<Компонент>` был видим;
- `widget()` — возвращает ссылку на компонент, который расположен внутри области, или значение `None`;
- `takeWidget()` — удаляет компонент из области и возвращает ссылку на него. Сам компонент не удаляется.

Класс `QScrollArea` наследует следующие методы из класса `QAbstractScrollArea` (перечислены только основные методы; полный список смотрите в документации):

- `horizontalScrollBar()` — возвращает ссылку на горизонтальную полосу прокрутки (экземпляр класса `QScrollBar`);

- `verticalScrollBar()` — возвращает ссылку на вертикальную полосу прокрутки (экземпляр класса `QScrollBar`);
- `cornerWidget()` — возвращает ссылку на компонент, расположенный в правом нижнем углу между двумя полосами прокрутки, или значение `None`;
- `viewport()` — возвращает ссылку на окно области прокрутки;
- `setHorizontalScrollBarPolicy(<Режим>)` — устанавливает режим отображения горизонтальной полосы прокрутки;
- `setVerticalScrollBarPolicy(<Режим>)` — устанавливает режим отображения вертикальной полосы прокрутки. В параметре `<Режим>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `ScrollBarAsNeeded` — `0` — полоса прокрутки отображается только в том случае, если размеры компонента больше размеров области;
  - `ScrollBarAlwaysOff` — `1` — полоса прокрутки никогда не отображается;
  - `ScrollBarAlwaysOn` — `2` — полоса прокрутки всегда отображается;
- `setViewportMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)` — задает отступ от границ области до компонента. По умолчанию отступы равны нулю.

ГЛАВА 23

Основные компоненты

Практически все компоненты графического интерфейса наследуют классы `QObject` и `QWidget`. Следовательно, методы этих классов, которые мы рассматривали в предыдущих главах, доступны всем компонентам. Если компонент не имеет родителя, то он обладает собственным окном и, например, его положение отсчитывается относительно экрана. Если же компонент имеет родителя, то его положение отсчитывается относительно родительского компонента. Это обстоятельство важно учитывать при работе с компонентами. Обращайте внимание на иерархию наследования, которую мы будем показывать для каждого компонента.

23.1. Надпись

Надпись применяется для вывода подсказки пользователю, информирования пользователя о ходе выполнения операции, назначения клавиш быстрого доступа применительно к другому компоненту, а также для вывода изображений и анимации. Кроме того, надписи позволяют отображать текст в формате HTML, отформатированный с помощью CSS, что делает надпись самым настоящим браузером. В библиотеке PyQt надпись реализуется с помощью класса `QLabel`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLabel
```

Конструктор класса `QLabel` имеет два формата:

```
<Объект> = QLabel([parent=<Родитель>][, flags=<Тип окна>])
<Объект> = QLabel(<Текст>[, parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном, тип которого можно задать с помощью параметра `flags`. Параметр `<Текст>` позволяет задать текст, который будет отображен на надписи. Пример:

```
label = QtGui.QLabel("Текст надписи", flags=QtCore.Qt.Window)
label.resize(300, 50)
label.show()
```

Класс `QLabel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText(<Текст>)` — задает текст, который будет отображен на надписи. Можно указать как обычный текст, так и текст в формате HTML, который содержит форматирование с помощью CSS.

Пример:

```
label.setText("Текст <b>полужирный</b>")
```

Перевод строки в простом тексте осуществляется с помощью символа \n, а в тексте в формате HTML с помощью тега <br>. Пример:

```
label.setText("Текст\nна двух строках")
```

Внутри текста символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент, ссылка на который передана в метод setBuddy(), окажется в фокусе ввода. Чтобы вывести символ &, необходимо его удвоить. Если надпись не связана с другим компонентом, то символ & выводится в составе текста. Пример:

```
label = QtGui.QLabel("&Пароль")
lineEdit = QtGui.QLineEdit()
label.setBuddy(lineEdit)
```

Метод является слотом с сигнатурой setText(const QString&);

- setNum(<Число>) — преобразует целое или вещественное число в строку и отображает ее на надписи. Метод является слотом с сигнатурами setNum(int) и setNum(double);
- setWordWrap(<Флаг>) — если в параметре указано значение True, то текст может переноситься на другую строку. По умолчанию перенос строк не осуществляется;
- setTextFormat(<Режим>) — задает режим отображения текста. Могут быть указаны следующие атрибуты из класса QtCore.Qt:
  - PlainText — 0 — простой текст;
  - RichText — 1 — форматированный текст;
  - AutoText — 2 — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим RichText, в противном случае — режим PlainText;
- text() — возвращает текст надписи;
- setAlignment(<Режим>) — задает режим выравнивания текста внутри надписи. Допустимые значения мы рассматривали в *разд. 22.2*. Пример:

```
label.setAlignment(QtCore.Qt.AlignRight | QtCore.Qt.AlignBottom)
```

- setOpenExternalLinks(<Флаг>) — если в качестве параметра указано значение True, то щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке указанной страницы. Пример:

```
label.setText('<a href="http://google.ru/">Это гиперссылка</a>')
label.setOpenExternalLinks(True)
```

- setBuddy(<Компонент>) — позволяет связать надпись с другим компонентом. В этом случае в тексте надписи можно задавать клавиши быстрого доступа, указав символ & перед буквой или цифрой. После нажатия комбинации клавиш в фокусе ввода окажется компонент, ссылка на который передана в качестве параметра;
- setPixmap(<QPixmap>) — позволяет вывести изображение на надпись. В качестве параметра указывается экземпляр класса QPixmap. Метод является слотом с сигнатурой setPixmap(const QPixmap&).

Пример:

```
label.setPixmap(QtGui.QPixmap("foto.jpg"))
```

- `setPicture(<QPicture>)` — позволяет вывести рисунок. В качестве параметра указывается экземпляр класса `QPicture`. Метод является слотом с сигнатурой `setPicture(const QPicture&);`
- `setMovie(<QMovie>)` — позволяет вывести анимацию. В качестве параметра указывается экземпляр класса `QMovie`. Метод является слотом с сигнатурой `setMovie(QMovie *);`
- `setScaledContents(<Флаг>)` — если в параметре указано значение `True`, то при изменении размеров надписи размер содержимого также будет изменяться. По умолчанию изменение размеров содержимого не осуществляется;
- `setMargin(<Отступ>)` — задает отступ от рамки до содержимого надписи;
- `setIndent(<Отступ>)` — задает отступ от рамки до текста надписи в зависимости от значения выравнивания. Если выравнивание производится по левой стороне, то задает отступ слева, если по правой стороне, то справа и т. д.;
- `clear()` — удаляет содержимое надписи. Метод является слотом;
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом надписи. Можно указать следующие атрибуты (или их комбинацию через оператор `|`) из класса `QtCore.Qt`:
  - `NoTextInteraction` — `0` — пользователь не может взаимодействовать с текстом надписи;
  - `TextSelectableByMouse` — `1` — текст можно выделить и скопировать в буфер обмена;
  - `TextSelectableByKeyboard` — `2` — текст можно выделить с помощью клавиш на клавиатуре. Внутри надписи будет отображен текстовый курсор;
  - `LinksAccessibleByMouse` — `4` — на гиперссылке можно щелкнуть мышью и скопировать ее адрес;
  - `LinksAccessibleByKeyboard` — `8` — с гиперссылкой можно взаимодействовать с помощью клавиатуры. Перемещаться между гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке при нажатии клавиши <Enter>;
  - `TextEditable` — `16` — текст надписи можно редактировать;
  - `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;
  - `TextBrowserInteraction` — комбинация `TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard`;
- `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`;
- `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;
- `selectedText()` — возвращает выделенный текст или пустую строку;
- `hasSelectedText()` — возвращает значение `True`, если существует выделенный фрагмент, и `False` — в противном случае.

Класс QLabel содержит следующие сигналы:

- linkActivated(const QString&) — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес;
- linkHovered(const QString&) — генерируется при наведении указателя мыши на гиперссылку и выведении указателя. Через параметр внутри обработчика доступен URL-адрес или пустая строка.

23.2. Командная кнопка

Командная кнопка является наиболее часто используемым компонентом. При нажатии кнопки внутри обработчика обычно выполняется какая-либо операция. Кнопка реализуется с помощью класса QPushButton. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QPushButton
```

Конструктор класса QPushButton имеет три формата:

```
<Объект> = QPushButton([parent=<Родитель>])
<Объект> = QPushButton(<Текст>[, parent=<Родитель>])
<Объект> = QPushButton(<QIcon>, <Текст>[, parent=<Родитель>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, то компонент будет обладать своим собственным окном. Параметр <Текст> позволяет задать текст, который будет отображен на кнопке, а параметр <QIcon> позволяет добавить перед текстом иконку.

Класс QPushButton наследует следующие методы из класса QAbstractButton (перечислены только основные методы; полный список смотрите в документации):

- setText(<Текст>) — задает текст, который будет отображен на кнопке. Внутри текста символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы кнопка будет нажата. Чтобы вывести символ &, необходимо его удвоить;
- text() — возвращает текст, отображаемый на кнопке;
- setShortcut(<QKeySequence>) — задает комбинацию клавиш быстрого доступа. Примеры указания значения:

```
button.setShortcut("Alt+B")
button.setShortcut(QtGui.QKeySequence.mnemonic("&B"))
button.setShortcut(QtGui.QKeySequence("Alt+B"))
button.setShortcut(
       QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E))
```

- setIcon(<QIcon>) — позволяет вставить иконку перед текстом;
- setIconSize(<QSize>) — задает размеры иконки. В качестве параметра указывается экземпляр класса QSize. Метод является слотом с сигнатурой setIconSize(const QSize&);
- setAutoRepeat(<Флаг>) — если в качестве параметра указано значение True, то сигнал clicked() будет периодически генерироваться, пока кнопка находится в нажатом состоянии. Примером являются кнопки, изменяющие значение полосы прокрутки;

- `animateClick([<Интервал>])` — имитирует нажатие кнопки пользователем. После нажатия кнопка находится в этом состоянии указанный промежуток времени, по истечении которого кнопка отпускается. Значение указывается в миллисекундах. Если параметр не указан, то интервал равен 100 миллисекундам. Метод является слотом;
- `click()` — имитирует нажатие кнопки без анимации. Метод является слотом;
- `toggle()` — переключает кнопку. Метод является слотом;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка является переключателем, который может находиться в двух состояниях — установленном и не установленном;
- `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка-переключатель будет находиться в установленном состоянии. Метод является слотом с сигнатурой `setChecked(bool)`;
- `isChecked()` — возвращает значение `True`, если кнопка находится в установленном состоянии, и `False` — в противном случае;
- `setAutoExclusive(<Флаг>)` — если в качестве параметра указано значение `True`, то внутри группы только одна кнопка-переключатель может быть установлена;
- `setDown(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка будет находиться в нажатом состоянии;
- `isDown()` — возвращает значение `True`, если кнопка находится в нажатом состоянии, и `False` — в противном случае.

Кроме перечисленных состояний кнопка может находиться в неактивном состоянии. Для этого необходимо передать значение `False` в метод `setEnabled()` из класса `QWidget`. Проверить активна кнопка или нет позволяет метод `isEnabled()`. Метод возвращает значение `True`, если кнопка находится в активном состоянии, и `False` — в противном случае.

Класс `QAbstractButton` содержит следующие сигналы:

- `pressed()` — генерируется при нажатии кнопки;
- `released()` — генерируется при отпускании ранее нажатой кнопки;
- `clicked(bool=0)` — генерируется при нажатии, а затем отпускании кнопки мыши над кнопкой. Именно для этого сигнала обычно назначают обработчики;
- `toggled(bool)` — генерируется при переключении кнопки. Через параметр внутри обработчика доступно текущее состояние кнопки.

Класс `QPushButton` содержит дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setFlat(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка будет отображаться без границ;
- `setAutoDefault(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка может быть нажата с помощью клавиши <Enter>, при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>. В диалоговых окнах для всех кнопок по умолчанию указано значение `True`, а для остальных окон — значение `False`;
- `setDefault(<Флаг>)` — задает кнопку по умолчанию. Метод работает только в диалоговых окнах. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, например на текстовое поле;

- `setMenu(<QMenu>)` — устанавливает всплывающее меню, которое будет отображаться при нажатии кнопки. В качестве параметра указывается экземпляр класса `QMenu`;
- `menu()` — возвращает ссылку на всплывающее меню или значение `None`;
- `showMenu()` — отображает всплывающее меню. Метод является слотом.

23.3. Переключатель

Переключатели (иногда их называют радиокнопками) обычно используются в группе. Внутри группы может быть включен только один переключатель. При попытке включить другой переключатель, ранее включенный переключатель автоматически отключается. Для объединения переключателей в группу можно воспользоваться классом `QGroupBox`, который мы уже рассматривали в *разд. 22.7*, а также классом `QButtonGroup`. Переключатель реализуется с помощью класса `QRadioButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QRadioButton
```

Конструктор класса `QRadioButton` имеет два формата:

```
<Объект> = QRadioButton([parent=<Родитель>])
<Объект> = QRadioButton(<Текст>[, parent=<Родитель>])
```

Класс `QRadioButton` наследует все методы из класса `QAbstractButton` (см. *разд. 23.2*). Включить или отключить переключатель позволяет метод `setChecked()`, проверить текущий статус можно с помощью метода `isChecked()`, а чтобы перехватить переключение, следует назначить обработчик сигнала `toggled(bool)`. Через параметр внутри обработчика доступно текущее состояние переключателя.

23.4. Флажок

Флажок предназначен для включения или выключения каких-либо опций настроек программы пользователем и может иметь несколько состояний: установлен, сброшен и частично установлен. Флажок реализуется с помощью класса `QCheckBox`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QCheckBox
```

Конструктор класса `QCheckBox` имеет два формата:

```
<Объект> = QCheckBox([parent=<Родитель>])
<Объект> = QCheckBox(<Текст>[, parent=<Родитель>])
```

Класс `QCheckBox` наследует все методы из класса `QAbstractButton` (см. *разд. 23.2*), а также добавляет несколько новых:

- `setCheckState(<Статус>)` — задает статус флажка. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `checkState()` — возвращает текущий статус флажка;
- `setTristate([<Флаг>=True])` — если в качестве параметра указано значение `True` (значение по умолчанию), то флажок может поддерживать все три статуса. По умолчанию поддерживаются только статусы установлен и сброшен;

- `isTristate()` — возвращает значение `True`, если флажок поддерживает три статуса, и `False` — в противном случае.

Чтобы перехватить смену статуса флажка, следует назначить обработчик сигнала `stateChanged(int)`. Через параметр внутри обработчика доступен текущий статус флажка.

Если используется флажок, поддерживающий только два состояния, то установить или сбросить флажок позволяет метод `setChecked()`, проверить текущий статус можно с помощью метода `isChecked()`, а чтобы перехватить изменение статуса следует назначить обработчик сигнала `toggled(bool)`. Через параметр внутри обработчика доступен текущий статус флажка.

23.5. Однострочное текстовое поле

Однострочное текстовое поле предназначено для ввода и редактирования текста небольшого объема. С его помощью можно также отобразить вводимые символы в виде звездочек (например, чтобы скрыть пароль) или вообще не отображать их (например, чтобы скрыть длину пароля). Поле по умолчанию поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Однострочное текстовое поле реализуется с помощью класса `QLineEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QLineEdit
```

Конструктор класса `QLineEdit` имеет два формата:

```
<Объект> = QLineEdit([parent=<Родитель>])
<Объект> = QLineEdit(<Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст, который будет отображен в однострочном текстовом поле.

23.5.1. Основные методы и сигналы

Класс `QLineEdit` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setEchoMode(<Режим>)` — задает режим отображения текста. Могут быть указаны следующие атрибуты из класса `QLineEdit`:
  - `Normal` — `0` — показывать символы как они были введены;
  - `NoEcho` — `1` — не показывать вводимые символы;
  - `Password` — `2` — вместо символов указывать символ `*`;
  - `PasswordEchoOnEdit` — `3` — показывать символы при вводе, а при потере фокуса отображать символ `*`;
- `setCompleter(<QCompleter>)` — позволяет предлагать возможные варианты значений, начинающиеся с введенных пользователем символов. В качестве параметра указывается экземпляр класса `QCompleter`. Пример:

```
lineEdit = QtGui.QLineEdit()
arr = ["кадр", "каменный", "камень", "камера"]
```

```
completer = QtGui.QCompleter(arr, window)
lineEdit.setCompleter(completer)
```

- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, то поле будет доступно только для чтения;
- `isReadOnly()` — возвращает значение `True`, если поле доступно только для чтения, и `False` — в противном случае;
- `setAlignment(<Выравнивание>)` — задает выравнивание текста внутри поля;
- `setMaxLength(<Количество>)` — задает максимальное количество символов;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, то поле будет отображаться без рамки;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то режим перетаскивания текста из текстового поля с помощью мыши будет включен. По умолчанию однострочное текстовое поле только принимает перетаскиваемый текст;
- `setPlaceholderText(<Текст>)` — задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода;
- `setText(<Текст>)` — вставляет указанный текст в поле. Метод является слотом с сигнатурой `setText(const QString&)`;
- `insert(<Текст>)` — вставляет текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален;
- `text()` — возвращает текст, содержащийся в текстовом поле;
- `displayText()` — возвращает текст, который видит пользователь. Результат зависит от режима отображения, заданного с помощью метода `setEchoMode()`, например, в режиме `Password` строка будет состоять из символов `*`;
- `selectedText()` — возвращает выделенный фрагмент или пустую строку;
- `clear()` — удаляет весь текст из поля. Метод является слотом;
- `backspace()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, то удаляет символ, стоящий слева от текстового курсора;
- `del_()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, то удаляет символ, стоящий справа от текстового курсора;
- `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`. Во втором параметре можно указать отрицательное значение;
- `selectAll()` — выделяет весь текст в поле. Метод является слотом;
- `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;
- `hasSelectedText()` — возвращает значение `True`, если поле содержит выделенный фрагмент, и `False` — в противном случае;
- `deselect()` — снимает выделение;
- `setCursorPosition(<Индекс>)` — задает положение текстового курсора;
- `cursorPosition()` — возвращает текущее положение текстового курсора;
- `cursorForward(<Флаг>[, steps=1])` — перемещает текстовый курсор вперед на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, то фрагмент выделяется;

- `cursorBackward(<Флаг>[, steps=1])` — перемещает текстовый курсор назад на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, то фрагмент выделяется;
- `cursorWordForward(<Флаг>)` — перемещает текстовый курсор вперед на одно слово. Если в параметре указано значение `True`, то фрагмент выделяется;
- `cursorWordBackward(<Флаг>)` — перемещает текстовый курсор назад на одно слово. Если в параметре указано значение `True`, то фрагмент выделяется;
- `home(<Флаг>)` — перемещает текстовый курсор в начало поля. Если в параметре указано значение `True`, то фрагмент выделяется;
- `end(<Флаг>)` — перемещает текстовый курсор в конец поля. Если в параметре указано значение `True`, то фрагмент выделяется;
- `cut()` — копирует выделенный текст в буфер обмена, а затем удаляет его из поля, при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена, при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора, при условии, что поле доступно для редактирования. Метод является слотом;
- `undo()` — отменяет последнюю операцию ввода пользователем, при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `createStandardContextMenu()` — создает стандартное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, наследующий класс `QLineEdit`, и переопределить метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню.

Класс `QLineEdit` содержит следующие сигналы:

- `cursorPositionChanged(int,int)` — генерируется при перемещении текстового курсора. Внутри обработчика через первый параметр доступна старая позиция курсора, а через второй параметр — новая позиция;
- `editingFinished()` — генерируется при нажатии клавиши <Enter> или потере полем фокуса ввода;
- `returnPressed()` — генерируется при нажатии клавиши <Enter>;
- `selectionChanged()` — генерируется при изменении выделения;
- `textChanged(const QString&)` — генерируется при изменении текста внутри поля пользователем или программно. Внутри обработчика через параметр доступно новое значение;

- `textEdited(const QString&)` — генерируется при изменении текста внутри поля пользователем. Сигнал не генерируется при изменении текста с помощью метода `setText()`. Внутри обработчика через параметр доступно новое значение.

23.5.2. Ввод данных по маске

С помощью метода `setInputMask(<Маска>)` можно ограничить ввод символов допустимым диапазоном значений. В качестве значения указывается строка, имеющая следующий формат:

```
"<Последовательность символов>[;<Символ-заполнитель>]"
```

В первом параметре указывается комбинация из следующих специальных символов:

- `9` — обязательна цифра от `0` до `9`;
- `0` — разрешена, но не обязательна цифра от `0` до `9`;
- `D` — обязательна цифра от `1` до `9`;
- `d` — разрешена, но не обязательна цифра от `1` до `9`;
- `B` — обязательна цифра `0` или `1`;
- `b` — разрешена, но не обязательна цифра `0` или `1`;
- `H` — обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `h` — разрешен, но не обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `#` — разрешена, но не обязательна цифра или знак плюс или минус;
- `A` — обязательна буква в любом регистре;
- `a` — разрешена, но не обязательна буква;
- `N` — обязательна буква в любом регистре или цифра от `0` до `9`;
- `n` — разрешена, но не обязательна буква или цифра от `0` до `9`;
- `X` — обязателен любой символ;
- `x` — разрешен, но не обязателен любой символ;
- `>` — все последующие буквы переводятся в верхний регистр;
- `<` — все последующие буквы переводятся в нижний регистр;
- `!` — отключает изменение регистра;
- `\` — используется для отмены действия спецсимволов.

Все остальные символы трактуются как есть. В необязательном параметре `<Символ-заполнитель>` можно указать символ, который будет отображаться в поле, обозначая место ввода. Если параметр не указан, то символом является пробел. Пример:

```
lineEdit.setInputMask("Дата: 99.B9.9999;_")  # Дата: __.__.____
lineEdit.setInputMask("Дата: 99.B9.9999;#")  # Дата: ##.##.####
lineEdit.setInputMask("Дата: 99.B9.9999 г.") # Дата:   .  .     г.
```

Проверить соответствие введенных данных маске позволяет метод `hasAcceptableInput()`. Если данные соответствуют маске, то метод возвращает значение `True`, а в противном случае — `False`.

23.5.3. Контроль ввода

Контролировать ввод данных позволяет метод `setValidator(<QValidator>)`. В качестве значения указывается экземпляр класса, наследующего класс `QValidator`. Существуют следующие стандартные классы, позволяющие контролировать ввод данных:

- `QIntValidator` — допускает ввод только целых чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

```
QIntValidator([parent=None])
QIntValidator(<Начальное значение>, <Конечное значение>,
              <Родитель>)
```

  Пример ограничения ввода диапазоном целых чисел от `0` до `100`:

```
lineEdit.setValidator(QtGui.QIntValidator(0, 100, window))
```

- `QDoubleValidator` — допускает ввод только вещественных чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

```
QDoubleValidator([parent=None])
QDoubleValidator(<Начальное значение>, <Конечное значение>,
                 <Количество цифр после точки>, <Родитель>)
```

  Пример ограничения ввода диапазоном вещественных чисел от `0.0` до `100.0`. и двумя цифрами после десятичной точки:

```
lineEdit.setValidator(
         QtGui.QDoubleValidator(0.0, 100.0, 2, window))
```

  Чтобы позволить вводить числа в экспоненциальной форме, необходимо передать значение атрибута `ScientificNotation` в метод `setNotation()`. Если передать значение атрибута `StandardNotation`, то число должно быть только в десятичной форме. Пример:

```
validator = QtGui.QDoubleValidator(0.0, 100.0, 2, window)
validator.setNotation(QtGui.QDoubleValidator.StandardNotation)
lineEdit.setValidator(validator)
```

- `QRegExpValidator` — позволяет проверить данные на соответствие шаблону регулярного выражения. Форматы конструктора:

```
QRegExpValidator([parent=None])
QRegExpValidator(<QRegExp>, <Родитель>)
```

  Пример ввода только цифр от `0` до `9`:

```
validator = QtGui.QRegExpValidator(
                  QtCore.QRegExp("[0-9]+"), window)
lineEdit.setValidator(validator)
```

  Обратите внимание на то, что производится проверка полного соответствия шаблону, поэтому символы `^` и `$` явным образом указывать не нужно.

Проверить соответствие введенных данных условию позволяет метод `hasAcceptableInput()`. Если данные соответствуют условию, то метод возвращает значение `True`, а в противном случае — `False`.

23.6. Многострочное текстовое поле

Многострочное текстовое поле предназначено для ввода и редактирования как простого текста, так и текста в формате HTML. Поле по умолчанию поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Многострочное текстовое поле реализуется с помощью класса `QTextEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit
```

Конструктор класса `QTextEdit` имеет два формата:

```
<Объект> = QTextEdit([parent=<Родитель>])
<Объект> = QTextEdit(<Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст в формате HTML, который будет отображен в текстовом поле.

> ***Примечание***
>
> Класс `QTextEdit` предназначен для отображения как простого текста, так и текста в формате HTML. Если поддержка HTML не нужна, то следует воспользоваться классом `QPlainTextEdit`, который оптимизирован для работы с простым текстом большого объема.

23.6.1. Основные методы и сигналы

Класс `QTextEdit` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText(<Текст>)` — вставляет указанный текст в поле. Текст может быть простым или в формате HTML. Метод является слотом с сигнатурой `setText(const QString&)`;
- `setPlainText(<Текст>)` — вставляет простой текст. Метод является слотом с сигнатурой `setPlainText(const QString&)`;
- `setHtml(<Текст>)` — вставляет текст в формате HTML. Метод является слотом с сигнатурой `setHtml(const QString&)`;
- `insertPlainText(<Текст>)` — вставляет простой текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Метод является слотом с сигнатурой `insertPlainText(const QString&)`;
- `insertHtml(<Текст>)` — вставляет текст в формате HTML в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Метод является слотом с сигнатурой `insertHtml(const QString&)`;
- `append(<Текст>)` — добавляет новый абзац с указанным текстом в формате HTML в конец поля;
- `setDocumentTitle(<Текст>)` — задает текст заголовка документа (для тега `<title>`);
- `documentTitle()` — возвращает текст заголовка (из тега `<title>`);
- `toPlainText()` — возвращает простой текст, содержащийся в текстовом поле;

- `toHtml()` — возвращает текст в формате HTML;
- `clear()` — удаляет весь текст из поля. Метод является слотом;
- `selectAll()` — выделяет весь текст в поле. Метод является слотом;
- `zoomIn([range=1])` — увеличивает размер шрифта. Метод является слотом;
- `zoomOut([range=1])` — уменьшает размер шрифта. Метод является слотом;
- `cut()` — копирует выделенный текст в буфер обмена, а затем удаляет его из поля, при условии, что есть выделенный фрагмент. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена, при условии, что есть выделенный фрагмент. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора, при условии, что поле доступно для редактирования. Метод является слотом;
- `canPaste()` — возвращает `True`, если из буфера обмена можно вставить текст, и `False` — в противном случае;
- `setAcceptRichText(<Флаг>)` — если в качестве параметра указано значение `True`, то в поле можно будет вставить текст в формате HTML из буфера обмена или при помощи перетаскивания. Значение `False` отключает эту возможность;
- `acceptRichText()` — возвращает значение `True`, если в поле можно вставить текст в формате HTML, и `False` — в противном случае;
- `undo()` — отменяет последнюю операцию ввода пользователем, при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве значения указано значение `True`, то операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `createStandardContextMenu([<QPoint>])` — создает стандартное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, наследующий класс `QTextEdit`, и переопределить метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню;
- `ensureCursorVisible()` — прокручивает область таким образом, чтобы текстовый курсор оказался в зоне видимости;
- `find(<Текст>[, <Режим>])` — производит поиск фрагмента (по умолчанию в прямом направлении без учета регистра символов) в текстовом поле. Если фрагмент найден, то он выделяется и метод возвращает значение `True`, в противном случае — значение `False`. В необязательном параметре `<Режим>` можно указать комбинацию (через оператор `|`) следующих атрибутов из класса `QTextDocument`:
  - `FindBackward` — `1` — поиск в обратном направлении, а не в прямом;
  - `FindCaseSensitively` — `2` — поиск с учетом регистра символов;
  - `FindWholeWords` — `4` — поиск слов целиком, а не фрагментов;

- `print_(<QPrinter>)` — отправляет содержимое текстового поля на печать. В качестве параметра указывается экземпляр класса `QPrinter`. Пример печати в файл в формате PDF:

```
printer = QtGui.QPrinter()
printer.setOutputFormat(QtGui.QPrinter.PdfFormat)
printer.setOutputFileName("mypdf.pdf")
textEdit.print_(printer)
```

Класс `QTextEdit` содержит следующие сигналы:

- `currentCharFormatChanged(const QTextCharFormat&)` — генерируется при изменении формата. Внутри обработчика через параметр доступен новый формат;
- `cursorPositionChanged()` — генерируется при изменении положения текстового курсора;
- `selectionChanged()` — генерируется при изменении выделения;
- `textChanged()` — генерируется при изменении текста внутри поля;
- `copyAvailable(bool)` — генерируется при изменении возможности скопировать фрагмент. Внутри обработчика через параметр доступно значение `True`, если фрагмент можно скопировать, и `False` — в противном случае;
- `undoAvailable(bool)` — генерируется при изменении возможности отменить операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно отменить операцию ввода, и `False` — в противном случае;
- `redoAvailable(bool)` — генерируется при изменении возможности повторить отмененную операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно повторить отмененную операцию ввода, и `False` — в противном случае.

23.6.2. Изменение настроек поля

Для изменения настроек предназначены следующие методы из класса `QTextEdit` (перечислены только основные методы; полный список смотрите в документации):

- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом. Можно указать следующие атрибуты (или их комбинацию через оператор `|`) из класса `QtCore.Qt`:
  - `NoTextInteraction` — `0` — пользователь не может взаимодействовать с текстом;
  - `TextSelectableByMouse` — `1` — текст можно выделить и скопировать в буфер обмена;
  - `TextSelectableByKeyboard` — `2` — текст можно выделить с помощью клавиш на клавиатуре. Внутри поля будет отображен текстовый курсор;
  - `LinksAccessibleByMouse` — `4` — на гиперссылке можно щелкнуть мышью и скопировать ее адрес;
  - `LinksAccessibleByKeyboard` — `8` — с гиперссылкой можно взаимодействовать с помощью клавиатуры. Перемещаться между гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке при нажатии клавиши <Enter>;
  - `TextEditable` — `16` — текст можно редактировать;
  - `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;

- TextBrowserInteraction — комбинация TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard;

- setReadOnly(<Флаг>) — если в качестве параметра указано значение True, то поле будет доступно только для чтения;
- isReadOnly() — возвращает значение True, если поле доступно только для чтения, и False — в противном случае;
- setLineWrapMode(<Режим>) — задает режим переноса строк. В качестве значения могут быть указаны следующие атрибуты из класса QTextEdit:
  - NoWrap — 0 — перенос строк не производится;
  - WidgetWidth — 1 — перенос строки при достижении ширины поля;
  - FixedPixelWidth — 2 — перенос строки при достижении фиксированной ширины в пикселах, которую можно задать с помощью метода setLineWrapColumnOrWidth();
  - FixedColumnWidth — 3 — перенос строки при достижении фиксированной ширины в буквах, которую можно задать с помощью метода setLineWrapColumnOrWidth();
- setLineWrapColumnOrWidth(<Значение>) — задает ширину колонки;
- setWordWrapMode(<Режим>) — задает режим переноса по словам. В качестве значения могут быть указаны следующие атрибуты из класса QTextOption:
  - NoWrap — 0 — перенос по словам не производится;
  - WordWrap — 1 — перенос строк только по словам;
  - ManualWrap — 2 — аналогичен режиму NoWrap;
  - WrapAnywhere — 3 — перенос строки может быть внутри слова;
  - WrapAtWordBoundaryOrAnywhere — 4 — по возможности перенос по словам. Если это невозможно, то перенос строки может быть внутри слова;
- setOverwriteMode(<Флаг>) — если в качестве параметра указано значение True, то вводимый текст будет замещать ранее введенный. Значение False отключает замещение;
- overwriteMode() — возвращает значение True, если вводимый текст замещает ранее введенный, и False — в противном случае;
- setAutoFormatting(<Режим>) — задает режим автоматического форматирования. В качестве значения могут быть указаны следующие атрибуты из класса QTextEdit:
  - AutoNone — автоматическое форматирование не используется;
  - AutoBulletList — автоматически создавать маркированный список при вводе пользователем в начале строки символа *;
  - AutoAll — включить все режимы. В Qt версии 4.7 эквивалентно режиму AutoBulletList;
- setCursorWidth(<Ширина>) — задает ширину текстового курсора;
- setTabChangesFocus(<Флаг>) — если в качестве параметра указано значение False, то с помощью нажатия клавиши <Tab> можно вставить символ табуляции в поле. Если указано значение True, то клавиша <Tab> используется для передачи фокуса между компонентами;
- setTabStopWidth(<Ширина>) — задает ширину символа табуляции в пикселах;
- tabStopWidth() — возвращает ширину символа табуляции в пикселах.

23.6.3. Изменение характеристик текста и фона

Для изменения характеристик текста и фона предназначены следующие методы из класса `QTextEdit` (перечислены только основные методы; полный список смотрите в документации):

- `setCurrentFont(<QFont>)` — задает текущий шрифт. Метод является слотом с сигнатурой `setCurrentFont(const QFont&)`. В качестве параметра указывается экземпляр класса `QFont`. Конструктор класса `QFont` имеет следующий формат:

  ```
  <Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
                  [, italic=False])
  ```

  В первом параметре указывается название шрифта в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light`, `Normal`, `DemiBold`, `Bold` или `Black` из класса `QFont`. Если в параметре `italic` указано значение `True`, то шрифт будет курсивным;

- `currentFont()` — возвращает экземпляр класса `QFont` с текущими характеристиками шрифта;
- `setFontFamily(<Название шрифта>)` — задает название текущего шрифта. Метод является слотом с сигнатурой `setFontFamily(const QString&)`;
- `fontFamily()` — возвращает название текущего шрифта;
- `setFontPointSize(<Размер>)` — задает размер текущего шрифта. Метод является слотом с сигнатурой `setFontPointSize(qreal)`;
- `fontPointSize()` — возвращает размер текущего шрифта;
- `setFontWeight(<Жирность>)` — задает жирность текущего шрифта. Метод является слотом с сигнатурой `setFontWeight(int)`;
- `fontWeight()` — возвращает жирность текущего шрифта;
- `setFontItalic(<Флаг>)` — если в качестве параметра указано значение `True`, то шрифт будет курсивным. Метод является слотом с сигнатурой `setFontItalic(bool)`;
- `fontItalic()` — возвращает `True`, если шрифт курсивный, и `False` — в противном случае;
- `setFontUnderline(<Флаг>)` — если в качестве параметра указано значение `True`, то текст будет подчеркнутым. Метод является слотом с сигнатурой `setFontUnderline(bool)`;
- `fontUnderline()` — возвращает `True`, если текст подчеркнутый, и `False` — в противном случае;
- `setTextColor(<QColor>)` — задает цвет текущего текста. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Метод является слотом с сигнатурой `setTextColor(const QColor&)`;
- `textColor()` — возвращает экземпляр класса `QColor` с цветом текущего текста;
- `setTextBackgroundColor(<QColor>)` — задает цвет фона. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Метод является слотом с сигнатурой `setTextBackgroundColor(const QColor&)`;

- `textBackgroundColor()` — возвращает экземпляр класса `QColor` с цветом фона;
- `setAlignment(<Выравнивание>)` — задает горизонтальное выравнивание текста внутри абзаца. Допустимые значения мы рассматривали в *разд. 22.2*. Метод является слотом с сигнатурой `setAlignment(Qt::Alignment)`;
- `alignment()` — возвращает значение выравнивания текста внутри абзаца.

Задать формат символов можно также с помощью класса `QTextCharFormat`, который содержит дополнительные настройки. После создания экземпляра класса его следует передать в метод `setCurrentCharFormat(<QTextCharFormat>)`. Получить экземпляр класса с текущими настройками позволяет метод `currentCharFormat()`. За подробной информацией по классу `QTextCharFormat` обращайтесь к документации.

23.6.4. Класс *QTextDocument*

Класс `QTextDocument` реализует документ, который отображается в многострочном текстовом поле. Получить ссылку на текущий документ позволяет метод `document()` из класса `QTextEdit`. Установить новый документ можно с помощью метода `setDocument(<QTextDocument>)`. Иерархия наследования:

```
QObject — QTextDocument
```

Конструктор класса `QTextDocument` имеет два формата:

```
<Объект> = QTextDocument([parent=<Родитель>])
<Объект> = QTextDocument(<Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Параметр `<Текст>` позволяет задать текст в простом формате (не в HTML-формате), который будет отображен в текстовом поле.

Класс `QTextDocument` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPlainText(<Текст>)` — вставляет простой текст;
- `setHtml(<Текст>)` — вставляет текст в формате HTML;
- `toPlainText()` — возвращает простой текст, содержащийся в документе;
- `toHtml([<QByteArray>])` — возвращает текст в формате HTML. В качестве параметра можно указать кодировку документа, которая будет выведена в теге `<meta>`;
- `clear()` — удаляет весь текст из документа;
- `isEmpty()` — возвращает значение `True`, если документ пустой, и `False` — в противном случае;
- `isModified()` — возвращает значение `True`, если документ был изменен, и `False` — в противном случае;
- `undo()` — отменяет последнюю операцию ввода пользователем, при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;

- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `availableUndoSteps()` — возвращает количество возможных операций отмены;
- `availableRedoSteps()` — возвращает количество возможных повторов отмененных операций;
- `clearUndoRedoStacks([stacks=UndoAndRedoStacks])` — очищает список возможных отмен и/или повторов. В качестве параметра можно указать следующие атрибуты из класса `QTextDocument`:
  - `UndoStack` — только список возможных отмен;
  - `RedoStack` — только список возможных повторов;
  - `UndoAndRedoStacks` — очищаются оба списка;
- `print_(<QPrinter>)` — отправляет содержимое документа на печать. В качестве параметра указывается экземпляр класса `QPrinter`;
- `find()` — производит поиск фрагмента в документе. Метод возвращает экземпляр класса `QTextCursor`. Если фрагмент не найден, то объект курсора будет нулевым. Проверить успешность операции можно с помощью метода `isNull()` объекта курсора. Форматы метода:

```
find(<Текст>[, position=0][, options=0])
find(<QRegExp>[, position=0][, options=0])
find(<Текст>, <QTextCursor>[, options=0])
find(<QRegExp>, <QTextCursor>[, options=0])
```

  Параметр `<Текст>` задает искомый фрагмент, а параметр `<QRegExp>` позволяет указать регулярное выражение. По умолчанию обычный поиск производится без учета регистра символов в прямом направлении, начиная с позиции `position` или от текстового курсора, указанного в параметре `<QTextCursor>`. Поиск по регулярному выражению по умолчанию производится с учетом регистра символов. Чтобы поиск производился без учета регистра, необходимо передать атрибут `QtCore.Qt.CaseInsensitive` в метод `setCaseSensitivity()`. В необязательном параметре `options` можно указать комбинацию (через оператор `|`) следующих атрибутов из класса `QTextDocument`:
  - `FindBackward` — `1` — поиск в обратном направлении, а не в прямом;
  - `FindCaseSensitively` — `2` — поиск с учетом регистра символов. При использовании регулярного выражения значение игнорируется;
  - `FindWholeWords` — `4` — поиск слов целиком, а не фрагментов;
- `setDefaultFont(<QFont>)` — задает шрифт по умолчанию для документа. В качестве параметра указывается экземпляр класса `QFont`. Конструктор класса `QFont` имеет следующий формат:

```
<Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
                [, italic=False])
```

В первом параметре указывается название шрифта в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light`, `Normal`, `DemiBold`, `Bold` или `Black` из класса `QFont`. Если в параметре `italic` указано значение `True`, то шрифт будет курсивным;

- `setDefaultStyleSheet(<CSS>)` — устанавливает таблицу стилей CSS по умолчанию для документа;
- `setDocumentMargin(<Отступ>)` — задает отступ от краев поля до текста;
- `documentMargin()` — возвращает величину отступа от краев поля до текста;
- `setMaximumBlockCount(<Количество>)` — задает максимальное количество текстовых блоков в документе. Если количество блоков становится больше указанного значения, то первый блок будет удален;
- `maximumBlockCount()` — возвращает максимальное количество текстовых блоков;
- `characterCount()` — возвращает количество символов в документе;
- `lineCount()` — возвращает количество абзацев в документе;
- `blockCount()` — возвращает количество текстовых блоков в документе;
- `firstBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит первый текстовый блок документа;
- `lastBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит последний текстовый блок документа;
- `findBlock(<Индекс символа>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа, включающий символ с указанным индексом;
- `findBlockByNumber(<Индекс блока>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа с указанным индексом.

Класс `QTextDocument` содержит следующие сигналы:

- `undoAvailable(bool)` — генерируется при изменении возможности отменить операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно отменить операцию ввода, и `False` — в противном случае;
- `redoAvailable(bool)` — генерируется при изменении возможности повторить отмененную операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно повторить отмененную операцию ввода, и `False` — в противном случае.
- `undoCommandAdded()` — генерируется при добавлении операции ввода в список возможных отмен;
- `blockCountChanged(int)` — генерируется при изменении количества текстовых блоков. Внутри обработчика через параметр доступно новое количество текстовых блоков;
- `cursorPositionChanged(const QTextCursor&)` — генерируется при изменении позиции текстового курсора из-за операции редактирования. Обратите внимание на то, что при простом перемещении текстового курсора сигнал не генерируется;
- `contentsChange(int,int,int)` — генерируется при изменении текста. Внутри обработчика через первый параметр доступен индекс позиции внутри документа, через второй параметр — количество удаленных символов, а через третий параметр — количество добавленных символов;

- contentsChanged() — генерируется при любом изменении документа;
- modificationChanged(bool) — генерируется при изменении статуса документа.

23.6.5. Класс *QTextCursor*

Класс QTextCursor реализует текстовый курсор, выделение и позволяет изменять документ. Конструктор класса QTextCursor имеет следующие форматы:

```
<Объект> = QTextCursor()
<Объект> = QTextCursor(<QTextDocument>)
<Объект> = QTextCursor(<QTextFrame>)
<Объект> = QTextCursor(<QTextBlock>)
<Объект> = QTextCursor(<QTextCursor>)
```

Создать текстовый курсор, установить его в документе и управлять им позволяют следующие методы из класса QTextEdit:

- textCursor() — возвращает видимый в данный момент текстовый курсор (экземпляр класса QTextCursor). Чтобы изменения затронули текущий документ, необходимо передать этот объект в метод setTextCursor();
- setTextCursor(<QTextCursor>) — устанавливает текстовый курсор, ссылка на который указана в качестве параметра;
- cursorForPosition(<QPoint>) — возвращает текстовый курсор, который соответствует позиции, указанной в качестве параметра. Позиция задается с помощью экземпляра класса QPoint в координатах области;
- moveCursor(<Позиция>[, mode=MoveAnchor]) — перемещает текстовый курсор внутри документа. В первом параметре можно указать следующие атрибуты из класса QTextCursor:
  - NoMove — 0 — не перемещать курсор;
  - Start — 1 — в начало документа;
  - Up — 2 — на одну строку вверх;
  - StartOfLine — 3 — в начало текущей строки;
  - StartOfBlock — 4 — в начало текущего текстового блока;
  - StartOfWord — 5 — в начало текущего слова;
  - PreviousBlock — 6 — в начало предыдущего текстового блока;
  - PreviousCharacter — 7 — сдвинуть на один символ влево;
  - PreviousWord — 8 — в начало предыдущего слова;
  - Left — 9 — сдвинуть на один символ влево;
  - WordLeft — 10 — влево на одно слово;
  - End — 11 — в конец документа;
  - Down — 12 — на одну строку вниз;
  - EndOfLine — 13 — в конец текущей строки;
  - EndOfWord — 14 — в конец текущего слова;
  - EndOfBlock — 15 — в конец текущего текстового блока;

- NextBlock — 16 — в начало следующего текстового блока;
- NextCharacter — 17 — сдвинуть на один символ вправо;
- NextWord — 18 — в начало следующего слова;
- Right — 19 — сдвинуть на один символ вправо;
- WordRight — 20 — в начало следующего слова.

Помимо перечисленных атрибутов существуют также атрибуты NextCell, PreviousCell, NextRow и PreviousRow, позволяющие перемещать текстовый курсор внутри таблицы. В необязательном параметре mode можно указать следующие атрибуты из класса QTextCursor:

- MoveAnchor — 0 — если существует выделенный фрагмент, то выделение будет снято и текстовый курсор переместится в новое место (значение по умолчанию);
- KeepAnchor — 1 — фрагмент текста от старой позиции курсора до новой будет выделен.

Класс QTextCursor содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- isNull() — возвращает значение True, если объект курсора является нулевым (создан с помощью конструктора без параметра), и False — в противном случае;
- setPosition(<Позиция>[, mode=MoveAnchor]) — перемещает текстовый курсор внутри документа. В первом параметре указывается позиция внутри документа. Необязательный параметр mode аналогичен одноименному параметру в методе moveCursor() из класса QTextEdit;
- movePosition(<Позиция>[, mode=MoveAnchor][, n=1]) — перемещает текстовый курсор внутри документа. Параметры <Позиция> и mode аналогичны одноименным параметрам в методе moveCursor() из класса QTextEdit. Необязательный параметр n позволяет указать количество перемещений, например, переместить курсор на 10 символов вперед можно так:

```
cur = textEdit.textCursor()
cur.movePosition(QtGui.QTextCursor.NextCharacter,
                 mode=QtGui.QTextCursor.MoveAnchor, n=10)
textEdit.setTextCursor(cur)
```

  Метод movePosition() возвращает значение True, если операция успешно выполнена указанное количество раз. Если было выполнено меньшее количество перемещений (например, из-за достижения конца документа), то метод возвращает значение False;
- position() — возвращает позицию текстового курсора внутри документа;
- positionInBlock() — возвращает позицию текстового курсора внутри блока;
- block() — возвращает экземпляр класса QTextBlock, который описывает текстовый блок, содержащий курсор;
- blockNumber() — возвращает индекс текстового блока, содержащего курсор;
- atStart() — возвращает значение True, если текстовый курсор находится в начале документа, и False — в противном случае;
- atEnd() — возвращает значение True, если текстовый курсор находится в конце документа, и False — в противном случае;

- `atBlockStart()` — возвращает значение `True`, если текстовый курсор находится в начале блока, и `False` — в противном случае;
- `atBlockEnd()` — возвращает значение `True`, если текстовый курсор находится в конце блока, и `False` — в противном случае;
- `select(<Режим>)` — выделяет фрагмент в документе в соответствии с указанным режимом. В качестве параметра можно указать следующие атрибуты из класса `QTextCursor`:
  - `WordUnderCursor` — `0` — выделяет слово, в котором расположен курсор;
  - `LineUnderCursor` — `1` — выделяет текущую строку;
  - `BlockUnderCursor` — `2` — выделяет текущий текстовый блок;
  - `Document` — `3` — выделяет весь документ;
- `hasSelection()` — возвращает значение `True`, если существует выделенный фрагмент, и `False` — в противном случае;
- `hasComplexSelection()` — возвращает значение `True`, если выделенный фрагмент содержит сложное форматирование, а не просто текст, и `False` — в противном случае;
- `clearSelection()` — снимает выделение;
- `selectionStart()` — возвращает начальную позицию выделенного фрагмента;
- `selectionEnd()` — возвращает конечную позицию выделенного фрагмента;
- `selectedText()` — возвращает текст выделенного фрагмента. Обратите внимание, если выделенный фрагмент занимает несколько строк, то вместо символа перевода строки вставляется символ с кодом `\u2029`. Попытка вывести этот символ в окно консоли приведет к исключению, поэтому следует произвести замену символа с помощью метода `replace()`:

  ```
  print(cur.selectedText().replace("\u2029", "\n"))
  ```

- `selection()` — возвращает экземпляр класса `QTextDocumentFragment`, который описывает выделенный фрагмент. Получить текст позволяют методы `toPlainText()` (возвращает простой текст) и `toHtml()` (возвращает текст в формате HTML) из этого класса;
- `removeSelectedText()` — удаляет выделенный фрагмент;
- `deleteChar()` — если нет выделенного фрагмента, то удаляет символ справа от курсора, в противном случае удаляет выделенный фрагмент;
- `deletePreviousChar()` — если нет выделенного фрагмента, то удаляет символ слева от курсора, в противном случае удаляет выделенный фрагмент;
- `beginEditBlock()` и `endEditBlock()` — задают начало и конец блока инструкций. Эти инструкции могут быть отменены или повторены как единое целое с помощью методов `undo()` и `redo()`;
- `joinPreviousEditBlock()` — делает последующие инструкции частью предыдущего блока инструкций;
- `setKeepPositionOnInsert(<Флаг>)` — если в качестве параметра указано значение `True`, то после операции вставки курсор сохранит свою предыдущую позицию. По умолчанию позиция курсора при вставке изменяется;
- `insertText(<Текст>[, <QTextCharFormat>])` — вставляет простой текст;
- `insertHtml(<Текст>)` — вставляет текст в формате HTML.

С помощью методов insertBlock(), insertFragment(), insertFrame(), insertImage(), insertList() и insertTable() можно вставить различные элементы, например изображения, списки и др. Изменить формат выделенного фрагмента позволяют методы mergeBlockCharFormat(), mergeBlockFormat() и mergeCharFormat(). За подробной информацией по этим методам обращайтесь к документации.

23.7. Текстовый браузер

Класс QTextBrowser расширяет возможности класса QTextEdit и реализует текстовый браузер с возможностью перехода по гиперссылкам при щелчке мышью. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit — QTextBrowser
```

Формат конструктора класса QTextBrowser:

```
<Объект> = QTextBrowser([parent=<Родитель>])
```

Класс QTextBrowser содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setSource(<QUrl>) — загружает ресурс. В качестве параметра указывается экземпляр класса QUrl. Метод является слотом с сигнатурой setSource(const QUrl&);
- source() — возвращает экземпляр класса QUrl с адресом текущего ресурса;
- reload() — перезагружает текущий ресурс. Метод является слотом;
- home() — загружает первый ресурс из списка истории. Метод является слотом;
- backward() — загружает предыдущий ресурс из списка истории. Метод является слотом;
- forward() — загружает следующий ресурс из списка истории. Метод является слотом;
- backwardHistoryCount() — возвращает количество предыдущих ресурсов;
- forwardHistoryCount() — возвращает количество следующих ресурсов;
- isBackwardAvailable() — возвращает значение True, если существует предыдущий ресурс в списке истории, и False — в противном случае;
- isForwardAvailable() — возвращает значение True, если существует следующий ресурс в списке истории, и False — в противном случае;
- clearHistory() — очищает список истории;
- historyTitle(<Число>) — если в качестве параметра указано отрицательное число, то возвращает заголовок предыдущего ресурса, если 0 — то заголовок текущего ресурса, а если положительное число — то заголовок следующего ресурса;
- historyUrl(<Число>) — если в качестве параметра указано отрицательное число, то возвращает URL-адрес (экземпляр класса QUrl) предыдущего ресурса, если 0 — то URL-адрес текущего ресурса, а если положительное число — то URL-адрес следующего ресурса;
- setOpenLinks(<Флаг>) — если в качестве параметра указано значение True, то автоматический переход по гиперссылкам разрешен (значение по умолчанию). Значение False запрещает переход.

Класс QTextBrowser содержит следующие сигналы:

- anchorClicked(const QUrl&) — генерируется при переходе по гиперссылке. Внутри обработчика через параметр доступен URL-адрес гиперссылки;
- backwardAvailable(bool) — генерируется при изменении статуса списка предыдущих ресурсов. Внутри обработчика через параметр доступен статус;
- forwardAvailable(bool) — генерируется при изменении статуса списка следующих ресурсов. Внутри обработчика через параметр доступен статус;
- highlighted(const QUrl&) — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен URL-адрес ссылки (экземпляр класса QUrl) или пустой объект;
- highlighted(const QString&) — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен URL-адрес ссылки в виде строки или пустая строка;
- historyChanged() — генерируется при изменении списка истории;
- sourceChanged(const QUrl&) — генерируется при загрузке нового ресурса. Внутри обработчика через параметр доступен URL-адрес ресурса.

> ***Примечание***
>
> Если возможностей класса QTextBrowser вам покажется мало, то обратите внимание на модуль QtWebKit, который содержит множество классов в совокупности реализующих полноценный браузер с поддержкой HTML, XHTML, CSS и JavaScript.

23.8. Поля для ввода целых и вещественных чисел

Для ввода чисел предназначены классы QSpinBox (поле для ввода целых чисел) и QDoubleSpinBox (поле для ввода вещественных чисел). Поля могут содержать две кнопки, которые позволяют увеличивать и уменьшать значение внутри поля с помощью щелчка мышью. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QSpinBox
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDoubleSpinBox
```

Форматы конструкторов классов QSpinBox и QDoubleSpinBox:

```
<Объект> = QSpinBox([parent=<Родитель>])
<Объект> = QDoubleSpinBox([parent=<Родитель>])
```

Классы QSpinBox и QDoubleSpinBox наследуют следующие методы из класса QAbstractSpinBox (перечислены только основные методы; полный список смотрите в документации):

- setButtonSymbols(<Режим>) — задает режим отображения кнопок, предназначенных для изменения значения поля с помощью мыши. Можно указать следующие атрибуты из класса QAbstractSpinBox:
  - UpDownArrows — 0 — отображаются кнопки со стрелками;
  - PlusMinus — 1 — отображаются кнопки с символами + и -. Обратите внимание на то, что при использовании некоторых стилей данное значение может быть проигнорировано;
  - NoButtons — 2 — кнопки не отображаются;

- `setAlignment(<Режим>)` — задает режим выравнивания значения внутри поля;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то значение внутри поля будет изменяться по кругу при нажатии кнопок, например, максимальное значение сменится минимальным;
- `setSpecialValueText(<Строка>)` — позволяет задать строку, которая будет отображаться внутри поля вместо минимального значения;
- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, то поле будет доступно только для чтения;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, то поле будет отображаться без рамки;
- `stepDown()` — уменьшает значение на одно приращение. Метод является слотом;
- `stepUp()` — увеличивает значение на одно приращение. Метод является слотом;
- `stepBy(<Количество>)` — увеличивает (при положительном значении) или уменьшает (при отрицательном значении) значение поля на указанное количество приращений;
- `text()` — возвращает текст, содержащийся внутри поля;
- `clear()` — очищает поле. Метод является слотом;
- `selectAll()` — выделяет все содержимое поля. Метод является слотом.

Класс `QAbstractSpinBox` содержит сигнал `editingFinished()`, который генерируется при потере полем фокуса ввода или нажатии клавиши <Enter>.

Классы `QSpinBox` и `QDoubleSpinBox` содержат следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setValue(<Число>)` — задает значение поля. Метод является слотом с сигнатурами `setValue(int)` и `setValue(double)`;
- `value()` — возвращает целое или вещественное число, содержащееся в поле;
- `cleanText()` — возвращает целое или вещественное число в виде строки;
- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное допустимые значения;
- `setPrefix(<Текст>)` — задает текст, который будет отображаться внутри поля перед значением;
- `setSuffix(<Текст>)` — задает текст, который будет отображаться внутри поля после значения;
- `setSingleStep(<Число>)` — задает число, которое будет прибавляться или вычитаться из текущего значения поля на каждом шаге.

Класс `QDoubleSpinBox` содержит также метод `setDecimals(<Количество>)`, который задает количество цифр после десятичной точки.

Классы `QSpinBox` и `QDoubleSpinBox` содержат сигналы `valueChanged(int)` (только в классе `QSpinBox`), `valueChanged(double)` (только в классе `QDoubleSpinBox`) и `valueChanged(const QString&)`, которые генерируются при изменении значения внутри поля. Внутри обработчика через параметр доступно новое значение в виде числа или строки в зависимости от типа параметра.

23.9. Поля для ввода даты и времени

Для ввода даты и времени предназначены классы `QDateTimeEdit` (поле для ввода даты и времени), `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени). Поля могут содержать две кнопки, которые позволяют увеличивать и уменьшать значение внутри поля с помощью щелчка мышью. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit —
                          QDateEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit —
                          QTimeEdit
```

Форматы конструкторов классов:

```
<Объект> = QDateTimeEdit([parent=<Родитель>])
<Объект> = QDateTimeEdit(<QDateTime>[, parent=<Родитель>])
<Объект> = QDateTimeEdit(<QDate>[, parent=<Родитель>])
<Объект> = QDateTimeEdit(<QTime>[, parent=<Родитель>])
<Объект> = QDateEdit([parent=<Родитель>])
<Объект> = QDateEdit(<QDate>[, parent=<Родитель>])
<Объект> = QTimeEdit([parent=<Родитель>])
<Объект> = QTimeEdit(<QTime>[, parent=<Родитель>])
```

В параметре `<QDateTime>` можно указать экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python. Преобразовать экземпляр класса `QDateTime` в экземпляр класса `datetime` позволяет метод `toPyDateTime()`.

В качестве параметра `<QDate>` можно указать экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Преобразовать экземпляр класса `QDate` в экземпляр класса `date` позволяет метод `toPyDate()`.

В параметре `<QTime>` можно указать экземпляр класса `QTime` или экземпляр класса `time` из языка Python. Преобразовать экземпляр класса `QTime` в экземпляр класса `time` позволяет метод `toPyTime()`.

Класс `QDateTimeEdit` наследует все методы из класса `QAbstractSpinBox` (см. *разд. 23.8*) и дополнительно реализует следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setDateTime(<QDateTime>)` — устанавливает дату и время. В качестве параметра указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python. Метод является слотом с сигнатурой `setDateTime(const QDateTime&)`;
- `setDate(<QDate>)` — устанавливает дату. В качестве параметра указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Метод является слотом с сигнатурой `setDate(const QDate&)`;
- `setTime(<QTime>)` — устанавливает время. В качестве параметра указывается экземпляр класса `QTime` или экземпляр класса `time` из языка Python. Метод является слотом с сигнатурой `setTime(const QTime&)`;
- `dateTime()` — возвращает экземпляр класса `QDateTime` с датой и временем;
- `date()` — возвращает экземпляр класса `QDate` с датой;

- `time()` — возвращает экземпляр класса `QTime` со временем;
- `setDateTimeRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное допустимые значения для даты и времени. В параметрах указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python;
- `setMinimumDateTime(<Минимум>)` и `setMaximumDateTime(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты и времени. В параметрах указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python;
- `setDateRange(<Минимум>, <Максимум>)`, `setMinimumDate(<Минимум>)` и `setMaximumDate(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты. В параметрах указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python;
- `setTimeRange(<Минимум>, <Максимум>)`, `setMinimumTime(<Минимум>)` и `setMaximumTime(<Максимум>)` — задают минимальное и максимальное допустимые значения для времени. В параметрах указывается экземпляр класса `QTime` или экземпляр класса `time` из языка Python;
- `setDisplayFormat(<Формат>)` — задает формат отображения даты и времени. В качестве параметра указывается строка, содержащая специальные символы. Пример указания строки формата:

  ```
  dateTimeEdit.setDisplayFormat("dd.MM.yyyy HH:mm:ss")
  ```

- `setTimeSpec(<Зона>)` — задает зону времени. В качестве параметра можно указать атрибуты `LocalTime`, `UTC` или `OffsetFromUTC` из класса `QtCore.Qt`;
- `setCalendarPopup(<Флаг>)` — если в качестве параметра указано значение `True`, то дату можно будет выбрать с помощью календаря;
- `setSelectedSection(<Секция>)` — выделяет указанную секцию. В качестве параметра можно указать атрибуты `NoSection`, `DaySection`, `MonthSection`, `YearSection`, `HourSection`, `MinuteSection`, `SecondSection`, `MSecSection` или `AmPmSection` из класса `QDateTimeEdit`;
- `setCurrentSection(<Секция>)` — делает указанную секцию текущей;
- `setCurrentSectionIndex(<Индекс>)` — делает секцию с указанным индексом текущей;
- `currentSection()` — возвращает тип текущей секции;
- `currentSectionIndex()` — возвращает индекс текущей секции;
- `sectionCount()` — возвращает количество секций внутри поля;
- `sectionAt(<Индекс>)` — возвращает тип секции по указанному индексу;
- `sectionText(<Секция>)` — возвращает текст указанной секции.

При изменении внутри поля значений даты или времени генерируются сигналы `timeChanged(const QTime&)`, `dateChanged(const QDate&)` и `dateTimeChanged(const QDateTime&)`. Внутри обработчиков через параметр доступно новое значение.

Классы `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени) созданы для удобства и отличаются от класса `QDateTimeEdit` только форматом отображаемых данных. Эти классы наследуют методы базовых классов и не добавляют больше никаких своих методов.

23.10. Календарь

Класс `QCalendarWidget` реализует календарь с возможностью выбора даты и перемещения по месяцам с помощью мыши и клавиатуры. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QCalendarWidget
```

Формат конструктора класса `QCalendarWidget`:

```
<Объект> = QCalendarWidget([parent=<Родитель>])
```

Класс `QCalendarWidget` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setSelectedDate(<QDate>)` — устанавливает указанную дату. В качестве параметра указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Метод является слотом с сигнатурой `setSelectedDate(const QDate&)`;
- `selectedDate()` — возвращает экземпляр класса `QDate` с выбранной датой;
- `setDateRange(<Минимум>, <Максимум>)`, `setMinimumDate(<Минимум>)` и `setMaximumDate(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты. В параметрах указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python;
- `setCurrentPage(<Год>, <Месяц>)` — делает текущей страницу с указанным годом и месяцем. Выбранная дата при этом не изменяется. Метод является слотом с сигнатурой `setCurrentPage(int,int)`;
- `monthShown()` — возвращает месяц (число от `1` до `12`), отображаемый на текущей странице;
- `yearShown()` — возвращает год, отображаемый на текущей странице;
- `showSelectedDate()` — отображает страницу с выбранной датой. Выбранная дата при этом не изменяется. Метод является слотом;
- `showToday()` — отображает страницу с сегодняшней датой. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousMonth()` — отображает страницу с предыдущим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showNextMonth()` — отображает страницу со следующим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousYear()` — отображает страницу с текущим месяцем в предыдущем году. Выбранная дата при этом не изменяется. Метод является слотом;
- `showNextYear()` — отображает страницу с текущим месяцем в следующем году. Выбранная дата при этом не изменяется. Метод является слотом;
- `setFirstDayOfWeek(<День>)` — задает первый день недели. По умолчанию используется воскресенье. Чтобы первым днем недели сделать понедельник, следует в качестве параметра указать атрибут `Monday` из класса `QtCore.Qt`;
- `setNavigationBarVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то панель навигации выводиться не будет. Метод является слотом с сигнатурой `setNavigationBarVisible(bool)`;
- `setHorizontalHeaderFormat(<Формат>)` — задает формат горизонтального заголовка. В качестве параметра можно указать следующие атрибуты из класса `QCalendarWidget`:

- NoHorizontalHeader — 0 — заголовок не отображается;
- SingleLetterDayNames — 1 — отображается только первая буква из названия дня недели;
- ShortDayNames — 2 — отображается сокращенное название дня недели;
- LongDayNames — 3 — отображается полное название дня недели;

- setVerticalHeaderFormat(<Формат>) — задает формат вертикального заголовка. В качестве параметра можно указать следующие атрибуты из класса QCalendarWidget:
  - NoVerticalHeader — 0 — заголовок не отображается;
  - ISOWeekNumbers — 1 — отображается номер недели в году;
- setGridVisible(<Флаг>) — если в качестве параметра указано значение True, то будут отображены линии сетки;
- setSelectionMode(<Режим>) — задает режим выделения даты. В качестве параметра можно указать следующие атрибуты из класса QCalendarWidget:
  - NoSelection — 0 — дата не может быть выбрана пользователем;
  - SingleSelection — 1 — может быть выбрана одна дата;
- setHeaderTextFormat(<QTextCharFormat>) — задает формат ячеек заголовка. В параметре указывается экземпляр класса QTextCharFormat;
- setWeekdayTextFormat(<День недели>, <QTextCharFormat>) — задает формат ячеек для указанного дня недели. В первом параметре указываются атрибуты Monday, Tuesday, Wednesday, Thursday, Friday, Saturday или Sunday из класса QtCore.Qt, а во втором параметре — экземпляр класса QTextCharFormat;
- setDateTextFormat(<QDate>, <QTextCharFormat>) — задает формат ячейки с указанной датой. В первом параметре указывается экземпляр класса QDate или экземпляр класса date из языка Python, а во втором параметре — экземпляр класса QTextCharFormat.

Класс QCalendarWidget содержит следующие сигналы:

- activated(const QDate&) — генерируется при двойном щелчке мышью или нажатии клавиши <Enter>. Внутри обработчика через параметр доступна дата;
- clicked(const QDate&) — генерируется при щелчке мышью на доступной дате. Внутри обработчика через параметр доступна выбранная дата;
- currentPageChanged(int,int) — генерируется при изменении страницы. Внутри обработчика через первый параметр доступен год, а через второй — месяц;
- selectionChanged() — генерируется при изменении выбранной даты пользователем или из программы.

23.11. Электронный индикатор

Класс QLCDNumber реализует электронный индикатор, в котором цифры и буквы отображаются отдельными сегментами, как на электронных часах или дисплее калькулятора. Индикатор позволяет отображать числа в двоичной, восьмеричной, десятичной и шестнадцатеричной системах счисления. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLCDNumber
```

Форматы конструктора класса QLCDNumber:

```
<Объект> = QLCDNumber([parent=<Родитель>])
<Объект> = QLCDNumber(<Количество цифр>[, parent=<Родитель>])
```

В параметре <Количество цифр> указывается количество отображаемых цифр. Если параметр не указан, то по умолчанию используется значение 5.

Класс QLCDNumber содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- display(<Значение>) — задает новое значение. В качестве параметра можно указать целое число, вещественное число или строку. Метод является слотом с сигнатурами display(int), display(double) и display(const QString&);
- checkOverflow(<Число>) — возвращает значение True, если целое или вещественное число, указанное в параметре, не может быть отображено индикатором. В противном случае возвращает значение False;
- intValue() — возвращает значение индикатора в виде целого числа;
- value() — возвращает значение индикатора в виде вещественного числа;
- setSegmentStyle(<Стиль>) — задает стиль индикатора. В качестве параметра можно указать атрибуты Outline, Filled или Flat из класса QLCDNumber;
- setMode(<Режим>) — задает режим отображения чисел. В качестве параметра можно указать следующие атрибуты из класса QLCDNumber:
  - Hex — 0 — шестнадцатеричное значение;
  - Dec — 1 — десятичное значение;
  - Oct — 2 — восьмеричное значение;
  - Bin — 3 — двоичное значение.

  Вместо метода setMode() удобнее воспользоваться слотами setHexMode(), setDecMode(), setOctMode() и setBinMode();
- setSmallDecimalPoint(<Флаг>) — если в качестве параметра указано значение True, то десятичная точка будет отображаться как отдельный элемент (при этом значение выводится более компактно без пробелов до и после точки), а если значение False — то десятичная точка будет занимать позицию цифры (значение используется по умолчанию). Метод является слотом с сигнатурой setSmallDecimalPoint(bool);
- setDigitCount(<Число>) — задает количество отображаемых цифр. Если в методе setSmallDecimalPoint() указано значение False, то десятичная точка считается одной цифрой.

Класс QLCDNumber содержит сигнал overflow(), который генерируется при попытке задать значение, которое не может быть отображено индикатором.

23.12. Индикатор хода процесса

Класс QProgressBar реализует индикатор хода процесса, с помощью которого можно информировать пользователя о текущем состоянии выполнения длительной операции. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QProgressBar
```

Формат конструктора класса QProgressBar:

```
<Объект> = QProgressBar([parent=<Родитель>])
```

Класс QProgressBar содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setValue(<Значение>) — задает новое значение. Метод является слотом с сигнатурой setValue(int);
- value() — возвращает текущее значение индикатора в виде числа;
- text() — возвращает текст, отображаемый на индикаторе или рядом с ним;
- setRange(<Минимум>, <Максимум>), setMinimum(<Минимум>) и setMaximum(<Максимум>) — задают минимальное и максимальное значения. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами с сигнатурами setRange(int,int), setMinimum(int) и setMaximum(int);
- reset() — сбрасывает значение индикатора. Метод является слотом;
- setOrientation(<Ориентация>) — задает ориентацию индикатора. В качестве значения указываются атрибуты Horizontal или Vertical из класса QtCore.Qt. Метод является слотом с сигнатурой setOrientation(Qt::Orientation);
- setTextVisible(<Флаг>) — если в качестве параметра указано значение False, то текст с текущим значением индикатора отображаться не будет;
- setTextDirection(<Направление>) — задает направление вывода текста при вертикальной ориентации индикатора. Обратите внимание на то, что при использовании стилей "windows", "windowsxp" и "macintosh" при вертикальной ориентации текст вообще не отображается. В качестве значения указываются следующие атрибуты из класса QProgressBar:
  - TopToBottom — 0 — текст поворачивается на 90 градусов по часовой стрелке;
  - BottomToTop — 1 — текст поворачивается на 90 градусов против часовой стрелки;
- setInvertedAppearance(<Флаг>) — если в качестве параметра указано значение True, то направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево при горизонтальной ориентации).

При изменении значения индикатора генерируется сигнал valueChanged(int) . Внутри обработчика через параметр доступно новое значение.

23.13. Шкала с ползунком

Класс QSlider реализует шкалу с ползунком, который можно перемещать с помощью указателя мыши или клавиатуры. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QSlider
```

Форматы конструктора класса QSlider:

```
<Объект> = QSlider([parent=<Родитель>])
<Объект> = QSlider(<Ориентация>[, parent=<Родитель>])
```

Параметр <Ориентация> позволяет задать ориентацию шкалы. В качестве значения указываются атрибуты Horizontal или Vertical (значение по умолчанию) из класса QtCore.Qt.

Класс QSlider наследует следующие методы из класса QAbstractSlider (перечислены только основные методы; полный список смотрите в документации):

- setValue(<Значение>) — задает новое значение. Метод является слотом с сигнатурой setValue(int);
- value() — возвращает установленное положение ползунка в виде числа;
- setSliderPosition(<Значение>) — задает текущее положение ползунка;
- sliderPosition() — возвращает текущее положение ползунка в виде числа. Если отслеживание перемещения ползунка включено (по умолчанию), то возвращаемое значение будет совпадать со значением, возвращаемым методом value(). Если отслеживание выключено, то при перемещении метод sliderPosition() вернет текущее положение, а метод value() — положение, которое имел ползунок до перемещения;
- setRange(<Минимум>, <Максимум>), setMinimum(<Минимум>) и setMaximum(<Максимум>) — задают минимальное и максимальное значения;
- setOrientation(<Ориентация>) — задает ориентацию шкалы. В качестве значения указываются атрибуты Horizontal или Vertical из класса QtCore.Qt;
- setSingleStep(<Значение>) — задает значение, на которое сдвинется ползунок при нажатии клавиш со стрелками;
- setPageStep(<Значение>) — задает значение, на которое сдвинется ползунок при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши или щелчке мышью на шкале;
- setInvertedAppearance(<Флаг>) — если в качестве параметра указано значение True, то направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево при горизонтальной ориентации);
- setInvertedControls(<Флаг>) — если в качестве параметра указано значение False, то при изменении направления увеличения значения будет изменено и направление перемещения ползунка при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши и нажатии клавиш со стрелками вверх и вниз;
- setTracking(<Флаг>) — если в качестве параметра указано значение True, то отслеживание перемещения ползунка будет включено (значение по умолчанию). При этом сигнал valueChanged(int) будет генерироваться постоянно при перемещении ползунка. Если в качестве параметра указано значение False, то сигнал valueChanged(int) будет сгенерирован только при отпускании ползунка;
- hasTracking() — возвращает значение True, если отслеживание перемещения ползунка включено, и False — в противном случае.

Класс QAbstractSlider содержит следующие сигналы:

- actionTriggered(int) — генерируется, когда производится взаимодействие с ползунком, например, при нажатии клавиши <Page Up>. Внутри обработчика через параметр доступно произведенное действие, которое описывается атрибутами SliderNoAction (0), SliderSingleStepAdd (1), SliderSingleStepSub (2), SliderPageStepAdd (3), SliderPageStepSub (4), SliderToMinimum (5), SliderToMaximum (6) и SliderMove (7) из класса QAbstractSlider;
- rangeChanged(int,int) — генерируется при изменении диапазона значений. Внутри обработчика через первый параметр доступно новое минимальное значение, а через второй параметр — новое максимальное значение;

- `sliderPressed()` — генерируется при нажатии ползунка;
- `sliderMoved(int)` — генерируется постоянно при перемещении ползунка. Внутри обработчика через параметр доступно новое положение ползунка;
- `sliderReleased()` — генерируется при отпускании ранее нажатого ползунка;
- `valueChanged(int)` — генерируется при изменении значения. Внутри обработчика через параметр доступно новое значение.

Класс `QSlider` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setTickPosition(<Позиция>)` — задает позицию рисок. В качестве параметра указываются следующие атрибуты из класса `QSlider`:
  - `NoTicks` — без рисок;
  - `TicksBothSides` — риски по обе стороны;
  - `TicksAbove` — риски выводятся сверху;
  - `TicksBelow` — риски выводятся снизу;
  - `TicksLeft` — риски выводятся слева;
  - `TicksRight` — риски выводятся справа;
- `setTickInterval(<Расстояние>)` — задает расстояние между рисками.

23.14. Класс *QDial*

Класс `QDial` реализует круглую шкалу с ползунком круглой или треугольной формы (вид зависит от используемого стиля), который можно перемещать по кругу с помощью указателя мыши или клавиатуры. Компонент напоминает регулятор, используемый в различных устройствах для изменения или отображения каких-либо настроек. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QDial
```

Формат конструктора класса `QDial`:

```
<Объект> = QDial([parent=<Родитель>])
```

Класс `QDial` наследует все методы и сигналы из класса `QAbstractSlider` (см. *разд. 23.13*) и содержит несколько дополнительных методов (перечислены только основные методы; полный список смотрите в документации):

- `setNotchesVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то риски будут отображены. По умолчанию риски не выводятся. Метод является слотом с сигнатурой `setNotchesVisible(bool)`;
- `setNotchTarget(<Значение>)` — задает рекомендуемое количество пикселов между рисками. В качестве параметра указывается вещественное число;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то начало шкалы будет совпадать с ее концом. По умолчанию между началом шкалы и концом расположено пустое пространство. Метод является слотом с сигнатурой `setWrapping(bool)`.

23.15. Полоса прокрутки

Класс QScrollBar реализует горизонтальную и вертикальную полосу прокрутки. Изменить значение можно с помощью нажатия кнопок, расположенных по краям полосы, щелчка мышью на полосе, путем перемещения ползунка, нажатия клавиш на клавиатуре, а также выбрав соответствующий пункт из контекстного меню. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QScrollBar
```

Форматы конструктора класса QScrollBar:

```
<Объект> = QScrollBar([parent=<Родитель>])
<Объект> = QScrollBar(<Ориентация>[, parent=<Родитель>])
```

Параметр <Ориентация> позволяет задать ориентацию полосы прокрутки. В качестве значения указываются атрибуты Horizontal или Vertical (значение по умолчанию) из класса QtCore.Qt.

Класс QScrollBar наследует все методы и сигналы из класса QAbstractSlider (см. *разд. 23.13*) и не содержит дополнительных методов.

> ***Примечание***
>
> Полоса прокрутки редко используется отдельно. Гораздо удобнее воспользоваться областью с полосами прокрутки, которую реализует класс QScrollArea (см. *разд. 22.12*).

ГЛАВА 24

Списки и таблицы

В PyQt имеется широкий выбор компонентов, позволяющих отображать как одномерный список строк (в свернутом или развернутом состояниях), так и табличные данные. Кроме того, можно отобразить данные, которые имеют очень сложную структуру, например иерархическую. Благодаря поддержке концепции "модель/представление", позволяющей отделить данные от их отображения, одни и те же данные можно отображать сразу в нескольких компонентах без их дублирования.

24.1. Раскрывающийся список

Класс `QComboBox` реализует раскрывающийся список с возможностью выбора одного пункта. При щелчке мышью на поле появляется список возможных вариантов, а при выборе пункта список сворачивается. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QComboBox
```

Формат конструктора класса `QComboBox`:

```
<Объект> = QComboBox([parent=<Родитель>])
```

24.1.1. Добавление, изменение и удаление элементов

Для добавления, изменения, удаления и получения значения элементов предназначены следующие методы из класса `QComboBox`:

- `addItem()` — добавляет один элемент в конец списка. Форматы метода:

  ```
  addItem(<Строка>[, <Данные>])
  addItem(<QIcon>, <Строка>[, <Данные>])
  ```

  В параметре `<Строка>` задается текст элемента списка, а в параметре `<QIcon>` — иконка, которая будет отображена перед текстом. Необязательный параметр `<Данные>` позволяет сохранить пользовательские данные, например, индекс в таблице базы данных;

- `addItems(<Список строк>)` — добавляет несколько элементов в конец списка;

- `insertItem()` — вставляет один элемент в указанную позицию списка. Все остальные элементы сдвигаются в конец списка. Форматы метода:

  ```
  insertItem(<Индекс>, <Строка>[, <Данные>])
  insertItem(<Индекс>, <QIcon>, <Строка>[, <Данные>])
  ```

- `insertItems(<Индекс>, <Список строк>)` — вставляет несколько элементов в указанную позицию списка. Все остальные элементы сдвигаются в конец списка;
- `insertSeparator(<Индекс>)` — вставляет разделительную линию в указанную позицию;
- `setItemText(<Индекс>, <Строка>)` — изменяет текст элемента с указанным индексом;
- `setItemIcon(<Индекс>, <QIcon>)` — изменяет иконку элемента с указанным индексом;
- `setItemData(<Индекс>, <Данные>[, role=UserRole])` — изменяет данные для элемента с указанным индексом. Необязательный параметр `role` позволяет указать роль, для которой задаются данные. Например, если указать атрибут `ToolTipRole` из класса `QtCore.Qt`, то данные задают текст всплывающей подсказки, которая будет отображена при наведении указателя мыши на элемент. По умолчанию изменяются пользовательские данные;
- `setCurrentIndex(<Индекс>)` — делает элемент с указанным индексом текущим. Метод является слотом с сигнатурой `setCurrentIndex(int)`;
- `currentIndex()` — возвращает индекс текущего элемента;
- `currentText()` — возвращает текст текущего элемента;
- `itemText(<Индекс>)` — возвращает текст элемента с указанным индексом;
- `itemData(<Индекс>[, role=UserRole])` — возвращает данные, сохраненные в роли `role` элемента с индексом `<Индекс>`;
- `count()` — возвращает общее количество элементов списка. Получить количество элементов можно также с помощью функции `len()`;
- `removeItem(<Индекс>)` — удаляет элемент с указанным индексом;
- `clear()` — удаляет все элементы списка.

24.1.2. Изменение настроек

Управлять настройками раскрывающегося списка позволяют следующие методы:

- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь сможет добавлять новые элементы в список путем ввода текста в поле и последующего нажатия клавиши <Enter>;
- `setInsertPolicy(<Режим>)` — задает режим добавления нового элемента пользователем. В качестве параметра указываются следующие атрибуты из класса `QComboBox`:
  - `NoInsert` — `0` — элемент не будет добавлен;
  - `InsertAtTop` — `1` — элемент вставляется в начало списка;
  - `InsertAtCurrent` — `2` — будет изменен текст текущего элемента;
  - `InsertAtBottom` — `3` — элемент вставляется в конец списка;
  - `InsertAfterCurrent` — `4` — элемент вставляется после текущего элемента;
  - `InsertBeforeCurrent` — `5` — элемент вставляется перед текущим элементом;
  - `InsertAlphabetically` — `6` — при вставке учитывается алфавитный порядок следования элементов;
- `setEditText(<Текст>)` — вставляет текст в поле редактирования. Метод является слотом с сигнатурой `setEditText(const QString&)`;

- `clearEditText()` — удаляет текст из поля редактирования. Метод является слотом;
- `setAutoCompletion(<Флаг>)` — если в качестве параметра указано значение `True`, то режим отображения возможных вариантов, начинающихся с введенных букв, будет включен. Значение `False` отключает отображение вариантов;
- `setCompleter(<QCompleter>)` — позволяет предлагать возможные варианты значений, начинающиеся с введенных пользователем символов. В качестве параметра указывается экземпляр класса `QCompleter`;
- `setAutoCompletionCaseSensitivity(<Режим>)` — задает режим учета регистра символов при отображении возможных вариантов. При указании атрибута `CaseInsensitive` из класса `QtCore.Qt` регистр букв учитываться не будет. Атрибут `CaseSensitive` задает регистрозависимый режим;
- `autoCompletion()` — возвращает значение `True`, если режим отображения возможных вариантов включен, и `False` — в противном случае;
- `setValidator(<QValidator>)` — устанавливает контроль ввода. В качестве значения указывается экземпляр класса, наследующего класс `QValidator` (см. *разд. 23.5.3*);
- `setDuplicatesEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может добавить элемент с повторяющимся текстом. По умолчанию повторы запрещены;
- `setMaxCount(<Количество>)` — задает максимальное количество элементов в списке. Если до вызова метода количество элементов превышало указанное количество, то лишние элементы будут удалены;
- `setMaxVisibleItems(<Количество>)` — задает максимальное количество видимых элементов в раскрывающемся списке;
- `setMinimumContentsLength(<Количество>)` — задает минимальное количество отображаемых символов;
- `setSizeAdjustPolicy(<Режим>)` — устанавливает режим изменения ширины при изменении содержимого. В качестве параметра указываются следующие атрибуты из класса `QComboBox`:
  - `AdjustToContents` — `0` — ширина будет соответствовать содержимому;
  - `AdjustToContentsOnFirstShow` — `1` — ширина будет соответствовать ширине, используемой при первом отображении списка;
  - `AdjustToMinimumContentsLength` — `2` — `AdjustToContents` или `AdjustToContentsOnFirstShow`;
  - `AdjustToMinimumContentsLengthWithIcon` — `3` — используется значение минимальной ширины, которое установлено с помощью метода `setMinimumContentsLength()`, плюс ширина иконки;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, то поле будет отображаться без рамки;
- `setIconSize(<QSize>)` — задает максимальный размер иконок;
- `showPopup()` — отображает список;
- `hidePopup()` — скрывает список.

24.1.3. Поиск элемента внутри списка

Произвести поиск элемента внутри списка позволяют методы `findText()` (поиск в тексте элемента) и `findData()` (поиск данных в указанной роли). Методы возвращают индекс найденного элемента или значение `-1`, если элемент не найден. Форматы методов:

```
findText(<Текст>[,  flags=MatchExactly | MatchCaseSensitive])
findData(<Данные>[, role=UserRole][,
                    flags=MatchExactly | MatchCaseSensitive])
```

Параметр `flags` задает режим поиска. В качестве значения можно указать комбинацию (через оператор `|`) следующих атрибутов из класса `QtCore.Qt`:

- `MatchExactly` — `0` — поиск полного соответствия;
- `MatchFixedString` — `8` — поиск полного соответствия внутри строки, выполняемый по умолчанию без учета регистра символов;
- `MatchContains` — `1` — поиск совпадения с любой частью;
- `MatchStartsWith` — `2` — совпадение с началом;
- `MatchEndsWith` — `3` — совпадение с концом;
- `MatchRegExp` — `4` — поиск с помощью регулярного выражения;
- `MatchWildcard` — `5` — используются подстановочные знаки;
- `MatchCaseSensitive` — `16` — поиск с учетом регистра символов;
- `MatchWrap` — `32` — поиск по кругу;
- `MatchRecursive` — `64` — просмотр всей иерархии.

24.1.4. Сигналы

Класс `QComboBox` содержит следующие сигналы:

- `activated(int)` и `activated(const QString&)` — генерируются при выборе пункта в списке (даже если индекс не изменился) пользователем. Внутри обработчика доступен индекс или текст элемента;
- `currentIndexChanged(int)` и `currentIndexChanged(const QString&)` — генерируются при изменении текущего индекса. Внутри обработчика доступен индекс (значение `-1`, если список пуст) или текст элемента;
- `editTextChanged(const QString&)` — генерируется при изменении текста в поле. Внутри обработчика через параметр доступен новый текст;
- `highlighted(int)` и `highlighted(const QString&)` — генерируются при наведении указателя мыши на пункт в списке. Внутри обработчика доступен индекс или текст элемента.

24.2. Список для выбора шрифта

Класс `QFontComboBox` реализует раскрывающийся список с названиями шрифтов. Шрифт можно выбрать из списка или ввести название в поле, при этом будут отображаться названия, начинающиеся с введенных букв. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QComboBox — QFontComboBox
```

Формат конструктора класса QFontComboBox:

```
<Объект> = QFontComboBox([parent=<Родитель>])
```

Класс QFontComboBox наследует все методы и сигналы из класса QComboBox (см. *разд. 24.1*) и содержит несколько дополнительных методов:

- setCurrentFont(<QFont>) — делает текущим элемент, соответствующий указанному шрифту. В качестве параметра указывается экземпляр класса QFont. Метод является слотом с сигнатурой setCurrentFont(const QFont&). Пример:

  ```
  comboBox.setCurrentFont(QtGui.QFont("Verdana"))
  ```

- currentFont() — возвращает экземпляр класса QFont с выбранным шрифтом. Пример вывода названия шрифта:

  ```
  print(comboBox.currentFont().family())
  ```

- setFontFilters(<Фильтр>) — ограничивает список указанными типами шрифтов. В качестве параметра указывается комбинация следующих атрибутов из класса QFontComboBox:
  - AllFonts — 0 — все типы шрифтов;
  - ScalableFonts — 1 — масштабируемые шрифты;
  - NonScalableFonts — 2 — не масштабируемые шрифты;
  - MonospacedFonts — 4 — моноширинные шрифты;
  - ProportionalFonts — 8 — пропорциональные шрифты.

Класс QFontComboBox содержит сигнал currentFontChanged(const QFont&), который генерируется при изменении текущего шрифта. Внутри обработчика доступен экземпляр класса QFont с текущим шрифтом.

24.3. Роли элементов

Каждый элемент списка содержит данные, распределенные по ролям. С помощью этих данных можно указать текст элемента, каким шрифтом и цветом отображается текст, задать текст всплывающей подсказки и многое другое. Перечислим роли элементов (атрибуты из класса QtCore.Qt):

- DisplayRole — 0 — отображаемые данные (обычно текст);
- DecorationRole — 1 — изображение (обычно иконка);
- EditRole — 2 — данные в виде, удобном для редактирования;
- ToolTipRole — 3 — текст всплывающей подсказки;
- StatusTipRole — 4 — текст для строки состояния;
- WhatsThisRole — 5 — текст для справки;
- FontRole — 6 — шрифт элемента. Указывается экземпляр класса QFont;
- TextAlignmentRole — 7 — выравнивание текста внутри элемента;
- BackgroundRole — 8 — фон элемента. Указывается экземпляр класса QBrush;
- ForegroundRole — 9 — цвет текста. Указывается экземпляр класса QBrush;

- `CheckStateRole` — `10` — статус флажка. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `AccessibleTextRole` — `11` — текст для плагинов;
- `AccessibleDescriptionRole` — `12` — описание элемента;
- `SizeHintRole` — `13` — рекомендуемый размер элемента. Указывается экземпляр класса `QSize`;
- `UserRole` — `32` — любые пользовательские данные, например индекс элемента в базе данных. Можно сохранить несколько данных, указав их в роли с индексом более `32`. Пример:

```
comboBox.setItemData(0, 50, role=QtCore.Qt.UserRole)
comboBox.setItemData(0, "Другие данные",
                            role=QtCore.Qt.UserRole + 1)
```

24.4. Модели

Для отображения данных в виде списков и таблиц применяется концепция "модель/представление", позволяющая отделить данные от их представления и избежать дублирования данных. В основе концепции лежат следующие составляющие:

- *модель* — является "оберткой" над данными. Позволяет добавлять, изменять и удалять данные, а также содержит методы для чтения данных и управления ими;
- *представление* — предназначено для отображения элементов модели. В нескольких представлениях можно установить одну модель;
- *модель выделения* — позволяет управлять выделением. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении;
- *промежуточная модель* — является прослойкой между моделью и представлением. Позволяет производить сортировку и фильтрацию данных на основе базовой модели без изменения порядка следования элементов в базовой модели;
- *делегат* — выполняет рисование каждого элемента в отдельности, а также позволяет произвести редактирование данных. В своих программах вы можете наследовать стандартные классы делегатов и полностью управлять отображением данных и их редактированием. За подробной информацией по классам делегатов обращайтесь к документации.

24.4.1. Доступ к данным внутри модели

Доступ к данным внутри модели реализуется с помощью класса `QModelIndex`. Формат конструктора класса:

```
<Объект> = QModelIndex([<QModelIndex или QPersistentModelIndex>])
```

Если параметр не указан, то создается невалидный объект, который ни на что не указывает. Такой объект обычно возвращается, когда элемента не существует в модели. Наиболее часто экземпляр класса `QModelIndex` создается с помощью метода `index()` из класса модели или метода `currentIndex()` из класса `QAbstractItemView`.

Класс `QModelIndex` содержит следующие методы:

- `isValid()` — возвращает значение `True`, если объект является валидным, и `False` — в противном случае;
- `data([DisplayRole])` — возвращает данные, хранящиеся в указанной роли;
- `flags()` — содержит свойства элемента. Возвращает комбинацию следующих атрибутов из класса `QtCore.Qt`:
  - `NoItemFlags` — `0` — свойства не установлены;
  - `ItemIsSelectable` — `1` — можно выделить;
  - `ItemIsEditable` — `2` — можно редактировать;
  - `ItemIsDragEnabled` — `4` — можно перетаскивать;
  - `ItemIsDropEnabled` — `8` — можно сбрасывать перетаскиваемые данные;
  - `ItemIsUserCheckable` — `16` — может быть включен и выключен;
  - `ItemIsEnabled` — `32` — пользователь может взаимодействовать с элементом;
  - `ItemIsTristate` — `64` — флажок имеет три состояния;
- `row()` — возвращает индекс строки;
- `column()` — возвращает индекс столбца;
- `parent()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень выше по иерархии. Если элемента нет, то возвращается невалидный экземпляр класса `QModelIndex`;
- `child(<Строка>, <Столбец>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень ниже по иерархии на указанной позиции. Если элемента нет, то возвращается невалидный экземпляр класса `QModelIndex`;
- `sibling(<Строка>, <Столбец>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на том же уровне вложенности на указанной позиции. Если элемента нет, то возвращается невалидный экземпляр класса `QModelIndex`;
- `model()` — возвращает ссылку на модель.

Следует учитывать, что модель может измениться и экземпляр класса `QModelIndex` будет ссылаться на несуществующий уже элемент. Если необходимо сохранить ссылку на элемент, то следует воспользоваться классом `QPersistentModelIndex`, который содержит те же самые методы, но обеспечивает валидность ссылки.

24.4.2. Класс *QStringListModel*

Класс `QStringListModel` реализует одномерную модель, содержащую список строк. Модель можно отобразить с помощью классов `QListView`, `QComboBox` и др., передав в метод `setModel()`. Иерархия наследования:

```
QObject — QAbstractItemModel — QAbstractListModel — QStringListModel
```

Форматы конструктора класса QStringListModel:

```
<Объект> = QStringListModel([parent=<Родитель>])
<Объект> = QStringListModel(<Список строк>[, parent=<Родитель>])
```

Класс QStringListModel содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setStringList(<Список строк>) — устанавливает список строк;
- stringList() — возвращает список строк, хранимых в модели;
- insertRows(<Индекс>, <Количество>[, parent]) — вставляет указанное количество элементов в позицию, заданную первым параметром. Остальные элементы сдвигаются в конец списка. Метод возвращает значение True, если операция успешно выполнена;
- removeRows(<Индекс>, <Количество>[, parent]) — удаляет указанное количество элементов, начиная с позиции, заданной первым параметром. Метод возвращает значение True, если операция успешно выполнена;
- setData(<QModelIndex>, <Значение>[, role=EditRole]) — задает значение для роли role элемента, на который указывает индекс <QModelIndex>. Метод возвращает значение True, если операция успешно выполнена;
- data(<QModelIndex>, <Роль>) — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс <QModelIndex>;
- rowCount([parent]) — возвращает количество строк в модели;
- sort(<Индекс столбца>[, order=AscendingOrder]) — производит сортировку. Если во втором параметре указан атрибут AscendingOrder из класса QtCore.Qt, то сортировка производится в прямом порядке, а если DescendingOrder — то в обратном.

Класс QStringListModel наследует метод index() из класса QAbstractListModel, который возвращает индекс (экземпляр класса QModelIndex) внутри модели. Формат метода:

```
index(<Строка>[, column=0][, parent=QModelIndex()])
```

24.4.3. Класс *QStandardItemModel*

Класс QStandardItemModel реализует двумерную (таблица) и иерархическую модели. Каждый элемент такой модели представлен классом QStandardItem. Модель можно отобразить с помощью классов QTableView, QTreeView и др., передав в метод setModel(). Иерархия наследования:

```
QObject — QAbstractItemModel — QStandardItemModel
```

Форматы конструктора класса QStandardItemModel:

```
<Объект> = QStandardItemModel([parent=<Родитель>])
<Объект> = QStandardItemModel(<Количество строк>, <Количество столбцов>
                              [, parent=<Родитель>])
```

Класс QStandardItemModel содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setRowCount(<Количество строк>) — задает количество строк;
- setColumnCount(<Количество столбцов>) — задает количество столбцов;
- rowCount([parent=QModelIndex()]) — возвращает количество строк;

- `columnCount([parent=QModelIndex()])` — возвращает количество столбцов;
- `setItem(<Строка>, <Столбец>, <QStandardItem>)` — устанавливает элемент в указанную ячейку. Пример заполнения таблицы:

```
model = QtGui.QStandardItemModel(3, 4)
for row in range(0, 3):
    for column in range(0, 4):
        item = QtGui.QStandardItem(
                                "({0}, {1})".format(row, column))
        model.setItem(row, column, item)
view.setModel(model)
```

- `appendRow(<Список>)` — добавляет одну строку в конец модели. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `appendColumn(<Список>)` — добавляет один столбец в конец модели. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `insertRow(<Индекс строки>, <Список>)` — добавляет одну строку в указанную позицию модели. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;
- `insertRow(<Индекс>[, parent=QModelIndex()])` — добавляет одну строку в указанную позицию модели. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько строк в указанную позицию модели. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertColumn(<Индекс столбца>, <Список>)` — добавляет один столбец в указанную позицию модели. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;
- `insertColumn(<Индекс>[, parent=QModelIndex()])` — добавляет один столбец в указанную позицию. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько столбцов в указанную позицию. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество строк, начиная со строки, имеющей индекс `<Индекс>`. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс `<Индекс>`. Метод возвращает значение `True`, если операция успешно выполнена;
- `takeItem(<Строка>[, <Столбец>=0])` — удаляет указанный элемент из модели и возвращает его (экземпляр класса `QStandardItem`);
- `takeRow(<Индекс>)` — удаляет указанную строку из модели и возвращает ее (список экземпляров класса `QStandardItem`);
- `takeColumn(<Индекс>)` — удаляет указанный столбец из модели и возвращает его (список экземпляров класса `QStandardItem`);
- `clear()` — удаляет все элементы из модели;

- item(<Строка>[, <Столбец>=0]) — возвращает ссылку на элемент (экземпляр класса QStandardItem), расположенный в указанной ячейке;
- invisibleRootItem() — возвращает ссылку на невидимый корневой элемент модели (экземпляр класса QStandardItem);
- itemFromIndex(<QModelIndex>) — возвращает ссылку на элемент (экземпляр класса QStandardItem), на который ссылается индекс <QModelIndex>;
- index(<Строка>, <Столбец>[, parent=QModelIndex()]) — возвращает индекс элемента (экземпляр класса QModelIndex), расположенного в указанной ячейке;
- indexFromItem(<QStandardItem>) — возвращает индекс элемента (экземпляр класса QModelIndex), ссылка на который передана в качестве параметра;
- setData(<QModelIndex>, <Значение>[, role=EditRole]) — задает значение для роли role элемента, на который указывает индекс <QModelIndex>. Метод возвращает значение True, если операция успешно выполнена;
- data(<QModelIndex>[, role=DisplayRole]) — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс <QModelIndex>;
- setHorizontalHeaderLabels(<Список строк>) — задает заголовки столбцов. В качестве параметра указывается список строк;
- setVerticalHeaderLabels(<Список строк>) — задает заголовки строк. В качестве параметра указывается список строк;
- setHorizontalHeaderItem(<Индекс>, <QStandardItem>) — задает заголовок столбца. В первом параметре указывается индекс столбца, а во втором параметре — экземпляр класса QStandardItem;
- setVerticalHeaderItem(<Индекс>, <QStandardItem>) — задает заголовок строки. В первом параметре указывается индекс строки, а во втором параметре — экземпляр класса QStandardItem;
- horizontalHeaderItem(<Индекс>) — возвращает ссылку на указанный заголовок столбца (экземпляр класса QStandardItem);
- verticalHeaderItem(<Индекс>) — возвращает ссылку на указанный заголовок строки (экземпляр класса QStandardItem);
- setHeaderData(<Индекс>, <Ориентация>, <Значение>[, role]) — задает данные для указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором параметре — ориентация (атрибуты Horizontal или Vertical из класса QtCore.Qt). Если параметр role не указан, то используется значение EditRole. Метод возвращает значение True, если операция успешно выполнена;
- headerData(<Индекс>, <Ориентация>[, role]) — возвращает данные, хранящиеся в указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором параметре — ориентация. Если параметр role не указан, то используется значение DisplayRole;
- findItems(<Текст>[, flags=MatchExactly][, column=0]) — производит поиск элемента внутри модели в указанном в параметре column столбце. Допустимые значения параметра flags мы уже рассматривали в *разд. 24.1.3*. В качестве значения метод возвращает список элементов (список экземпляров класса QStandardItem) или пустой список;

- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном;
- `setSortRole(<Роль>)` — задает роль (см. *разд. 24.3*), по которой производится сортировка;
- `parent(<QModelIndex>)` — возвращает индекс (экземпляр класса `QModelIndex`) родительского элемента. В качестве параметра указывается индекс (экземпляр класса `QModelIndex`) элемента-потомка;
- `hasChildren([parent=QModelIndex()])` — возвращает значение `True`, если существует элемент, расположенный на один уровень ниже по иерархии, и `False` — в противном случае.

При изменении значения элемента генерируется сигнал `itemChanged(QStandardItem *)`. Внутри обработчика через параметр доступна ссылка на элемент (экземпляр класса `QStandardItem`).

24.4.4. Класс *QStandardItem*

Каждый элемент модели `QStandardItemModel` представлен классом `QStandardItem`. Этот класс не только описывает элемент, но и позволяет создавать вложенные структуры. Форматы конструктора класса:

```
<Объект> = QStandardItem()
<Объект> = QStandardItem(<Текст>)
<Объект> = QStandardItem(<QIcon>, <Текст>)
<Объект> = QStandardItem(<Количество строк>[, <Количество столбцов>=1])
<Объект> = QStandardItem(<QStandardItem>)
```

Класс `QStandardItem` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowCount(<Количество строк>)` — задает количество дочерних строк;
- `setColumnCount(<Количество столбцов>)` — задает количество дочерних столбцов;
- `rowCount()` — возвращает количество дочерних строк;
- `columnCount()` — возвращает количество дочерних столбцов;
- `row()` — возвращает индекс строки в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя;
- `column()` — возвращает индекс столбца в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя;
- `setChild(<Строка>, <Столбец>, <QStandardItem>)` — устанавливает элемент в указанную ячейку дочерней таблицы. Пример создания иерархии:

```
parent = QtGui.QStandardItem(3, 4)
parent.setText("Элемент-родитель")
for row in range(0, 3):
    for column in range(0, 4):
        item = QtGui.QStandardItem(
                    "({0}, {1})".format(row, column))
        parent.setChild(row, column, item)
model.appendRow(parent)
```

- appendRow(<Список>) — добавляет одну строку в конец дочерней таблицы. В качестве параметра указывается список экземпляров класса QStandardItem;
- appendRow(<QStandardItem>) — добавляет один элемент в конец дочерней таблицы. В качестве параметра указывается экземпляр класса QStandardItem;
- appendRows(<Список>) — добавляет несколько строк в конец дочерней таблицы. В качестве параметра указывается список экземпляров класса QStandardItem;
- appendColumn(<Список>) — добавляет один столбец в конец дочерней таблицы. В качестве параметра указывается список экземпляров класса QStandardItem;
- insertRow(<Индекс строки>, <Список>) — добавляет одну строку в указанную позицию дочерней таблицы. В качестве параметра <Список> указывается список экземпляров класса QStandardItem;
- insertRow(<Индекс строки>, <QStandardItem>) — добавляет один элемент в указанную позицию дочерней таблицы. В качестве второго параметра указывается экземпляр класса QStandardItem;
- insertRows(<Индекс строки>, <Список>) — добавляет несколько строк в указанную позицию дочерней таблицы. В качестве параметра <Список> указывается список экземпляров класса QStandardItem;
- insertRows(<Индекс строки>, <Количество>) — добавляет несколько строк в указанную позицию дочерней таблицы;
- insertColumn(<Индекс столбца>, <Список>) — добавляет один столбец в указанную позицию дочерней таблицы. В качестве параметра <Список> указывается список экземпляров класса QStandardItem;
- insertColumns(<Индекс>, <Количество>) — добавляет несколько столбцов в указанную позицию;
- removeRow(<Индекс>) — удаляет строку с указанным индексом;
- removeRows(<Индекс>, <Количество>) — удаляет указанное количество строк, начиная со строки, имеющей индекс <Индекс>;
- removeColumn(<Индекс>) — удаляет столбец с указанным индексом;
- removeColumns(<Индекс>, <Количество>) — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс <Индекс>;
- takeChild(<Строка>[, <Столбец>=0]) — удаляет указанный дочерний элемент и возвращает его (экземпляр класса QStandardItem);
- takeRow(<Индекс>) — удаляет указанную строку из дочерней таблицы и возвращает ее (список экземпляров класса QStandardItem);
- takeColumn(<Индекс>) — удаляет указанный столбец из дочерней таблицы и возвращает его (список экземпляров класса QStandardItem);
- parent() — возвращает ссылку на родительский элемент (экземпляр класса QStandardItem) или значение None;
- child(<Строка>[, <Столбец>=0]) — возвращает ссылку на дочерний элемент (экземпляр класса QStandardItem) или значение None;
- hasChildren() — возвращает значение True, если существует дочерний элемент, и False — в противном случае;

- `setData(<Значение>[, role=UserRole+1])` — устанавливает значение для указанной роли;
- `data([UserRole+1])` — возвращает значение, хранимое в указанной роли;
- `setText(<Текст>)` — задает текст элемента;
- `text()` — возвращает текст элемента;
- `setTextAlignment(<Выравнивание>)` — задает выравнивание текста внутри элемента;
- `setIcon(<QIcon>)` — задает иконку, которая будет отображена перед текстом;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст для справки;
- `setFont(<QFont>)` — задает шрифт элемента;
- `setBackground(<QBrush>)` — задает цвет фона;
- `setForeground(<QBrush>)` — задает цвет текста;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может взаимодействовать с флажком;
- `isCheckable()` — возвращает значение `True`, если пользователь может взаимодействовать с флажком, и `False` — в противном случае;
- `setCheckState(<Статус>)` — задает статус флажка. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `checkState()` — возвращает текущий статус флажка;
- `setTristate(<Флаг>)` — если в качестве параметра указано значение `True`, то флажок может иметь три состояния: установлен, сброшен и частично установлен;
- `isTristate()` — возвращает значение `True`, если флажок может иметь три состояния, и `False` — в противном случае;
- `setFlags(<Флаги>)` — задает свойства элемента (см. *разд. 24.4.1*);
- `flags()` — возвращает значение установленных свойств элемента;
- `setSelectable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может выделить элемент;
- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может редактировать текст элемента;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то перетаскивание элемента разрешено;
- `setDropEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то сброс разрешен;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может взаимодействовать с элементом. Значение `False` делает элемент недоступным;
- `clone()` — возвращает копию элемента (экземпляр класса `QStandardItem`);

- `index()` — возвращает индекс элемента (экземпляр класса `QModelIndex`);
- `model()` — возвращает ссылку на модель (экземпляр класса `QStandardItemModel`);
- `sortChildren(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку дочерней таблицы. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном.

24.5. Представления

Для отображения элементов модели предназначены следующие классы представлений:

- `ListView` — реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме того, с помощью этого класса можно отображать иконки;
- `QTableView` — реализует таблицу;
- `QTreeView` — реализует иерархический список.

Помимо этих классов для отображения элементов модели можно воспользоваться классами `QComboBox` (раскрывающийся список; см. *разд. 24.1*), `QListWidget` (простой список), `QTableWidget` (таблица) и `QTreeWidget` (иерархический список). Последние три класса нарушают концепцию "модель/представление", хотя и базируются на этой концепции. За подробной информацией по этим классам обращайтесь к документации.

24.5.1. Класс *QAbstractItemView*

Абстрактный класс `QAbstractItemView` является базовым классом для всех представлений. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                             QAbstractItemView
```

Класс `QAbstractItemView` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setCurrentIndex(<QModelIndex>)` — делает элемент с указанным индексом (экземпляр класса `QModelIndex`) текущим. Метод является слотом с сигнатурой `setCurrentIndex(const QModelIndex&)`;
- `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента;
- `setRootIndex(<QModelIndex>)` — задает корневой элемент. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом с сигнатурой `setRootIndex(const QModelIndex&)`;
- `rootIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) корневого элемента;
- `setAlternatingRowColors(<Флаг>)` — если в качестве параметра указано значение `True`, то четные и нечетные строки будут иметь разный цвет фона;
- `setIndexWidget(<QModelIndex>, <QWidget>)` — устанавливает компонент в позицию, указанную индексом (экземпляр класса `QModelIndex`);
- `indexWidget(<QModelIndex>)` — возвращает ссылку на компонент, установленный ранее в позицию, указанную индексом (экземпляр класса `QModelIndex`);

- `setSelectionModel(<QItemSelectionModel>)` — устанавливает модель выделения;
- `selectionModel()` — возвращает модель выделения;
- `setSelectionMode(<Режим>)` — задает режим выделения элементов. В качестве параметра указываются следующие атрибуты из класса `QAbstractItemView`:
  - `NoSelection` — `0` — элементы не могут быть выделены;
  - `SingleSelection` — `1` — можно выделить только один элемент;
  - `MultiSelection` — `2` — можно выделить несколько элементов. Повторный щелчок на элементе снимает выделение;
  - `ExtendedSelection` — `3` — можно выделить несколько элементов, удерживая нажатой клавишу <Ctrl> или щелкнув на элементе левой кнопкой мыши и перемещая мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
  - `ContiguousSelection` — `4` — можно выделить несколько элементов, щелкнув на элементе левой кнопкой мыши и перемещая мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
- `setSelectionBehavior(<Режим>)` — задает режим выделения. В качестве параметра указываются следующие атрибуты из класса `QAbstractItemView`:
  - `SelectItems` — `0` — выделяется отдельный элемент;
  - `SelectRows` — `1` — выделяется строка целиком;
  - `SelectColumns` — `2` — выделяется столбец целиком;
- `selectAll()` — выделяет все элементы. Метод является слотом;
- `clearSelection()` — снимает выделение. Метод является слотом;
- `setEditTriggers(<Режим>)` — задает действие, при котором производится начало редактирования текста элемента. В качестве параметра указывается комбинация следующих атрибутов из класса `QAbstractItemView`:
  - `NoEditTriggers` — `0` — редактировать нельзя;
  - `CurrentChanged` — `1` — при выделении элемента;
  - `DoubleClicked` — `2` — при двойном щелчке мышью;
  - `SelectedClicked` — `4` — при одинарном щелчке мышью на выделенном элементе;
  - `EditKeyPressed` — `8` — при нажатии клавиши <F2>;
  - `AnyKeyPressed` — `16` — при нажатии любой символьной клавиши;
  - `AllEditTriggers` — `31` — при любом вышеперечисленном действии;
- `setIconSize(<QSize>)` — задает размер иконок. В качестве параметра указывается экземпляр класса `QSize`;
- `setTextElideMode(<Режим>)` — задает режим обрезки текста, если он не помещается в отведенную область. В месте пропуска выводится троеточие. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `ElideLeft` — `0` — текст обрезается слева;
  - `ElideRight` — `1` — текст обрезается справа;

- ElideMiddle — 2 — текст обрезается посередине;
- ElideNone — 3 — текст не обрезается;

- setTabKeyNavigation(<Флаг>) — если в качестве параметра указано значение True, то между элементами можно перемещаться с помощью клавиши <Tab>;
- scrollTo(<QModelIndex>[, hint=EnsureVisible]) — прокручивает представление таким образом, чтобы элемент, на который ссылается индекс (экземпляр класса QModelIndex) был видим. В параметре hint указываются следующие атрибуты из класса QAbstractItemView:
  - EnsureVisible — 0 — элемент должен быть в области видимости;
  - PositionAtTop — 1 — элемент отображается в верхней части;
  - PositionAtBottom — 2 — элемент отображается в нижней части;
  - PositionAtCenter — 3 — элемент отображается в центре;
- scrollToTop() — прокручивает представление в самое начало. Метод является слотом;
- scrollToBottom() — прокручивает представление в самый конец. Метод является слотом;
- setDragEnabled(<Флаг>) — если в качестве параметра указано значение True, то перетаскивание элементов разрешено;
- setDragDropMode(<Режим>) — задает режим технологии drag & drop. В качестве параметра указываются следующие атрибуты из класса QAbstractItemView:
  - NoDragDrop — 0 — drag & drop не поддерживается;
  - DragOnly — 1 — поддерживается только перетаскивание;
  - DropOnly — 2 — поддерживается только сбрасывание;
  - DragDrop — 3 — поддерживается перетаскивание и сбрасывание;
  - InternalMove — 4 — перетаскивание и сбрасывание самого элемента, а не его копии;
- setDropIndicatorShown(<Флаг>) — если в качестве параметра указано значение True, то позиция возможного сброса элемента будет выделена;
- setAutoScroll(<Флаг>) — если в качестве параметра указано значение True, то при перетаскивании пункта будет производиться автоматическая прокрутка;
- setAutoScrollMargin(<Отступ>) — задает расстояние от края области, при достижении которого будет производиться автоматическая прокрутка области.

Класс QAbstractItemView содержит следующие сигналы:

- activated(const QModelIndex&) — генерируется при активизации элемента, например, путем нажатия клавиши <Enter>. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса QModelIndex);
- pressed(const QModelIndex&) — генерируется при нажатии кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса QModelIndex);
- clicked(const QModelIndex&) — генерируется при нажатии и отпускании кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса QModelIndex);

- doubleClicked(const QModelIndex&) — генерируется при двойном щелчке мышью над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса QModelIndex);
- entered(const QModelIndex&) — генерируется при вхождении указателя мыши в область элемента. Чтобы сигнал сработал, необходимо включить обработку перемещения указателя с помощью метода setMouseTracking() из класса QWidget. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса QModelIndex);
- viewportEntered() — генерируется при вхождении указателя мыши в область компонента. Чтобы сигнал сработал, необходимо включить обработку перемещения указателя с помощью метода setMouseTracking() из класса QWidget.

24.5.2. Простой список

Класс QListView реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме того, с помощью этого класса можно отображать иконки. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QListView
```

Формат конструктора класса QListView:

```
<Объект> = QListView([parent=<Родитель>])
```

Класс QListView наследует все методы и сигналы из класса QabstractItemView (см. *разд. 24.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setModel(<Модель>) — устанавливает модель;
- model() — возвращает ссылку на модель;
- setModelColumn(<Индекс>) — задает индекс отображаемого столбца в табличной модели;
- setViewMode(<Режим>) — задает режим отображения элементов. В качестве параметра указываются следующие атрибуты из класса QListView:
  - ListMode — 0 — элементы размещаются сверху вниз, а иконки имеют маленькие размеры;
  - IconMode — 1 — элементы размещаются слева направо, а иконки имеют большие размеры. Элементы можно свободно перемещать с помощью мыши;
- setMovement(<Режим>) — задает режим перемещения элементов. В качестве параметра указываются следующие атрибуты из класса QListView:
  - Static — 0 — пользователь не может перемещать элементы;
  - Free — 1 — свободное перемещение;
  - Snap — 2 — перемещение по сетке (размеры задаются методом setGridSize());
- setGridSize(<QSize>) — задает размеры вспомогательной сетки;
- setResizeMode(<Режим>) — задает режим положения элементов при изменении размера списка. В качестве параметра указываются следующие атрибуты из класса QListView:
  - Fixed — 0 — элементы остаются в том же положении;
  - Adjust — 1 — положение элементов изменяется при изменении размеров;

- `setFlow(<Режим>)` — задает порядок вывода элементов. В качестве параметра указываются следующие атрибуты из класса `QListView`:
  - `LeftToRight` — `0` — слева направо;
  - `TopToBottom` — `1` — сверху вниз;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `False`, то перенос элементов на новую строку (если они не помещаются в ширину области) запрещен;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, то текст элемента может быть перенесен на другую строку;
- `setLayoutMode(<Режим>)` — задает режим размещения элементов. В качестве параметра указываются следующие атрибуты из класса `QListView`:
  - `SinglePass` — `0` — элементы размещаются все сразу. Если список слишком большой, то окно будет заблокировано, пока все элементы не будут отображены;
  - `Batched` — `1` — элементы размещаются блоками. Размер блока задается с помощью метода `setBatchSize(<Количество>)`;
- `setUniformItemSizes(<Флаг>)` — если в качестве параметра указано значение `True`, то все элементы будут иметь одинаковый размер;
- `setSpacing(<Отступ>)` — задает отступ вокруг элемента;
- `setSelectionRectVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то при выделении будет отображаться вспомогательная рамка, показывающая область выделения. Метод доступен только при использовании режима множественного выделения;
- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то строка с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список.

Класс `QListView` содержит сигнал `indexesMoved(const QModelIndexList&)`, который генерируется при перемещении элементов. Внутри обработчика через параметр доступен список экземпляров класса `QModelIndex`.

24.5.3. Таблица

Класс `QTableView` реализует таблицу. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QTableView
```

Формат конструктора класса `QTableView`:

```
<Объект> = QTableView([parent=<Родитель>])
```

Класс `QTableView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 24.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel(<Модель>)` — устанавливает модель;
- `model()` — возвращает ссылку на модель;
- `horizontalHeader()` — возвращает ссылку на горизонтальный заголовок (экземпляр класса `QHeaderView`);
- `verticalHeader()` — возвращает ссылку на вертикальный заголовок (экземпляр класса `QHeaderView`). Например, вывести таблицу без заголовков можно следующим образом:

```
view.horizontalHeader().hide()
view.verticalHeader().hide()
```

- `setRowHeight(<Индекс>, <Высота>)` — задает высоту строки с указанным в первом параметре индексом;
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `rowHeight(<Индекс>)` — возвращает высоту строки;
- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `resizeRowToContents(<Индекс строки>)` — изменяет размер указанной строки таким образом, чтобы поместилось все содержимое. Метод является слотом с сигнатурой `resizeRowToContents(int)`;
- `resizeRowsToContents()` — изменяет размер всех строк таким образом, чтобы поместилось все содержимое. Метод является слотом;
- `resizeColumnToContents(<Индекс столбца>)` — изменяет размер указанного столбца таким образом, чтобы поместилось все содержимое. Метод является слотом с сигнатурой `resizeColumnToContents(int)`;
- `resizeColumnsToContents()` — изменяет размер всех столбцов таким образом, чтобы поместилось все содержимое. Метод является слотом;
- `setSpan(<Индекс строки>, <Индекс столбца>, <Количество строк>, <Количество столбцов>)` — растягивает элемент с указанными в первых двух параметрах индексами на заданное количество строк и столбцов. Происходит как бы объединение ячеек таблицы;
- `rowSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в строке, которое занимает элемент с указанными индексами;
- `columnSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в столбце, которое занимает элемент с указанными индексами;
- `clearSpans()` — отменяет все объединения ячеек;
- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то строка с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает строку;
- `hideRow(<Индекс>)` — скрывает строку с указанным индексом. Метод является слотом с сигнатурой `hideRow(int)`;
- `showRow(<Индекс>)` — отображает строку с указанным индексом. Метод является слотом с сигнатурой `showRow(int)`;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `False` отображает столбец;

- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом с сигнатурой `hideColumn(int)`;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом с сигнатурой `showColumn(int)`;
- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае;
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `isIndexHidden(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) скрыт, и `False` — в противном случае;
- `selectRow(<Индекс>)` — выделяет строку с указанным индексом. Метод является слотом с сигнатурой `selectRow(int)`;
- `selectColumn(<Индекс>)` — выделяет столбец с указанным индексом. Метод является слотом с сигнатурой `selectColumn(int)`;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список;
- `setGridStyle(<Стиль>)` — задает стиль линий сетки. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt`:
  - `NoPen` — `0` — линии не выводятся;
  - `SolidLine` — `1` — сплошная линия;
  - `DashLine` — `2` — штриховая линия;
  - `DotLine` — `3` — пунктирная линия;
  - `DashDotLine` — `4` — штрих и точка, штрих и точка и т. д.;
  - `DashDotDotLine` — `5` — штрих и две точки, штрих и две точки и т. д.;
- `setShowGrid(<Флаг>)` — если в качестве параметра указано значение `True`, то сетка будет отображена, а если `False` — то скрыта. Метод является слотом с сигнатурой `setShowGrid(bool)`;
- `setSortingEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки;
- `setCornerButtonEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то с помощью кнопки в левом верхнем углу заголовка можно выделить всю таблицу. Значение `False` отключает кнопку;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, то текст элемента может быть перенесен на другую строку;
- `sortByColumn(<Индекс столбца>[, AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном.

24.5.4. Иерархический список

Класс `QTreeView` реализует иерархический список. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QTreeView
```

Формат конструктора класса `QTreeView`:

```
<Объект> = QTreeView([parent=<Родитель>])
```

Класс `QTreeView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 24.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel(<Модель>)` — устанавливает модель;
- `model()` — возвращает ссылку на модель;
- `header()` — возвращает ссылку на горизонтальный заголовок (экземпляр класса `QHeaderView`);
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `rowHeight(<QModelIndex>)` — возвращает высоту строки, в которой находится элемент с указанным индексом (задается экземпляром класса `QModelIndex`);
- `resizeColumnToContents(<Индекс столбца>)` — изменяет ширину указанного столбца таким образом, чтобы поместилось все содержимое. Метод является слотом с сигнатурой `resizeColumnToContents(int)`;
- `setUniformRowHeights(<Флаг>)` — если в качестве параметра указано значение `True`, то все элементы будут иметь одинаковую высоту;
- `setHeaderHidden(<Флаг>)` — если в качестве параметра указано значение `True`, то заголовок будет скрыт. Значение `False` отображает заголовок;
- `isHeaderHidden()` — возвращает значение `True`, если заголовок скрыт, и `False` — в противном случае;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `False` отображает столбец;
- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом с сигнатурой `hideColumn(int)`;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом с сигнатурой `showColumn(int)`;
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `setRowHidden(<Индекс>, <QModelIndex>, <Флаг>)` — если в третьем параметре указано значение `True`, то строка с индексом `<Индекс>` и родителем `<QModelIndex>` будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с указанным индексом `<Индекс>` и родителем `<QModelIndex>` скрыта, и `False` — в противном случае;

- `isIndexHidden(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) скрыт, и `False` — в противном случае;
- `setExpanded(<QModelIndex>, <Флаг>)` — если во втором параметре указано значение `True`, то элементы, которые являются дочерними для элемента с указанным в первом параметре индексом, будут отображены, а если `False` — то скрыты. В первом параметре указывается экземпляр класса `QModelIndex`;
- `expand(<QModelIndex>)` — отображает элементы, которые являются дочерними для элемента с указанным индексом. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом с сигнатурой `expand(const QModelIndex&)`;
- `expandToDepth(<Уровень>)` — отображает все дочерние элементы до указанного уровня. Метод является слотом с сигнатурой `expandToDepth(int)`;
- `expandAll()` — отображает все дочерние элементы. Метод является слотом;
- `collapse(<QModelIndex>)` — скрывает элементы, которые являются дочерними для элемента с указанным индексом. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом с сигнатурой `collapse(const QModelIndex&)`;
- `collapseAll()` — скрывает все дочерние элементы. Метод является слотом;
- `isExpanded(<QModelIndex>)` — возвращает значение `True`, если элементы, которые являются дочерними для элемента с указанным индексом, отображены, и `False` — в противном случае. В качестве параметра указывается экземпляр класса `QModelIndex`;
- `setItemsExpandable(<Флаг>)` — если в качестве параметра указано значение `False`, то пользователь не сможет отображать или скрывать дочерние элементы;
- `setAnimated(<Флаг>)` — если в качестве параметра указано значение `True`, то отображение и сокрытие дочерних элементов будет производиться с анимацией;
- `setIndentation(<Оступ>)` — задает отступ для дочерних элементов;
- `setRootIsDecorated(<Флаг>)` — если в качестве параметра указано значение `False`, то для элементов верхнего уровня не будут показываться компоненты, с помощью которых производится отображение и сокрытие дочерних элементов;
- `setSortingEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки;
- `sortByColumn(<Индекс столбца>[, AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, то текст элемента может быть перенесен на другую строку;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список.

Класс `QTreeView` содержит следующие сигналы:

- `expanded(const QModelIndex&)` — генерируется при отображении дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса `QModelIndex`) элемента;
- `collapsed(const QModelIndex&)` — генерируется при сокрытии дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса `QModelIndex`) элемента.

24.5.5. Управление заголовками строк и столбцов

Класс `QHeaderView` реализует заголовки строк и столбцов в представлениях `QTableView` и `QTreeView`. Получить ссылки на заголовки в классе `QTableView` позволяют методы `horizontalHeader()` и `verticalHeader()`, а для установки заголовков предназначены методы `setHorizontalHeader(<QHeaderView>)` и `setVerticalHeader(<QHeaderView>)`. Получить ссылку на заголовок в классе `QTreeView` позволяет метод `header()`, а для установки заголовка предназначен метод `setHeader(<QHeaderView>)`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                    QAbstractItemView — QHeaderView
```

Формат конструктора класса `QHeaderView`:

```
<Объект> = QHeaderView(<Ориентация>[, parent=<Родитель>])
```

Параметр `<Ориентация>` позволяет задать ориентацию заголовка. В качестве значения указываются атрибуты `Horizontal` или `Vertical` из класса `QtCore.Qt`.

Класс `QHeaderView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 24.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `count()` — возвращает количество секций в заголовке. Получить количество секций можно также с помощью функции `len()`;
- `setDefaultSectionSize(<Размер>)` — задает размер секций по умолчанию;
- `defaultSectionSize()` — возвращает размер секций по умолчанию;
- `setMinimumSectionSize(<Размер>)` — задает минимальный размер секций;
- `minimumSectionSize()` — возвращает минимальный размер секций;
- `resizeSection(<Индекс>, <Размер>)` — изменяет размер секции с указанным индексом;
- `sectionSize(<Индекс>)` — возвращает размер секции с указанным индексом;
- `setResizeMode(<Режим>)` — задает режим изменения размеров для всех секций. В качестве параметра могут быть указаны следующие атрибуты из класса `QHeaderView`:
  - `Interactive` — `0` — размер может быть изменен пользователем или программно;
  - `Stretch` — `1` — секции автоматически равномерно распределяют свободное пространство между собой. Размер не может быть изменен ни пользователем, ни программно;
  - `Fixed` — `2` — размер может быть изменен только программно;
  - `ResizeToContents` — `3` — размер определяется автоматически по содержимому секции. Размер не может быть изменен ни пользователем, ни программно;
- `setResizeMode(<Индекс>, <Режим>)` — задает режим изменения размеров для секции с указанным индексом;
- `setStretchLastSection(<Флаг>)` — если в качестве параметра указано значение `True`, то последняя секция будет занимать все свободное пространство;
- `setCascadingSectionResizes(<Флаг>)` — если в качестве параметра указано значение `True`, то изменение размеров одной секции может привести к изменению размеров других секций;

- `setSectionHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то секция с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает секцию;
- `hideSection(<Индекс>)` — скрывает секцию с указанным индексом;
- `showSection(<Индекс>)` — отображает секцию с указанным индексом;
- `isSectionHidden(<Индекс>)` — возвращает значение `True`, если секция с указанным индексом скрыта, и `False` — в противном случае;
- `sectionsHidden()` — возвращает значение `True`, если существует скрытая секция, и `False` — в противном случае;
- `hiddenSectionCount()` — возвращает количество скрытых секций;
- `setDefaultAlignment(<Выравнивание>)` — задает выравнивание текста внутри заголовков;
- `setHighlightSections(<Флаг>)` — если в качестве параметра указано значение `True`, то текст заголовка текущей секции будет выделен;
- `setClickable(<Флаг>)` — если в качестве параметра указано значение `True`, то заголовок будет реагировать на щелчок мышью, при этом выделяя все элементы секции;
- `setMovable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может перемещать секции с помощью мыши;
- `isMovable()` — возвращает значение `True`, если пользователь может перемещать секции с помощью мыши, и `False` — в противном случае;
- `moveSection(<Откуда>, <Куда>)` — позволяет переместить секцию. В параметрах указываются визуальные индексы;
- `swapSections(<Секция1>, <Секция2>)` — меняет две секции местами. В параметрах указываются визуальные индексы;
- `visualIndex(<logicalIndex>)` — преобразует логический (первоначальный порядок следования) индекс в визуальный (отображаемый порядок следования) индекс. Если преобразование прошло неудачно, то возвращается значение `-1`;
- `logicalIndex(<visualIndex>)` — преобразует визуальный (отображаемый порядок следования) индекс в логический (первоначальный порядок следования) индекс. Если преобразование прошло неудачно, то возвращается значение `-1`;
- `saveState()` — возвращает экземпляр класса `QByteArray` с текущими размерами и положением секций;
- `restoreState(<QByteArray>)` — восстанавливает размеры и положение секций на основе экземпляра класса `QByteArray`, возвращаемого методом `saveState()`.

Класс `QHeaderView` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

- `sectionPressed(int)` — генерируется при нажатии левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionClicked(int)` — генерируется при нажатии и отпускании левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен логический индекс секции;

- `sectionDoubleClicked(int)` — генерируется при двойном щелчке мышью на заголовке секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionMoved(int, int, int)` — генерируется при изменении положения секции. Внутри обработчика через первый параметр доступен логический индекс секции, через второй параметр — старый визуальный индекс, а через третий — новый визуальный индекс;
- `sectionResized(int, int, int)` — генерируется непрерывно при изменении размера секции. Внутри обработчика через первый параметр доступен логический индекс секции, через второй параметр — старый размер, а через третий — новый размер.

24.6. Управление выделением элементов

Класс `QItemSelectionModel` реализует модель, позволяющую централизованно управлять выделением сразу в нескольких представлениях. Установить модель выделения позволяет метод `setSelectionModel(<QItemSelectionModel>)` из класса `QAbstractItemView`, а получить ссылку на модель можно с помощью метода `selectionModel()`. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении. Иерархия наследования выглядит так:

```
QObject — QItemSelectionModel
```

Форматы конструктора класса `QItemSelectionModel`:

```
<Объект> = QItemSelectionModel(<Модель>)
<Объект> = QItemSelectionModel(<Модель>, <Родитель>)
```

Класс `QItemSelectionModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `hasSelection()` — возвращает значение `True`, если существует выделенный элемент, и `False` — в противном случае;
- `isSelected(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) выделен, и `False` — в противном случае;
- `isRowSelected(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с индексом `<Индекс>` и родителем `<QModelIndex>` выделена, и `False` — в противном случае;
- `isColumnSelected(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если столбец с индексом `<Индекс>` и родителем `<QModelIndex>` выделен, и `False` — в противном случае;
- `rowIntersectsSelection(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с индексом `<Индекс>` и родителем `<QModelIndex>` содержит выделенный элемент, и `False` — в противном случае;
- `columnIntersectsSelection(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если столбец с индексом `<Индекс>` и родителем `<QModelIndex>` содержит выделенный элемент, и `False` — в противном случае;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список;

- `selectedRows([<Индекс столбца>=0])` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов из указанного столбца. Элемент попадет в список только в том случае, если строка выделена полностью;
- `selectedColumns([<Индекс строки>=0])` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов из указанной строки. Элемент попадет в список только в том случае, если столбец выделен полностью;
- `selection()` — возвращает ссылку на экземпляр класса `QItemSelection`;
- `select(<QModelIndex>, <Режим>)` — изменяет выделение элемента с указанным индексом. Во втором параметре указываются следующие атрибуты (или их комбинация через оператор `|`) из класса `QItemSelectionModel`:
  - `NoUpdate` — без изменений;
  - `Clear` — снимает выделение всех элементов;
  - `Select` — выделяет элемент;
  - `Deselect` — снимает выделение с элемента;
  - `Toggle` — выделяет элемент, если он не выделен, или снимает выделение, если элемент был выделен;
  - `Current` — изменяет выделение текущего элемента;
  - `Rows` — индекс будет расширен так, чтобы охватить всю строку;
  - `Columns` — индекс будет расширен так, чтобы охватить весь столбец;
  - `SelectCurrent` — комбинация `Select | Current`;
  - `ToggleCurrent` — комбинация `Toggle | Current`;
  - `ClearAndSelect` — комбинация `Clear | Select`.

  Метод является слотом с сигнатурой `select(const QModelIndex&, QItemSelectionModel::SelectionFlags)`;
- `select(<QItemSelection>, <Режим>)` — изменяет выделение элементов. Метод является слотом с сигнатурой `select(const QItemSelection&, QItemSelectionModel::SelectionFlags)`;
- `setCurrentIndex(<QModelIndex>, <Режим>)` — делает элемент текущим и изменяет режим выделения. Метод является слотом с сигнатурой `setCurrentIndex(const QModelIndex&, QItemSelectionModel::SelectionFlags)`;
- `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента;
- `clearSelection()` — снимает все выделения. Метод является слотом.

Класс `QItemSelectionModel` содержит следующие сигналы:

- `currentChanged(const QModelIndex&, const QModelIndex&)` — генерируется при изменении индекса текущего элемента. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй параметр — индекс нового элемента;
- `currentRowChanged(const QModelIndex&, const QModelIndex&)` — генерируется, когда текущий элемент перемещается в другую строку. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй параметр — индекс нового элемента;

- currentColumnChanged(const QModelIndex&, const QModelIndex&) — генерируется, когда текущий элемент перемещается в другой столбец. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй параметр — индекс нового элемента;
- selectionChanged(const QItemSelection&, const QItemSelection&) — генерируется при изменении выделения.

24.7. Промежуточные модели

Как вы уже знаете, одну модель можно установить в нескольких представлениях. При этом изменение порядка следования элементов в одном представлении повлечет за собой изменение порядка следования элементов в другом представлении. Чтобы предотвратить изменение порядка следования элементов в базовой модели, следует создать промежуточную модель с помощью класса QSortFilterProxyModel и установить ее в представлении. Иерархия наследования для класса QSortFilterProxyModel выглядит так:

```
QObject — QAbstractItemModel — QAbstractProxyModel —
                                QSortFilterProxyModel
```

Формат конструктора класса QSortFilterProxyModel:

```
<Объект> = QSortFilterProxyModel([parent=<Родитель>])
```

Класс QSortFilterProxyModel наследует следующие методы из класса QAbstractProxyModel (перечислены только основные методы; полный список смотрите в документации):

- setSourceModel(<Модель>) — устанавливает базовую модель;
- sourceModel() — возвращает ссылку на базовую модель.

Класс QSortFilterProxyModel поддерживает основные методы обычных моделей и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- sort(<Индекс столбца>[, order=AscendingOrder]) — производит сортировку. Если во втором параметре указан атрибут AscendingOrder из класса QtCore.Qt, то сортировка производится в прямом порядке, а если DescendingOrder — то в обратном. Если в параметре <Индекс столбца> указать значение -1, то будет использован порядок следования элементов из базовой модели.

  Примечание

  Чтобы включить сортировку столбцов пользователем, следует передать значение True в метод setSortingEnabled() объекта представления.

- setSortRole(<Роль>) — задает роль (см. *разд. 24.3*), по которой производится сортировка. По умолчанию сортировка производится по роли DisplayRole;
- setSortCaseSensitivity(<Режим>) — если в качестве параметра указать атрибут CaseInsensitive из класса QtCore.Qt, то при сортировке не будет учитываться регистр символов, а если CaseSensitive — то регистр будет учитываться;
- setSortLocaleAware(<Флаг>) — если в качестве параметра указать значение True, то при сортировке будут учитываться настройки локали;
- setFilterFixedString(<Фрагмент>) — в результат попадут только строки, которые содержат заданный фрагмент. Если указать пустую строку, то в результат попадут

все строки из базовой модели. Метод является слотом с сигнатурой `setFilterFixedString(const QString&);`

- `setFilterRegExp()` — производит фильтрацию элементов в соответствии с указанным регулярным выражением. Если указать пустую строку, то в результат попадут все строки из базовой модели. Форматы метода:

  ```
  setFilterRegExp(<QRegExp>)
  setFilterRegExp(<Строка с шаблоном>)
  ```

  В первом формате указывается экземпляр класса `QRegExp`, а во втором формате строка с шаблоном регулярного выражения. Второй формат метода является слотом с сигнатурой `setFilterRegExp(const QString&);`

- `setFilterWildcard(<Шаблон>)` — производит фильтрацию элементов в соответствии с указанной строкой, содержащей подстановочные знаки:

  - `?` — один любой символ;
  - `*` — нуль или более любых символов;
  - `[...]` — диапазон значений.

  Остальные символы трактуются как есть. Если в качестве параметра указать пустую строку, то в результат попадут все строки из базовой модели. Метод является слотом с сигнатурой `setFilterWildcard(const QString&);`

- `setFilterKeyColumn(<Индекс>)` — задает индекс столбца, по которому будет производиться фильтрация. Если в качестве параметра указать значение `-1`, то будут просматриваться элементы во всех столбцах. По умолчанию фильтрация производится по первому столбцу;

- `setFilterRole(<Роль>)` — задает роль (см. *разд. 24.3*), по которой производится фильтрация. По умолчанию сортировка производится по роли `DisplayRole`;

- `setFilterCaseSensitivity(<Режим>)` — если в качестве параметра указать атрибут `CaseInsensitive` из класса `QtCore.Qt`, то при фильтрации не будет учитываться регистр символов, а если `CaseSensitive` — то регистр будет учитываться;

- `setDynamicSortFilter(<Флаг>)` — если в качестве параметра указано значение `True`, то при изменении базовой модели будет производиться повторная сортировка или фильтрация.

ГЛАВА 25

Работа с графикой

Все компоненты, которые мы рассматривали в предыдущих главах, на самом деле нарисованы. Когда компонент становится видимым (в первый раз, при отображении части компонента, ранее перекрытой другим окном, или после изменения настроек) вызывается метод с названием `paintEvent()` (описание метода см. в *разд. 21.8.3*). Вызвать событие перерисовки компонента можно также искусственно с помощью методов `repaint()` и `update()` из класса `QWidget`. Внутри метода `paintEvent()` выполняется рисование компонента с помощью методов класса `QPainter`.

Класс `QPainter` содержит все необходимые средства, позволяющие выполнять рисование геометрических фигур и текста на поверхности, которая реализуется классом `QPaintDevice`. Класс `QWidget` наследует класс `QPaintDevice`. В свою очередь класс `QWidget` наследуют все компоненты, поэтому мы можем рисовать на поверхности любого компонента. Класс `QPaintDevice` наследуют также классы `QPicture`, `QPixmap`, `QImage`, `QPrinter` и некоторые другие. Класс `QPicture` позволяет сохранить команды рисования в метафайл, а затем считать их из файла и воспроизвести на какой-либо поверхности. Классы `QPixmap` и `QImage` позволяют обрабатывать изображения, а класс `QPrinter` предназначен для вывода данных на принтер или в PDF-файл. Основные методы этих классов мы рассмотрим далее в этой главе.

Библиотека PyQt позволяет также работать с SVG-графикой, анимацией, видео, а также содержит поддержку библиотеки OpenGL, предназначенной для обработки двумерной и трехмерной графики. Рассмотрение этих возможностей выходит за рамки данной книги. За подробной информацией обращайтесь к документации.

25.1. Вспомогательные классы

Прежде чем изучать работу с графикой, необходимо рассмотреть несколько вспомогательных классов, с помощью которых производится настройка различных параметров (например, цвета, характеристик шрифта, стиля пера и кисти). Кроме того, мы рассмотрим классы, описывающие геометрические фигуры (например, линию и многоугольник).

25.1.1. Класс *QColor*. Цвет

Класс `QColor` из модуля `QtGui` описывает цвет в цветовых моделях RGB, CMYK, HSV или HSL. Форматы конструктора класса `QColor`:

```
<Объект> = QColor()
<Объект> = QColor(<Красный>, <Зеленый>, <Синий>[, alpha=255])
```

```
<Объект> = QColor(<Строка>)
<Объект> = QColor(<Атрибут>)
<Объект> = QColor(<Число>)
<Объект> = QColor(<QColor>)
```

Первый конструктор создает невалидный объект. Проверить объект на валидность можно с помощью метода `isValid()`. Метод возвращает значение `True`, если объект является валидным, и `False` — в противном случае.

Второй конструктор позволяет указать целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `alpha` задает степень прозрачности цвета. Значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному. Пример указания красного цвета:

```
red = QtGui.QColor(255, 0, 0)
```

В третьем конструкторе цвет указывается в виде строки в форматах `"#RGB"`, `"#RRGGBB"`, `"#RRRGGGBBB"`, `"#RRRRGGGGBBBB"`, `"Название цвета"` или `"transparent"` (для прозрачного цвета). Пример:

```
red = QtGui.QColor("#f00")
darkBlue = QtGui.QColor("#000080")
white = QtGui.QColor("white")
```

Получить список всех поддерживаемых названий цветов позволяет статический метод `colorNames()`. Проверить правильность строки с названием цвета можно с помощью статического метода `isValidColor(<Строка>)`. Метод возвращает значение `True`, если строка является валидной, и `False` — в противном случае. Пример:

```
print(QtGui.QColor.colorNames()) # ['aliceblue', 'antiquewhite', ...]
print(QtGui.QColor.isValidColor("lightcyan")) # True
```

В четвертом конструкторе указываются следующие атрибуты из класса `QtCore.Qt`: `white`, `black`, `red`, `darkRed`, `green`, `darkGreen`, `blue`, `darkBlue`, `cyan`, `darkCyan`, `magenta`, `darkMagenta`, `yellow`, `darkYellow`, `gray`, `darkGray`, `lightGray`, `color0`, `color1` или `transparent` (прозрачный цвет). Атрибуты `color0` (прозрачный цвет) и `color1` (непрозрачный цвет) используются в двухцветных изображениях. Пример:

```
black = QtCore.Qt.black
```

В пятом конструкторе указывается целочисленное значение цвета, а шестой конструктор создает новый объект на основе указанного в параметре.

Задать или получить значения в цветовой модели RGB (Red, Green, Blue; красный, зеленый, синий) позволяют следующие методы:

- `setNamedColor(<Строка>)` — задает название цвета в виде строки в форматах `"#RGB"`, `"#RRGGBB"`, `"#RRRGGGBBB"`, `"#RRRRGGGGBBBB"`, `"Название цвета"` или `"transparent"` (для прозрачного цвета);
- `name()` — возвращает строковое представление цвета в формате `"#RRGGBB"`;
- `setRgb(<Красный>, <Зеленый>, <Синий>[, alpha=255])` — задает целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `alpha` задает степень прозрачности цвета. Значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному;

- `setRed(<Красный>)`, `setGreen(<Зеленый>)`, `setBlue(<Синий>)` и `setAlpha(<alpha>)` — задают значения отдельных составляющих цвета. В качестве параметров указываются числа от `0` до `255`;
- `setRgb(<Число>)` и `setRgba(<Число>)` — задают целочисленное значение цвета;
- `fromRgb(<Красный>, <Зеленый>, <Синий>[, alpha=255])` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются числа от `0` до `255`. Метод является статическим. Пример:

```
white = QtGui.QColor.fromRgb(255, 255, 255, 255)
```

- `fromRgb(<Число>)` и `fromRgba(<Число>)` — возвращают экземпляр класса `QColor` со значениями, соответствующими целым числам, которые указаны в параметрах. Метод является статическим. Пример:

```
white = QtGui.QColor.fromRgba(4294967295)
```

- `getRgb()` — возвращает кортеж из четырех целочисленных значений (`<Красный>`, `<Зеленый>`, `<Синий>`, `<alpha>`);
- `red()`, `green()`, `blue()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета;
- `rgb()` и `rgba()` — возвращают целочисленное значение цвета;
- `setRgbF(<Красный>, <Зеленый>, <Синий>[, alpha=1.0])` — задает значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `alpha` задает степень прозрачности цвета. Значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному;
- `setRedF(<Красный>)`, `setGreenF(<Зеленый>)`, `setBlueF(<Синий>)` и `setAlphaF(<alpha>)` — задают значения отдельных составляющих цвета. В качестве параметров указываются вещественные числа от `0.0` до `1.0`;
- `fromRgbF(<Красный>, <Зеленый>, <Синий>[, alpha=1.0])` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Метод является статическим. Пример:

```
white = QtGui.QColor.fromRgbF(1.0, 1.0, 1.0, 1.0)
```

- `getRgbF()` — возвращает кортеж из четырех вещественных значений (`<Красный>`, `<Зеленый>`, `<Синий>`, `<alpha>`);
- `redF()`, `greenF()`, `blueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета;
- `lighter([factor=150])` — если параметр имеет значение больше `100`, то возвращает новый объект с более светлым цветом, а если меньше `100` — то с более темным;
- `darker([factor=200])` — если параметр имеет значение больше `100`, то возвращает новый объект с более темным цветом, а если меньше `100` — то с более светлым.

Задать или получить значения в цветовой модели CMYK (Cyan, Magenta, Yellow, Key; голубой, пурпурный, желтый, "ключевой" (черный)) позволяют следующие методы:

- `setCmyk(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=255])` — задает целочисленные значения составляющих цвета модели CMYK. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `alpha` задает степень прозрачности

цвета. Значение 0 соответствует прозрачному цвету, а значение 255 — полностью непрозрачному;

- fromCmyk(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=255]) — возвращает экземпляр класса QColor с указанными значениями. В качестве параметров указываются числа от 0 до 255. Метод является статическим. Пример:

  ```
  white = QtGui.QColor.fromCmyk(0, 0, 0, 0, 255)
  ```

- getCmyk() — возвращает кортеж из пяти целочисленных значений (<Голубой>, <Пурпурный>, <Желтый>, <Черный>, <alpha>);

- cyan(), magenta(), yellow(), black() и alpha() — возвращают целочисленные значения отдельных составляющих цвета;

- setCmykF(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=1.0]) — задает значения составляющих цвета модели CMYK. В качестве параметров указываются вещественные числа от 0.0 до 1.0. Необязательный параметр alpha задает степень прозрачности цвета. Значение 0.0 соответствует прозрачному цвету, а значение 1.0 — полностью непрозрачному;

- fromCmykF(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=1.0]) — возвращает экземпляр класса QColor с указанными значениями. В качестве параметров указываются вещественные числа от 0.0 до 1.0. Метод является статическим. Пример:

  ```
  white = QtGui.QColor.fromCmykF(0.0, 0.0, 0.0, 0.0, 1.0)
  ```

- getCmykF() — возвращает кортеж из пяти вещественных значений (<Голубой>, <Пурпурный>, <Желтый>, <Черный>, <alpha>);

- cyanF(), magentaF(), yellowF(), blackF() и alphaF() — возвращают вещественные значения отдельных составляющих цвета.

Задать или получить значения в цветовой модели HSV (Hue, Saturation, Value; оттенок, насыщенность, значение (яркость)) позволяют следующие методы:

- setHsv(<Оттенок>, <Насыщенность>, <Значение>[, alpha=255]) — задает целочисленные значения составляющих цвета модели HSV. В первом параметре указывается число от 0 до 359, а в остальных параметрах — числа от 0 до 255;

- fromHsv(<Оттенок>, <Насыщенность>, <Значение>[, alpha=255]) — возвращает экземпляр класса QColor с указанными значениями. Метод является статическим. Пример:

  ```
  white = QtGui.QColor.fromHsv(0, 0, 255, 255)
  ```

- getHsv() — возвращает кортеж из четырех целочисленных значений (<Оттенок>, <Насыщенность>, <Значение>, <alpha>);

- hsvHue(), hsvSaturation(), value() и alpha() — возвращают целочисленные значения отдельных составляющих цвета;

- setHsvF(<Оттенок>, <Насыщенность>, <Значение>[, alpha=1.0]) — задает значения составляющих цвета модели HSV. В качестве параметров указываются вещественные числа от 0.0 до 1.0;

- fromHsvF(<Оттенок>, <Насыщенность>, <Значение>[, alpha=1.0]) — возвращает экземпляр класса QColor с указанными значениями. В качестве параметров указываются вещественные числа от 0.0 до 1.0. Метод является статическим. Пример:

  ```
  white = QtGui.QColor.fromHsvF(0.0, 0.0, 1.0, 1.0)
  ```

- `getHsvF()` — возвращает кортеж из четырех вещественных значений (`<Оттенок>`, `<Насыщенность>`, `<Значение>`, `<alpha>`);
- `hsvHueF()`, `hsvSaturationF()`, `valueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета.

Цветовая модель HSL (Hue, Saturation, Lightness) отличается от модели HSV только последней составляющей. Описание этой модели и полный перечень методов для установки и получения значений смотрите в документации.

Для получения типа используемой модели и преобразования между моделями предназначены следующие методы:

- `spec()` — позволяет узнать тип используемой модели. Возвращает значение одного из следующих атрибутов, определенных в классе `QColor`: `Invalid` (`0`), `Rgb` (`1`), `Hsv` (`2`), `Cmyk` (`3`) или `Hsl` (`4`);
- `convertTo(<Тип модели>)` — преобразует тип модели. В качестве параметра указываются атрибуты, которые перечислены в описании метода `spec()`. Метод возвращает новый объект. Пример преобразования:

```
whiteHSV = QtGui.QColor.fromHsv(0, 0, 255)
whiteRGB = whiteHSV.convertTo(QtGui.QColor.Rgb)
```

  Вместо метода `convertTo()` удобнее воспользоваться методами `toRgb()`, `toCmyk()`, `toHsv()` или `toHsl()`, которые возвращают новый объект. Пример:

```
whiteHSV = QtGui.QColor.fromHsv(0, 0, 255)
whiteRGB = whiteHSV.toRgb()
```

25.1.2. Класс *QPen*. Перо

Класс `QPen` из модуля `QtGui` описывает перо, с помощью которого производится рисование точек, линий и контуров фигур. Форматы конструктора класса:

```
<Объект> = QPen()
<Объект> = QPen(<QColor>)
<Объект> = QPen(<Стиль>)
<Объект> = QPen(<QBrush>, <Ширина>[, style=SolidLine][, cap=SquareCap][,
                join=BevelJoin])
<Объект> = QPen(<QPen>)
```

Первый конструктор создает перо черного цвета с настройками по умолчанию. Второй конструктор задает только цвет пера с помощью экземпляра класса `QColor`. Третий конструктор позволяет указать стиль линии. В качестве значения указываются следующие атрибуты из класса `QtCore.Qt`:

- `NoPen` — `0` — линия не выводится;
- `SolidLine` — `1` — сплошная линия;
- `DashLine` — `2` — штриховая линия;
- `DotLine` — `3` — пунктирная линия;
- `DashDotLine` — `4` — штрих и точка, штрих и точка и т. д.;
- `DashDotDotLine` — `5` — штрих и две точки, штрих и две точки и т. д.
- `CustomDashLine` — `6` — пользовательский стиль.

Четвертый конструктор позволяет задать все характеристики пера за один раз. В первом параметре указывается экземпляр класса `QBrush` или класса `QColor`. Ширина линии передается во втором параметре, а стиль линии в необязательном параметре `style`. Необязательный параметр `cap` задает стиль концов линии. В качестве значения указываются следующие атрибуты из класса `QtCore.Qt`:

- `FlatCap` — `0` — квадратный конец линии. Длина линии не превышает указанных граничных точек;
- `SquareCap` — `16` — квадратный конец линии. Длина линии увеличивается с обоих концов на половину ширины линии;
- `RoundCap` — `32` — скругленные концы. Длина линии увеличивается с обоих концов на половину ширины линии.

Необязательный параметр `join` задает стиль перехода одной линии в другую. В качестве значения указываются следующие атрибуты из класса `QtCore.Qt`:

- `MiterJoin` — `0` — линии соединяются под острым углом;
- `BevelJoin` — `64` — пространство между концами линий заполняется цветом линии;
- `RoundJoin` — `128` — скругленные углы;
- `SvgMiterJoin` — `256`.

Класс `QPen` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setColor(<QColor>)` — задает цвет линии;
- `setBrush(<QBrush>)` — задает кисть;
- `setWidth(<int>)` и `setWidthF(<float>)` — задают ширину линии;
- `setStyle(<Стиль>)` — задает стиль линии (см. значения параметра `style` в четвертом формате конструктора класса `QPen`);
- `setCapStyle(<Стиль>)` — задает стиль концов линии (см. значения параметра `cap` в четвертом формате конструктора класса `QPen`);
- `setJoinStyle(<Стиль>)` — задает стиль перехода одной линии в другую (см. значения параметра `join` в четвертом формате конструктора класса `QPen`).

25.1.3. Класс *QBrush*. Кисть

Класс `QBrush` из модуля `QtGui` описывает кисть, с помощью которой производится заполнение фона фигур. Форматы конструктора класса:

```
<Объект> = QBrush()
<Объект> = QBrush(<QColor>[, style=SolidPattern])
<Объект> = QBrush(<Стиль кисти>)
<Объект> = QBrush(<QGradient>)
<Объект> = QBrush(<QColor>, <QPixmap>)
<Объект> = QBrush(<QPixmap>)
<Объект> = QBrush(<QImage>)
<Объект> = QBrush(<QBrush>)
```

Параметр <QColor> задает цвет кисти. Можно передать экземпляр класса QColor или атрибут из класса QtCore.Qt, например, black.

В параметрах <Стиль кисти> и style указываются атрибуты из класса QtCore.Qt, задающие стиль кисти, например, NoBrush, SolidPattern, Dense1Pattern, Dense2Pattern, Dense3Pattern, Dense4Pattern, Dense5Pattern, Dense6Pattern, Dense7Pattern, CrossPattern и др. С помощью этого параметра можно сделать цвет сплошным (SolidPattern) или имеющим текстуру (например, атрибут CrossPattern задает текстуру в виде сетки).

Параметр <QGradient> позволяет установить градиентную заливку. В качестве значения указываются экземпляры классов QLinearGradient (линейный градиент), QConicalGradient (конический градиент) или QRadialGradient (радиальный градиент). За подробной информацией по этим классам обращайтесь к документации.

Параметры <QPixmap> и <QImage> предназначены для установки изображения в качестве текстуры.

Класс QBrush содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setColor(<QColor>) — задает цвет кисти;
- setStyle(<Стиль>) — задает стиль кисти (см. значения параметра style в конструкторе класса QBrush);
- setTexture(<QPixmap>) — устанавливает растровое изображение. В качестве параметра можно указать экземпляр классов QPixmap или QBitmap;
- setTextureImage(<QImage>) — устанавливает изображение.

25.1.4. Класс *QLine*. Линия

Класс QLine из модуля QtCore описывает координаты линии. Форматы конструктора класса:

```
<Объект> = QLine()
<Объект> = QLine(<QPoint>, <QPoint>)
<Объект> = QLine(<X1>, <Y1>, <X2>, <Y2>)
<Объект> = QLine(<QLine>)
```

Первый конструктор создает нулевой объект. Во втором и третьем конструкторах указываются координаты начальной и конечной точек в виде экземпляров класса QPoint или значений через запятую. Четвертый конструктор создает новый объект на основе другого объекта.

> ***Примечание***
>
> Класс QLine предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс QLineF.

Класс QLine содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- isNull() — возвращает значение True, если начальная или конечная точка не установлена, и False — в противном случае;
- setPoints(<QPoint>, <QPoint>) — задает координаты начальной и конечной точек в виде экземпляров класса QPoint;

- `setLine(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты начальной и конечной точек в виде значений через запятую;
- `setP1(<QPoint>)` — задает координаты начальной точки;
- `setP2(<QPoint>)` — задает координаты конечной точки;
- `p1()` — возвращает координаты (экземпляр класса `QPoint`) начальной точки;
- `p2()` — возвращает координаты (экземпляр класса `QPoint`) конечной точки;
- `x1()`, `y1()`, `x2()` и `y2()` — возвращают значения отдельных составляющих координат начальной и конечной точек в виде целых чисел;
- `dx()` — возвращает горизонтальную составляющую вектора линии;
- `dy()` — возвращает вертикальную составляющую вектора линии.

25.1.5. Класс *QPolygon*. Многоугольник

Класс `QPolygon` из модуля `QtGui` описывает координаты вершин многоугольника. Форматы конструктора класса:

```
<Объект> = QPolygon()
<Объект> = QPolygon(<Список с экземплярами класса QPoint>)
<Объект> = QPolygon(<Список с координатами>)
<Объект> = QPolygon(<QRect>[, closed=False])
<Объект> = QPolygon(<Количество вершин>)
<Объект> = QPolygon(<QPolygon>)
```

Первый конструктор создает пустой объект. Заполнить объект координатами вершин можно с помощью оператора <<. Пример добавления координат вершин треугольника:

```
polygon = QtGui.QPolygon()
polygon << QtCore.QPoint(20, 50) << QtCore.QPoint(280, 50)
polygon << QtCore.QPoint(150, 280)
```

Во втором конструкторе указывается список с экземплярами класса `QPoint`, которые задают координаты отдельных вершин. Пример:

```
polygon = QtGui.QPolygon([QtCore.QPoint(20, 50), QtCore.QPoint(280, 50),
                          QtCore.QPoint(150, 280)])
```

В третьем конструкторе указывается список с координатами вершин через запятую:

```
polygon = QtGui.QPolygon([20,50, 280,50, 150,280])
```

Четвертый конструктор создает многоугольник на основе экземпляра класса `QRect`. Если параметр `closed` имеет значение `False`, то будут созданы четыре вершины, а если значение `True` — то пять вершин.

В пятом конструкторе можно указать количество вершин, а затем задать координаты путем присваивания значения по индексу:

```
polygon = QtGui.QPolygon(3)
polygon[0] = QtCore.QPoint(20, 50)
polygon[1] = QtCore.QPoint(280, 50)
polygon[2] = QtCore.QPoint(150, 280)
```

Шестой конструктор создает новый объект на основе другого объекта.

Примечание

Класс `QPolygon` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QPolygonF`.

Класс `QPolygon` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPoints()` — устанавливает координаты вершин. Ранее установленные значения удаляются. Форматы метода:

```
setPoints(<Список с координатами>)
setPoints(<X1>, <Y1>[, ..., <Xn>, <Yn>])
```

  Пример указания значений:

```
polygon = QtGui.QPolygon()
polygon.setPoints([20,50, 280,50, 150,280])
```

- `prepend(<QPoint>)` — добавляет новую вершину в начало объекта;
- `append(<QPoint>)` — добавляет новую вершину в конец объекта. Добавить вершину можно также с помощью операторов << и +=;
- `insert(<Индекс>, <QPoint>)` — добавляет новую вершину в указанную позицию;
- `setPoint()` — задает координаты для вершины с указанным индексом. Форматы метода:

```
setPoint(<Индекс>, <QPoint>)
setPoint(<Индекс>, <X>, <Y>)
```

  Можно также задать координаты путем присваивания значения по индексу:

```
polygon = QtGui.QPolygon(3)
polygon.setPoint(0, QtCore.QPoint(20, 50))
polygon.setPoint(1, 280, 50)
polygon[2] = QtCore.QPoint(150, 280)
```

- `point(<Индекс>)` — возвращает экземпляр класса `QPoint` с координатами вершины, индекс которой указан в параметре. Получить значение можно также с помощью операции доступа по индексу. Пример:

```
polygon = QtGui.QPolygon([20,50, 280,50, 150,280])
print(polygon.point(0)) # PyQt4.QtCore.QPoint(20, 50)
print(polygon[1])       # PyQt4.QtCore.QPoint(280, 50)
```

- `remove(<Индекс>[, <Количество>])` — удаляет указанное количество вершин, начиная с индекса `<Индекс>`. Если второй параметр не указан, то удаляется только одна вершина. Удалить вершину можно также с помощью оператора `del` по индексу или срезу;
- `clear()` — удаляет все вершины;
- `size()` и `count([<QPoint>])` — возвращают количество вершин. Если в методе `count()` указан параметр, то возвращается только количество вершин с указанными координатами. Получить количество вершин можно также с помощью функции `len()`;
- `isEmpty()` — возвращает значение `True`, если объект пустой, и `False` — в противном случае;
- `boundingRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которую вписан многоугольник.

25.1.6. Класс *QFont*. Шрифт

Класс `QFont` из модуля `QtGui` описывает характеристики шрифта. Форматы конструктора класса:

```
<Объект> = QFont()
<Объект> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
                                  [, italic=False])
<Объект> = QFont(<QFont>)
```

Первый конструктор создает объект шрифта с настройками, используемыми приложением по умолчанию. Установить шрифт приложения по умолчанию позволяет статический метод `setFont()` из класса `QApplication`.

Второй конструктор позволяет указать основные характеристики шрифта. В первом параметре указывается название шрифта или семейства в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light` (`25`), `Normal` (`50`), `DemiBold` (`63`), `Bold` (`75`) или `Black` (`87`) из класса `QFont`. Если в параметре `italic` указано значение `True`, то шрифт будет курсивным.

Третий конструктор создает новый объект на основе другого объекта.

Класс `QFont` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setFamily(<Название шрифта>)` — задает название шрифта или семейства;
- `family()` — возвращает название шрифта;
- `setPointSize(<int>)` и `setPointSizeF(<float>)` — задают размер шрифта в пунктах;
- `pointSize()` — возвращает размер шрифта в пунктах в виде целого числа или значение `-1`, если размер шрифта был установлен в пикселах;
- `pointSizeF()` — возвращает размер шрифта в пунктах в виде вещественного числа или значение `-1`, если размер шрифта был установлен в пикселах;
- `setPixelSize(<int>)` — задает размер шрифта в пикселах;
- `pixelSize()` — возвращает размер шрифта в пикселах или значение `-1`, если размер шрифта был установлен в пунктах;
- `setWeight(<Жирность>)` — задает степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light` (`25`), `Normal` (`50`), `DemiBold` (`63`), `Bold` (`75`) или `Black` (`87`) из класса `QFont`;
- `weight()` — возвращает степень жирности шрифта;
- `setBold(<Флаг>)` — если в качестве параметра указано значение `True`, то жирность шрифта устанавливается равной значению атрибута `Bold`, а если `False` — то равной значению атрибута `Normal`;
- `bold()` — возвращает значение `True`, если степень жирности шрифта больше значения атрибута `Normal`, и `False` — в противном случае;
- `setItalic(<Флаг>)` — если в качестве параметра указано значение `True`, то шрифт будет курсивным, а если `False` — то нормальным;
- `italic()` — возвращает значение `True`, если шрифт курсивный, и `False` — в противном случае;

- setUnderline(<Флаг>) — если в качестве параметра указано значение True, то текст будет подчеркнутым, а если False — то не подчеркнутым;
- underline() — возвращает значение True, если текст будет подчеркнут, и False — в противном случае;
- setOverline(<Флаг>) — если в качестве параметра указано значение True, то над текстом будет выводиться черта;
- overline() — возвращает значение True, если над текстом будет выводиться черта, и False — в противном случае;
- setStrikeOut(<Флаг>) — если в качестве параметра указано значение True, то текст будет перечеркнутым;
- strikeOut() — возвращает значение True, если текст будет перечеркнутым, и False — в противном случае.

Получить список всех доступных в системе шрифтов позволяет метод families() из класса QFontDatabase. Метод возвращает список строк. Пример:

```
print(QtGui.QFontDatabase().families())
```

Чтобы получить список доступных стилей для указанного шрифта, следует воспользоваться методом styles(<Название шрифта>) из класса QFontDatabase:

```
print(QtGui.QFontDatabase().styles("Tahoma"))
# ['Normal', 'Italic', 'Bold', 'Bold Italic']
```

Получить допустимые размеры для указанного стиля можно с помощью метода smoothSizes(<Название шрифта>, <Стиль>) из класса QFontDatabase:

```
print(QtGui.QFontDatabase().smoothSizes("Tahoma", "Normal"))
# [6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72]
```

Очень часто необходимо произвести выравнивание выводимого текста внутри некоторой области. Чтобы это сделать, нужно знать размеры области, в которую вписан текст. Получить эти значения позволяют следующие методы из класса QFontMetrics:

- width(<Текст>[, length=-1]) — возвращает расстояние от начала текста <Текст> до позиции, в которой должен начаться другой текст. Параметр length позволяет ограничить количество символов;
- height() — возвращает высоту шрифта;
- boundingRect(<Текст>) — возвращает экземпляр класса QRect с координатами и размерами прямоугольной области, в которую вписан текст.

Пример получения размеров области:

```
font = QtGui.QFont("Tahoma", 16)
fm = QtGui.QFontMetrics(font)
print(fm.width("Строка"))         # 67
print(fm.height())                # 25
print(fm.boundingRect("Строка")) # PyQt4.QtCore.QRect(0, -21, 65, 25)
```

Обратите внимание на то, что значения, возвращаемые методами width() и QRect.width(), отличаются.

> ***Примечание***
>
> Класс QFontMetrics предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс QFontMetricsF.

25.2. Класс *QPainter*

Класс `QPainter` содержит все необходимые средства, позволяющие выполнять рисование геометрических фигур и текста на поверхности, которая реализуется классом `QPaintDevice`. Класс `QPaintDevice` наследуют классы `QWidget`, `QPicture`, `QPixmap`, `QImage`, `QPrinter` и некоторые другие. Таким образом, мы можем рисовать как на поверхности любого компонента, так и на изображении. Форматы конструктора класса:

```
<Объект> = QPainter()
<Объект> = QPainter(<QPaintDevice>)
```

Первый конструктор создает объект, который не подключен ни к одному устройству. Чтобы подключиться к устройству и захватить контекст рисования, необходимо вызвать метод `begin(<QPaintDevice>)` и передать ему ссылку на экземпляр класса, являющегося наследником класса `QPaintDevice`. Метод возвращает значение `True`, если контекст успешно захвачен, и `False` — в противном случае. В один момент времени только один объект может рисовать на устройстве, поэтому после окончания рисования необходимо освободить контекст рисования с помощью метода `end()`. С учетом сказанного код, позволяющий рисовать на компоненте, будет выглядеть так:

```
def paintEvent(self, e):
    painter = QtGui.QPainter()
    painter.begin(self)
    # Здесь производим рисование на компоненте
    painter.end()
```

Второй конструктор принимает ссылку на экземпляр класса, являющегося наследником класса `QPaintDevice`, подключается к этому устройству и сразу захватывает контекст рисования. Контекст рисования автоматически освобождается внутри деструктора класса `QPainter` при уничтожении объекта. Так как объект автоматически уничтожается при выходе из метода `paintEvent()`, то метод `end()` можно и не вызывать. Пример рисования на компоненте:

```
def paintEvent(self, e):
    painter = QtGui.QPainter(self)
    # Здесь производим рисование на компоненте
```

Проверить успешность захвата контекста рисования можно с помощью метода `isActive()`. Метод возвращает значение `True`, если контекст захвачен, и `False` — в противном случае.

25.2.1. Рисование линий и фигур

После захвата контекста рисования следует установить перо и кисть. С помощью пера производится рисование точек, линий и контуров фигур, а с помощью кисти — заполнение фона фигур. Установить перо позволяет метод `setPen()` из класса `QPainter`. Форматы метода:

```
setPen(<QPen>)
setPen(<QColor>)
setPen(<Стиль пера>)
```

Для установки кисти предназначен метод `setBrush()`. Форматы метода:

```
setBrush(<QBrush>)
setBrush(<Стиль кисти>)
```

Устанавливать перо или кисть нужно будет перед каждой операцией рисования, требующей изменения цвета или стиля. Если перо или кисть не установлены, то будут использоваться объекты с настройками по умолчанию. После установки пера и кисти можно приступать к рисованию точек, линий, фигур, текста и др.

Для рисования точек, линий и фигур класс `QPainter` предоставляет следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QPainter`):

- `drawPoint()` — рисует точку. Форматы метода:

```
drawPoint(<X>, <Y>)
drawPoint(<QPoint>)
drawPoint(<QPointF>)
```

- `drawPoints()` — рисует несколько точек. Форматы метода:

```
drawPoints(<QPoint 1>[, ..., <QPoint N>])
drawPoints(<QPointF 1>[, ..., <QPointF N>])
drawPoints(<QPolygon>)
drawPoints(<QPolygonF>)
```

- `drawLine()` — рисует линию. Форматы метода:

```
drawLine(<QLine>)
drawLine(<QLineF>)
drawLine(<QPoint>, <QPoint>)
drawLine(<QPointF>, <QPointF>)
drawLine(<X1>, <Y1>, <X2>, <Y2>)
```

- `drawLines()` — рисует несколько отдельных линий. Форматы метода:

```
drawLines(<QLine 1>[, ..., <QLine N>])
drawLines(<QLineF 1>[, ..., <QLineF N>])
drawLines(<Список с экземплярами класса QLine>)
drawLines(<Список с экземплярами класса QLineF>)
drawLines(<QPoint 1>[, ..., <QPoint N>])
drawLines(<QPointF 1>[, ..., <QPointF N>])
drawLines(<Список с экземплярами класса QPoint>)
drawLines(<Список с экземплярами класса QPointF>)
```

- `drawPolyline()` — рисует несколько линий, которые соединяют указанные точки. Первая и последняя точки не соединяются. Форматы метода:

```
drawPolyline(<QPoint 1>[, ..., <QPoint N>])
drawPolyline(<QPointF 1>[, ..., <QPointF N>])
drawPolyline(<QPolygon>)
drawPolyline(<QPolygonF>)
```

- `drawRect()` — рисует прямоугольник с границей и заливкой. Чтобы убрать границу, следует использовать перо со стилем `NoPen`, а чтобы убрать заливку — кисть со стилем `NoBrush`. Форматы метода:

```
drawRect(<X>, <Y>, <Ширина>, <Высота>)
drawRect(<QRect>)
drawRect(<QRectF>)
```

- `fillRect()` — рисует прямоугольник с заливкой без границы. Форматы метода:

```
fillRect(<X>, <Y>, <Ширина>, <Высота>, <QBrush или QColor>)
fillRect(<QRect>, <QBrush или QColor>)
fillRect(<QRectF>, <QBrush или QColor>)
```

- `drawRoundedRect()` — рисует прямоугольник с границей, заливкой и скругленными краями. Форматы метода:

```
drawRoundRect(<X>, <Y>, <Ширина>, <Высота>
              [, xRound=25][, yRound=25])
drawRoundRect(<QRect>[, xRound=25][, yRound=25])
drawRoundRect(<QRectF>[, xRound=25][, yRound=25])
```

- `drawPolygon()` — рисует многоугольник с границей и заливкой. Форматы метода:

```
drawPolygon(<QPoint 1>[, ..., <QPoint N>])
drawPolygon(<QPointF 1>[, ..., <QPointF N>])
drawPolygon(<QPolygon>[, fillRule=OddEvenFill])
drawPolygon(<QPolygonF>[, fillRule=OddEvenFill])
```

- `drawEllipse()` — рисует эллипс с границей и заливкой. Форматы метода:

```
drawEllipse(<X>, <Y>, <Ширина>, <Высота>)
drawEllipse(<QRect>)
drawEllipse(<QRectF>)
drawEllipse(<QPoint>, <int rX>, <int rY>)
drawEllipse(<QPointF>, <float rX>, <float rY>)
```

  В первых трех форматах указываются координаты и размеры прямоугольника, в который необходимо вписать эллипс. В двух последних форматах первый параметр задает координаты центра, параметр `rX` — радиус по оси `X`, а параметр `rY` — радиус по оси `Y`;

- `drawArc()` — рисует дугу. Форматы метода:

```
drawArc(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>,
        <Угол>)
drawArc(<QRect>, <Начальный угол>, <Угол>)
drawArc(<QRectF>, <Начальный угол>, <Угол>)
```

  Следует учитывать, что значения углов задаются в значениях `1/16` градуса. Полный круг эквивалентен значению `16 * 360`. Нулевой угол находится в позиции трех часов. Положительные значения углов отсчитываются против часовой стрелки, а отрицательные по часовой стрелке;

- `drawChord()` — рисует замкнутую дугу. Метод `drawChord()` аналогичен методу `drawArc()`, но замыкает крайние точки дуги прямой линией. Форматы метода:

```
drawChord(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>,
          <Угол>)
drawChord(<QRect>, <Начальный угол>, <Угол>)
drawChord(<QRectF>, <Начальный угол>, <Угол>)
```

- `drawPie()` — рисует замкнутый сектор. Метод `drawPie()` аналогичен методу `drawArc()`, но замыкает крайние точки дуги с центром окружности. Форматы метода:

```
drawPie(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>,
        <Угол>)
```

```
drawPie(<QRect>, <Начальный угол>, <Угол>)
drawPie(<QRectF>, <Начальный угол>, <Угол>)
```

При выводе некоторых фигур (например, эллипса) контур может отображаться в виде "лесенки". Чтобы сгладить контуры фигур, следует вызвать метод `setRenderHint()` и передать ему атрибут `Antialiasing`. Пример:

```
painter.setRenderHint(QtGui.QPainter.Antialiasing)
```

25.2.2. Вывод текста

Вывести текст позволяет метод `drawText()` из класса `QPainter`. Форматы метода:

```
drawText(<X>, <Y>, <Текст>)
drawText(<QPoint>, <Текст>)
drawText(<QPointF>, <Текст>)
drawText(<X>, <Y>, <Ширина>, <Высота>, <Флаги>, <Текст>)
drawText(<QRect>, <Флаги>, <Текст>)
drawText(<QRectF>, <Флаги>, <Текст>)
drawText(<QRectF>, <Текст>[, option=QTextOption()])
```

Первые три формата метода выводят текст, начиная с указанных координат.

Следующие три формата выводят текст в указанную прямоугольную область. Если текст не помещается в прямоугольную область, то он будет обрезан, если не указан флаг `TextDontClip`. Методы возвращают экземпляр класса `QRect` (`QRectF` для шестого формата) с координатами и размерами прямоугольника, в который вписан текст. В параметре `<Флаги>` через оператор `|` указываются атрибуты `AlignLeft`, `AlignRight`, `AlignHCenter`, `AlignTop`, `AlignBottom`, `AlignVCenter` или `AlignCenter` из класса `QtCore.Qt`, задающие выравнивание текста внутри прямоугольной области, а также следующие атрибуты:

- `TextDontClip` — текст может выйти за пределы указанной прямоугольной области;
- `TextSingleLine` — все специальные символы трактуются как пробелы и текст выводится в одну строку;
- `TextWordWrap` — если текст не помещается на одной строке, то будет произведен перенос по границам слова;
- `TextWrapAnywhere` — перенос строки может быть внутри слова;
- `TextShowMnemonic` — символ, перед которым указан символ `&`, будет подчеркнут. Чтобы вывести символ `&`, его необходимо удвоить;
- `TextHideMnemonic` — то же самое, что и `TextShowMnemonic`, но символ не подчеркивается;
- `TextExpandTabs` — символы табуляции будут обрабатываться.

Седьмой формат метода `drawText()` также выводит текст в указанную прямоугольную область, но выравнивание текста и другие опции задаются с помощью экземпляра класса `QTextOption`. Например, с помощью этого класса можно отобразить непечатаемые символы (символ пробела, табуляцию и др.).

Получить координаты и размеры прямоугольника, в который вписывается текст, позволяет метод `boundingRect()`. Форматы метода:

```
boundingRect(<X>, <Y>, <Ширина>, <Высота>, <Флаги>, <Текст>)
boundingRect(<QRect>, <Флаги>, <Текст>)
```

```
boundingRect(<QRectF>, <Флаги>, <Текст>)
boundingRect(<QRectF>, <Текст>[, option=QTextOption()])
```

Первые два формата метода `boundingRect()` возвращают экземпляр класса `QRect`, а последние два — экземпляр класса `QRectF`.

При выводе текста линии букв могут отображаться в виде "лесенки". Чтобы сгладить контуры, следует вызвать метод `setRenderHint()` и передать ему атрибут `TextAntialiasing`. Пример:

```
painter.setRenderHint(QtGui.QPainter.TextAntialiasing)
```

25.2.3. Вывод изображения

Для вывода растровых изображений предназначены методы `drawPixmap()` и `drawImage()` из класса `QPainter`. Метод `drawPixmap()` предназначен для вывода изображений, хранимых в экземпляре класса `QPixmap`. Форматы метода `drawPixmap()`:

```
drawPixmap(<X>, <Y>, <QPixmap>)
drawPixmap(<QPoint>, <QPixmap>)
drawPixmap(<QPointF>, <QPixmap>)
drawPixmap(<X>, <Y>, <Ширина>, <Высота>, <QPixmap>)
drawPixmap(<QRect>, <QPixmap>)
drawPixmap(<X1>, <Y1>, <QPixmap>, <X2>, <Y2>, <Ширина2>, <Высота2>)
drawPixmap(<QPoint>, <QPixmap>, <QRect>)
drawPixmap(<QPointF>, <QPixmap>, <QRectF>)
drawPixmap(<X1>, <Y1>, <Ширина1>, <Высота1>, <QPixmap>,
           <X2>, <Y2>, <Ширина2>, <Высота2>)
drawPixmap(<QRect>, <QPixmap>, <QRect>)
drawPixmap(<QRectF>, <QPixmap>, <QRectF>)
```

Первые три формата задают координаты, в которые будет установлен левый верхний угол изображения, и экземпляр класса `QPixmap`. Пример:

```
pixmap = QtGui.QPixmap("foto.jpg")
painter.drawPixmap(3, 3, pixmap)
```

Четвертый и пятый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Шестой, седьмой и восьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QPixmap`, в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`.

Последние три формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QPixmap`, в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Метод drawImage() предназначен для вывода изображений, хранимых в экземпляре класса QImage. При выводе экземпляр класса QImage преобразуется в экземпляр класса QPixmap. Тип преобразования задается с помощью необязательного параметра flags. Форматы метода drawImage():

```
drawImage(<QPoint>, <QImage>)
drawImage(<QPointF>, <QImage>)
drawImage(<QRect>, <QImage>)
drawImage(<QRectF>, <QImage>)
drawImage(<X1>, <Y1>, <QImage>[, sx=0][, sy=0][, sw=-1][, sh=-1]
          [, flags=AutoColor])
drawImage(<QPoint>, <QImage>, <QRect>[, flags=AutoColor])
drawImage(<QPointF>, <QImage>, <QRectF>[, flags=AutoColor])
drawImage(<QRect>, <QImage>, <QRect>[, flags=AutoColor])
drawImage(<QRectF>, <QImage>, <QRectF>[, flags=AutoColor])
```

Первые два формата (а также пятый формат со значениями по умолчанию) задают координаты, в которые будет установлен левый верхний угол изображения, и экземпляр класса QImage. Пример:

```
img = QtGui.QImage("foto.jpg")
painter.drawImage(3, 3, img)
```

Третий и четвертый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Пятый, шестой и седьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса QImage, в виде отдельных составляющих или экземпляров классов QRect или QRectF.

Последние два формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса QImage, в виде экземпляров классов QRect или QRectF. Если размеры области не совпадают с размерами фрагмента изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

25.2.4. Преобразование систем координат

Существуют две системы координат: физическая (viewport; система координат устройства) и логическая (window). При рисовании координаты из логической системы координат преобразуются в систему координат устройства. По умолчанию эти две системы координат совпадают. В некоторых случаях возникает необходимость изменить координаты. Выполнить изменение физической системы координат позволяет метод setViewport(), а получить текущие значения можно с помощью метода viewport(). Выполнить изменение логической системы координат позволяет метод setWindow(), а получить текущие значения можно с помощью метода window().

Произвести дополнительную трансформацию системы координат позволяют следующие методы из класса QPainter:

- `translate()` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз. Форматы метода:

```
translate(<X>, <Y>)
translate(<QPoint>)
translate(<QPointF>)
```

- `rotate(<Угол>)` — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;
- `scale(<По оси X>, <По оси Y>)` — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- `shear(<По горизонтали>, <По вертикали>)` — сдвигает систему координат. В качестве значений указываются вещественные числа.

Все указанные трансформации влияют на последующие операции рисования и не изменяют ранее нарисованные фигуры. Чтобы после трансформации восстановить систему координат, следует предварительно сохранить состояние в стеке с помощью метода `save()`, а после окончания рисования вызвать метод `restore()`:

```
painter.save()      # Сохраняем состояние
                    # Трансформируем и рисуем
painter.restore()   # Восстанавливаем состояние
```

Несколько трансформаций можно произвести последовательно друг за другом. В этом случае следует учитывать, что порядок следования трансформаций имеет значение. Если одна и та же последовательность выполняется несколько раз, то ее можно сохранить в экземпляре класса `QMatrix`, а затем установить с помощью метода `setMatrix()`. Пример:

```
matrix = QtGui.QMatrix()
matrix.translate(105, 105)
matrix.rotate(45.0)
painter.setMatrix(matrix)
painter.fillRect(-25, -25, 50, 50, QtCore.Qt.green)
```

25.2.5. Сохранение команд рисования в файл

Класс `QPicture` исполняет роль устройства для рисования с возможностью сохранения команд рисования в файл специального формата и последующего восстановления команд. Иерархия наследования:

```
QPaintDevice — QPicture
```

Форматы конструктора класса:

```
<Объект> = QPicture([formatVersion=-1])
<Объект> = QPicture(<QPicture>)
```

Первый конструктор создает пустой рисунок. Необязательный параметр `formatVersion` задает версию формата. Если параметр не указан, то используется формат, принятый в текущей версии PyQt. Второй конструктор создает копию рисунка.

Для сохранения и загрузки рисунка предназначены следующие методы:

- `save(<Название файла>)` — сохраняет рисунок в файл. Метод возвращает значение `True`, если рисунок успешно сохранен, и `False` — в противном случае;
- `load(<Название файла>)` — загружает рисунок из файла. Метод возвращает значение `True`, если рисунок успешно загружен, и `False` — в противном случае.

Для вывода загруженного рисунка на устройство рисования предназначен метод `drawPicture()` из класса `QPainter`. Форматы метода:

```
drawPicture(<X>, <Y>, <QPicture>)
drawPicture(<QPoint>, <QPicture>)
drawPicture(<QPointF>, <QPicture>)
```

Пример сохранения рисунка:

```
painter = QtGui.QPainter()
pic = QtGui.QPicture()
painter.begin(pic)
# Здесь что-то рисуем
painter.end()
pic.save("pic.dat")
```

Пример вывода загруженного рисунка на поверхность компонента:

```
def paintEvent(self, e):
    painter = QtGui.QPainter(self)
    pic = QtGui.QPicture()
    pic.load("pic.dat")
    painter.drawPicture(0, 0, pic)
```

25.3. Работа с изображениями

Библиотека PyQt содержит несколько классов, позволяющих работать с растровыми изображениями в контекстно-зависимом (классы `QPixmap` и `QBitmap`) и контекстно-независимом (класс `QImage`) представлениях. Получить список форматов, которые можно загрузить, позволяет статический метод `supportedImageFormats()` из класса `QImageReader`. Метод возвращает список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов для чтения:

```
for i in QtGui.QImageReader.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Результат выполнения:

```
BMP GIF ICO JPEG JPG MNG PBM PGM PNG PPM SVG SVGZ TIF TIFF XBM XPM
```

Получить список форматов, в которых можно сохранить изображение, позволяет статический метод `supportedImageFormats()` из класса `QImageWriter`. Метод возвращает список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов для записи:

```
for i in QtGui.QImageWriter.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Результат выполнения:

```
BMP ICO JPEG JPG PNG PPM TIF TIFF XBM XPM
```

Обратите внимание на то, что мы можем загрузить изображение в формате GIF, но не имеем возможности сохранить изображение в этом формате, т. к. алгоритм сжатия, используемый в этом формате, защищен патентом.

25.3.1. Класс *QPixmap*

Класс QPixmap предназначен для работы с изображениями в контекстно-зависимом представлении. Данные хранятся в виде, позволяющем отображать изображение на экране наиболее эффективным способом, поэтому класс часто используется в качестве буфера для рисования перед выводом на экран. Иерархия наследования:

```
QPaintDevice — QPixmap
```

Так как класс QPixmap наследует класс QPaintDevice, мы можем использовать его как поверхность для рисования. Вывести изображение позволяет метод drawPixmap() из класса QPainter (см. *разд. 25.2.3*).

Форматы конструктора класса:

```
<Объект> = QPixmap()
<Объект> = QPixmap(<Ширина>, <Высота>)
<Объект> = QPixmap(<QSize>)
<Объект> = QPixmap(<Путь к файлу>[, format=None][, flags=AutoColor])
<Объект> = QPixmap(<QPixmap>)
```

Первый конструктор создает нулевой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения. Если размеры равны нулю, то будет создан нулевой объект изображения. Четвертый конструктор предназначен для загрузки изображения из файла, а пятый конструктор создает копию изображения.

Класс QPixmap содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- isNull() — возвращает значение True, если объект является нулевым, и False — в противном случае;
- load(<Путь к файлу>[, format=None][, flags=AutoColor]) — загружает изображение из файла. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно указать тип изображения в виде строки. Если параметр не указан, то тип определяется автоматически. Необязательный параметр flags задает тип преобразования. Метод возвращает значение True, если изображение успешно загружено, и False — в противном случае;
- loadFromData(<QByteArray>[, format=None][, flags=AutoColor]) — загружает изображение из экземпляра класса QByteArray. В первом параметре можно указать данные, имеющие тип bytes. Метод возвращает значение True, если изображение успешно загружено, и False — в противном случае;
- save(<Путь к файлу>[, format=None][, quality=-1]) — сохраняет изображение в файл. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно задать тип изображения в виде строки. Если параметр не указан, то тип определяется автоматически по расширению файла. Необязательный параметр quality позволяет задать качество изображения. Можно передать значение в диапазоне от 0 до 100. Метод возвращает значение True, если изображение успешно сохранено, и False — в противном случае;

- convertFromImage(<QImage>[, flags=AutoColor]) — преобразует экземпляр класса QImage в экземпляр класса QPixmap. Метод возвращает значение True, если изображение успешно преобразовано, и False — в противном случае;
- fromImage(<QImage>[, flags=AutoColor]) — преобразует экземпляр класса QImage в экземпляр класса QPixmap и возвращает его. Метод является статическим;
- toImage() — преобразует экземпляр класса QPixmap в экземпляр класса QImage и возвращает его;
- fill() — производит заливку изображения указанным цветом. Форматы метода:

```
fill([color=white])
fill(<QWidget>, <X>, <Y>)
fill(<QWidget>, <QPoint>)
```

- width() — возвращает ширину изображения;
- height() — возвращает высоту изображения;
- size() — возвращает экземпляр класса QSize с размерами изображения;
- rect() — возвращает экземпляр класса QRect с координатами и размерами прямоугольной области, ограничивающей изображение;
- depth() — возвращает глубину цвета;
- isQBitmap() — возвращает значение True, если глубина цвета равна одному биту, и False — в противном случае;
- setMask(<QBitmap>) — устанавливает маску;
- mask() — возвращает экземпляр класса QBitmap с маской изображения;
- copy() — возвращает экземпляр класса QPixmap с фрагментом изображения. Если параметр rect не указан, то изображение копируется полностью. Форматы метода:

```
copy([rect=QRect()])
copy(<X>, <Y>, <Ширина>, <Высота>)
```

- scaled() — изменяет размер изображения и возвращает экземпляр класса QPixmap. Исходное изображение не изменяется. Форматы метода:

```
scaled(<Ширина>, <Высота>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
scaled(<QSize>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
```

  В необязательном параметре aspectRatioMode могут быть указаны следующие атрибуты из класса QtCore.Qt:

  - IgnoreAspectRatio — 0 — непосредственно изменяет размеры без сохранения пропорций сторон;
  - KeepAspectRatio — 1 — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - KeepAspectRatioByExpanding — 2 — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

  В необязательном параметре transformMode могут быть указаны следующие атрибуты из класса QtCore.Qt:

- FastTransformation — 0 — сглаживание выключено;
- SmoothTransformation — 1 — сглаживание включено;

- scaledToWidth(<Ширина>[, mode=FastTransformation]) — изменяет ширину изображения и возвращает экземпляр класса QPixmap. Высота изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();
- scaledToHeight(<Высота>[, mode=FastTransformation]) — изменяет высоту изображения и возвращает экземпляр класса QPixmap. Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр mode аналогичен параметру transformMode в методе scaled();
- transformed(<QMatrix>[, mode=FastTransformation]) — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса QPixmap. Исходное изображение не изменяется. Трансформация задается с помощью экземпляра класса QMatrix. Параметр mode аналогичен параметру transformMode в методе scaled();
- grabWindow(<winId>[, x=0][, y=0][, width=-1][, height=-1]) — создает скриншот указанного окна (без заголовка и границ). Метод возвращает экземпляр класса QPixmap. В первом параметре указывается результат выполнения метода winId() из класса QWidget. Остальные параметры задают координаты и размеры области. Если параметры не указаны, то делается скриншот всего окна. Координаты и размеры области могут выходить за границы окна. Метод является статическим. Пример сохранения скриншота окна в файл:

```
QtGui.QPixmap.grabWindow(window.winId()).save("window.png",
                                              "PNG")
```

Пример сохранения скриншота всего экрана в файл:

```
desktop = QtGui.QApplication.desktop()
QtGui.QPixmap.grabWindow(desktop.screen().winId()
                         ).save("screen.png", "PNG")
```

- grabWidget() — создает скриншот указанного компонента. Метод возвращает экземпляр класса QPixmap. Метод является статическим. Форматы метода:

```
grabWidget(<QWidget>[, x=0][, y=0][, width=-1][, height=-1])
grabWidget(<QWidget>, <QRect>)
```

25.3.2. Класс *QBitmap*

Класс QBitmap предназначен для работы с изображениями, имеющими глубину цвета равной одному биту, в контекстно-зависимом представлении. Наиболее часто класс QBitmap используется для создания масок изображений. Иерархия наследования:

```
QPaintDevice — QPixmap — QBitmap
```

Так как класс QBitmap наследует класс QPaintDevice, мы можем использовать его как поверхность для рисования. Цвет пера и кисти задается атрибутами color0 (прозрачный цвет) и color1 (непрозрачный цвет) из класса QtCore.Qt. Вывести изображение позволяет метод drawPixmap() из класса QPainter (см. *разд. 25.2.3*).

Форматы конструктора класса:

```
<Объект> = QBitmap()
<Объект> = QBitmap(<Ширина>, <Высота>)
<Объект> = QBitmap(<QSize>)
<Объект> = QBitmap(<Путь к файлу>[, format=None])
<Объект> = QBitmap(<QPixmap>)
<Объект> = QBitmap(<QBitmap>)
```

Класс `QBitmap` наследует все методы из класса `QPixmap` и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `fromImage(<QImage>[, flags=AutoColor])` — преобразует экземпляр класса `QImage` в экземпляр класса `QBitmap` и возвращает его. Метод является статическим;
- `transformed(<QMatrix>)` — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса `QBitmap`. Исходное изображение не изменяется. Трансформация задается с помощью экземпляра класса `QMatrix`;
- `clear()` — устанавливает все биты изображения в значение `color0`.

25.3.3. Класс *QImage*

Класс `QImage` предназначен для работы с изображениями в контекстно-независимом представлении. Перед выводом на экран экземпляр класса `QImage` преобразуется в экземпляр класса `QPixmap`. Иерархия наследования:

```
QPaintDevice — QImage
```

Так как класс `QImage` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Однако следует учитывать, что не на всех форматах изображения можно рисовать. Для рисования лучше использовать изображение формата `Format_ARGB32_Premultiplied`. Вывести изображение позволяет метод `drawImage()` из класса `QPainter` (см. *разд. 25.2.3*).

Форматы конструктора класса:

```
<Объект> = QImage()
<Объект> = QImage(<Ширина>, <Высота>, <Формат>)
<Объект> = QImage(<QSize>, <Формат>)
<Объект> = QImage(<Путь к файлу>[, <Тип изображения>])
<Объект> = QImage(<QImage>)
```

Первый конструктор создает нулевой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения. Если размеры равны нулю, то будет создан нулевой объект изображения. Четвертый конструктор предназначен для загрузки изображения из файла. Во втором параметре указывается тип изображения в виде строки (например, `"PNG"`). Если второй параметр не указан, то формат определяется автоматически. Пятый конструктор создает копию изображения.

В параметре `<Формат>` можно указать следующие атрибуты из класса `QImage` (перечислены только основные атрибуты; полный список смотрите в документации):

- `Format_Invalid` — `0` — формат не определен;
- `Format_Mono` — `1` — глубина цвета 1 бит;

- `Format_MonoLSB` — `2` — глубина цвета 1 бит;
- `Format_Indexed8` — `3` — глубина цвета 8 бит;
- `Format_RGB32` — `4` — RGB без альфа-канала, глубина цвета 32 бита;
- `Format_ARGB32` — `5` — RGB с альфа-каналом, глубина цвета 32 бита;
- `Format_ARGB32_Premultiplied` — `6` — RGB с альфа-каналом, глубина цвета 32 бита. Этот формат лучше использовать для рисования.

Класс `QImage` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `True`, если объект является нулевым, и `False` — в противном случае;
- `load(<Путь к файлу>[, format=None])` — загружает изображение из файла. В первом параметре задается абсолютный или относительный путь к файлу. Во втором параметре указывается тип изображения в виде строки (например, `"PNG"`). Если параметр не указан, то тип определяется автоматически. Метод возвращает значение `True`, если изображение успешно загружено, и `False` — в противном случае;
- `loadFromData(<QByteArray>[, format=None])` — загружает изображение из экземпляра класса `QByteArray`. В первом параметре можно указать данные, имеющие тип `bytes`. Во втором параметре указывается тип изображения в виде строки (например, `"PNG"`). Метод возвращает значение `True`, если изображение успешно загружено, и `False` — в противном случае;
- `fromData(<QByteArray>[, format=None])` — загружает изображение из экземпляра класса `QByteArray`. В первом параметре можно указать данные, имеющие тип `bytes`. Во втором параметре указывается тип изображения в виде строки (например, `"PNG"`). Метод возвращает экземпляр класса `QImage`. Метод является статическим;
- `save(<Путь к файлу>[, format=None][, quality=-1])` — сохраняет изображение в файл. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно задать тип изображения в виде строки. Если параметр не указан, то тип определяется автоматически по расширению файла. Необязательный параметр `quality` позволяет задать качество изображения. Можно передать значение в диапазоне от `0` до `100`. Метод возвращает значение `True`, если изображение успешно сохранено, и `False` — в противном случае;
- `fill(<Число>)` — производит заливку изображения определенным цветом. В качестве параметра указывается целочисленное значение цвета. Получить это значение позволяют методы `rgb()` и `rgba()` из класса `QColor`. Пример:

```
img.fill(QtGui.QColor("#ff0000").rgb())
```

- `width()` — возвращает ширину изображения;
- `height()` — возвращает высоту изображения;
- `size()` — возвращает экземпляр класса `QSize` с размерами изображения;
- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, ограничивающей изображение;
- `depth()` — возвращает глубину цвета;
- `format()` — возвращает формат изображения (см. значения параметра `<Формат>` в конструкторе класса `QImage`);

- `setPixel()` — задает цвет указанного пиксела. Форматы метода:

```
setPixel(<X>, <Y>, <Индекс или цвет>)
setPixel(<QPoint>, <Индекс или цвет>)
```

  В параметре `<Индекс или цвет>` для 8-битных изображений задается индекс цвета в палитре, а для 32-битных — целочисленное значение цвета. Получить целочисленное значение цвета позволяют методы `rgb()` и `rgba()` из класса `QColor`;

- `pixel()` — возвращает целочисленное значение цвета указанного пиксела. Это значение можно передать конструктору класса `QColor`, а затем получить значения различных составляющих. Форматы метода:

```
pixel(<X>, <Y>)
pixel(<QPoint>)
```

- `convertToFormat()` — преобразует формат изображения (см. значения параметра `<Формат>` в конструкторе класса `QImage`) и возвращает новый экземпляр класса `QImage`. Исходное изображение не изменяется. Форматы метода:

```
convertToFormat(<Формат>[, flags=AutoColor])
convertToFormat(<Формат>, <Таблица цветов>[, flags=AutoColor])
```

- `copy()` — возвращает экземпляр класса `QImage` с фрагментом изображения. Если параметр `rect` не указан, то изображение копируется полностью. Форматы метода:

```
copy([rect=QRect()])
copy(<X>, <Y>, <Ширина>, <Высота>)
```

- `scaled()` — изменяет размер изображения и возвращает экземпляр класса `QImage`. Исходное изображение не изменяется. Форматы метода:

```
scaled(<Ширина>, <Высота>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
scaled(<QSize>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
```

  В необязательном параметре `aspectRatioMode` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

  - `IgnoreAspectRatio` — `0` — непосредственно изменяет размеры без сохранения пропорций сторон;
  - `KeepAspectRatio` — `1` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - `KeepAspectRatioByExpanding` — `2` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

  В необязательном параметре `transformMode` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

  - `FastTransformation` — `0` — сглаживание выключено;
  - `SmoothTransformation` — `1` — сглаживание включено;

- `scaledToWidth(<Ширина>[, mode=FastTransformation])` — изменяет ширину изображения и возвращает экземпляр класса `QImage`. Высота изображения изменяется пропорцио-

нально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;

- `scaledToHeight(<Высота>[, mode=FastTransformation])` — изменяет высоту изображения и возвращает экземпляр класса `QImage`. Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;
- `transformed(<QMatrix>[, mode=FastTransformation])` — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса `QImage`. Исходное изображение не изменяется. Трансформация задается с помощью экземпляра класса `QMatrix`. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;
- `invertPixels([mode=InvertRgb])` — инвертирует значения всех пикселов в изображении. В необязательном параметре `mode` могут быть указаны атрибуты `InvertRgb` или `InvertRgba` из класса `QImage`;
- `mirrored([horizontal=False][, vertical=True])` — создает зеркальный образ изображения. Метод возвращает экземпляр класса `QImage`. Исходное изображение не изменяется.

25.3.4. Класс *QIcon*

Класс `QIcon` описывает иконки в различных размерах, режимах и состояниях. Обратите внимание на то, что класс `QIcon` не наследует класс `QPaintDevice`, следовательно, мы не можем использовать его как поверхность для рисования. Форматы конструктора:

```
<Объект> = QIcon()
<Объект> = QIcon(<Путь к файлу>)
<Объект> = QIcon(<QPixmap>)
<Объект> = QIcon(<QIcon>)
```

Первый конструктор создает нулевой объект иконки. Второй конструктор предназначен для загрузки иконки из файла. Обратите внимание на то, что файл загружается при первой попытке использования, а не сразу. Третий конструктор создает иконку на основе экземпляра класса `QPixmap`, а четвертый конструктор создает копию иконки.

Класс `QIcon` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `True`, если объект является нулевым, и `False` — в противном случае;
- `addFile(<Путь к файлу>[, size=QSize()][, mode=Normal][, state=Off])` — добавляет иконку для указанного размера, режима и состояния. Можно добавить несколько иконок, вызывая метод с разными значениями параметров. Параметр `size` задает размер иконки (с помощью экземпляра класса `QSize`). Так как загрузка иконки производится при первой попытке использования, заранее размер иконки неизвестен. В параметре `mode` можно указать следующие атрибуты из класса `QIcon`: `Normal`, `Disabled`, `Active` или `Selected`. В параметре `state` указываются атрибуты `Off` или `On` из класса `QIcon`;
- `addPixmap(<QPixmap>[, mode=Normal][, state=Off])` — добавляет иконку для указанного режима и состояния. Иконка загружается из экземпляра класса `QPixmap`;
- `availableSizes([mode=Normal][, state=Off])` — возвращает доступные размеры (список с экземплярами класса `QSize`) иконок для указанного режима и состояния;

- `actualSize(<QSize>[, mode=Normal][, state=Off])` — возвращает фактический размер (экземпляр класса `QSize`) для указанного размера, режима и состояния. Фактический размер может быть меньше размера, указанного в первом параметре, но не больше;
- `pixmap()` — возвращает иконку (экземпляр класса `QPixmap`), которая примерно соответствует указанному размеру, режиму и состоянию. Форматы метода:

```
pixmap(<Ширина>, <Высота>[, mode=Normal][, state=Off])
pixmap(<QSize>[, mode=Normal][, state=Off])
```

Вместо загрузки иконки из файла можно воспользоваться одной из встроенных иконок. Загрузить стандартную иконку позволяет следующий код:

```
ico = window.style().standardIcon(QtGui.QStyle.SP_MessageBoxCritical)
```

Посмотреть список всех встроенных иконок можно в документации к классу `QStyle` (C:/Python32/Lib/site-packages/PyQt4/doc/html/qstyle.html#StandardPixmap-enum).

ГЛАВА 26

Графическая сцена

Графическая сцена позволяет отображать объекты (например, линию, прямоугольник и др.) и производить с ними различные манипуляции (например, перемещать с помощью мыши, трансформировать и др.). Для отображения графических объектов применяется концепция "модель/представление", позволяющая отделить данные от их отображения и избежать дублирования данных. Благодаря этому одну и ту же сцену можно отобразить сразу в нескольких представлениях без дублирования. В основе концепции лежат следующие классы:

- `QGraphicsScene` — исполняет роль сцены, на которой расположены графические объекты. Этот класс содержит также множество методов для управления этими объектами;
- `QGraphicsView` — предназначен для отображения сцены. Одну сцену можно отображать с помощью нескольких представлений;
- `QGraphicsItem` — является базовым классом для графических объектов. Можно наследовать этот класс и реализовать свой графический объект или воспользоваться готовыми классами, например, `QGraphicsRectItem` (прямоугольник), `QGraphicsEllipseItem` (эллипс) и др.

26.1. Класс *QGraphicsScene*. Сцена

Класс `QGraphicsScene` исполняет роль сцены, на которой расположены графические объекты. Этот класс содержит также множество методов для управления этими объектами. Иерархия наследования выглядит так:

```
QObject — QGraphicsScene
```

Форматы конструктора:

```
<Объект> = QGraphicsScene([parent=None])
<Объект> = QGraphicsScene(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
<Объект> = QGraphicsScene(<QRectF>[, parent=None])
```

Первый конструктор создает сцену, не имеющую определенного размера. Второй и третий конструкторы позволяют указать размеры сцены в виде вещественных чисел или экземпляра класса `QRectF`. В качестве параметра `parent` можно указать ссылку на родительский компонент.

26.1.1. Настройка параметров сцены

Для настройки различных параметров сцены предназначены следующие методы из класса QGraphicsScene:

- setSceneRect() — задает координаты и размеры сцены. Форматы метода:

  setSceneRect(<X>, <Y>, <Ширина>, <Высота>)
  setSceneRect(<QRectF>)

- sceneRect() — возвращает экземпляр класса QRectF с координатами и размерами сцены;
- width() и height() — возвращают ширину и высоту сцены соответственно в виде вещественного числа;
- itemsBoundingRect() — возвращает экземпляр класса QRectF с координатами и размерами прямоугольника, в который можно вписать все объекты, расположенные на сцене;
- setBackgroundBrush(<QBrush>) — задает кисть для заднего плана (расположен под графическими объектами). Чтобы изменить задний фон, можно также переопределить метод drawBackground() и внутри него выполнять перерисовку заднего фона при каждом вызове;
- setForegroundBrush(<QBrush>) — задает кисть для переднего плана (расположен над графическими объектами). Чтобы изменить передний фон, можно также переопределить метод drawForeground() и внутри него выполнять перерисовку переднего фона при каждом вызове;
- setFont(<QFont>) — задает шрифт сцены по умолчанию;
- setItemIndexMethod(<Режим>) — задает режим индексации объектов сцены. В качестве параметра указываются следующие атрибуты из класса QGraphicsScene:
  - BspTreeIndex — 0 — для поиска объектов используется индекс в виде бинарного дерева. Этот режим следует применять для сцен, большинство объектов которых являются статическими;
  - NoIndex — -1 — индекс не используется. Этот режим следует использовать для динамических сцен;
- setBspTreeDepth(<Число>) — задает глубину дерева при использовании режима BspTreeIndex. По умолчанию установлено значение 0, которое говорит, что глубина выбирается автоматически;
- bspTreeDepth() — возвращает текущее значение глубины дерева при использовании режима BspTreeIndex.

26.1.2. Добавление и удаление графических объектов

Для добавления графических объектов на сцену и удаления их предназначены следующие методы из класса QGraphicsScene:

- addItem(<QGraphicsItem>) — добавляет графический объект на сцену. В качестве значения указывается экземпляр класса, который наследует класс QGraphicsItem, например, QGraphicsEllipseItem (эллипс);
- addLine() — создает линию, добавляет ее на сцену и возвращает ссылку на нее (экземпляр класса QGraphicsLineItem).

Форматы метода:

```
addLine(<X1>, <Y1>, <X2>, <Y2>[, pen=QPen()])
addLine(<QLineF>[, pen=QPen()])
```

- `addRect()` — создает прямоугольник, добавляет его на сцену и возвращает ссылку на него (экземпляр класса `QGraphicsRectItem`). Форматы метода:

```
addRect(<X>, <Y>, <Ширина>, <Высота>[,
        pen=QPen()][, brush=QBrush()])
addRect(<QRectF>[, pen=QPen()][, brush=QBrush()])
```

- `addPolygon()` — создает многоугольник, добавляет его на сцену и возвращает ссылку на него (экземпляр класса `QGraphicsPolygonItem`). Формат метода:

```
addPolygon(<QPolygonF>[, pen=QPen()][, brush=QBrush()])
```

- `addEllipse()` — создает эллипс, добавляет его на сцену и возвращает ссылку на него (экземпляр класса `QGraphicsEllipseItem`). Форматы метода:

```
addEllipse(<X>, <Y>, <Ширина>, <Высота>[,
           pen=QPen()][, brush=QBrush()])
addEllipse(<QRectF>[, pen=QPen()][, brush=QBrush()])
```

- `addPixmap(<QPixmap>)` — создает изображение, добавляет его на сцену и возвращает ссылку на него (экземпляр класса `QGraphicsPixmapItem`);
- `addSimpleText(<Текст>[, font=QFont()])` — создает экземпляр класса `QGraphicsSimpleTextItem`, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает ссылку на него;
- `addText(<Текст>[, font=QFont()])` — создает экземпляр класса `QGraphicsTextItem`, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает ссылку на него;
- `addPath(<QPainterPath>[, pen=QPen()][, brush=QBrush()])` — создает экземпляр класса `QGraphicsPathItem`, добавляет его на сцену и возвращает ссылку на него;
- `removeItem(<QGraphicsItem>)` — убирает графический объект (и всех его потомков) со сцены. Графический объект при этом не удаляется и, например, может быть добавлен на другую сцену. В качестве значения указывается экземпляр класса, который наследует класс `QGraphicsItem`, например, `QGraphicsEllipseItem` (эллипс);
- `clear()` — удаляет все элементы со сцены. Метод является слотом;
- `createItemGroup(<Список с объектами>)` — группирует объекты, добавляет группу на сцену и возвращает экземпляр класса `QGraphicsItemGroup`;
- `destroyItemGroup(<QGraphicsItemGroup>)` — удаляет группу со сцены.

26.1.3. Добавление компонентов на сцену

Помимо графических объектов на сцену можно добавить компоненты, которые будут функционировать как обычно. Добавить компонент на сцену позволяет метод `addWidget()` из класса `QGraphicsScene`. Формат метода:

```
addWidget(<QWidget>[, flags=0])
```

В первом параметре указывается экземпляр класса, который наследует класс `QWidget`. Во втором параметре задается тип окна (см. *разд. 20.2*). Метод возвращает ссылку на добавленный компонент (экземпляр класса `QGraphicsProxyWidget`).

Чтобы изменить текст в заголовке окна, следует воспользоваться методом `setWindowTitle()` из класса `QGraphicsWidget`.

Сделать окно активным позволяет метод `setActiveWindow(<QGraphicsWidget>)`. Получить ссылку на активное окно можно с помощью метода `activeWindow()`. Если активного окна нет, то метод возвращает значение `None`.

26.1.4. Поиск объектов

Для поиска объектов предназначены следующие методы из класса `QGraphicsScene`:

- `itemAt()` — возвращает ссылку на верхний видимый объект, который расположен по указанным координатам, или значение `None`, если объекта нет. Форматы метода:

```
itemAt(<X>, <Y>[, <QTransform>])
itemAt(<QPointF>[, <QTransform>])
```

- `collidingItems(<QGraphicsItem>[, mode=IntersectsItemShape])` — возвращает список со ссылками на объекты, которые сталкиваются с указанным в первом параметре объектом. Если таких объектов нет, то метод возвращает пустой список;
- `items()` — возвращает список со ссылками на все объекты или на объекты, расположенные по указанным координатам, или объекты, попадающие в указанную область. Если объектов нет, то возвращается пустой список. Форматы метода:

```
items()
items(<Режим сортировки>)
items(<QPointF>)
items(<QPointF>, <Режим попадания>, <Режим сортировки>[,
      deviceTransform=QTransform()])
items(<X>, <Y>, <Ширина>, <Высота>[, mode=IntersectsItemShape])
items(<X>, <Y>, <Ширина>, <Высота>, <Режим попадания>,
      <Режим сортировки>[, deviceTransform=QTransform()])
items(<QRectF>[, mode=IntersectsItemShape])
items(<QRectF>, <Режим попадания>, <Режим сортировки>[,
      deviceTransform=QTransform()])
items(<QPolygonF>[, mode=IntersectsItemShape])
items(<QPolygonF>, <Режим попадания>, <Режим сортировки>[,
      deviceTransform=QTransform()])
items(<QPainterPath>[, mode=IntersectsItemShape])
items(<QPainterPath>, <Режим попадания>, <Режим сортировки>[,
      deviceTransform=QTransform()])
```

В параметре `<Режим сортировки>` указываются атрибуты `AscendingOrder` (в алфавитном порядке) или `DescendingOrder` (в обратном порядке) из класса `QtCore.Qt`.

В параметрах `<Режим попадания>` и `mode` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `ContainsItemShape` — `0` — объект попадет в список, если все точки объекта находятся внутри области;
- `IntersectsItemShape` — `1` — объект попадет в список, если любая точка объекта попадет в область;

- `ContainsItemBoundingRect` — `2` — объект попадет в список, если охватывающий прямоугольник полностью находится внутри области;
- `IntersectsItemBoundingRect` — `3` — объект попадет в список, если любая точка охватывающего прямоугольника попадет в область.

26.1.5. Управление фокусом ввода

Обладать фокусом ввода с клавиатуры может как сцена в целом, так и отдельный объект на сцене. Если фокус установлен на отдельный объект, то все события клавиатуры перенаправляются этому объекту. Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable`, например, с помощью метода `setFlag()` из класса `QGraphicsItem`. Для управления фокусом ввода предназначены следующие методы из класса `QGraphicsScene`:

- `setFocus([focusReason=OtherFocusReason])` — устанавливает фокус ввода на сцену. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 21.9.1*);
- `setFocusItem(<QGraphicsItem>[, focusReason=OtherFocusReason])` — устанавливает фокус ввода на указанный графический объект на сцене. Если сцена была вне фокуса ввода, то она автоматически получит фокус. Если в первом параметре указано значение `None` или объект не может принимать фокус, то метод просто убирает фокус с объекта, который обладает фокусом ввода в данный момент. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 21.9.1*);
- `clearFocus()` — убирает фокус ввода со сцены. Объект на сцене, который обладает фокусом ввода в данный момент, потеряет фокус, но получит его снова, когда фокус будет снова установлен на сцену;
- `hasFocus()` — возвращает значение `True`, если сцена имеет фокус ввода, и `False` — в противном случае;
- `focusItem()` — возвращает ссылку на объект, который обладает фокусом ввода, или значение `None`;
- `setStickyFocus(<Флаг>)` — если в качестве параметра указано значение `True`, то при щелчке мышью на фоне сцены или на объекте, который не может принимать фокус, объект, владеющий фокусом, не потеряет фокус ввода. По умолчанию фокус убирается;
- `stickyFocus()` — возвращает значение `True`, если фокус ввода не будет убран с объекта при щелчке мышью на фоне или на объекте, который не может принимать фокус.

26.1.6. Управление выделением объектов

Чтобы объект можно было выделить (с помощью мыши или из программы), необходимо установить флаг `ItemIsSelectable`, например, с помощью метода `setFlag()` из класса `QGraphicsItem`. Для управления выделением объектов предназначены следующие методы из класса `QGraphicsScene`:

- `setSelectionArea()` — выделяет объекты внутри указанной области. Чтобы выделить только один объект, следует воспользоваться методом `setSelected()` из класса `QGraphicsItem`. Форматы метода `setSelectionArea()`:

```
setSelectionArea(<QPainterPath>)
setSelectionArea(<QPainterPath>, <Режим>)
```

```
setSelectionArea(<QPainterPath>, <QTransform>)
setSelectionArea(<QPainterPath>, <Режим>, <QTransform>)
```

В параметре `<Режим>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `ContainsItemShape` — `0` — объект будет выделен, если все точки объекта находятся внутри области выделения;
- `IntersectsItemShape` — `1` — объект будет выделен, если любая точка объекта попадет в область выделения;
- `ContainsItemBoundingRect` — `2` — объект будет выделен, если охватывающий прямоугольник полностью находится внутри области выделения;
- `IntersectsItemBoundingRect` — `3` — объект будет выделен, если любая точка охватывающего прямоугольника попадет в область выделения;

- `selectionArea()` — возвращает область выделения (экземпляр класса `QPainterPath`);
- `selectedItems()` — возвращает список со ссылками на выделенные объекты или пустой список, если выделенных объектов нет;
- `clearSelection()` — снимает выделение. Метод является слотом.

26.1.7. Прочие методы и сигналы

Помимо рассмотренных методов класс `QGraphicsScene` содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `isActive()` — возвращает значение `True`, если сцена отображается представлением, и `False` — в противном случае;
- `views()` — возвращает список с представлениями (экземпляры класса `QGraphicsView`), к которым подключена сцена. Если сцена не подключена к представлениям, то возвращается пустой список;
- `mouseGrabberItem()` — возвращает ссылку на объект, который владеет мышью, или значение `None`, если такого объекта нет;
- `render()` — позволяет вывести содержимое сцены на принтер или на устройство рисования. Формат метода:

```
render(<QPainter>[, target=QRectF()][, source=QRectF()][,
       mode=KeepAspectRatio])
```

Параметр `target` задает координаты и размеры устройства рисования, а параметр `source` — координаты и размеры прямоугольной области на сцене. Если параметры не указаны, то используются размеры устройства рисования и сцены;

- `invalidate()` — вызывает перерисовку указанных слоев внутри прямоугольной области на сцене. Форматы метода:

```
invalidate(<X>, <Y>, <Ширина>, <Высота>[, layers=AllLayers])
invalidate([rect=QRectF()][, layers=AllLayers])
```

В параметре `layers` могут быть указаны следующие атрибуты из класса `QGraphicsScene`:

- `ItemLayer` — `1` — слой объекта;
- `BackgroundLayer` — `2` — слой заднего плана;

- `ForegroundLayer` — `4` — слой переднего плана;
- `AllLayers` — `65535` — все слои. Вначале отрисовывается слой заднего плана, затем слой объекта и в конце слой переднего плана.

Метод является слотом с сигнатурой:

```
invalidate(const QRectF& = QRectF(),
    QGraphicsScene::SceneLayers = QGraphicsScene.AllLayers)
```

- `update()` — вызывает перерисовку указанной прямоугольной области сцены. Форматы метода:

```
update(<X>, <Y>, <Ширина>, <Высота>)
update([rect=QRectF()])
```

Метод является слотом с сигнатурой `update(const QRectF& = QRectF())`.

Класс `QGraphicsScene` содержит следующие сигналы:

- `changed(const QList<QRectF>&)` — генерируется при изменении сцены. Внутри обработчика через параметр доступен список с экземплярами класса `QRectF` или пустой список;
- `sceneRectChanged(const QRectF&)` — генерируется при изменении размеров сцены. Внутри обработчика через параметр доступен экземпляр класса `QRectF` с новыми координатами и размерами сцены;
- `selectionChanged()` — генерируется при изменении выделения объектов.

26.2. Класс *QGraphicsView*. Представление

Класс `QGraphicsView` предназначен для отображения сцены. Одну сцену можно отображать с помощью нескольких представлений. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QGraphicsView
```

Форматы конструктора класса:

```
<Объект> = QGraphicsView([parent=None])
<Объект> = QGraphicsView(<QGraphicsScene>[, parent=None])
```

26.2.1. Настройка параметров представления

Для настройки различных параметров представления предназначены следующие методы из класса `QGraphicsView` (перечислены только основные методы; полный список смотрите в документации):

- `setScene(<QGraphicsScene>)` — устанавливает сцену в представление;
- `scene()` — возвращает ссылку на сцену (экземпляр класса `QGraphicsScene`);
- `setSceneRect()` — задает координаты и размеры сцены. Форматы метода:

```
setSceneRect(<X>, <Y>, <Ширина>, <Высота>)
setSceneRect(<QRectF>)
```

- `sceneRect()` — возвращает экземпляр класса `QRectF` с координатами и размерами сцены;

- `setBackgroundBrush(<QBrush>)` — задает кисть для заднего плана сцены (расположен под графическими объектами);
- `setForegroundBrush(<QBrush>)` — задает кисть для переднего плана сцены (расположен над графическими объектами);
- `setCacheMode(<Режим>)` — задает режим кеширования. В качестве параметра могут быть указаны следующие атрибуты из класса `QGraphicsView`:
  - `CacheNone` — `0` — без кеширования;
  - `CacheBackground` — `1` — кешируется задний фон;
- `resetCachedContent()` — сбрасывает кеш;
- `setAlignment(<Выравнивание>)` — задает выравнивание сцены внутри представления при отсутствии полос прокрутки. По умолчанию сцена центрируется по горизонтали и вертикали. Пример установки сцены в левом верхнем углу представления:

  ```
  view.setAlignment(QtCore.Qt.AlignLeft | QtCore.Qt.AlignTop)
  ```

- `setInteractive(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может взаимодействовать с объектами на сцене (интерактивный режим используется по умолчанию). Значение `False` устанавливает режим только для чтения;
- `isInteractive()` — возвращает значение `True`, если используется интерактивный режим, и `False` — в противном случае;
- `setDragMode(<Режим>)` — задает действие, которое производится при щелчке левой кнопкой мыши на фоне и перемещении мыши. Получить текущее действие позволяет метод `dragMode()`. В качестве параметра могут быть указаны следующие атрибуты из класса `QGraphicsView`:
  - `NoDrag` — `0` — действие не определено;
  - `ScrollHandDrag` — `1` — перемещение мыши при нажатой левой кнопки будет приводить к прокрутке сцены. При этом указатель мыши примет вид сжатой или разжатой руки;
  - `RubberBandDrag` — `2` — создается область выделения. Объекты, частично или полностью (задается с помощью метода `setRubberBandSelectionMode()`) попавшие в эту область, будут выделены (при условии, что установлен флаг `ItemIsSelectable`). Действие выполняется только в интерактивном режиме;
- `setRubberBandSelectionMode(<Режим>)` — задает режим выделения объектов при установленном флаге `RubberBandDrag`. В параметре `<Режим>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `ContainsItemShape` — `0` — объект будет выделен, если все точки объекта находятся внутри области выделения;
  - `IntersectsItemShape` — `1` — объект будет выделен, если любая точка объекта попадет в область выделения;
  - `ContainsItemBoundingRect` — `2` — объект будет выделен, если охватывающий прямоугольник полностью находится внутри области выделения;
  - `IntersectsItemBoundingRect` — `3` — объект будет выделен, если любая точка охватывающего прямоугольника попадет в область выделения.

26.2.2. Преобразования между координатами представления и сцены

Для преобразования между координатами представления и сцены предназначены следующие методы из класса `QGraphicsView`:

- `mapFromScene(<X>, <Y>)` и `mapFromScene(<QPointF>)` — преобразуют координаты точки из системы координат сцены в систему координат представления. Методы возвращают экземпляр класса `QPoint`.

  Можно также воспользоваться следующими форматами метода `mapFromScene()`:

```
mapFromScene(<X>, <Y>, <Ширина>, <Высота>) -> QPolygon
mapFromScene(<QRectF>)                     -> QPolygon
mapFromScene(<QPolygonF>)                  -> QPolygon
mapFromScene(<QPainterPath>)               -> QPainterPath
```

- `mapToScene(<X>, <Y>)` и `mapToScene(<QPoint>)` — преобразуют координаты точки из системы координат представления в систему координат сцены. Методы возвращают экземпляр класса `QPointF`.

  Можно также воспользоваться следующими форматами метода `mapToScene()`:

```
mapToScene(<X>, <Y>, <Ширина>, <Высота>)   -> QPolygonF
mapToScene(<QRect>)                        -> QPolygonF
mapToScene(<QPolygon>)                     -> QPolygonF
mapToScene(<QPainterPath>)                 -> QPainterPath
```

26.2.3. Поиск объектов

Для поиска объектов на сцене предназначены следующие методы:

- `itemAt()` — возвращает ссылку на верхний видимый объект, который расположен по указанным координатам, или значение `None`, если объекта нет. В качестве значений указываются координаты в системе координат представления, а не сцены. Форматы метода:

```
itemAt(<X>, <Y>)
itemAt(<QPoint>)
```

- `items()` — возвращает список со ссылками на все объекты или на объекты, расположенные по указанным координатам, или объекты, попадающие в указанную область. Если объектов нет, то возвращается пустой список. В качестве значений указываются координаты в системе координат представления, а не сцены. Форматы метода:

```
items()
items(<X>, <Y>)
items(<QPoint>)
items(<X>, <Y>, <Ширина>, <Высота>[, mode=IntersectsItemShape])
items(<QRect>[, mode=IntersectsItemShape])
items(<QPolygon>[, mode=IntersectsItemShape])
items(<QPainterPath>[, mode=IntersectsItemShape])
```

  Допустимые значения параметра `mode` мы уже рассматривали в *разд. 26.1.4*.

26.2.4. Трансформация систем координат

Произвести трансформацию системы координат позволяют следующие методы из класса `QGraphicsView`:

- `translate(<X>, <Y>)` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз;
- `rotate(<Угол>)` — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;
- `scale(<По оси X>, <По оси Y>)` — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- `shear(<По горизонтали>, <По вертикали>)` — сдвигает систему координат. В качестве значений указываются вещественные числа;
- `resetMatrix()` — отменяет все трансформации.

Несколько трансформаций можно произвести последовательно друг за другом. В этом случае следует учитывать, что порядок следования трансформаций имеет значение. Если одна и та же последовательность выполняется несколько раз, то ее можно сохранить в экземпляре класса `QMatrix`, а затем установить с помощью метода `setMatrix()`. Получить ссылку на установленную матрицу позволяет метод `matrix()`.

26.2.5. Прочие методы

Помимо рассмотренных методов класс `QGraphicsView` содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `centerOn()` — прокручивает область таким образом, чтобы указанная точка или объект находились в центре видимой области представления. Форматы метода:

```
centerOn(<X>, <Y>)
centerOn(<QPointF>)
centerOn(<QGraphicsItem>)
```

- `ensureVisible()` — прокручивает область таким образом, чтобы указанный прямоугольник или объект находились в видимой области представления. Форматы метода:

```
ensureVisible(<X>, <Y>, <Ширина>, <Высота>[, xMargin=50][,
              yMargin=50])
ensureVisible(<QRectF>[, xMargin=50][, yMargin=50])
ensureVisible(<QGraphicsItem>[, xMargin=50][, yMargin=50])
```

- `fitInView()` — прокручивает и масштабирует область таким образом, чтобы указанный прямоугольник или объект занимали всю видимую область представления. Форматы метода:

```
fitInView(<X>, <Y>, <Ширина>, <Высота>[, mode=IgnoreAspectRatio])
fitInView(<QRectF>[, mode=IgnoreAspectRatio])
fitInView(<QGraphicsItem>[, mode=IgnoreAspectRatio])
```

- `render()` — позволяет вывести содержимое представления на принтер или на устройство рисования. Формат метода:

```
render(<QPainter>[, target=QRectF()][, source=QRectF()][,
       mode=KeepAspectRatio])
```

- `invalidateScene([rect=QRectF()][, layers=AllLayers])` — вызывает перерисовку указанных слоев внутри прямоугольной области на сцене. Метод является слотом с сигнатурой:

```
invalidateScene(const QRectF& = QRectF(),
    QGraphicsScene::SceneLayers = QGraphicsScene.AllLayers)
```

- `updateSceneRect(<QRectF>)` — вызывает перерисовку указанной прямоугольной области сцены. Метод является слотом с сигнатурой `updateSceneRect(const QRectF&)`;
- `updateScene(<Список с экземплярами класса QRectF>)` — вызывает перерисовку указанных прямоугольных областей. Метод является слотом с сигнатурой `updateScene(const QList<QRectF>&)`.

26.3. Класс *QGraphicsItem*. Базовый класс для графических объектов

Абстрактный класс `QGraphicsItem` является базовым классом для графических объектов. Формат конструктора класса:

```
QGraphicsItem([parent=None][, scene=None])
```

В параметре `parent` может быть указана ссылка на родительский объект (экземпляр класса, наследующего класс `QGraphicsItem`). Необязательный параметр `scene` позволяет указать ссылку на сцену (экземпляр класса `QGraphicsScene`).

Так как класс `QGraphicsItem` является абстрактным, создать экземпляр этого класса нельзя. Чтобы создать новый графический объект, следует наследовать класс `QGraphicsItem` и переопределить минимум методы `boundingRect()` и `paint()`. Метод `boundingRect()` должен возвращать экземпляр класса `QRectF` с координатами и размерами прямоугольной области, ограничивающей объект. Внутри метода `paint()` необходимо выполнить рисование объекта. Формат метода `paint()`:

```
paint(self, <QPainter>, <QStyleOptionGraphicsItem>, widget=None)
```

Для обработки столкновений следует также перегрузить метод `shape()`. Метод должен возвращать экземпляр класса `QPainterPath`.

26.3.1. Настройка параметров объекта

Для настройки различных параметров объекта предназначены следующие методы из класса `QGraphicsItem` (перечислены только основные методы; полный список смотрите в документации):

- `setPos()` — задает позицию объекта относительно родителя или сцены (при отсутствии родителя). Форматы метода:

```
setPos(<X>, <Y>)
setPos(<QPointF>)
```

- `pos()` — возвращает экземпляр класса `QPointF` с текущими координатами относительно родителя или сцены (при отсутствии родителя);
- `scenePos()` — возвращает экземпляр класса `QPointF` с текущими координатами относительно сцены;
- `sceneBoundingRect()` — возвращает экземпляр класса `QRectF`, который содержит координаты (относительно сцены) и размеры прямоугольника, ограничивающего объект;
- `setX(<X>)` и `setY(<Y>)` — задают позицию объекта по отдельным осям;
- `x()` и `y()` — возвращают позицию объекта по отдельным осям;
- `setZValue(<Z>)` — задает позицию объекта по оси `Z`. Объект с большим значением рисуется выше объекта с меньшим значением. По умолчанию для всех объектов значение равно `0.0`;
- `zValue()` — возвращает позицию объекта по оси `Z`;
- `moveBy(<По оси X>, <По оси Y>)` — сдвигает объект на указанное смещение относительно текущей позиции;
- `prepareGeometryChange()` — этот метод следует вызвать перед изменением размеров объекта, чтобы поддержать индекс сцены в актуальном состоянии;
- `scene()` — возвращает ссылку на сцену (экземпляр класса `QGraphicsScene`) или значение `None`;
- `setFlag(<Флаг>[, enabled=True])` — устанавливает (если второй параметр имеет значение `True`) или сбрасывает (если второй параметр имеет значение `False`) указанный флаг. В первом параметре могут быть указаны следующие атрибуты из класса `QGraphicsItem` (перечислены только основные атрибуты; полный список смотрите в документации):
  - `ItemIsMovable` — `0x1` — объект можно перемещать с помощью мыши;
  - `ItemIsSelectable` — `0x2` — объект можно выделять;
  - `ItemIsFocusable` — `0x4` — объект может получить фокус ввода;
  - `ItemIgnoresTransformations` — `0x20` — объект игнорирует наследуемые трансформации;
  - `ItemIgnoresParentOpacity` — `0x40` — объект игнорирует прозрачность родителя;
  - `ItemDoesntPropagateOpacityToChildren` — `0x80` — прозрачность объекта не распространяется на потомков;
  - `ItemStacksBehindParent` — `0x100` — объект располагается позади родителя;
  - `ItemIsPanel` — `0x4000` — объект является панелью;
- `setFlags(<Флаги>)` — устанавливает несколько флагов. Атрибуты (см. описание метода `setFlag()`) указываются через оператор `|`;
- `flags()` — возвращает комбинацию установленных флагов (см. описание метода `setFlag()`);
- `setOpacity(<Число>)` — задает степень прозрачности объекта. В качестве значения указывается вещественное число от `0.0` (полностью прозрачный) до `1.0` (полностью непрозрачный);
- `opacity()` — возвращает степень прозрачности объекта;

- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `setCursor(<Курсор>)` — задает внешний вид указателя мыши при наведении указателя на объект (см. *разд. 21.10.5*);
- `unsetCursor()` — отменяет изменение указателя мыши;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то объект будет видим. Значение `False` скрывает объект;
- `show()` — делает объект видимым;
- `hide()` — скрывает объект;
- `isVisible()` — возвращает значение `True`, если объект видим, и `False` — если скрыт;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то объект будет доступен. Значение `False` делает объект недоступным. Недоступный объект не получает никаких событий и его нельзя выделить;
- `isEnabled()` — возвращает значение `True`, если объект доступен, и `False` — если недоступен;
- `setSelected(<Флаг>)` — если в качестве параметра указано значение `True`, то объект будет выделен. Значение `False` снимает выделение. Чтобы объект можно было выделить, необходимо установить флаг `ItemIsSelectable`, например, с помощью метода `setFlag()` из класса `QGraphicsItem`;
- `isSelected()` — возвращает значение `True`, если объект выделен, и `False` — в противном случае;
- `setFocus([focusReason=OtherFocusReason])` — устанавливает фокус ввода на объект. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 21.9.1*). Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable`, например, с помощью метода `setFlag()` из класса `QGraphicsItem`;
- `clearFocus()` — убирает фокус ввода с объекта;
- `hasFocus()` — возвращает значение `True`, если объект находится в фокусе ввода, и `False` — в противном случае;
- `grabKeyboard()` — захватывает ввод с клавиатуры;
- `ungrabKeyboard()` — освобождает ввод с клавиатуры;
- `grabMouse()` — захватывает мышь;
- `ungrabMouse()` — освобождает мышь.

26.3.2. Трансформация объекта

Произвести трансформацию графического объекта позволяют следующие методы из класса `QGraphicsItem`:

- `translate(<X>, <Y>)` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз;
- `rotate(<Угол>)` — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;

- scale(<По оси X>, <По оси Y>) — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- shear(<По горизонтали>, <По вертикали>) — сдвигает систему координат. В качестве значений указываются вещественные числа;
- resetMatrix() — отменяет все трансформации.

Несколько трансформаций можно произвести последовательно друг за другом. В этом случае следует учитывать, что порядок следования трансформаций имеет значение. Если одна и та же последовательность выполняется несколько раз, то ее можно сохранить в экземпляре класса QMatrix, а затем установить с помощью метода setMatrix(). Получить ссылку на установленную матрицу позволяет метод matrix(). Для получения ссылки на матрицу трансформации сцены предназначен метод sceneMatrix().

Вместо трансформаций, осуществляемых с помощью класса QMatrix, можно воспользоваться возможностями класса QTransform. Установить матрицу трансформаций позволяет метод setTransform(), а получить ссылку на матрицу можно с помощью метода transform(). Для получения ссылки на матрицу трансформации сцены предназначен метод sceneTransform(). Вместо установки матрицы с помощью метода setTransform() можно воспользоваться следующими методами из класса QGraphicsItem:

- setTransformOriginPoint(<X>, <Y>) — перемещает начало координат в указанную точку. Форматы метода:

  setTransformOriginPoint(<X>, <Y>)

  setTransformOriginPoint(<QPointF>)
- setRotation(<Угол>) — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;
- rotation() — возвращает текущий угол поворота;
- setScale(<Значение>) — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- scale() — возвращает текущий масштаб;
- resetTransform() — отменяет все трансформации.

26.3.3. Прочие методы

Помимо рассмотренных методов класс QGraphicsItem содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- setParentItem(<QGraphicsItem>) — задает родительский объект. Местоположение дочернего объекта задается в координатах родительского объекта;
- parentItem() — возвращает ссылку на родительский объект;
- topLevelItem() — возвращает ссылку на родительский объект верхнего уровня;
- childItems() — возвращает список с дочерними объектами;
- collidingItems([mode=IntersectsItemShape]) — возвращает список со ссылками на объекты, которые сталкиваются с данным объектом. Если таких объектов нет, то метод возвращает пустой список. Возможные значения параметра mode см. в *разд. 26.1.4*;

- `collidesWithItem(<QGraphicsItem>[, mode=IntersectsItemShape])` — возвращает значение `True`, если данный объект сталкивается с объектом, указанным в первом параметре. Возможные значения параметра `mode` см. в *разд. 26.1.4*;
- `ensureVisible()` — прокручивает область таким образом, чтобы указанный прямоугольник находился в видимой области представления. Форматы метода:

```
ensureVisible(<X>, <Y>, <Ширина>, <Высота>[, xMargin=50][,
              yMargin=50])
ensureVisible([rect=QRectF()][, xMargin=50][, yMargin=50])
```

- `update(<QRectF>)` — вызывает перерисовку указанной прямоугольной области. Форматы метода:

```
update(<X>, <Y>, <Ширина>, <Высота>)
update([rect=QRectF()])
```

26.4. Графические объекты

Вместо создания собственного объекта, путем наследования класса `QGraphicsItem`, можно воспользоваться следующими стандартными классами:

- `QGraphicsLineItem` — линия;
- `QGraphicsRectItem` — прямоугольник;
- `QGraphicsPolygonItem` — многоугольник;
- `QGraphicsEllipseItem` — эллипс;
- `QGraphicsPixmapItem` — изображение;
- `QGraphicsSimpleTextItem` — простой текст;
- `QGraphicsTextItem` — форматированный текст;
- `QGraphicsPathItem` — путь;
- `QGraphicsSvgItem` — SVG-графика.

26.4.1. Линия

Класс `QGraphicsLineItem` описывает линию. Иерархия наследования:

```
QGraphicsItem — QGraphicsLineItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsLineItem([parent=None][, scene=None])
<Объект> = QGraphicsLineItem(<X1>, <Y1>, <X2>, <Y2>[, parent=None][,
                             scene=None])
<Объект> = QGraphicsLineItem(<QLineF>[, parent=None][, scene=None])
```

В параметре `parent` можно указать ссылку на родительский объект, а в параметре `scene` — ссылку на сцену.

Класс `QGraphicsLineItem` наследует все методы из класса `QGraphicsItem` и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setLine()` — устанавливает линию. Форматы метода:

```
setLine(<X1>, <Y1>, <X2>, <Y2>)
setLine(<QLineF>)
```

- `line()` — возвращает экземпляр класса `QLineF`;
- `setPen(<QPen>)` — устанавливает перо.

26.4.2. Класс *QAbstractGraphicsShapeItem*

Класс `QAbstractGraphicsShapeItem` является базовым классом для графических фигур. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem
```

Класс `QAbstractGraphicsShapeItem` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPen(<QPen>)` — устанавливает перо;
- `setBrush(<QBrush>)` — устанавливает кисть.

26.4.3. Прямоугольник

Класс `QGraphicsRectItem` описывает прямоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsRectItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsRectItem([parent=None][, scene=None])
<Объект> = QGraphicsRectItem(<X>, <Y>, <Ширина>, <Высота>[,
                             parent=None][, scene=None])
<Объект> = QGraphicsRectItem(<QRectF>[, parent=None][, scene=None])
```

В параметре `parent` можно указать ссылку на родительский объект, а в параметре `scene` — ссылку на сцену.

Класс `QGraphicsRectItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRect()` — устанавливает прямоугольник. Форматы метода:

```
setRect(<X>, <Y>, <Ширина>, <Высота>)
setRect(<QRectF>)
```

- `rect()` — возвращает экземпляр класса `QRectF`.

26.4.4. Многоугольник

Класс `QGraphicsPolygonItem` описывает многоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsPolygonItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsPolygonItem([parent=None][, scene=None])
<Объект> = QGraphicsPolygonItem(<QPolygonF>[, parent=None][, scene=None])
```

В параметре parent можно указать ссылку на родительский объект, а в параметре scene — ссылку на сцену.

Класс QGraphicsPolygonItem наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setPolygon(<QPolygonF>) — устанавливает многоугольник;
- polygon() — возвращает экземпляр класса QPolygonF.

26.4.5. Эллипс

Класс QGraphicsEllipseItem описывает эллипс. Иерархия наследования:

QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsEllipseItem

Форматы конструктора класса:

```
<Объект> = QGraphicsEllipseItem([parent=None][, scene=None])
<Объект> = QGraphicsEllipseItem(<X>, <Y>, <Ширина>, <Высота>[,
                                parent=None][, scene=None])
<Объект> = QGraphicsEllipseItem(<QRectF>[, parent=None][, scene=None])
```

В параметре parent можно указать ссылку на родительский объект, а в параметре scene — ссылку на сцену.

Класс QGraphicsEllipseItem наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setRect() — устанавливает прямоугольник, в который необходимо вписать эллипс. Форматы метода:

  ```
  setRect(<X>, <Y>, <Ширина>, <Высота>)
  setRect(<QRectF>)
  ```

- rect() — возвращает экземпляр класса QRectF;
- setStartAngle(<Угол>) и setSpanAngle(<Угол>) — задают начальный и конечный углы сектора соответственно. Следует учитывать, что значения углов задаются в значениях 1/16 градуса. Полный круг эквивалентен значению 16 * 360. Нулевой угол находится в позиции трех часов. Положительные значения углов отсчитываются против часовой стрелки, а отрицательные по часовой стрелке;
- startAngle() и spanAngle() — возвращают значения начального и конечного углов сектора соответственно.

26.4.6. Изображение

Класс QGraphicsPixmapItem описывает изображение. Иерархия наследования:

QGraphicsItem — QGraphicsPixmapItem

Форматы конструктора класса:

```
<Объект> = QGraphicsPixmapItem([parent=None][, scene=None])
<Объект> = QGraphicsPixmapItem(<QPixmap>[, parent=None][, scene=None])
```

В параметре parent можно указать ссылку на родительский объект, а в параметре scene — ссылку на сцену.

Класс QGraphicsPixmapItem наследует все методы из класса QGraphicsItem и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setPixmap(<QPixmap>) — устанавливает изображение;
- pixmap() — возвращает экземпляр класса QPixmap;
- setOffset() — задает местоположение изображения. Форматы метода:

```
setOffset(<X>, <Y>)
setOffset(<QPointF>)
```

- offset() — возвращает местоположение изображения (экземпляр класса QPointF);
- setShapeMode(<Режим>) — задает режим определения формы изображения. В качестве параметра могут быть указаны следующие атрибуты из класса QGraphicsPixmapItem:
  - MaskShape — 0 — используется результат выполнения метода mask() из класса QPixmap (значение по умолчанию);
  - BoundingRectShape — 1 — форма определяется по контуру изображения;
  - HeuristicMaskShape — 2 — используется результат выполнения метода createHeuristicMask() из класса QPixmap;
- setTransformationMode(<Режим>) — задает режим сглаживания. В качестве параметра могут быть указаны следующие атрибуты из класса QtCore.Qt:
  - FastTransformation — 0 — сглаживание выключено (по умолчанию);
  - SmoothTransformation — 1 — сглаживание включено.

26.4.7. Простой текст

Класс QGraphicsSimpleTextItem описывает простой текст. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsSimpleTextItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsSimpleTextItem([parent=None][, scene=None])
<Объект> = QGraphicsSimpleTextItem(<Текст>[, parent=None][, scene=None])
```

В параметре parent можно указать ссылку на родительский объект, а в параметре scene — ссылку на сцену.

Класс QGraphicsSimpleTextItem наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setText(<Текст>) — задает текст;
- text() — возвращает текст;
- setFont(<QFont>) — устанавливает шрифт;
- font() — возвращает объект шрифта (экземпляр класса QFont).

26.4.8. Форматированный текст

Класс QGraphicsTextItem описывает форматированный текст. Иерархия наследования:

```
(QObject, QGraphicsItem) — QGraphicsObject — QGraphicsTextItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsTextItem([parent=None][, scene=None])
<Объект> = QGraphicsTextItem(<Текст>[, parent=None][, scene=None])
```

В параметре `parent` можно указать ссылку на родительский объект, а в параметре `scene` — ссылку на сцену.

Класс `QGraphicsTextItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPlainText(<Простой текст>)` — задает простой текст;
- `toPlainText()` — возвращает простой текст;
- `setHtml(<HTML-текст>)` — задает HTML-текст;
- `toHtml()` — возвращает HTML-текст;
- `setFont(<QFont>)` — устанавливает шрифт;
- `font()` — возвращает объект шрифта (экземпляр класса `QFont`);
- `setDefaultTextColor(<QColor>)` — задает цвет шрифта по умолчанию;
- `setTextWidth(<Ширина>)` — задает предпочитаемую ширину строки. Если текст не помещается в установленную ширину, то он будет перенесен на новую строку;
- `textWidth()` — возвращает предпочитаемую ширину текста;
- `setDocument(<QTextDocument>)` — устанавливает объект документа (экземпляр класса `QTextDocument`; см. *разд. 23.6.4*);
- `document()` — возвращает ссылку на объект документа (экземпляр класса `QTextDocument`; см. *разд. 23.6.4*);
- `setTextCursor(<QTextCursor>)` — устанавливает объект курсора (экземпляр класса `QTextCursor`; см. *разд. 23.6.5*);
- `textCursor()` — возвращает объект курсора (экземпляр класса `QTextCursor`; см. *разд. 23.6.5*);
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом. По умолчанию используется режим `NoTextInteraction`, при котором пользователь не может взаимодействовать с текстом. Допустимые режимы перечислены в *разд. 23.1* (см. описание метода `setTextInteractionFlags()`);
- `setTabChangesFocus(<Флаг>)` — если в качестве параметра указано значение `False`, то с помощью нажатия клавиши <Tab> можно вставить символ табуляции. Если указано значение `True`, то клавиша <Tab> используется для передачи фокуса;
- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, то щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке указанной страницы. Метод работает только при использовании режима `TextBrowserInteraction`.

Класс `QGraphicsTextItem` содержит следующие сигналы:

- `linkActivated(const QString&)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес;
- `linkHovered(const QString&)` — генерируется при наведении указателя мыши на гиперссылку и выведении указателя. Через параметр внутри обработчика доступен URL-адрес или пустая строка.

26.5. Группировка объектов

Объединить несколько объектов в группу позволяет класс QGraphicsItemGroup. После группировки над объектами можно выполнять различные преобразования, например перемещать одновременно все объекты группы. Иерархия наследования для класса QGraphicsItemGroup выглядит так:

```
QGraphicsItem — QGraphicsItemGroup
```

Формат конструктора класса:

```
<Объект> = QGraphicsItemGroup([parent=None][, scene=None])
```

В параметре parent можно указать ссылку на родительский объект, а в параметре scene — ссылку на сцену.

Класс QGraphicsItemGroup наследует все методы из класса QGraphicsItem и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- addToGroup(<QGraphicsItem>) — добавляет объект в группу;
- removeFromGroup(<QGraphicsItem>) — удаляет объект из группы.

Создать группу и добавить ее на сцену можно также с помощью метода createItemGroup(<Список с объектами>) из класса QGraphicsScene. Метод возвращает ссылку на группу (экземпляр класса QGraphicsItemGroup). Удалить группу со сцены позволяет метод destroyItemGroup(<QGraphicsItemGroup>) из класса QGraphicsScene.

Добавить объект в группу позволяет также метод setGroup(<QGraphicsItemGroup>) из класса QGraphicsItem. Получить ссылку на группу, к которой прикреплен объект, можно с помощью метода group() из класса QGraphicsItem. Если объект не прикреплен к группе, то метод возвращает значение None.

26.6. Эффекты

К графическим объектам можно применить различные эффекты, например изменить прозрачность или цвет, отобразить тень или сделать объект размытым. Наследуя класс QGraphicsEffect и переопределяя метод draw(), можно создать свой эффект.

Для установки эффекта и получения ссылки на него предназначены следующие методы из класса QGraphicsItem:

- setGraphicsEffect(<QGraphicsEffect>) — устанавливает эффект;
- graphicsEffect() — возвращает ссылку на эффект или значение None, если эффект не был установлен.

26.6.1. Класс *QGraphicsEffect*

Класс QGraphicsEffect является базовым классом для всех эффектов. Иерархия наследования выглядит так:

```
QObject — QGraphicsEffect
```

Формат конструктора класса:

```
QGraphicsEffect([<QObject>])
```

Класс QGraphicsEffect содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- draw(self, <QPainter>) — производит рисование эффекта. Этот абстрактный метод должен быть переопределен в производных классах;
- setEnabled(<Флаг>) — если в качестве параметра указано значение False, то эффект отключен. Значение True разрешает использование эффекта. Метод является слотом с сигнатурой setEnabled(bool);
- isEnabled() — возвращает значение True, если эффект разрешено использовать, и False — в противном случае;
- update() — вызывает перерисовку эффекта. Метод является слотом.

Класс QGraphicsEffect содержит сигнал enabledChanged(bool), который генерируется при изменении статуса эффекта. Внутри обработчика через параметр доступно значение True, если эффект разрешено использовать, и False — в противном случае.

26.6.2. Тень

Класс QGraphicsDropShadowEffect реализует тень. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsDropShadowEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsDropShadowEffect([<QObject>])
```

Класс QGraphicsDropShadowEffect наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setColor(<QColor>) — задает цвет тени. По умолчанию используется цвет QColor(63, 63, 63, 180). Метод является слотом с сигнатурой setColor(const QColor&);
- color() — возвращает цвет тени (экземпляр класса QColor);
- setBlurRadius(<Значение>) — задает радиус размытия тени. Метод является слотом с сигнатурой setBlurRadius(qreal);
- blurRadius() — возвращает радиус размытия тени;
- setOffset() — задает смещение тени. Форматы метода:

  ```
  setOffset(<По оси X>, <По оси Y>)
  setOffset(<QPointF>)
  setOffset(<Смещение>)
  ```

  Метод является слотом с сигнатурами:

  ```
  setOffset(qreal,qreal)
  setOffset(const QPointF&)
  setOffset(qreal)
  ```

- offset() — возвращает смещение тени (экземпляр класса QPointF);
- setXOffset(<Смещение>) — задает смещение по оси X. Метод является слотом с сигнатурой setXOffset(qreal);
- xOffset() — возвращает смещение по оси X;

- setYOffset(<Смещение>) — задает смещение по оси Y. Метод является слотом с сигнатурой setYOffset(qreal);
- yOffset() — возвращает смещение по оси Y.

Класс QGraphicsDropShadowEffect содержит следующие сигналы:

- colorChanged(const QColor&) — генерируется при изменении цвета тени. Внутри обработчика через параметр доступен новый цвет (экземпляр класса QColor);
- blurRadiusChanged(qreal) — генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение;
- offsetChanged(const QPointF&) — генерируется при изменении смещения. Внутри обработчика через параметр доступно новое значение (экземпляр класса QPointF).

26.6.3. Размытие

Класс QGraphicsBlurEffect реализует эффект размытия. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsBlurEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsBlurEffect([<QObject>])
```

Класс QGraphicsBlurEffect наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setBlurRadius(<Значение>) — задает радиус размытия. Метод является слотом с сигнатурой setBlurRadius(qreal);
- blurRadius() — возвращает радиус размытия.

Класс QGraphicsBlurEffect содержит сигнал blurRadiusChanged(qreal), который генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение.

26.6.4. Изменение цвета

Класс QGraphicsColorizeEffect реализует эффект изменения цвета. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsColorizeEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsColorizeEffect([<QObject>])
```

Класс QGraphicsColorizeEffect наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setColor(<QColor>) — задает цвет. По умолчанию используется цвет QColor(0, 0, 192, 255). Метод является слотом с сигнатурой setColor(const QColor&);
- color() — возвращает текущий цвет (экземпляр класса QColor);
- setStrength(<Значение>) — задает интенсивность цвета. В качестве значения указывается вещественное число от 0.0 до 1.0 (значение по умолчанию). Метод является слотом с сигнатурой setStrength(qreal);
- strength() — возвращает интенсивность цвета.

Класс QGraphicsColorizeEffect содержит следующие сигналы:

- colorChanged(const QColor&) — генерируется при изменении цвета. Внутри обработчика через параметр доступен новый цвет (экземпляр класса QColor);
- strengthChanged(qreal) — генерируется при изменении интенсивности цвета. Внутри обработчика через параметр доступно новое значение.

26.6.5. Изменение прозрачности

Класс QGraphicsOpacityEffect реализует эффект прозрачности. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsOpacityEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsOpacityEffect([<QObject>])
```

Класс QGraphicsOpacityEffect наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setOpacity(<Значение>) — задает степень прозрачности. В качестве значения указывается вещественное число от 0.0 до 1.0. По умолчанию используется значение 0.7. Метод является слотом с сигнатурой setOpacity(qreal);
- opacity() — возвращает степень прозрачности;
- setOpacityMask(<QBrush>) — задает маску. Метод является слотом с сигнатурой setOpacityMask(const QBrush&);
- opacityMask() — возвращает маску.

Класс QGraphicsOpacityEffect содержит следующие сигналы:

- opacityChanged(qreal) — генерируется при изменении степени прозрачности. Внутри обработчика через параметр доступно новое значение;
- opacityMaskChanged(const QBrush&) — генерируется при изменении маски.

26.7. Обработка событий

Первоначально события получает представление, затем представление преобразует события и передает их объекту сцены. В свою очередь сцена передает событие объекту, который способен обработать данное событие. Например, щелчок мыши передается объекту, который расположен по координатам щелчка.

Обработка событий в классе представления ничем не отличается от обычной обработки событий, рассмотренной в *гл. 21*. Обработка событий в классе объекта имеет свои отличия, которые мы и рассмотрим в этом разделе.

26.7.1. События клавиатуры

При обработке событий клавиатуры следует учитывать, что:

- графический объект должен иметь возможность принимать фокус ввода. Для этого необходимо установить флаг ItemIsFocusable, например, с помощью метода setFlag() из класса QGraphicsItem;

- объект должен быть в фокусе ввода. Методы, позволяющие управлять фокусом ввода, мы рассматривали в *разд. 26.1.5* и *26.3.1*;
- чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `ungrabKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему объекту соответственно;
- если событие обработано, то нужно вызвать метод `accept()` через объект события. В противном случае необходимо вызвать метод `ignore()`.

Для обработки событий клавиатуры следует наследовать класс, реализующий графический объект, и переопределить следующие методы:

- `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода. Через параметр `<event>` доступен экземпляр класса `QFocusEvent` (см. *разд. 21.9.1*);
- `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода. Через параметр `<event>` доступен экземпляр класса `QFocusEvent` (см. *разд. 21.9.1*);
- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши на клавиатуре. Если клавишу удерживать нажатой, то событие генерируется постоянно, пока клавиша не будет отпущена. Через параметр `<event>` доступен экземпляр класса `QKeyEvent` (см. *разд. 21.9.3*);
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой клавиши. Через параметр `<event>` доступен экземпляр класса `QKeyEvent` (см. *разд. 21.9.3*).

С помощью метода `setFocusProxy(<QGraphicsItem>)` из класса `QGraphicsItem` можно указать объект, который будет обрабатывать события клавиатуры вместо текущего объекта. Получить ссылку на такой объект позволяет метод `focusProxy()`.

26.7.2. События мыши

Для обработки нажатия кнопки мыши и перемещения мыши следует наследовать класс, реализующий графический объект, и переопределить следующие методы:

- `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши над объектом. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`. Если событие принято, то необходимо вызвать метод `accept()` через объект события, в противном случае следует вызвать метод `ignore()`. Если вызван метод `ignore()`, то методы `mouseReleaseEvent()` и `mouseMoveEvent()` вызваны не будут.

  С помощью метода `setAcceptedMouseButtons(<Кнопки>)` из класса `QGraphicsItem` можно указать кнопки, события от которых объект принимает. По умолчанию объект принимает события от всех кнопок мыши. Если в параметре `<Кнопки>` указать атрибут `NoButton` из класса `QtCore.Qt`, то обработка событий мыши будет отключена;
- `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`;
- `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью в области объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`;
- `mouseMoveEvent(self, <event>)` — вызывается при перемещении мыши с нажатой кнопкой. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`.

Класс QGraphicsSceneMouseEvent наследует все методы из классов QGraphicsSceneEvent и QEvent и добавляет следующие методы:

- pos() — возвращает экземпляр класса QPointF с координатами в пределах области объекта;
- scenePos() — возвращает экземпляр класса QPointF с координатами в пределах сцены;
- screenPos() — возвращает экземпляр класса QPoint с координатами в пределах экрана;
- lastPos() — возвращает экземпляр класса QPointF с координатами последней запомненной представлением позиции в пределах области объекта;
- lastScenePos() — возвращает экземпляр класса QPointF с координатами последней запомненной представлением позиции в пределах сцены;
- lastScreenPos() — возвращает экземпляр класса QPoint с координатами последней запомненной представлением позиции в пределах экрана;
- buttonDownPos(<Кнопка>) — возвращает экземпляр класса QPointF с координатами щелчка указанной кнопки мыши в пределах области объекта;
- buttonDownScenePos(<Кнопка>) — возвращает экземпляр класса QPointF с координатами щелчка указанной кнопки мыши в пределах сцены;
- buttonDownScreenPos(<Кнопка>) — возвращает экземпляр класса QPoint с координатами щелчка указанной кнопки мыши в пределах экрана;
- button() — позволяет определить, какая кнопка мыши вызвала событие;
- buttons() — позволяет определить все кнопки, которые нажаты одновременно;
- modifiers() — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши.

По умолчанию событие мыши перехватывает объект, над которым произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне объекта, следует захватить мышь с помощью метода grabMouse() из класса QGraphicsItem. Освободить захваченную ранее мышь позволяет метод ungrabMouse(). Получить ссылку на объект, захвативший мышь, можно с помощью метода mouseGrabberItem() из класса QGraphicsScene.

Для обработки прочих событий мыши следует наследовать класс, реализующий графический объект, и переопределить следующие методы:

- hoverEnterEvent(self, <event>) — вызывается при наведении указателя мыши на область объекта. Через параметр <event> доступен экземпляр класса QGraphicsSceneHoverEvent;
- hoverLeaveEvent(self, <event>) — вызывается, когда указатель мыши покидает область объекта. Через параметр <event> доступен экземпляр класса QGraphicsSceneHoverEvent;
- hoverMoveEvent(self, <event>) — вызывается при перемещении указателя мыши внутри области объекта. Через параметр <event> доступен экземпляр класса QGraphicsSceneHoverEvent;
- wheelEvent(self, <event>) — вызывается при повороте колесика мыши при нахождении указателя мыши над объектом. Чтобы обрабатывать событие, в любом случае следует захватить мышь. Через параметр <event> доступен экземпляр класса QGraphicsSceneWheelEvent.

Следует учитывать, что методы `hoverEnterEvent()`, `hoverLeaveEvent()` и `hoverMoveEvent()` будут вызваны только в том случае, если обработка этих событий разрешена. Чтобы разрешить обработку событий перемещения мыши, следует вызвать метод `setAcceptHoverEvents(<Флаг>)` из класса `QGraphicsItem` и передать ему значение `True`. Значение `False` запрещает обработку событий перемещения указателя. Получить текущее состояние позволяет метод `acceptHoverEvents()`.

Класс `QGraphicsSceneHoverEvent` наследует все методы из классов `QGraphicsSceneEvent` и `QEvent` и добавляет следующие методы:

- `pos()` — возвращает экземпляр класса `QPointF` с координатами в пределах области объекта;
- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами в пределах сцены;
- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана;
- `lastPos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции в пределах области объекта;
- `lastScenePos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции в пределах сцены;
- `lastScreenPos()` — возвращает экземпляр класса `QPoint` с координатами последней запомненной представлением позиции в пределах экрана;
- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты.

Класс `QGraphicsSceneWheelEvent` наследует все методы из классов `QGraphicsSceneEvent` и `QEvent` и добавляет следующие методы:

- `delta()` — возвращает расстояние поворота колесика;
- `orientation()` — возвращает ориентацию в виде значения одного из следующих атрибутов из класса `QtCore.Qt`:
  - `Horizontal` — `1` — по горизонтали;
  - `Vertical` — `2` — по вертикали;
- `pos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах области объекта;
- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах сцены;
- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши в пределах экрана;
- `buttons()` — позволяет определить кнопки, которые нажаты одновременно с поворотом колесика;
- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты.

26.7.3. Обработка перетаскивания и сброса

Прежде чем обрабатывать перетаскивание и сброс, необходимо сообщить системе, что графический объект может обрабатывать эти события. Для этого следует вызвать метод `setAcceptDrops()` из класса `QGraphicsItem` и передать ему значение `True`.

Обработка перетаскивания и сброса выполняется следующим образом:

- внутри метода `dragEnterEvent()` проверяется MIME-тип перетаскиваемых данных и действие. Если графический объект способен обработать сброс этих данных и соглашается с предложенным действием, то необходимо вызвать метод `acceptProposedAction()` через объект события. Если нужно изменить действие, то методу `setDropAction()` передается новое действие, а затем вызывается метод `accept()`, а не метод `acceptProposedAction()`;
- если необходимо ограничить область сброса некоторым участком графического объекта, то можно дополнительно определить метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области графического объекта. При согласии со сбрасыванием следует вызвать метод `accept()`;
- внутри метода `dropEvent()` производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемый объект входит в область графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`;
- `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемый объект покидает область графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`;
- `dragMoveEvent(self, <event>)` — вызывается при перетаскивании объекта внутри области графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`;
- `dropEvent(self, <event>)` — вызывается при сбрасывании объекта в области графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`.

Класс `QGraphicsSceneDragDropEvent` наследует все методы из классов `QGraphicsSceneEvent` и `QEvent` и добавляет следующие методы:

- `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе;
- `pos()` — возвращает экземпляр класса `QPointF` с координатами в пределах области объекта;
- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами в пределах сцены;
- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана;
- `possibleActions()` — возвращает комбинацию возможных действий при сбрасывании;
- `proposedAction()` — возвращает действие по умолчанию при сбрасывании;
- `acceptProposedAction()` — устанавливает флаг готовности принять перемещаемые данные и согласия с действием, возвращаемым методом `proposedAction()`;
- `setDropAction(<Действие>)` — позволяет изменить действие при сбрасывании. После изменения действия следует вызвать метод `accept()`, а не `acceptProposedAction()`;
- `dropAction()` — возвращает действие, которое должно быть выполнено при сбрасывании;

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты;
- `buttons()` — позволяет определить кнопки мыши, которые нажаты;
- `source()` — возвращает ссылку на источник события или значение `None`.

26.7.4. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы специализированному методу. Для этого необходимо переопределить метод `sceneEvent(self, <event>)` в классе объекта. Через параметр `<event>` доступен объект с дополнительной информацией о событии. Тип этого объекта отличается для разных типов событий. Внутри метода следует вернуть значение `True`, если событие обработано, и `False` — в противном случае. Если вернуть значение `True`, то специализированный метод (например, `mousePressEvent()`) вызван не будет.

Чтобы произвести фильтрацию событий какого-либо объекта, необходимо переопределить метод `sceneEventFilter(self, <QGraphicsItem>, <event>)` в классе объекта. Через параметр `<QGraphicsItem>` доступна ссылка на объект, а через параметр `<event>` — объект с дополнительной информацией о событии. Тип этого объекта отличается для разных типов событий. Внутри метода следует вернуть значение `True`, если событие обработано, и `False` — в противном случае. Если вернуть значение `True`, то объект не получит событие.

Указать, события какого объекта фильтруются, позволяют следующие методы из класса `QGraphicsItem`:

- `installSceneEventFilter(<QGraphicsItem>)` — задает объект, который будет производить фильтрацию событий текущего объекта;
- `removeSceneEventFilter(<QGraphicsItem>)` — удаляет фильтр;
- `setFiltersChildEvents(<Флаг>)` — если в качестве параметра указано значение `True`, то объект будет производить фильтрацию событий всех своих дочерних объектов.

26.7.5. Обработка изменения состояния объекта

Чтобы обработать изменение состояния объекта, следует переопределить метод `itemChange(self, <Состояние>, <Значение>)` в классе объекта. Метод должен возвращать новое значение. Через параметр `<Состояние>` доступно состояние, которое было изменено, в виде значения одного из следующих атрибутов из класса `QGraphicsItem` (перечислены только основные атрибуты; полный список смотрите в документации):

- `ItemEnabledChange` — `3` — изменилось состояние доступности;
- `ItemEnabledHasChanged` — `13` — изменилось состояние доступности. Возвращаемое значение игнорируется;
- `ItemMatrixChange` — `1` — изменилась матрица трансформаций;
- `ItemPositionChange` — `0` — изменилась позиция объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `ItemPositionHasChanged` — `9` — изменилась позиция объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;

- `ItemScenePositionHasChanged` — `27` — изменилась позиция объекта на сцене. Метод будет вызван, только если установлен флаг `ItemSendsScenePositionChanges`. Возвращаемое значение игнорируется;
- `ItemTransformChange` — `8` — изменилась матрица трансформаций. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `ItemTransformHasChanged` — `10` — изменилась матрица трансформаций. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `ItemSelectedChange` — `4` — изменилось выделение объекта;
- `ItemSelectedHasChanged` — `14` — изменилось выделение объекта. Возвращаемое значение игнорируется;
- `ItemVisibleChange` — `2` — изменилось состояние видимости объекта;
- `ItemVisibleHasChanged` — `12` — изменилось состояние видимости объекта. Возвращаемое значение игнорируется;
- `ItemCursorChange` — `17` — изменился курсор;
- `ItemCursorHasChanged` — `18` — изменился курсор. Возвращаемое значение игнорируется;
- `ItemToolTipChange` — `19` — изменилась всплывающая подсказка;
- `ItemToolTipHasChanged` — `20` — изменилась всплывающая подсказка. Возвращаемое значение игнорируется;
- `ItemFlagsChange` — `21` — изменились флаги;
- `ItemFlagsHaveChanged` — `22` — изменились флаги. Возвращаемое значение игнорируется;
- `ItemZValueChange` — `23` — изменилось положение по оси `Z`;
- `ItemZValueHasChanged` — `24` — изменилось положение по оси `Z`. Возвращаемое значение игнорируется;
- `ItemOpacityChange` — `25` — изменилась прозрачность объекта;
- `ItemOpacityHasChanged` — `26` — изменилась прозрачность объекта. Возвращаемое значение игнорируется.

Внимание!

Вызов некоторых методов из метода `itemChange()` может привести к рекурсии. За подробной информацией обращайтесь к документации по классу `QGraphicsItem`.

ГЛАВА 27

Диалоговые окна

Диалоговые окна предназначены для информирования пользователя, а также для получения данных от пользователя. В большинстве случаев диалоговые окна являются модальными (т. е. блокирующими все окна приложения или только родительское окно) и отображаются на непродолжительный промежуток времени. Для работы с диалоговыми окнами в PyQt предназначен класс `QDialog`, который предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и многое другое. Класс `QDialog` наследуют другие классы, которые реализуют готовые диалоговые окна. Например, класс `QMessageBox` предоставляет готовые диалоговые окна для вывода сообщений, класс `QInputDialog` — для ввода данных, класс `QFileDialog` — для выбора каталога или файла и т. д.

> ***Примечание***
>
> Помимо рассмотренных в этой главе классов в PyQt существуют классы для реализации диалога изменения настроек принтера (класс `QPrintDialog`) и предварительного просмотра перед печатью (класс `QPrintPreviewDialog`). За подробной информацией по этим классам обращайтесь к документации.

27.1. Пользовательские диалоговые окна

Класс `QDialog` реализует диалоговое окно. По умолчанию окно выводится с рамкой и заголовком, в котором расположены кнопки **Справка** и **Закрыть**. Размеры окна можно изменять с помощью мыши. Иерархия наследования для класса `QDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog
```

Конструктор класса `QDialog` имеет следующий формат:

```
<Объект> = QDialog([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительское окно. Если параметр не указан или имеет значение `None`, то диалоговое окно будет центрироваться относительно экрана. Если указана ссылка на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна. Кроме того, это позволяет создать модальное диалоговое окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `flags`, мы уже рассматривали в *разд. 20.2*. Тип окна по умолчанию — `Dialog`.

Класс QDialog наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- exec_() — отображает модальное диалоговое окно и возвращает код возврата в виде значения следующих атрибутов из класса QDialog:
  - Accepted — 1 — нажата кнопка **OK**;
  - Rejected — 0 — нажата кнопка **Cancel**, кнопка **Закрыть** в заголовке окна или клавиша <Esc>.

  Метод является слотом с сигнатурой int exec().

  Пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки (класс MyDialog является наследником класса QDialog, а window — ссылка на главное окно):

```
def on_clicked():
    dialog = MyDialog(window)
    result = dialog.exec_()
    if result == QtGui.QDialog.Accepted:
        print("Нажата кнопка OK")
        # Здесь получаем данные из диалогового окна
    else:
        print("Нажата кнопка Cancel")
```

- accept() — скрывает модальное диалоговое окно и устанавливает код возврата равным значению атрибута Accepted из класса QDialog. Метод является слотом. Метод accept() следует соединить с сигналом нажатия кнопки **OK**:

```
self.btnOK.clicked.connect(self.accept)
```

- reject() — скрывает модальное диалоговое окно и устанавливает код возврата равным значению атрибута Rejected из класса QDialog. Метод является слотом. Метод reject() следует соединить с сигналом нажатия кнопки **Cancel**:

```
self.btnCancel.clicked.connect(self.reject)
```

- done(<Код возврата>) — скрывает модальное диалоговое окно и устанавливает код возврата равным значению параметра. Метод является слотом с сигнатурой done(int);
- setResult(<Код возврата>) — устанавливает код возврата;
- result() — возвращает код завершения;
- setExtension(<QWidget>) — устанавливает компонент, который будет отображаться в расширенной части диалогового окна. Метод должен быть вызван, когда диалоговое окно закрыто;
- extension() — возвращает ссылку на компонент, расположенный в расширенной части диалогового окна, или значение None;
- showExtension(<Флаг>) — если в качестве параметра указано значение True, то расширенная часть диалогового окна будет отображена, а если False — то скрыта. Метод является слотом с сигнатурой showExtension(bool). Этот метод можно соединить с сигналом toggled(bool) кнопки-переключателя:

```
self.btnExt = QtGui.QPushButton("&Больше...")
self.btnExt.setCheckable(True)
self.btnExt.toggled["bool"].connect(self.showExtension)
```

- `setOrientation(<Ориентация>)` — задает местоположение расширенной части диалогового окна. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt`:
  - `Horizontal` — `1` — расширенная часть расположена справа (значение по умолчанию);
  - `Vertical` — `2` — расширенная часть расположена снизу;
- `setSizeGripEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то в правом нижнем углу диалогового окна будет отображен значок изменения размера, а если `False` — то скрыт (значение по умолчанию);
- `isSizeGripEnabled()` — возвращает значение `True`, если значок изменения размера отображается в правом нижнем углу диалогового окна, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в параметре указано значение `True`, то диалоговое окно будет отображено, а если значение `False` — то скрыто. Вместо этого метода можно воспользоваться методами `show()` и `hide()` из класса `QWidget`;
- `open()` — отображает диалоговое окно в модальном режиме. Блокируется только родительское окно, а не все окна приложения. Метод является слотом;
- `setModal(<Флаг>)` — если в качестве параметра указано значение `True`, то окно будет модальным, а если `False` — то обычным. Обратите внимание на то, что окно, открываемое с помощью метода `exec_()`, всегда будет модальным независимо от значения метода `setModal()`. Чтобы диалоговое окно было не модальным, следует отображать его с помощью метода `show()` или `setVisible()`. После вызова этих методов следует вызвать методы `raise_()` (чтобы поместить окно поверх всех окон) и `activateWindow()` (чтобы сделать окно активным (имеющим фокус ввода)).

  Указать, что окно является модальным, позволяет также метод `setWindowModality(<Флаг>)` из класса `QWidget`. В качестве параметра могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `NonModal` — `0` — окно не является модальным;
  - `WindowModal` — `1` — окно блокирует только родительские окна в пределах иерархии;
  - `ApplicationModal` — `2` — окно блокирует все окна в приложении.

  Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

  Получить текущее значение позволяет метод `windowModality()` из класса `QWidget`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` из класса `QWidget`. Метод возвращает `True`, если окно является модальным, и `False` — в противном случае.

Класс `QDialog` содержит следующие сигналы:

- `accepted()` — генерируется при установке флага `Accepted` (нажата кнопка **OK**). Сигнал не генерируется при сокрытии окна с помощью метода `hide()` или `setVisible()`;
- `rejected()` — генерируется при установке флага `Rejected` (нажата кнопка **Cancel**, кнопка **Закрыть** в заголовке окна или клавиша <Esc>). Сигнал не генерируется при сокрытии окна с помощью метода `hide()` или `setVisible()`;

- `finished(int)` — генерируется при установке кода завершения, например, пользователем или с помощью методов `accept()`, `reject()` или `done()`. Внутри обработчика через параметр доступен код завершения. Сигнал не генерируется при сокрытии окна с помощью метода `hide()` или `setVisible()`.

Для всех кнопок, добавляемых в диалоговое окно, автоматически вызывается метод `setAutoDefault()` и ему передается значение `True`. В этом случае кнопка может быть нажата с помощью клавиши <Enter>, при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>.

С помощью метода `setDefault()` можно указать кнопку по умолчанию. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, например на текстовое поле.

27.2. Класс *QDialogButtonBox*

Класс `QDialogButtonBox` реализует контейнер, в который можно добавить различные кнопки, как пользовательские, так и стандартные. Внешний вид контейнера и расположение кнопок в нем зависит от используемой операционной системы. Иерархия наследования для класса `QDialogButtonBox`:

```
(QObject, QPaintDevice) — QWidget — QDialogButtonBox
```

Форматы конструктора класса `QDialogButtonBox`:

```
<Объект> = QDialogButtonBox([parent=None])
<Объект> = QDialogButtonBox(<Ориентация>[, parent=None])
<Объект> = QDialogButtonBox(<Стандартные кнопки>[,
                            orientation=Horizontal][, parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент. Параметры `<Ориентация>` и `orientation` задают порядок расположения кнопок внутри контейнера. В качестве значения указываются атрибуты `Horizontal` (по горизонтали; значение по умолчанию) или `Vertical` (по вертикали) из класса `QtCore.Qt`. В параметре `<Стандартные кнопки>` указываются следующие атрибуты (или их комбинация через оператор `|`) из класса `QDialogButtonBox`:

- `NoButton` — кнопки не установлены;
- `Ok` — кнопка **OK** с ролью `AcceptRole`;
- `Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `Yes` — кнопка **Yes** с ролью `YesRole`;
- `YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `No` — кнопка **No** с ролью `NoRole`;
- `NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `Open` — кнопка **Open** с ролью `AcceptRole`;
- `Close` — кнопка **Close** с ролью `RejectRole`;
- `Save` — кнопка **Save** с ролью `AcceptRole`;
- `SaveAll` — кнопка **Save All** с ролью `AcceptRole`;

- `Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `Reset` — кнопка **Reset** с ролью `ResetRole`;
- `RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `Help` — кнопка **Help** с ролью `HelpRole`;
- `Abort` — кнопка **Abort** с ролью `RejectRole`;
- `Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

Поведение кнопок описывается с помощью ролей. В качестве роли можно указать следующие атрибуты из класса `QDialogButtonBox`:

- `InvalidRole` — `-1` — ошибочная роль;
- `AcceptRole` — `0` — нажатие кнопки устанавливает код возврата равным значению атрибута `Accepted`;
- `RejectRole` — `1` — нажатие кнопки устанавливает код возврата равным значению атрибута `Rejected`;
- `DestructiveRole` — `2` — кнопка для отказа от изменений;
- `ActionRole` — `3` — нажатие кнопки приводит к выполнению операции, которая не связана с закрытием окна;
- `HelpRole` — `4` — кнопка для отображения справки;
- `YesRole` — `5` — кнопка **Yes**;
- `NoRole` — `6` — кнопка **No**;
- `ResetRole` — `7` — кнопка для установки значений по умолчанию;
- `ApplyRole` — `8` — кнопка для принятия изменений.

Класс `QDialogButtonBox` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setOrientation(<Ориентация>)` — задает порядок расположения кнопок внутри контейнера. В качестве значения указываются атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) из класса `QtCore.Qt`;
- `setStandardButtons(<Стандартные кнопки>)` — устанавливает несколько стандартных кнопок. Пример:

```
self.box.setStandardButtons(QtGui.QDialogButtonBox.Ok |
                            QtGui.QDialogButtonBox.Cancel)
```

- `addButton(<QAbstractButton>, <Роль>)` — добавляет пользовательскую кнопку в контейнер. В первом параметре указывается ссылка на объект кнопки, а во втором параметре — роль. Если роль недействительна, то кнопка добавлена не будет. Пример:

```
self.btnOK = QtGui.QPushButton("&OK")
self.box.addButton(self.btnOK,
                   QtGui.QDialogButtonBox.AcceptRole)
```

- `addButton(<Стандартная кнопка>)` — добавляет стандартную кнопку в контейнер и возвращает ссылку на нее. Если кнопка не добавлена, то метод вернет значение `None`. Пример:

```
self.btnOK = self.box.addButton(QtGui.QDialogButtonBox.Ok)
```

- `addButton(<Текст>, <Роль>)` — создает кнопку, добавляет ее в контейнер и возвращает ссылку на кнопку. Параметр `<Текст>` передается конструктору класса `QPushButton`. Если роль недействительна, то кнопка добавлена не будет, и метод вернет значение `None`. Пример:

```
self.btnOK = self.box.addButton("&OK",
                    QtGui.QDialogButtonBox.AcceptRole)
```

- `button(<Стандартная кнопка>)` — возвращает ссылку на кнопку, соответствующую указанному значению, или значение `None`, если стандартная кнопка не была добавлена в контейнер ранее;
- `buttonRole(<QAbstractButton>)` — возвращает роль указанной в параметре кнопки. Если кнопка не была добавлена в контейнер, то метод возвращает значение атрибута `InvalidRole`;
- `buttons()` — возвращает список со ссылками на кнопки, которые были добавлены в контейнер;
- `removeButton(<QAbstractButton>)` — удаляет кнопку из контейнера, при этом не удаляя объект кнопки;
- `clear()` — очищает контейнер и удаляет все кнопки;
- `setCenterButtons(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопки будут выравниваться по центру контейнера.

Класс `QDialogButtonBox` содержит следующие сигналы:

- `accepted()` — генерируется при нажатии кнопки с ролью `AcceptRole` или `YesRole`. Этот сигнал можно соединить со слотом `accept()` объекта диалогового окна. Пример:

```
self.box.accepted.connect(self.accept)
```

- `rejected()` — генерируется при нажатии кнопки с ролью `RejectRole` или `NoRole`. Этот сигнал можно соединить со слотом `reject()` объекта диалогового окна. Пример:

```
self.box.rejected.connect(self.reject)
```

- `helpRequested()` — генерируется при нажатии кнопки с ролью `HelpRole`;
- `clicked(QAbstractButton *)` — генерируется при нажатии любой кнопки внутри контейнера. Внутри обработчика через параметр доступна ссылка на кнопку.

27.3. Класс *QMessageBox*

Класс `QMessageBox` реализует модальные диалоговые окна для вывода сообщений. Иерархия наследования для класса `QMessageBox` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QMessageBox
```

Форматы конструктора класса `QMessageBox`:

```
<Объект> = QMessageBox([parent=None])
<Объект> = QMessageBox(<Иконка>, <Текст заголовка>, <Текст сообщения>[,
                       buttons=NoButton][, parent=None][,
                       flags=Dialog | MSWindowsFixedSizeDialogHint])
```

Если в параметре `parent` указана ссылка на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна (см. *разд. 20.2*). В параметре `<Иконка>` могут быть указаны следующие атрибуты из класса `QMessageBox`:

- `NoIcon` — `0` — нет иконки;
- `Question` — `4` — иконка со знаком вопроса;
- `Information` — `1` — иконка информационного сообщения;
- `Warning` — `2` — иконка предупреждающего сообщения;
- `Critical` — `3` — иконка критического сообщения.

В параметре `buttons` указываются следующие атрибуты (или их комбинация через оператор `|`) из класса `QMessageBox`:

- `NoButton` — кнопки не установлены;
- `Ok` — кнопка **OK** с ролью `AcceptRole`;
- `Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `Yes` — кнопка **Yes** с ролью `YesRole`;
- `YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `No` — кнопка **No** с ролью `NoRole`;
- `NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `Open` — кнопка **Open** с ролью `AcceptRole`;
- `Close` — кнопка **Close** с ролью `RejectRole`;
- `Save` — кнопка **Save** с ролью `AcceptRole`;
- `SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `Reset` — кнопка **Reset** с ролью `ResetRole`;
- `RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `Help` — кнопка **Help** с ролью `HelpRole`;
- `Abort` — кнопка **Abort** с ролью `RejectRole`;
- `Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

Поведение кнопок описывается с помощью ролей. В качестве роли можно указать следующие атрибуты из класса `QMessageBox`:

- `InvalidRole` — `-1` — ошибочная роль;
- `AcceptRole` — `0` — нажатие кнопки устанавливает код возврата равным значению атрибута `Accepted`;
- `RejectRole` — `1` — нажатие кнопки устанавливает код возврата равным значению атрибута `Rejected`;
- `DestructiveRole` — `2` — кнопка для отказа от изменений;
- `ActionRole` — `3` — нажатие кнопки приводит к выполнению операции, которая не связана с закрытием окна;
- `HelpRole` — `4` — кнопка для отображения справки;
- `YesRole` — `5` — кнопка **Yes**;
- `NoRole` — `6` — кнопка **No**;
- `ResetRole` — `7` — кнопка для установки значений по умолчанию;
- `ApplyRole` — `8` — кнопка для принятия изменений.

После создания экземпляра класса следует вызвать метод `exec_()` для отображения окна. Метод возвращает код нажатой кнопки. Пример:

```
dialog = QtGui.QMessageBox(QtGui.QMessageBox.Critical,
                           "Текст заголовка", "Текст сообщения",
                           buttons = QtGui.QMessageBox.Ok |
                           QtGui.QMessageBox.Cancel,
                           parent=window)
result = dialog.exec_()
```

27.3.1. Основные методы и сигналы

Класс `QMessageBox` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setIcon(<Иконка>)` — устанавливает стандартную иконку;
- `setIconPixmap(<QPixmap>)` — устанавливает пользовательскую иконку. В качестве параметра указывается экземпляр класса `QPixmap`;
- `setWindowTitle(<Текст заголовка>)` — задает текст заголовка окна;
- `setText(<Текст сообщения>)` — задает текст сообщения. Можно указать как обычный текст, так и текст в формате HTML. Перенос строки в обычной строке осуществляется с помощью символа `\n`, а в строке в формате HTML с помощью тега `<br>`;
- `setInformativeText(<Текст>)` — задает дополнительный текст сообщения, который отображается под обычным текстом сообщения. Можно указать как обычный текст, так и текст в формате HTML;
- `setDetailedText(<Текст>)` — задает текст описания деталей сообщения. Если текст задан, то будет добавлена кнопка **Show Details**, с помощью которой можно отобразить скрытую панель с описанием;
- `setTextFormat(<Режим>)` — задает режим отображения текста сообщения. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- PlainText — 0 — простой текст;
- RichText — 1 — форматированный текст;
- AutoText — 2 — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим RichText, в противном случае — режим PlainText;

♦ setStandardButtons(<Стандартные кнопки>) — устанавливает несколько стандартных кнопок. Пример:

```
dialog.setStandardButtons(QtGui.QMessageBox.Ok |
                          QtGui.QMessageBox.Cancel)
```

♦ addButton(<QAbstractButton>, <Роль>) — добавляет пользовательскую кнопку в окно. В первом параметре указывается ссылка на объект кнопки, а во втором параметре — роль. Пример:

```
btnYes = QtGui.QPushButton("&Да")
dialog.addButton(btnYes, QtGui.QMessageBox.AcceptRole)
```

♦ addButton(<Стандартная кнопка>) — добавляет стандартную кнопку в окно и возвращает ссылку на нее. Если кнопка не добавлена, то метод вернет значение None. Пример:

```
btnSave = dialog.addButton(QtGui.QMessageBox.Save)
```

♦ addButton(<Текст>, <Роль>) — создает кнопку, добавляет ее в окно и возвращает ссылку на кнопку. Параметр <Текст> передается конструктору класса QPushButton. Если роль недействительна, то кнопка добавлена не будет, и метод вернет значение None. Пример:

```
btnYes = dialog.addButton("&Да", QtGui.QMessageBox.AcceptRole)
```

♦ setDefaultButton() — задает кнопку по умолчанию. Форматы метода:

```
setDefaultButton(<Стандартная кнопка>)
setDefaultButton(<QPushButton>)
```

♦ setEscapeButton() — задает кнопку, которая будет нажата при нажатии клавиши <Esc>. Форматы метода:

```
setEscapeButton(<Стандартная кнопка>)
setEscapeButton(<QAbstractButton>)
```

♦ clickedButton() — возвращает ссылку на кнопку, которая была нажата, или значение None;

♦ button(<Стандартная кнопка>) — возвращает ссылку на кнопку, соответствующую указанному значению, или значение None, если стандартная кнопка не была добавлена в окно ранее;

♦ buttonRole(<QAbstractButton>) — возвращает роль указанной в параметре кнопки. Если кнопка не была добавлена в окно, то метод возвращает значение атрибута InvalidRole;

♦ buttons() — возвращает список со ссылками на кнопки, которые были добавлены в окно;

♦ removeButton(<QAbstractButton>) — удаляет кнопку из окна, при этом не удаляя объект кнопки.

Класс QMessageBox содержит сигнал buttonClicked(QAbstractButton *), который генерируется при нажатии кнопки в окне. Внутри обработчика через параметр доступна ссылка на кнопку.

27.3.2. Окно для вывода обычного сообщения

Помимо рассмотренных методов класс `QMessageBox` содержит несколько статических методов, реализующих готовые диалоговые окна. Для вывода модального окна с информационным сообщением предназначен статический метод `information()`. Формат метода:

```
information(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
            buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `information()` возвращает код нажатой кнопки. Пример:

```
QtGui.QMessageBox.information(window, "Текст заголовка",
                              "Текст сообщения",
                              buttons=QtGui.QMessageBox.Close,
                              defaultButton=QtGui.QMessageBox.Close)
```

Результат выполнения этого кода показан на рис. 27.1.

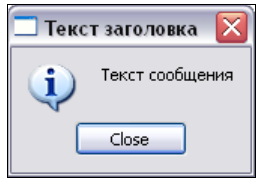

Рис. 27.1. Окно для вывода обычного сообщения

27.3.3. Окно запроса подтверждения

Для вывода модального окна с запросом подтверждения каких-либо действий предназначен статический метод `question()`. Формат метода:

```
question(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
         buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `question()` возвращает код нажатой кнопки. Пример:

```
result = QtGui.QMessageBox.question(window, "Текст заголовка",
                "Вы действительно хотите выполнить действие?",
                buttons=QtGui.QMessageBox.Yes | QtGui.QMessageBox.No |
                QtGui.QMessageBox.Cancel,
                defaultButton=QtGui.QMessageBox.Cancel)
```

Результат выполнения этого кода показан на рис. 27.2.

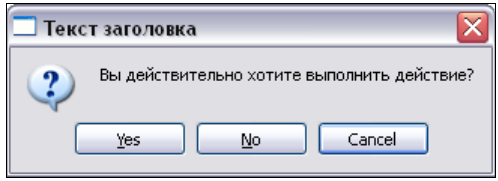

Рис. 27.2. Окно запроса подтверждения

27.3.4. Окно для вывода предупреждающего сообщения

Для вывода модального окна с предупреждающим сообщением предназначен статический метод `warning()`. Формат метода:

```
warning(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
        buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `warning()` возвращает код нажатой кнопки. Пример:

```
result = QtGui.QMessageBox.warning(window, "Текст заголовка",
                "Действие может быть опасным. Продолжить?",
                buttons=QtGui.QMessageBox.Yes | QtGui.QMessageBox.No |
                QtGui.QMessageBox.Cancel,
                defaultButton=QtGui.QMessageBox.Cancel)
```

Результат выполнения этого кода показан на рис. 27.3.

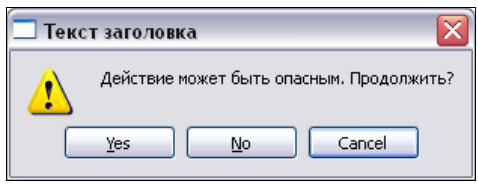

Рис. 27.3. Окно для вывода предупреждающего сообщения

27.3.5. Окно для вывода критического сообщения

Для вывода модального окна с критическим сообщением предназначен статический метод `critical()`. Формат метода:

```
critical(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
         buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через

оператор |). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `critical()` возвращает код нажатой кнопки. Пример:

```
QtGui.QMessageBox.critical(window, "Текст заголовка",
      "Программа выполнила недопустимую ошибку и будет закрыта",
      buttons=QtGui.QMessageBox.Ok,
      defaultButton=QtGui.QMessageBox.Ok)
```

Результат выполнения этого кода показан на рис. 27.4.

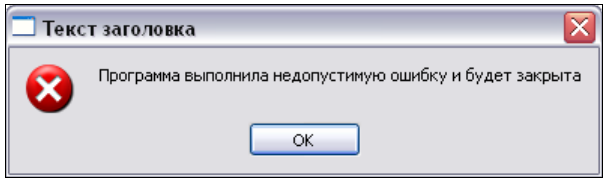

Рис. 27.4. Окно для вывода критического сообщения

27.3.6. Окно "О программе"

Для вывода модального окна с описанием программы и информацией об авторских правах предназначен статический метод `about()`. Формат метода:

```
about(<Родитель>, <Текст заголовка>, <Текст сообщения>)
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Слева от текста сообщения отображается иконка приложения (см. *разд. 20.9*), если она была установлена. Пример:

```
QtGui.QMessageBox.about(window, "Текст заголовка", "Описание программы")
```

27.3.7. Окно "About Qt"

Для вывода модального окна с описанием используемой версии библиотеки Qt предназначен статический метод `aboutQt()`. Формат метода:

```
aboutQt(<Родитель>[, title=QString()])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. В параметре `title` можно указать текст, выводимый в заголовке окна. Если параметр не указан, то выводится заголовок "About Qt". Пример:

```
QtGui.QMessageBox.aboutQt(window)
```

27.4. Класс *QInputDialog*

Класс `QInputDialog` реализует модальные диалоговые окна для ввода различных данных. Иерархия наследования для класса `QInputDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QInputDialog
```

Формат конструктора класса QInputDialog:

```
<Объект> = QInputDialog([parent=None][, flags=0])
```

Если в параметре parent указана ссылка на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр flags задает тип окна (см. *разд. 20.2*).

После создания экземпляра класса следует вызвать метод exec_() для отображения окна. Метод возвращает код возврата в виде значения следующих атрибутов из класса QDialog: Accepted или Rejected. Пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки:

```
def on_clicked():
    dialog = QtGui.QInputDialog(window)
    result = dialog.exec_()
    if result == QtGui.QDialog.Accepted:
        print("Нажата кнопка OK")
        # Здесь получаем данные из диалогового окна
    else:
        print("Нажата кнопка Cancel")
```

27.4.1. Основные методы и сигналы

Класс QInputDialog наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- setLabelText(<Текст>) — задает текст, отображаемый над полем ввода;
- setOkButtonText(<Текст>) — задает текст, отображаемый на кнопке **OK**:

  ```
  dialog.setOkButtonText("&Ввод")
  ```

- setCancelButtonText(<Текст>) — задает текст, отображаемый на кнопке **Cancel**. Пример:

  ```
  dialog.setCancelButtonText("&Отмена")
  ```

- setInputMode(<Режим>) — задает режим ввода данных. В качестве параметра указываются следующие атрибуты из класса QInputDialog:
  - TextInput — 0 — ввод текста;
  - IntInput — 1 — ввод целого числа;
  - DoubleInput — 2 — ввод вещественного числа;
- setTextEchoMode(<Режим>) — задает режим отображения текста в поле. Могут быть указаны следующие атрибуты из класса QLineEdit:
  - Normal — 0 — показывать символы как они были введены;
  - NoEcho — 1 — не показывать вводимые символы;
  - Password — 2 — вместо символов указывать символ *;
  - PasswordEchoOnEdit — 3 — показывать символы при вводе, а при потере фокуса отображать символ *;
- setTextValue(<Текст>) — задает текст по умолчанию, отображаемый в текстовом поле;

- `textValue()` — возвращает текст, введенный в текстовое поле;
- `setIntValue(<Значение>)` — задает целочисленное значение при использовании режима `IntInput`;
- `intValue()` — возвращает целочисленное значение, введенное в поле, при использовании режима `IntInput`;
- `setIntRange(<Минимум>, <Максимум>)`, `setIntMinimum(<Минимум>)` и `setIntMaximum(<Максимум>)` — задают диапазон допустимых целочисленных значений при использовании режима `IntInput`;
- `setIntStep(<Шаг>)` — задает шаг приращения значения при нажатии кнопок со стрелками в правой части поля при использовании режима `IntInput`;
- `setDoubleValue(<Значение>)` — задает вещественное значение при использовании режима `DoubleInput`;
- `doubleValue()` — возвращает вещественное значение, введенное в поле, при использовании режима `DoubleInput`;
- `setDoubleRange(<Минимум>, <Максимум>)`, `setDoubleMinimum(<Минимум>)` и `setDoubleMaximum(<Максимум>)` — задают диапазон допустимых вещественных значений при использовании режима `DoubleInput`;
- `setDoubleDecimals(<Значение>)` — задает количество цифр после десятичной точки при использовании режима `DoubleInput`;
- `setComboBoxItems(<Список строк>)` — задает строки, которые будут отображаться в раскрывающемся списке;
- `setComboBoxEditable(<Флаг>)` — если в качестве параметра указано значение `True`, то значения из раскрывающегося списка можно будет редактировать;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, то производит установку указанной в первом параметре опции, а если `False` — то сбрасывает опцию. В первом параметре можно указать следующие атрибуты из класса `QInputDialog`:
  - `NoButtons` — `1` — кнопки не отображаются;
  - `UseListViewForComboBoxItems` — `2` — для отображения списка строк будет использоваться класс `QListView`, а не `QComboBox`;
- `setOptions(<Опции>)` — устанавливает несколько опций (см. описание метода `setOption()`) сразу.

Класс `QInputDialog` содержит следующие сигналы:

- `textValueChanged(const QString&)` — генерируется при изменении значения в текстовом поле. Внутри обработчика через параметр доступно новое значение. Сигнал генерируется при использовании режима `TextInput`;
- `textValueSelected(const QString&)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение. Сигнал генерируется при использовании режима `TextInput`;
- `intValueChanged(int)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение. Сигнал генерируется при использовании режима `IntInput`;

- `intValueSelected(int)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение. Сигнал генерируется при использовании режима `IntInput`;
- `doubleValueChanged(double)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение. Сигнал генерируется при использовании режима `DoubleInput`;
- `doubleValueSelected(double)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение. Сигнал генерируется при использовании режима `DoubleInput`.

27.4.2. Окно для ввода строки

Помимо рассмотренных методов класс `QInputDialog` содержит несколько статических методов, реализующих готовые диалоговые окна. Окно для ввода обычной строки или пароля реализуется с помощью статического метода `getText()`. Формат метода:

```
getText(<Родитель>, <Текст заголовка>, <Текст подсказки>[,
        mode=Normal][, text=QString()][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `mode` задает режим отображения текста в поле (см. описание метода `setTextEchoMode()` в *разд. 27.4.1*). Параметр `text` устанавливает значение поля по умолчанию, а в параметре `flags` можно указать тип окна. Метод `getText()` возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй элемент — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна. Пример:

```
s, ok = QtGui.QInputDialog.getText(window, "Это заголовок окна",
                                   "Это текст подсказки",
                                   text="Значение по умолчанию")
if ok:
    print("Введено значение:", s)
```

Результат выполнения этого кода показан на рис. 27.5.

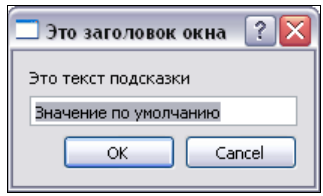

Рис. 27.5. Окно для ввода строки

27.4.3. Окно для ввода целого числа

Окно для ввода целого числа реализуется с помощью статических методов `getInt()` или `getInteger()`.

Форматы методов:

```
getInt(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
       min=-2147483647][, max=2147483647][, step=1][, flags=0])
getInteger(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
       min=-2147483647][, max=2147483647][, step=1][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, а параметр `step` — шаг приращения. Параметр `flags` позволяет указать тип окна. Методы возвращают кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй элемент — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна. Пример:

```
n, ok = QtGui.QInputDialog.getInt(window, "Это заголовок окна",
                                  "Это текст подсказки",
                                  value=50, min=0, max=100, step=2)
if ok:
    print("Введено значение:", n)
```

Результат выполнения этого кода показан на рис. 27.6.

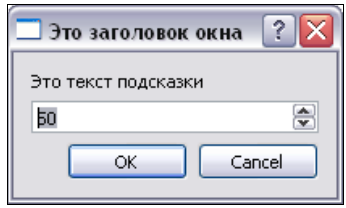

Рис. 27.6. Окно для ввода целого числа

27.4.4. Окно для ввода вещественного числа

Окно для ввода вещественного числа реализуется с помощью статического метода `getDouble()`. Формат метода:

```
getDouble(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
          min=-2147483647][, max=2147483647][, decimals=1][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, а параметр `decimals` — количество цифр после десятичной точки. Параметр `flags` позволяет указать тип окна. Метод возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй элемент — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна.

Пример:

```
n, ok = QtGui.QInputDialog.getDouble(window, "Это заголовок окна",
                                      "Это текст подсказки",
                                      value=50.0, min=0.0, max=100.0,
                                      decimals=2)
if ok:
    print("Введено значение:", n)
```

27.4.5. Окно для выбора пункта из списка

Окно для выбора пункта из списка реализуется с помощью статического метода `getItem()`. Формат метода:

```
getItem(<Родитель>, <Текст заголовка>, <Текст подсказки>,
        <Список строк>[, current=0][, editable=True][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `current` устанавливает индекс пункта, выбранного по умолчанию. Если в параметре `editable` указано значение `True`, то текст пункта списка можно редактировать. Параметр `flags` позволяет указать тип окна. Метод возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступен текст выбранного пункта в списке, а через второй элемент — значение `True`, если была нажата кнопка **OK**, или значение `False`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна. Пример:

```
s, ok = QtGui.QInputDialog.getItem(window, "Это заголовок окна",
        "Это текст подсказки", ["Пункт 1", "Пункт 2", "Пункт 3"],
        current=1, editable=False)
if ok:
    print("Текст выбранного пункта:", s)
```

Результат выполнения этого кода показан на рис. 27.7.

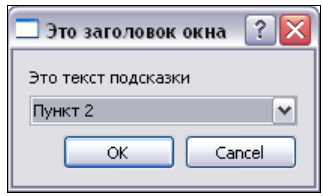

Рис. 27.7. Окно для выбора пункта из списка

27.5. Класс *QFileDialog*

Класс `QFileDialog` реализует модальные диалоговые окна для выбора файла или каталога. Иерархия наследования для класса `QFileDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QFileDialog
```

Форматы конструктора класса QFileDialog:

```
<Объект> = QFileDialog(<Родитель>, <Тип окна>)
<Объект> = QFileDialog([parent=None][, caption=QString()][,
                        directory=QString()][, filter=QString()])
```

Если в параметрах <Родитель> и parent указана ссылка на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр <Тип окна> задает тип окна (см. *разд. 20.2*). Необязательный параметр caption позволяет указать заголовок окна, параметр directory — начальный каталог, а параметр filter — ограничивает отображение файлов указанным фильтром (например, "Images (*.png *.jpg)").

После создания экземпляра класса следует вызвать метод exec_() для отображения окна. Метод возвращает код возврата в виде значения следующих атрибутов из класса QDialog: Accepted или Rejected.

27.5.1. Основные методы и сигналы

Класс QFileDialog наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- setAcceptMode(<Тип>) — задает тип окна. В качестве параметра указываются следующие атрибуты из класса QFileDialog:
  - AcceptOpen — 0 — окно для открытия файла (по умолчанию);
  - AcceptSave — 1 — окно для сохранения файла;
- setViewMode(<Режим>) — задает режим отображения файлов. В качестве параметра указываются следующие атрибуты из класса QFileDialog:
  - Detail — 0 — отображается детальная информация о файлах;
  - List — 1 — отображается только иконка и название файла;
- setFileMode(<Тип>) — задает тип возвращаемого значения. В качестве параметра указываются следующие атрибуты из класса QFileDialog:
  - AnyFile — 0 — имя файла независимо от того, существует он или нет;
  - ExistingFile — 1 — имя существующего файла;
  - Directory — 2 — имя каталога;
  - ExistingFiles — 3 — список из нескольких существующих файлов. Несколько файлов можно выбрать, удерживая нажатой клавишу <Ctrl>;
- setOption(<Опция>[, on=True]) — если во втором параметре указано значение True, то производит установку указанной в первом параметре опции, а если False — то сбрасывает опцию. В первом параметре можно указать следующие атрибуты из класса QFileDialog (перечислены только основные атрибуты; полный список смотрите в документации):
  - ShowDirsOnly — отображать только названия каталогов. Опция работает только при использовании типа возвращаемого значения Directory;
  - DontConfirmOverwrite — не спрашивать разрешения, если выбран существующий файл;

- `ReadOnly` — режим только для чтения;
- `HideNameFilterDetails` — скрывает детали фильтра;

- `setOptions(<Опции>)` — позволяет установить сразу несколько опций;
- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, то будет использоваться режим только для чтения;
- `setNameFilterDetailsVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то детали фильтра будут скрыты;
- `setDirectory()` — задает отображаемый каталог. Форматы метода:

  ```
  setDirectory(<Путь>)
  setDirectory(<QDir>)
  ```

- `directory()` — возвращает экземпляр класса `QDir` с путем к отображаемому каталогу;
- `setNameFilter(<Фильтр>)` — устанавливает фильтр. Чтобы установить несколько фильтров, необходимо перечислить их через две точки с запятой. Пример:

  ```
  dialog.setNameFilter("All (*);;Images (*.png *.jpg)")
  ```

- `setNameFilters(<Список фильтров>)` — устанавливает несколько фильтров:

  ```
  dialog.setNameFilters(["All (*)", "Images (*.png *.jpg)"])
  ```

- `selectFile(<Название файла>)` — выбирает указанный файл;
- `selectedFiles()` — возвращает список с выбранными файлами;
- `setDefaultSuffix(<Расширение>)` — задает расширение, которое добавляется к файлу при отсутствии указанного расширения;
- `setHistory(<Список>)` — задает список истории;
- `setSidebarUrls(<Список с QUrl>)` — задает список URL-адресов, отображаемый на боковой панели. Пример:

  ```
  dialog.setSidebarUrls([QtCore.QUrl.fromLocalFile("C:\\book"),
            QtCore.QUrl.fromLocalFile("C:\\book\\eclipse")])
  ```

- `setLabelText(<Тип надписи>, <Текст>)` — позволяет изменить текст указанной надписи. В первом параметре указываются следующие атрибуты из класса `QFileDialog`:
  - `LookIn` — `0` — надпись слева от списка с каталогами;
  - `FileName` — `1` — надпись слева от поля с названием файла;
  - `FileType` — `2` — надпись слева от поля с типами файлов;
  - `Accept` — `3` — надпись на кнопке, нажатие которой приведет к принятию диалога (по умолчанию **Open** или **Save**);
  - `Reject` — `4` — надпись на кнопке, нажатие которой приведет к отмене диалога (по умолчанию **Cancel**);
- `saveState()` — возвращает экземпляр класса `QByteArray` с текущими настройками;
- `restoreState(<QByteArray>)` — восстанавливает настройки и возвращает статус успешности выполненной операции.

Класс `QFileDialog` содержит следующие сигналы:

- `currentChanged(const QString&)` — генерируется при изменении текущего файла. Внутри обработчика через параметр доступен новый путь;

- `directoryEntered(const QString&)` — генерируется при изменении каталога. Внутри обработчика через параметр доступен новый путь;
- `filterSelected(const QString&)` — генерируется при изменении фильтра. Внутри обработчика через параметр доступен новый фильтр;
- `fileSelected(const QString&)` — генерируется при выборе файла и принятии диалога. Внутри обработчика через параметр доступен путь;
- `filesSelected(const QStringList&)` — генерируется при выборе нескольких файлов и принятии диалога. Внутри обработчика через параметр доступен список с путями.

27.5.2. Окно для выбора каталога

Помимо рассмотренных методов класс `QFileDialog` содержит несколько статических методов, реализующих готовые диалоговые окна. Окно для выбора каталога реализуется с помощью статического метода `getExistingDirectory()`. Формат метода:

```
getExistingDirectory([parent=None][, caption=QString()][,
                     directory=QString()][, options=ShowDirsOnly])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `directory` задает текущий каталог, а параметр `options` — устанавливает опции (см. описание метода `setOption()` в *разд. 27.5.1*). Метод возвращает выбранный каталог или пустую строку. Пример:

```
dir = QtGui.QFileDialog.getExistingDirectory(parent=window,
                          directory=QtCore.QDir.currentPath())
```

Результат выполнения этого кода показан на рис. 27.8.

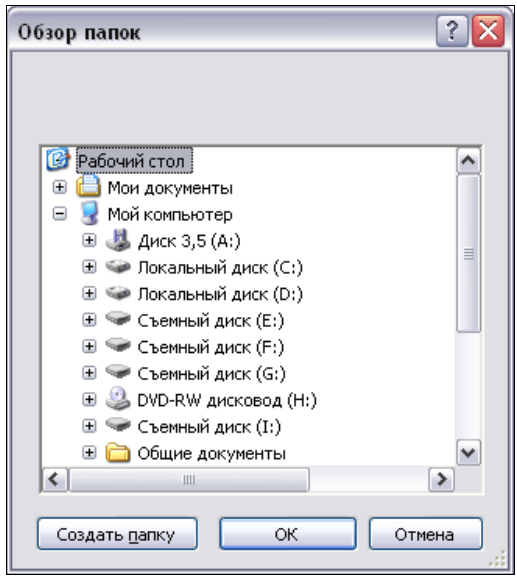

Рис. 27.8. Окно для выбора каталога

27.5.3. Окна для открытия файла

Окно для открытия одного файла реализуется с помощью статического метода `getOpenFileName()`. Формат метода:

```
getOpenFileName([parent=None][, caption=QString()][,
                directory=QString()][, filter=QString()][,
                options=0])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `caption` задает текст заголовка окна, параметр `directory` — текущий каталог, параметр `filter` — фильтр, а параметр `options` — устанавливает опции (см. описание метода `setOption()` в *разд. 27.5.1*). Метод возвращает выбранный файл или пустую строку. Пример:

```
f = QtGui.QFileDialog.getOpenFileName(parent=window,
    caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
    filter="All (*);;Images (*.png *.jpg)")
```

Результат выполнения этого кода показан на рис. 27.9.

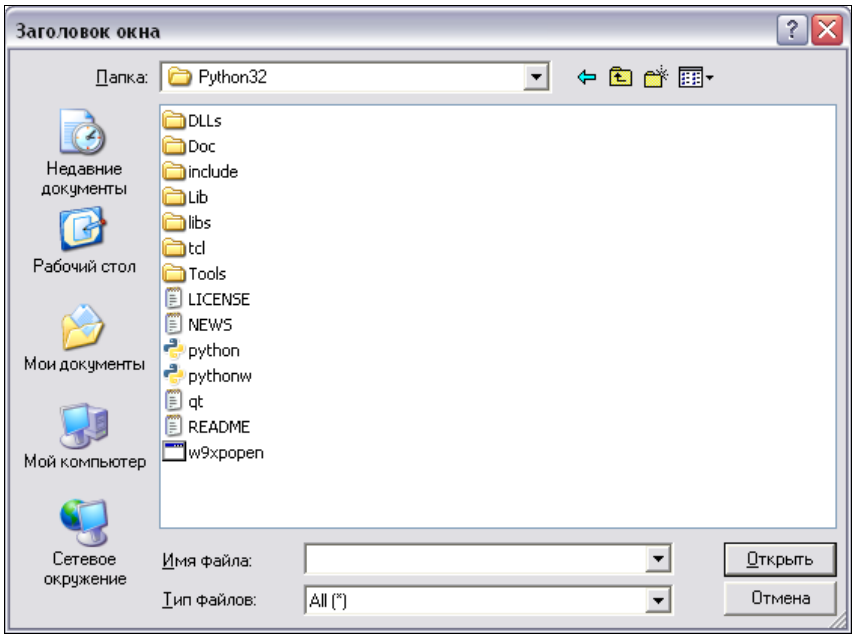

Рис. 27.9. Окно для открытия файла

Кроме того, для открытия одного файла можно воспользоваться статическим методом `getOpenFileNameAndFilter()`. Формат метода:

```
getOpenFileNameAndFilter([parent=None][, caption=QString()][,
                         directory=QString()][, filter=QString()][,
                         initialFilter=QString()][, options=0])
```

С помощью параметра `initialFilter` можно сделать указанный фильтр текущим. Метод возвращает кортеж из двух элементов. Первый элемент кортежа содержит выбранный файл или пустую строку, а второй элемент — фильтр.

Пример:

```
f, filter_ = QtGui.QFileDialog.getOpenFileNameAndFilter(
    parent=window,
    caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
    filter="All (*);;Images (*.png *.jpg)",
    initialFilter="Images (*.png *.jpg)")
```

Окно для открытия нескольких файлов реализуется с помощью статического метода `getOpenFileNames()`. Формат метода:

```
getOpenFileNames([parent=None][, caption=QString()][,
                 directory=QString()][, filter=QString()][, options=0])
```

Метод возвращает список с выбранными файлами или пустой список. Пример:

```
arr = QtGui.QFileDialog.getOpenFileNames(parent=window,
      caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
      filter="All (*);;Images (*.png *.jpg)")
```

Кроме того, для открытия нескольких файлов можно воспользоваться статическим методом `getOpenFileNamesAndFilter()`. Формат метода:

```
getOpenFileNamesAndFilter([parent=None][, caption=QString()][,
                          directory=QString()][, filter=QString()][,
                          initialFilter=QString()][, options=0])
```

С помощью параметра `initialFilter` можно сделать указанный фильтр текущим. Метод возвращает кортеж из двух элементов. Первый элемент кортежа содержит список с выбранными файлами или пустой список, а второй элемент — фильтр.

27.5.4. Окна для сохранения файла

Окно для сохранения файла реализуется с помощью статического метода `getSaveFileName()`. Формат метода:

```
getSaveFileName([parent=None][, caption=QString()][,
                directory=QString()][, filter=QString()][, options=0])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `caption` задает текст заголовка окна, параметр `directory` — текущий каталог, параметр `filter` — фильтр, а параметр `options` — устанавливает опции (см. описание метода `setOption()` в *разд. 27.5.1*). Метод возвращает выбранный файл или пустую строку. Пример:

```
f = QtGui.QFileDialog.getSaveFileName(parent=window,
    caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
    filter="All (*);;Images (*.png *.jpg)")
```

Результат выполнения этого кода показан на рис. 27.10.

Кроме того, для сохранения файла можно воспользоваться статическим методом `getSaveFileNameAndFilter()`. Формат метода:

```
getSaveFileNameAndFilter([parent=None][, caption=QString()][,
                         directory=QString()][, filter=QString()][,
                         initialFilter=QString()][, options=0])
```

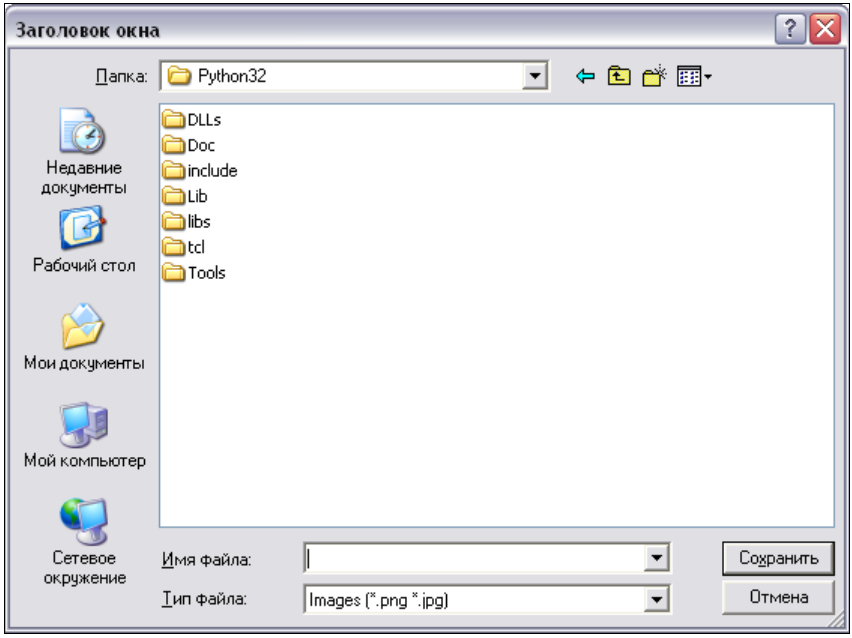

Рис. 27.10. Окно для сохранения файла

С помощью параметра `initialFilter` можно сделать указанный фильтр текущим. Метод возвращает кортеж из двух элементов. Первый элемент кортежа содержит выбранный файл или пустую строку, а второй элемент — фильтр. Пример:

```
f, filter_ = QtGui.QFileDialog.getSaveFileNameAndFilter(
    parent=window,
    caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
    filter="All (*);;Images (*.png *.jpg)",
    initialFilter="Images (*.png *.jpg)")
```

27.6. Окно для выбора цвета

Окно для выбора цвета (рис. 27.11) реализуется с помощью статического метода `getColor()` из класса `QColorDialog`. Форматы метода:

```
getColor([initial=white][, parent=None])
getColor(<QColor>, <Родитель>, <Текст заголовка>[, options=0])
```

В параметрах `<Родитель>` и `parent` указывается ссылка на родительское окно или значение `None`. Параметры `initial` и `<QColor>` задают начальный цвет. В параметре `options` могут быть указаны следующие атрибуты (или их комбинация) из класса `QColorDialog`:

- `ShowAlphaChannel` — пользователь может выбрать значение альфа-канала;
- `NoButtons` — кнопки **OK** и **Cancel** не отображаются;
- `DontUseNativeDialog`.

Метод возвращает экземпляр класса `QColor`. Если пользователь нажимает кнопку **Cancel**, то объект будет невалидным.

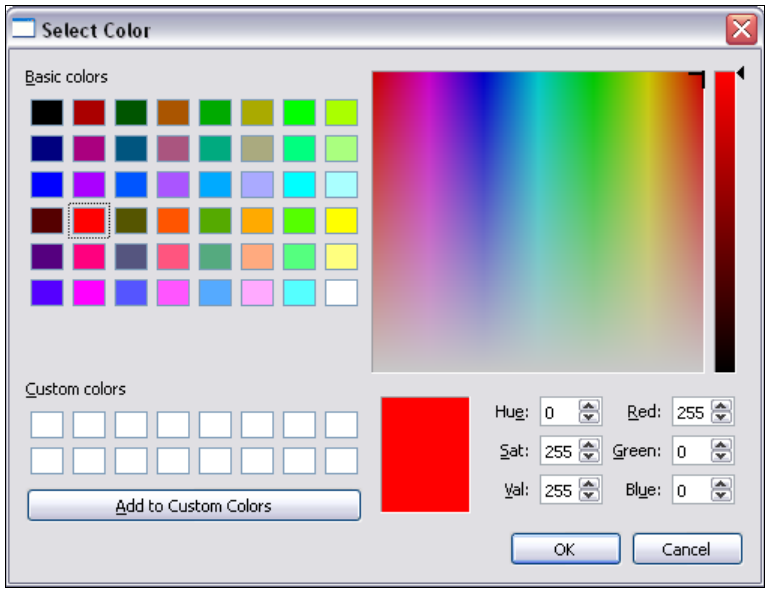

Рис. 27.11. Окно для выбора цвета

Пример:

```
color = QtGui.QColorDialog.getColor(QtGui.QColor("#ff0000"),
        window, "Заголовок окна", QtGui.QColorDialog.ShowAlphaChannel)
if color.isValid():
    print(color.red(), color.green(), color.blue(), color.alpha())
```

Окно для выбора цвета с альфа-каналом реализуется также с помощью статического метода `getRgba()` из класса `QColorDialog`. Формат метода:

```
getRgba([initial=4294967295][, parent=None])
```

В параметре `initial` указывается целочисленное значение начального цвета. Метод возвращает кортеж из двух элементов: `(<Цвет>, <Статус>)`. Если второй элемент содержит значение `True`, то первый элемент будет содержать целочисленное значение выбранного цвета. Пример:

```
(n, ok) = QtGui.QColorDialog.getRgba(
          initial=QtGui.QColor("#ff0000").rgba(), parent=window)
if ok:
    color = QtGui.QColor.fromRgba(n)
    print(color.red(), color.green(), color.blue(), color.alpha())
```

27.7. Окно для выбора шрифта

Окно для выбора шрифта реализуется с помощью статического метода `getFont()` из класса `QFontDialog`. Форматы метода:

```
getFont([parent=None])
getFont(<QFont>[, parent=None])
```

```
getFont(<QFont>, <Родитель>, <Текст заголовка>)
getFont(<QFont>, <Родитель>, <Текст заголовка>, <Опции>)
```

В параметрах `<Родитель>` и `parent` указывается ссылка на родительское окно или значение `None`. Параметр `<QFont>` задает начальный шрифт. В параметре `<Опции>` могут быть указаны атрибуты `NoButtons` и `DontUseNativeDialog` из класса `QFontDialog`. Метод возвращает кортеж из двух элементов: `(<QFont>, <Статус>)`. Если второй элемент содержит значение `True`, то первый элемент будет содержать экземпляр класса `QFont` с выбранным шрифтом. Пример:

```
(font, ok) = QtGui.QFontDialog.getFont(QtGui.QFont("Tahoma", 16),
                                       window, "Заголовок окна")
if ok:
    print(font.family(), font.pointSize(), font.weight(),
          font.italic(), font.underline())
```

Результат выполнения этого кода показан на рис. 27.12.

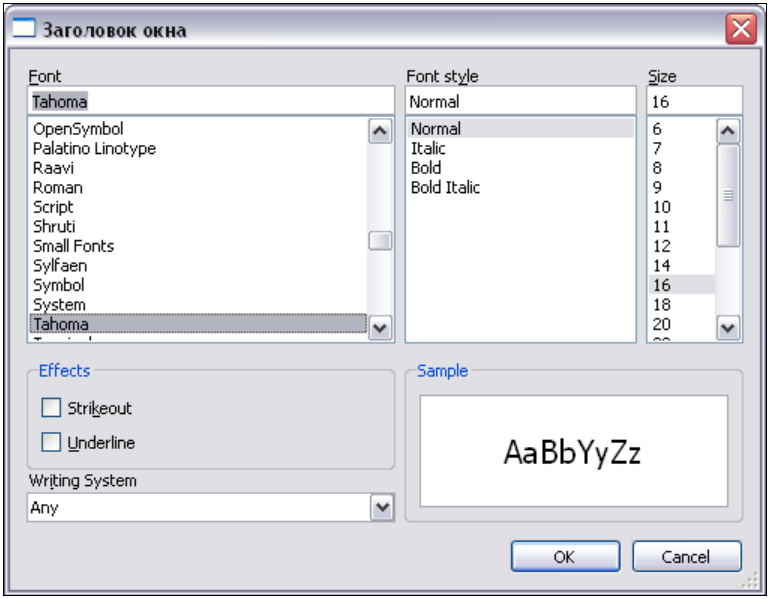

Рис. 27.12. Окно для выбора шрифта

27.8. Окно для вывода сообщения об ошибке

Класс `QErrorMessage` реализует немодальное диалоговое окно для вывода сообщения об ошибке (рис. 27.13). Окно содержит текстовое поле, в которое можно вставить текст сообщения об ошибке, и флажок. Если пользователь снимает флажок, то окно больше отображаться не будет. Иерархия наследования для класса `QErrorMessage` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QErrorMessage
```

Формат конструктора класса `QErrorMessage`:

```
<Объект> = QErrorMessage([parent=None])
```

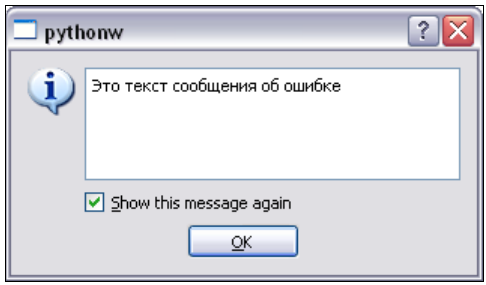

Рис. 27.13. Окно для вывода сообщения об ошибке

Для отображения окна предназначен метод `showMessage(<Текст сообщения>)`. Метод является слотом с сигнатурой `showMessage(const QString&)`.

27.9. Окно с индикатором хода процесса

Класс `QProgressDialog` реализует диалоговое окно с индикатором хода процесса и кнопкой **Cancel** (рис. 27.14). Иерархия наследования для класса `QProgressDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QProgressDialog
```

Форматы конструктора класса `QProgressDialog`:

```
<Объект> = QProgressDialog([parent=None][, flags=0])
<Объект> = QProgressDialog(<Текст над индикатором>,
           <Текст на кнопке Cancel>, <Минимум>, <Максимум>[,
           parent=None][, flags=0])
```

Если в параметре `parent` указана ссылка на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна (см. *разд. 20.2*).

Класс `QProgressDialog` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setLabel(<QLabel>)` — позволяет заменить объект надписи;
- `setBar(<QProgressBar>)` — позволяет заменить объект индикатора;
- `setCancelButton(<QPushButton>)` — позволяет заменить объект кнопки;
- `setValue(<Значение>)` — задает новое значение индикатора. Если диалоговое окно является модальным, то при установке значения автоматически вызывается метод `processEvents()` объекта приложения. Метод является слотом с сигнатурой `setValue(int)`;

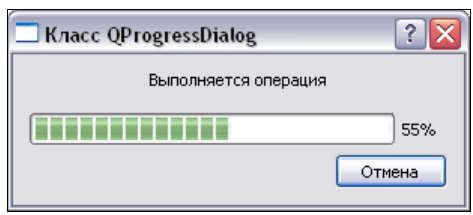

Рис. 27.14. Окно с индикатором хода процесса

- `value()` — возвращает текущее значение индикатора в виде числа;
- `setLabelText(<Текст над индикатором>)` — задает надпись, выводимую над индикатором. Метод является слотом с сигнатурой `setLabelText(const QString&)`;
- `setCancelButtonText(<Текст на кнопке Cancel>)` — задает надпись, выводимую на кнопке **Cancel**. Метод является слотом с сигнатурой `setCancelButtonText(const QString&)`;
- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное значения. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы `setMinimum()` и `setMaximum()` являются слотами с сигнатурами `setMinimum(int)` и `setMaximum(int)`;
- `setMinimumDuration(<Значение>)` — задает промежуток времени в миллисекундах перед отображением окна (по умолчанию значение равно `4000`). Окно может быть отображено ранее этого срока при установке значения. Метод является слотом с сигнатурой `setMinimumDuration(int)`;
- `reset()` — сбрасывает значение индикатора. Метод является слотом;
- `cancel()` — имитирует нажатие кнопки **Cancel**. Метод является слотом;
- `setAutoClose(<Флаг>)` — если в качестве параметра указано значение `True`, то при сбросе значения окно скрывается;
- `setAutoReset(<Флаг>)` — если в качестве параметра указано значение `True`, то при достижении максимального значения будет автоматически произведен сброс значения;
- `wasCanceled()` — возвращает значение `True`, если была нажата кнопка **Cancel**.

Класс `QProgressDialog` содержит сигнал `canceled()`, который генерируется при нажатии кнопки **Cancel**.

27.10. Создание многостраничного мастера

С помощью классов `QWizard` и `QWizardPage` можно создать диалоговое окно, в котором последовательно (или в произвольном порядке) отображаются различные страницы при нажатии кнопок **Back** (Назад) и **Next** (Далее). Класс `QWizard` реализует контейнер для страниц, а отдельная страница описывается с помощью класса `QWizardPage`.

27.10.1. Класс *QWizard*

Класс `QWizard` реализует набор страниц, отображаемых последовательно или в произвольном порядке. Иерархия наследования для класса `QWizard` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QWizard
```

Формат конструктора класса `QWizard`:

```
<Объект> = QWizard([parent=None][, flags=0])
```

Класс `QWizard` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- addPage(<QWizardPage>) — добавляет страницу в конец мастера и возвращает ее идентификатор. В качестве параметра указывается экземпляр класса QWizardPage;
- setPage(<Идентификатор>, <QWizardPage>) — добавляет страницу в указанную позицию;
- removePage(<Идентификатор>) — удаляет страницу с указанным идентификатором;
- page(<Идентификатор>) — возвращает ссылку на страницу (экземпляр класса QWizardPage), соответствующую указанному идентификатору, или значение None, если страницы не существует;
- pageIds() — возвращает список с идентификаторами страниц;
- currentId() — возвращает идентификатор текущей страницы;
- currentPage() — возвращает ссылку на текущую страницу (экземпляр класса QWizardPage) или значение None, если страницы не существует;
- setStartId(<Идентификатор>) — задает идентификатор начальной страницы;
- startId() — возвращает идентификатор начальной страницы;
- visitedPages() — возвращает список с идентификаторами посещенных страниц или пустой список;
- hasVisitedPage(<Идентификатор>) — возвращает значение True, если страница была посещена, и False — в противном случае;
- back() — имитирует нажатие кнопки **Back**. Метод является слотом;
- next() — имитирует нажатие кнопки **Next**. Метод является слотом;
- restart() — перезапускает мастера. Метод является слотом;
- nextId(self) — этот метод следует переопределить в классе, наследующем класс QWizard, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next**. Метод должен возвращать идентификатор следующей страницы или значение -1;
- initializePage(self, <Идентификатор>) — этот метод следует переопределить в классе, наследующем класс QWizard, если необходимо производить настройку свойств компонентов на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения следующей страницы. Если установлена опция IndependentPages, то метод вызывается только при первом отображении страницы;
- cleanupPage(self, <Идентификатор>) — этот метод следует переопределить в классе, наследующем класс QWizard, если необходимо контролировать нажатие кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция IndependentPages, то метод не вызывается;
- validateCurrentPage(self) — этот метод следует переопределить в классе, наследующем класс QWizard, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен вернуть значение True, если данные корректны, и False — в противном случае. Если метод возвращает значение False, то переход на следующую страницу не производится;
- setField(<Свойство>, <Значение>) — устанавливает значение указанного свойства. Создание свойства производится с помощью метода registerField() из класса

QWizardPage. С помощью этого метода можно изменять значения компонентов, расположенных на разных страницах мастера;

- field(<Свойство>) — возвращает значение указанного свойства. Создание свойства производится с помощью метода registerField() из класса QWizardPage. С помощью этого метода можно получить значения компонентов, расположенных на разных страницах мастера;
- setWizardStyle(<Стиль>) — задает стилевое оформление мастера. В качестве параметра указываются следующие атрибуты из класса QWizard:
  - ClassicStyle — 0;
  - ModernStyle — 1;
  - MacStyle — 2;
  - AeroStyle — 3 — доступен только в операционной системе Windows Vista. В остальных системах будет соответствовать стилю ModernStyle;
- setTitleFormat(<Формат>) — задает формат отображения названия страницы. В качестве параметра указываются следующие атрибуты из класса QtCore.Qt:
  - PlainText — 0 — простой текст;
  - RichText — 1 — форматированный текст;
  - AutoText — 2 — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим RichText, в противном случае — режим PlainText;
- setSubTitleFormat(<Формат>) — задает формат отображения описания страницы. Допустимые значения см. в описании метода setTitleFormat();
- setButton(<Роль>, <QAbstractButton>) — добавляет кнопку для указанной роли. В первом параметре указываются следующие атрибуты из класса QWizard:
  - BackButton — 0 — кнопка **Back**;
  - NextButton — 1 — кнопка **Next**;
  - CommitButton — 2 — кнопка **Commit**;
  - FinishButton — 3 — кнопка **Finish**;
  - CancelButton — 4 — кнопка **Cancel** (если установлена опция NoCancelButton, то кнопка не отображается);
  - HelpButton — 5 — кнопка **Help** (чтобы отобразить кнопку, необходимо установить опцию HaveHelpButton);
  - CustomButton1 — 6 — первая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton1);
  - CustomButton2 — 7 — вторая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton2);
  - CustomButton3 — 8 — третья пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton3);
- button(<Роль>) — возвращает ссылку на кнопку с указанной ролью;

- `setButtonText(<Роль>, <Текст надписи>)` — устанавливает текст надписи для кнопки с указанной ролью;
- `buttonText(<Роль>)` — возвращает текст надписи кнопки с указанной ролью;
- `setButtonLayout(<Список с ролями>)` — задает порядок отображения кнопок. В качестве параметра указывается список с ролями кнопок. Список может также содержать значение атрибута `Stretch` из класса `QWizard`, который задает фактор растяжения;
- `setPixmap(<Роль>, <QPixmap>)` — добавляет изображение для указанной роли. В первом параметре указываются следующие атрибуты из класса `QWizard`:
  - `WatermarkPixmap` — `0` — изображение, которое занимает всю левую сторону страницы при использовании стилей `ClassicStyle` или `ModernStyle`;
  - `LogoPixmap` — `1` — небольшое изображение, отображаемое в правой части заголовка при использовании стилей `ClassicStyle` или `ModernStyle`;
  - `BannerPixmap` — `2` — фоновое изображение, отображаемое в заголовке страницы при использовании стиля `ModernStyle`;
  - `BackgroundPixmap` — `3` — фоновое изображение при использовании стиля `MacStyle`;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, то производит установку опции, указанной в первом параметре, а если указано значение `False`, то сбрасывает опцию. В первом параметре указываются следующие атрибуты из класса `QWizard`:
  - `IndependentPages` — страницы не зависят друг от друга. Если опция установлена, то метод `initializePage()` будет вызван только при первом отображении страницы, а метод `cleanupPage()` не вызывается;
  - `IgnoreSubTitles` — не отображать описание страницы в заголовке;
  - `ExtendedWatermarkPixmap` — изображение с ролью `WatermarkPixmap` будет занимать всю левую сторону страницы вплоть до нижнего края окна;
  - `NoDefaultButton` — не делать кнопки **Next** и **Finish** кнопками по умолчанию;
  - `NoBackButtonOnStartPage` — не отображать кнопку **Back** на стартовой странице;
  - `NoBackButtonOnLastPage` — не отображать кнопку **Back** на последней странице;
  - `DisabledBackButtonOnLastPage` — сделать кнопку **Back** неактивной на последней странице;
  - `HaveNextButtonOnLastPage` — показать неактивную кнопку **Next** на последней странице;
  - `HaveFinishButtonOnEarlyPages` — показать неактивную кнопку **Finish** на не последних страницах;
  - `NoCancelButton` — не отображать кнопку **Cancel**;
  - `CancelButtonOnLeft` — поместить кнопку **Cancel** слева от кнопки **Back** (по умолчанию кнопка расположена справа от кнопок **Next** и **Finish**);
  - `HaveHelpButton` — показать кнопку **Help**;
  - `HelpButtonOnRight` — поместить кнопку **Help** у правого края окна;
  - `HaveCustomButton1` — показать пользовательскую кнопку с ролью `CustomButton1`;

- HaveCustomButton2 — показать пользовательскую кнопку с ролью CustomButton2;
- HaveCustomButton3 — показать пользовательскую кнопку с ролью CustomButton3;

♦ setOptions(<Опции>) — устанавливает несколько опций.

Класс QWizard содержит следующие сигналы:

♦ currentIdChanged(int) — генерируется при изменении текущей страницы. Внутри обработчика через параметр доступен идентификатор текущей страницы;

♦ customButtonClicked(int) — генерируется при нажатии кнопок с ролями CustomButton1, CustomButton2 и CustomButton3;

♦ helpRequested() — генерируется при нажатии кнопки **Help**;

♦ pageAdded(int) — генерируется при добавлении страницы;

♦ pageRemoved(int) — генерируется при удалении страницы.

27.10.2. Класс *QWizardPage*

Класс QWizardPage описывает одну страницу в многостраничном мастере. Иерархия наследования для класса QWizardPage выглядит так:

```
(QObject, QPaintDevice) — QWidget — QWizardPage
```

Формат конструктора класса QWizardPage:

```
<Объект> = QWizardPage([parent=None])
```

Класс QWizardPage наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

♦ wizard() — возвращает ссылку на объект мастера (экземпляр класса QWizard) или значение None, если страница не была добавлена;

♦ setTitle(<Текст>) — задает название страницы;

♦ title() — возвращает название страницы;

♦ setSubTitle(<Текст>) — задает описание страницы;

♦ subTitle() — возвращает описание страницы;

♦ setButtonText(<Роль>, <Текст надписи>) — устанавливает текст надписи для кнопки с указанной ролью (допустимые значения параметра <Роль> см. в описании метода setButton() из класса QWizard);

♦ buttonText(<Роль>) — возвращает текст надписи кнопки с указанной ролью;

♦ setPixmap(<Роль>, <QPixmap>) — добавляет изображение для указанной роли (допустимые значения параметра <Роль> см. в описании метода setPixmap() из класса QWizard);

♦ registerField() — регистрирует свойство, с помощью которого можно получить доступ к значению компонента с любой страницы мастера. Формат метода:

```
registerField(<Свойство>, <QWidget>[, property=None][,
              changedSignal=0])
```

В параметре <Свойство> указывается произвольное название свойства в виде строки. Если в конце строки указать символ *, то компонент должен обязательно иметь значение

(например, в поле должно быть введено какое-либо значение), иначе кнопки **Next** и **Finish** будут недоступны. Во втором параметре указывается ссылка на компонент. После регистрации свойства изменить значение компонента позволяет метод `setField()`, а получить значение — метод `field()`.

В параметре `property` может быть указано свойство для получения и изменения значения, а в параметре `changedSignal` — сигнал, генерируемый при изменении значения. Назначить эти параметры для определенного класса позволяет также метод `setDefaultProperty(<Название класса>, property, changedSignal)` из класса `QWizard`. Для стандартных классов по умолчанию используются следующие свойства и сигналы:

Класс:	**Свойство:**	**Сигнал:**
QAbstractButton	checked	toggled()
QAbstractSlider	value	valueChanged()
QComboBox	currentIndex	currentIndexChanged()
QDateTimeEdit	dateTime	dateTimeChanged()
QLineEdit	text	textChanged()
QListWidget	currentRow	currentRowChanged()
QSpinBox	value	valueChanged()

- `setField(<Свойство>, <Значение>)` — устанавливает значение указанного свойства. С помощью этого метода можно изменять значения компонентов, расположенных на разных страницах мастера;
- `field(<Свойство>)` — возвращает значение указанного свойства. С помощью этого метода можно получить значения компонентов, расположенных на разных страницах мастера;
- `setFinalPage(<Флаг>)` — если в качестве параметра указано значение `True`, то на странице будет отображаться кнопка **Finish**;
- `isFinalPage()` — возвращает значение `True`, если на странице будет отображаться кнопка **Finish**, и `False` — в противном случае;
- `setCommitPage(<Флаг>)` — если в качестве параметра указано значение `True`, то на странице будет отображаться кнопка **Commit**;
- `isCommitPage()` — возвращает значение `True`, если на странице будет отображаться кнопка **Commit**, и `False` — в противном случае;
- `isComplete(self)` — этот метод вызывается, чтобы определить, должны ли кнопки **Next** и **Finish** быть доступными (метод возвращает значение `True`) или недоступными (метод возвращает значение `False`). Метод можно переопределить в классе, наследующем класс `QWizardPage`, и реализовать собственную проверку правильности ввода данных. При изменении возвращаемого значения необходимо генерировать сигнал `completeChanged()`;
- `nextId(self)` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next**. Метод должен возвращать идентификатор следующей страницы или значение `-1`;
- `initializePage(self)` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо производить настройку свойств компонентов на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения следующей страницы. Если

установлена опция `IndependentPages`, то метод вызывается только при первом отображении страницы;

- `cleanupPage(self)` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо контролировать нажатие кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция `IndependentPages`, то метод не вызывается;
- `validatePage(self)` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен вернуть значение `True`, если данные корректны, и `False` — в противном случае. Если метод возвращает значение `False`, то переход на следующую страницу не производится.

ГЛАВА 28

Создание SDI- и MDI-приложений

В PyQt существует поддержка двух типов приложений:

- *SDI-приложения* (Single Document Interface) — приложение для отображения одного документа. Чтобы отобразить новый документ, необходимо закрыть предыдущий или запустить отдельный экземпляр приложения. Типичным примером таких приложений являются программы Блокнот, WordPad и Paint в операционной системе Windows. Чтобы создать SDI-приложение, следует установить центральный компонент с помощью метода `setCentralWidget()` из класса `QMainWindow`;
- *MDI-приложения* (Multiple Document Interface) — центральный компонент может отображать сразу несколько документов одновременно в разных вложенных окнах. Типичным примером MDI-приложения является программа Photoshop, позволяющая редактировать сразу несколько фотографий одновременно. Чтобы создать MDI-приложение, следует в качестве центрального компонента установить компонент `QMdiArea` с помощью метода `setCentralWidget()` из класса `QMainWindow`. Отдельное окно внутри MDI-области реализуется с помощью класса `QMdiSubWindow`.

28.1. Создание главного окна приложения

Класс `QMainWindow` реализует главное окно приложения, содержащее меню, панели инструментов, прикрепляемые панели, центральный компонент и строку состояния. Иерархия наследования для класса `QMainWindow` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMainWindow
```

Конструктор класса `QMainWindow` имеет следующий формат:

```
<Объект> = QMainWindow([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительское окно. Какие именно значения можно указать в параметре `flags`, мы уже рассматривали в *разд. 20.2*.

Класс `QMainWindow` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setCentralWidget(<QWidget>)` — делает указанный компонент центральным компонентом главного окна;
- `centralWidget()` — возвращает ссылку на центральный компонент или значение `None`, если компонент не был установлен;

- menuBar() — возвращает ссылку на главное меню (экземпляр класса QMenuBar);
- menuWidget() — возвращает ссылку на компонент, в котором расположено главное меню (экземпляр класса QWidget);
- setMenuBar(<QMenuBar>) — позволяет установить пользовательское меню вместо стандартного;
- setMenuWidget(<QWidget>) — позволяет установить компонент главного меню;
- createPopupMenu() — создает контекстное меню с пунктами, позволяющими отобразить или скрыть панели инструментов и прикрепляемые панели, и возвращает ссылку на меню (экземпляр класса QMenu). Это меню по умолчанию отображается при щелчке правой кнопкой мыши в области меню, панели инструментов или прикрепляемых панелей. Переопределив этот метод можно реализовать собственное контекстное меню;
- statusBar() — возвращает ссылку (экземпляр класса QStatusBar) на строку состояния;
- setStatusBar(<QStatusBar>) — позволяет заменить стандартную строку состояния;
- addToolBar() — добавляет панель инструментов. Форматы метода:

  ```
  addToolBar(<QToolBar>)
  addToolBar(<Область>, <QToolBar>)
  addToolBar(<Название панели>)
  ```

  Первый формат добавляет панель инструментов в верхнюю часть окна. Второй формат позволяет указать местоположение панели. В качестве параметра <Область> могут быть указаны следующие атрибуты из класса QtCore.Qt: LeftToolBarArea (слева), RightToolBarArea (справа), TopToolBarArea (сверху) или BottomToolBarArea (снизу). Третий формат создает панель инструментов с указанным именем, добавляет ее в верхнюю область окна и возвращает ссылку на нее (экземпляр класса QToolBar);
- insertToolBar(<QToolBar 1>, <QToolBar 2>) — добавляет панель <QToolBar 2> перед панелью <QToolBar 1>;
- removeToolBar(<QToolBar>) — удаляет панель инструментов из окна и скрывает ее. При этом объект панели инструментов не удаляется и далее может быть добавлен в другое место;
- toolBarArea(<QToolBar>) — возвращает местоположение указанной панели инструментов в виде значений атрибутов LeftToolBarArea (слева), RightToolBarArea (справа), TopToolBarArea (сверху), BottomToolBarArea (снизу) или NoToolBarArea (положение не определено) из класса QtCore.Qt;
- setToolButtonStyle(<Стиль>) — задает стиль кнопок на панели инструментов. В качестве параметра указываются следующие атрибуты из класса QtCore.Qt:
  - ToolButtonIconOnly — 0 — отображается только иконка;
  - ToolButtonTextOnly — 1 — отображается только текст;
  - ToolButtonTextBesideIcon — 2 — текст отображается справа от иконки;
  - ToolButtonTextUnderIcon — 3 — текст отображается под иконкой;
  - ToolButtonFollowStyle — 4 — зависит от используемого стиля;
- toolButtonStyle() — возвращает стиль кнопок на панели инструментов;
- setIconSize(<QSize>) — задает размеры иконок;

- `iconSize()` — возвращает размеры иконок (экземпляр класса `QSize`);
- `setAnimated(<Флаг>)` — если в качестве параметра указано значение `True` (используется по умолчанию), то вставка панелей инструментов и прикрепляемых панелей в новое место по окончании перемещения будет производиться с анимацией. Метод является слотом с сигнатурой `setAnimated(bool)`;
- `addToolBarBreak([area=TopToolBarArea])` — вставляет разрыв в указанное место после всех добавленных ранее панелей. По умолчанию панели добавляются друг за другом на одной строке. С помощью этого метода можно поместить панели инструментов на двух и более строках;
- `insertToolBarBreak(<QToolBar>)` — вставляет разрыв перед указанной панелью инструментов;
- `removeToolBarBreak(<QToolBar>)` — удаляет разрыв перед указанной панелью;
- `toolBarBreak(<QToolBar>)` — возвращает значение `True`, если перед указанной панелью инструментов существует разрыв, и `False` — в противном случае;
- `addDockWidget()` — добавляет прикрепляемую панель. Форматы метода:

```
addDockWidget(<Область>, <QDockWidget>)
addDockWidget(<Область>, <QDockWidget>, <Ориентация>)
```

  Первый формат добавляет прикрепляемую панель в указанную область окна. В качестве параметра `<Область>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`: `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху) или `BottomDockWidgetArea` (снизу). Второй формат позволяет дополнительно указать ориентацию при добавлении панели. В качестве параметра `<Ориентация>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`: `Horizontal` или `Vertical`. Если указан атрибут `Horizontal`, то добавляемая панель будет расположена справа от ранее добавленной панели, а если `Vertical` — то снизу;
- `removeDockWidget(<QDockWidget>)` — удаляет панель из окна и скрывает ее. При этом объект панели не удаляется и далее может быть добавлен в другую область;
- `dockWidgetArea(<QDockWidget>)` — возвращает местоположение указанной панели в виде значений атрибутов `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху), `BottomDockWidgetArea` (снизу) или `NoDockWidgetArea` (положение не определено) из класса `QtCore.Qt`;
- `setDockOptions(<Опции>)` — устанавливает опции для прикрепляемых панелей. Значение по умолчанию: `AnimatedDocks | AllowTabbedDocks`. В качестве значения указывается комбинация (через оператор `|`) следующих атрибутов из класса `QMainWindow`:
  - `AnimatedDocks` — если опция установлена, то вставка панелей в новое место по окончании перемещения будет производиться с анимацией;
  - `AllowNestedDocks` — если опция установлена, то области (в которые можно добавить панели) могут быть разделены, чтобы вместить больше панелей;
  - `AllowTabbedDocks` — если опция установлена, то одна панель может быть наложена пользователем на другую. При этом добавляется панель с вкладками. С помощью щелчка мышью на заголовке вкладки можно выбрать отображаемую в данный момент панель;
  - `ForceTabbedDocks` — если опция установлена, то панели не могут быть расположены рядом друг с другом. При этом опция `AllowNestedDocks` игнорируется;

- `VerticalTabs` — если опция установлена, то заголовки вкладок отображаются с внешнего края области (если область справа, то заголовки вкладок справа, если область слева, то заголовки слева, если область сверху, то заголовки вкладок сверху, если область снизу, то заголовки вкладок снизу). Если опция не установлена, то заголовки вкладок отображаются снизу. Опция `AllowTabbedDocks` должна быть установлена.

Обратите внимание

Опции необходимо устанавливать до добавления прикрепляемых панелей. Исключением являются опции `AnimatedDocks` и `VerticalTabs`.

- `dockOptions()` — возвращает комбинацию установленных опций;
- `setDockNestingEnabled(<Флаг>)` — если указано значение `True`, то метод устанавливает опцию `AllowNestedDocks`, а если указано значение `False` — то сбрасывает опцию. Метод является слотом с сигнатурой `setDockNestingEnabled(bool)`;
- `isDockNestingEnabled()` — возвращает значение `True`, если опция `AllowNestedDocks` установлена, и `False` — в противном случае;
- `setTabPosition(<Область>, <Позиция>)` — задает позицию отображения заголовков вкладок прикрепляемых панелей для указанной области. По умолчанию заголовки вкладок отображаются снизу. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты из класса `QTabWidget`:
  - `North` — `0` — сверху;
  - `South` — `1` — снизу;
  - `West` — `2` — слева;
  - `East` — `3` — справа;
- `tabPosition(<Область>)` — возвращает позицию отображения заголовков вкладок прикрепляемых панелей для указанной области;
- `setTabShape(<Форма>)` — задает форму углов ярлыков вкладок в области заголовка. Могут быть указаны следующие атрибуты из класса `QTabWidget`:
  - `Rounded` — `0` — скругленные углы (значение по умолчанию);
  - `Triangular` — `1` — треугольная форма;
- `tabShape()` — возвращает форму углов ярлыков вкладок в области заголовка;
- `setCorner(<Угол>, <Область>)` — позволяет закрепить указанный угол за определенной областью. По умолчанию верхние углы закреплены за верхней областью, а нижние углы за нижней областью. В качестве параметра `<Область>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`: `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху) или `BottomDockWidgetArea` (снизу). В параметре `<Угол>` указываются следующие атрибуты из класса `QtCore.Qt`:
  - `TopLeftCorner` — левый верхний угол;
  - `TopRightCorner` — правый верхний угол;
  - `BottomLeftCorner` — левый нижний угол;
  - `BottomRightCorner` — правый нижний угол;
- `corner(<Угол>)` — возвращает область, за которой закреплен указанный угол;

- splitDockWidget() — разделяет область, занимаемую панелью <QDockWidget 1>, и добавляет панель <QDockWidget 1> в первую часть, а панель <QDockWidget 2> — во вторую часть. Порядок расположения частей зависит от параметра <Ориентация>. В качестве параметра <Ориентация> могут быть указаны следующие атрибуты из класса QtCore.Qt: Horizontal или Vertical. Если указан атрибут Horizontal, то панель <QDockWidget 2> будет расположена справа, а если Vertical — то снизу. Если панель <QDockWidget 1> расположена на вкладке, то панель <QDockWidget 2> будет добавлена на новую вкладку, при этом разделения области не происходит. Формат метода:

  ```
  splitDockWidget(<QDockWidget 1>, <QDockWidget 2>, <Ориентация>)
  ```

- tabifyDockWidget(<QDockWidget 1>, <QDockWidget 2>) — размещает панель <QDockWidget 2> над панелью <QDockWidget 1>, создавая, таким образом, область с вкладками;

- tabifiedDockWidgets(<QDockWidget>) — возвращает список ссылок на панели (экземпляры класса QDockWidget), которые расположены на других вкладках в области панели, указанной в качестве параметра;

- saveState([version=0]) — возвращает экземпляр класса QByteArray с размерами и положением всех панелей инструментов и прикрепляемых панелей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода restoreState(<QByteArray>[, version=0]).

  Обратите внимание на то, что все панели инструментов и прикрепляемые панели должны иметь уникальные объектные имена. Задать объектное имя можно с помощью метода setObjectName(<Имя в виде строки>) из класса QObject. Пример:

  ```
  self.dw = QtGui.QDockWidget("MyDockWidget1")
  self.dw.setObjectName("MyDockWidget1")
  ```

  Чтобы сохранить размеры окна, следует воспользоваться методом saveGeometry() из класса QWidget. Метод возвращает экземпляр класса QByteArray с размерами окна. Чтобы восстановить размеры окна, следует воспользоваться методом restoreGeometry(<QByteArray>);

- restoreDockWidget(<QDockWidget>) — восстанавливает состояние указанной панели, если она создана после вызова метода restoreState(). Метод возвращает значение True, если состояние панели успешно восстановлено, и False — в противном случае.

28.2. Меню

Главное меню является основным компонентом пользовательского интерфейса, позволяющим компактно поместить множество команд, объединяя их в логические группы. Главное меню состоит из горизонтальной панели (реализуемой классом QMenuBar), на которой расположены отдельные меню (реализуются с помощью класса QMenu) верхнего уровня. Каждое меню может содержать множество пунктов (реализуемых с помощью класса QAction), разделители, а также вложенные меню. Пункт меню может содержать иконку, текст и флажок, превращающий команду в переключатель.

Помимо главного меню в приложениях часто используются контекстные меню, которые обычно отображаются при щелчке правой кнопкой мыши в области компонента. Контекстное меню реализуется с помощью класса QMenu и отображается внутри метода с предопределенным названием contextMenuEvent() с помощью метода exec_(), в который передаются глобальные координаты щелчка мышью.

28.2.1. Класс *QMenuBar*

Класс QMenuBar описывает горизонтальную панель меню. Панель меню реализована в главном окне приложения по умолчанию. Получить ссылку на нее можно с помощью метода menuBar() из класса QMainWindow. Установить свою панель позволяет метод setMenuBar(<QMenuBar>). Иерархия наследования для класса QMenuBar выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenuBar
```

Конструктор класса QMenuBar имеет следующий формат:

```
<Объект> = QMenuBar([parent=<Родитель>])
```

Класс QMenuBar наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- addMenu(<QMenu>) — добавляет меню на панель и возвращает экземпляр класса QAction, с помощью которого, например, можно скрыть меню (с помощью метода setVisible()) или сделать его неактивным (с помощью метода setEnabled());
- addMenu([<QIcon>,]<Название>) — создает меню, добавляет его на панель и возвращает ссылку на него (экземпляр класса QMenu). Внутри текста в параметре <Название> символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ &, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню будет выбрано. Чтобы вывести символ &, необходимо его удвоить;
- insertMenu(<QAction>, <QMenu>) — добавляет меню <QMenu> перед пунктом <QAction>. Метод возвращает экземпляр класса QAction;
- addAction() — добавляет пункт в меню. Форматы метода:

```
addAction(<QAction>)
addAction(<Название>)                    -> QAction
addAction(<Название>, <Объект>, <Слот>) -> QAction
addAction(<Название>, <Обработчик>)      -> QAction
```

- clear() — удаляет все действия из панели меню;
- setActiveAction(<QAction>) — делает активным указанное действие;
- activeAction() — возвращает активное действие (экземпляр класса QAction) или значение None;
- setDefaultUp(<Флаг>) — если в качестве параметра указано значение True, то пункты меню будут отображаться выше панели меню, а не ниже;
- setVisible(<Флаг>) — если в качестве параметра указано значение False, то панель меню будет скрыта. Значение True отображает панель меню.

Класс QMenuBar содержит следующие сигналы:

- hovered(QAction *) — генерируется при наведении указателя мыши на пункт меню. Внутри обработчика через параметр доступен экземпляр класса QAction;
- triggered(QAction *) — генерируется при выборе пункта меню. Внутри обработчика через параметр доступен экземпляр класса QAction.

28.2.2. Класс *QMenu*

Класс `QMenu` реализует отдельное меню на панели меню, а также вложенное, плавающее и контекстное меню. Иерархия наследования для класса `QMenu` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenu
```

Форматы конструктора класса `QMenu`:

```
<Объект> = QMenu([parent=None])
<Объект> = QMenu(<Название>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент. Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ `&`, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню будет выбрано. Чтобы вывести символ `&`, необходимо его удвоить.

Класс `QMenu` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `addAction()` — добавляет пункт в меню. Форматы метода:

```
addAction(<QAction>)
addAction(<Название>)                        -> QAction
addAction(<QIcon>, <Название>)               -> QAction
addAction(<Название>, <Объект>, <Слот>[, shortcut=0])
                                             -> QAction
addAction(<Название>, <Обработчик>[, <shortcut>])
                                             -> QAction
addAction(<QIcon>, <Название>, <Объект>, <Слот>[, shortcut=0])
                                             -> QAction
addAction(<QIcon>, <Название>, <Обработчик>[, <shortcut>])
                                             -> QAction
```

  Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ `&`, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа пункт меню будет выбран. Чтобы вывести символ `&`, необходимо его удвоить. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным.

  Параметр `shortcut` задает комбинацию "горячих" клавиш, нажатие которых аналогично выбору пункта в меню. Нажатие комбинации "горячих" клавиш сработает даже в том случае, если меню не является активным. Примеры указания значения в параметре `shortcut`:

```
QtGui.QKeySequence("Ctrl+R")
QtGui.QKeySequence(QtCore.Qt.CTRL + QtCore.Qt.Key_R))
QtGui.QKeySequence.fromString("Ctrl+R")
```

  Добавить пункты в меню и удалить их позволяют следующие методы из класса `QWidget`:

  - `addActions(<Список с экземплярами класса QAction>)` — добавляет несколько пунктов в конец меню;

- `insertAction(<QAction 1>, <QAction 2>)` — добавляет пункт `<QAction 2>` перед пунктом `<QAction 1>`;
- `insertActions(<QAction>, <Список с экземплярами класса QAction>)` — добавляет несколько пунктов, указанных во втором параметре перед пунктом `<QAction>`;
- `actions()` — возвращает список с действиями (экземпляры класса `QAction`);
- `removeAction(<QAction>)` — удаляет указанное действие из меню;

- `addSeparator()` — добавляет разделитель в меню и возвращает экземпляр класса `QAction`;
- `insertSeparator(<QAction>)` — добавляет разделитель перед указанным пунктом и возвращает экземпляр класса `QAction`;
- `addMenu(<QMenu>)` — добавляет вложенное меню и возвращает экземпляр класса `QAction`;
- `addMenu([<QIcon>, ]<Название>)` — создает вложенное меню, добавляет его в меню и возвращает ссылку на него (экземпляр класса `QMenu`);
- `insertMenu(<QAction>, <QMenu>)` — добавляет вложенное меню `<QMenu>` перед пунктом `<QAction>`. Метод возвращает экземпляр класса `QAction`;
- `clear()` — удаляет все действия из меню;
- `isEmpty()` — возвращает значение `True`, если меню не содержит видимых пунктов, и `False` — в противном случае;
- `menuAction()` — возвращает объект действия (экземпляр класса `QAction`), связанный с данным меню. С помощью этого объекта можно скрыть меню (с помощью метода `setVisible()`) или сделать его неактивным (с помощью метода `setEnabled()`);
- `setTitle(<Название>)` — задает название меню;
- `title()` — возвращает название меню;
- `setIcon(<QIcon>)` — задает иконку меню;
- `icon()` — возвращает иконку меню (экземпляр класса `QIcon`);
- `setActiveAction(<QAction>)` — делает активным указанный пункт;
- `activeAction()` — возвращает активный пункт (экземпляр класса `QAction`) или значение `None`;
- `setDefaultAction(<QAction>)` — задает пункт по умолчанию;
- `defaultAction()` — возвращает пункт по умолчанию (экземпляр класса `QAction`) или значение `None`;
- `setTearOffEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то в начало меню добавляется пункт с пунктирной линией, с помощью щелчка на котором можно оторвать меню от панели и сделать его плавающим (отображаемым в отдельном окне, которое можно разместить в любой части экрана);
- `isTearOffEnabled()` — возвращает значение `True`, если меню может быть плавающим, и `False` — в противном случае;
- `isTearOffMenuVisible()` — возвращает значение `True`, если плавающее меню отображается в отдельном окне, и `False` — в противном случае;
- `hideTearOffMenu()` — скрывает плавающее меню;

- setSeparatorsCollapsible(<Флаг>) — если в качестве параметра указано значение True, то вместо нескольких разделителей, идущих подряд, будет отображаться один разделитель. Кроме того, разделители, расположенные по краям меню, также будут скрыты;
- popup(<QPoint>[, <QAction>]) — отображает меню по указанным глобальным координатам. Если указан второй параметр, то меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню;
- exec_([<QPoint>[, <QAction>]]) — отображает меню по указанным глобальным координатам и возвращает экземпляр класса QAction (соответствующий выбранному пункту) или значение None (если пункт не выбран, например, нажата клавиша <Esc>). Если указан второй параметр, то меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню.

Для отображения меню можно также воспользоваться статическим методом exec_(). Метод отображает меню по указанным глобальным координатам и возвращает экземпляр класса QAction (соответствующий выбранному пункту) или значение None (если пункт не выбран, например, нажата клавиша <Esc>). Форматы метода:

```
exec_(<Список с экземплярами класса QAction>, <QPoint>[, <QAction>])
exec_(<Список с экземплярами класса QAction>, <QPoint>, <QAction>,
      <QWidget>)
```

Если указан параметр <QAction>, то меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню. В параметре <QWidget> указывается ссылка на родительский компонент.

Класс QMenu содержит следующие сигналы:

- hovered(QAction *) — генерируется при наведении указателя мыши на пункт меню. Внутри обработчика через параметр доступен экземпляр класса QAction;
- triggered(QAction *) — генерируется при выборе пункта меню. Внутри обработчика через параметр доступен экземпляр класса QAction;
- aboutToShow() — генерируется перед отображением меню;
- aboutToHide() — генерируется перед сокрытием меню.

28.2.3. Контекстное меню

Чтобы создать контекстное меню, необходимо наследовать класс компонента и переопределить метод с названием contextMenuEvent(self, <event>). Этот метод будет автоматически вызываться при щелчке правой кнопкой мыши в области компонента. Внутри метода через параметр <event> доступен экземпляр класса QContextMenuEvent, который позволяет получить дополнительную информацию о событии. Класс QContextMenuEvent содержит следующие основные методы:

- x() и y() — возвращают координаты по осям X и Y соответственно в пределах области компонента;
- pos() — возвращает экземпляр класса QPoint с целочисленными координатами в пределах области компонента;
- globalX() и globalY() — возвращают координаты по осям X и Y соответственно в пределах экрана;
- globalPos() — возвращает экземпляр класса QPoint с координатами в пределах экрана.

Чтобы отобразить контекстное меню внутри метода `contextMenuEvent()`, следует вызвать метод `exec_()` объекта меню и передать ему результат выполнения метода `globalPos()`. Пример:

```
def contextMenuEvent(self, event):
    self.context_menu.exec_(event.globalPos())
```

В этом примере меню создано внутри конструктора класса и сохранено в атрибуте `context_menu`. Если контекстное меню постоянно обновляется, то можно создавать меню при каждом вызове внутри метода `contextMenuEvent()` и сразу отображать.

28.2.4. Класс *QAction*

Класс `QAction` описывает объект действия, который можно добавить в меню, на панель инструментов или просто прикрепить к какому-либо компоненту. Один и тот же объект действия допускается добавлять в несколько мест. Например, можно продублировать пункт меню на панели инструментов, что позволит одновременно сделать действие недоступным. Иерархия наследования для класса `QAction` выглядит так:

```
QObject — QAction
```

Форматы конструктора класса `QAction`:

```
<Объект> = QAction(<QObject>)
<Объект> = QAction(<Название>, <QObject>)
<Объект> = QAction(<QIcon>, <Название>, <QObject>)
```

В параметре `<QObject>` указывается ссылка на родительский компонент или значение `None`. Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ `&`, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню будет выбрано. Чтобы вывести символ `&`, необходимо его удвоить. Параметр `<QIcon>` устанавливает иконку.

Класс `QAction` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText(<Название>)` — задает название действия. Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным;
- `text()` — возвращает название действия;
- `setIcon(<QIcon>)` — устанавливает иконку;
- `icon()` — возвращает иконку (экземпляр класса `QIcon`);
- `setIconVisibleInMenu(<Флаг>)` — если в качестве параметра указано значение `False`, то иконка в меню отображаться не будет;
- `setSeparator(<Флаг>)` — если в качестве параметра указано значение `True`, то объект является разделителем;
- `isSeparator()` — возвращает значение `True`, если объект является разделителем, и `False` — в противном случае;

- `setShortcut(<QKeySequence>)` — задает комбинацию "горячих" клавиш. Нажатие комбинации "горячих" клавиш по умолчанию сработает даже в том случае, если меню не является активным. Примеры указания значения:

```
"Ctrl+O"
QtGui.QKeySequence("Ctrl+O")
QtGui.QKeySequence(QtCore.Qt.CTRL + QtCore.Qt.Key_O))
QtGui.QKeySequence.fromString("Ctrl+O")
QtGui.QKeySequence.Open
```

- `setShortcuts()` — позволяет задать сразу несколько комбинаций "горячих" клавиш. Форматы метода:

```
setShortcuts(<Список с экземплярами класса QKeySequence>)
setShortcuts(<Стандартная комбинация клавиш>)
```

  В параметре `<Стандартная комбинация клавиш>` указываются атрибуты из перечисления `StandardKey` класса `QKeySequence` (например, `Open`, `Close`, `Copy`, `Cut` и т. д.; полный список смотрите в документации по классу `QKeySequence`);

- `setShortcutContext(<Область>)` — задает область действия комбинации "горячих" клавиш. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt`:
  - `WidgetShortcut` — `0` — комбинация доступна, если родительский компонент имеет фокус;
  - `WidgetWithChildrenShortcut` — `3` — комбинация доступна, если родительский компонент имеет фокус или любой дочерний компонент;
  - `WindowShortcut` — `1` — комбинация доступна, если окно, в котором расположен компонент, является активным;
  - `ApplicationShortcut` — `2` — комбинация доступна, если любое окно приложения является активным;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст справки;
- `whatsThis()` — возвращает текст справки;
- `setStatusTip(<Текст>)` — задает текст, который будет отображаться в строке состояния, при наведении указателя мыши на пункт меню;
- `statusTip()` — возвращает текст для строки состояния;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то действие является переключателем, который может находиться в двух состояниях — установленном и не установленном;
- `isCheckable()` — возвращает значение `True`, если действие является переключателем, и `False` — в противном случае;
- `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, то действие-переключатель будет находиться в установленном состоянии. Метод является слотом с сигнатурой `setChecked(bool)`;
- `isChecked()` — возвращает значение `True`, если действие-переключатель находится в установленном состоянии, и `False` — в противном случае;

- `setDisabled(<Флаг>)` — если в качестве параметра указано значение `True`, то действие станет недоступным. Значение `False` делает действие снова доступным. Метод является слотом с сигнатурой `setDisabled(bool)`;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `False`, то действие станет недоступным. Значение `True` делает действие снова доступным. Метод является слотом с сигнатурой `setEnabled(bool)`;
- `isEnabled()` — возвращает значение `True`, если действие доступно, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то действие будет скрыто. Значение `True` делает действие снова доступным. Метод является слотом с сигнатурой `setVisible(bool)`;
- `isVisible()` — возвращает значение `False`, если действие скрыто, и `True` — в противном случае;
- `setMenu(<QMenu>)` — устанавливает вложенное меню;
- `menu()` — возвращает ссылку на вложенное меню (экземпляр класса `QMenu`) или значение `None`, если вложенного меню нет;
- `setFont(<QFont>)` — устанавливает шрифт;
- `font()` — возвращает экземпляр класса `QFont` с текущими настройками шрифта;
- `setAutoRepeat(<Флаг>)` — если в качестве параметра указано значение `True` (значение по умолчанию), то действие будет повторяться, пока удерживается нажатой комбинация "горячих" клавиш;
- `setPriority(<Приоритет>)` — задает приоритет действия. В качестве параметра указываются атрибуты `LowPriority` (низкий), `NormalPriority` (нормальный) или `HighPriority` (высокий) из класса `QAction`;
- `priority()` — возвращает текущий приоритет действия;
- `setData(<Данные>)` — позволяет сохранить пользовательские данные любого типа в объекте действия;
- `data()` — возвращает пользовательские данные, сохраненные ранее с помощью метода `setData()`, или значение `None`;
- `setActionGroup(<QActionGroup>)` — добавляет действие в указанную группу;
- `actionGroup()` — возвращает ссылку на группу (экземпляр класса `QActionGroup`) или значение `None`;
- `showStatusText([<QWidget>=None])` — отправляет событие `QEvent.StatusTip` указанному компоненту и возвращает значение `True`, если событие успешно отправлено. Чтобы обработать событие, необходимо наследовать класс компонента и переопределить метод `event(self, <event>)`. При событии `QEvent.StatusTip` через параметр `<event>` доступен экземпляр класса `QStatusTipEvent`. Получить текст для строки состояния можно через метод `tip()` объекта события. Пример обработки события в классе, наследующем класс `QLabel`:

```
def event(self, e):
    if e.type() == QtCore.QEvent.StatusTip:
        self.setText(e.tip())
        return True
    return QtGui.QLabel.event(self, e)
```

- `hover()` — посылает сигнал `hovered()`. Метод является слотом;
- `toggle()` — производит изменение состояния переключателя на противоположное. Метод является слотом;
- `trigger()` — посылает сигнал `triggered()`. Метод является слотом.

Класс `QAction` содержит следующие сигналы:

- `changed()` — генерируется при изменении действия;
- `hovered()` — генерируется при наведении указателя мыши на объект действия;
- `toggled(bool)` — генерируется при изменении состояния переключателя. Внутри обработчика через параметр доступно текущее состояние;
- `triggered(bool=0)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов, нажатии комбинации клавиш или вызове метода `trigger()`.

28.2.5. Объединение переключателей в группу

Класс `QActionGroup` позволяет объединить несколько переключателей в группу. По умолчанию внутри группы может быть включен только один переключатель. При попытке включить другой переключатель, ранее включенный переключатель автоматически отключается. Иерархия наследования для класса `QActionGroup` выглядит так:

```
QObject — QActionGroup
```

Конструктор класса `QActionGroup` имеет следующий формат:

```
<Объект> = QActionGroup(<QObject>)
```

В параметре `<QObject>` указывается ссылка на родительский компонент или значение `None`. После создания объекта группы он может быть указан в качестве родителя при создании объектов действия. В этом случае действие автоматически добавляется в группу.

Класс `QActionGroup` содержит следующие основные методы:

- `addAction()` — добавляет объект действия в группу. Метод возвращает экземпляр класса `QAction`. Форматы метода:

  ```
  addAction(<QAction>)
  addAction(<Название>)
  addAction(<QIcon>, <Название>)
  ```

- `removeAction(<QAction>)` — удаляет объект действия из группы;
- `actions()` — возвращает список с объектами класса `QAction`, которые были добавлены в группу, или пустой список;
- `checkedAction()` — возвращает ссылку (экземпляр класса `QAction`) на установленный переключатель внутри группы при использовании эксклюзивного режима или значение `None`;
- `setExclusive(<Флаг>)` — если в качестве параметра указано значение `True` (значение по умолчанию), то внутри группы может быть включен только один переключатель. При попытке включить другой переключатель, ранее включенный переключатель автоматически отключается. Значение `False` отключает эксклюзивный режим. Метод является слотом с сигнатурой `setExclusive(bool)`;

- `setDisabled(<Флаг>)` — если в качестве параметра указано значение `True`, то все действия внутри группы станут недоступными. Значение `False` делает действия снова доступными. Метод является слотом с сигнатурой `setDisabled(bool)`;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `False`, то все действия внутри группы станут недоступными. Значение `True` делает действия снова доступными. Метод является слотом с сигнатурой `setEnabled(bool)`;
- `isEnabled()` — возвращает значение `True`, если действия доступны, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то все действия внутри группы будут скрыты. Значение `True` делает действия снова доступными. Метод является слотом с сигнатурой `setVisible(bool)`;
- `isVisible()` — возвращает значение `False`, если действия скрыты, и `True` — в противном случае.

Класс `QActionGroup` содержит следующие сигналы:

- `hovered(QAction *)` — генерируется при наведении указателя мыши на объект действия внутри группы. Внутри обработчика через параметр доступна ссылка на объект действия (экземпляр класса `QAction`);
- `selected(QAction *)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов, нажатии комбинации клавиш. Внутри обработчика через параметр доступна ссылка на объект действия (экземпляр класса `QAction`);
- `triggered(QAction *)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов, нажатии комбинации клавиш. Внутри обработчика через параметр доступна ссылка на объект действия (экземпляр класса `QAction`).

28.3. Панели инструментов

Панели инструментов предназначены для отображения часто используемых команд. Добавить панель инструментов в главное окно позволяют методы `addToolBar()` и `insertToolBar()` из класса `QMainWindow`. Один из форматов метода `addToolBar()` позволяет указать область, к которой изначально прикреплена панель. По умолчанию с помощью мыши пользователь может переместить панель в другую область окна или отобразить панель в отдельном окне. Ограничить перечень областей, к которым можно прикрепить панель, позволяет метод `setAllowedAreas()` из класса `QToolBar`, а запретить отображение панели в отдельном окне позволяет метод `setFloatable()`.

28.3.1. Класс *QToolBar*

Класс `QToolBar` реализует панель инструментов, которую можно перемещать с помощью мыши. Иерархия наследования для класса `QToolBar` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QToolBar
```

Форматы конструктора класса `QToolBar`:

```
<Объект> = QToolBar([parent=<Родитель>])
<Объект> = QToolBar(<Название>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент, а в параметре `<Название>` задается название панели, которое отображается в контекстном меню, при щелчке пра-

вой кнопкой мыши в области меню, панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить панель инструментов.

Класс `QToolBar` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `addAction()` — добавляет действие на панель инструментов. Форматы метода:

```
addAction(<QAction>)
addAction(<Название>)                                -> QAction
addAction(<QIcon>, <Название>)                       -> QAction
addAction(<Название>, <Объект>, <Слот>)              -> QAction
addAction(<Название>, <Обработчик>)                  -> QAction
addAction(<QIcon>, <Название>, <Объект>, <Слот>)     -> QAction
addAction(<QIcon>, <Название>, <Обработчик>)         -> QAction
```

  Добавить действия на панель инструментов и удалить их позволяют следующие методы из класса `QWidget`:

  - `addActions(<Список с экземплярами класса QAction>)` — добавляет несколько пунктов в конец панели;
  - `insertAction(<QAction 1>, <QAction 2>)` — добавляет действие `<QAction 2>` перед действием `<QAction 1>`;
  - `insertActions(<QAction>, <Список с экземплярами класса QAction>)` — добавляет несколько действий, указанных во втором параметре перед действием `<QAction>`;
  - `actions()` — возвращает список с действиями (экземпляры класса `QAction`);
  - `removeAction(<QAction>)` — удаляет указанное действие;

- `addSeparator()` — добавляет разделитель и возвращает экземпляр класса `QAction`;
- `insertSeparator(<QAction>)` — добавляет разделитель перед указанным действием и возвращает экземпляр класса `QAction`;
- `addWidget(<QWidget>)` — позволяет добавить компонент, например, раскрывающийся список. Метод возвращает экземпляр класса `QAction`;
- `insertWidget(<QAction>, <QWidget>)` — добавляет компонент перед указанным действием и возвращает экземпляр класса `QAction`;
- `widgetForAction(<QAction>)` — возвращает ссылку на компонент, который связан с указанным действием;
- `clear()` — удаляет все действия из панели инструментов;
- `setAllowedAreas(<Области>)` — задает области, к которым можно прикрепить панель инструментов. В качестве параметра указываются атрибуты (или их комбинация через оператор `|`) `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху), `BottomToolBarArea` (снизу) или `AllToolBarAreas` из класса `QtCore.Qt`;
- `setMovable(<Флаг>)` — если в качестве параметра указано значение `True` (значение по умолчанию), то панель можно перемещать с помощью мыши. Значение `False` запрещает перемещение;
- `isMovable()` — возвращает значение `True`, если панель можно перемещать с помощью мыши, и `False` — в противном случае;

- `setFloatable(<Флаг>)` — если в качестве параметра указано значение `True` (значение по умолчанию), то панель можно использовать как отдельное окно, а если указано значение `False`, то нет;
- `isFloatable()` — возвращает значение `True`, если панель можно использовать как отдельное окно, и `False` — в противном случае;
- `isFloating()` — возвращает значение `True`, если панель отображается в отдельном окне, и `False` — в противном случае;
- `setToolButtonStyle(<Стиль>)` — задает стиль кнопок на панели инструментов. Метод является слотом с сигнатурой `setToolButtonStyle(Qt::ToolButtonStyle)`. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt`:
  - `ToolButtonIconOnly` — `0` — отображается только иконка;
  - `ToolButtonTextOnly` — `1` — отображается только текст;
  - `ToolButtonTextBesideIcon` — `2` — текст отображается справа от иконки;
  - `ToolButtonTextUnderIcon` — `3` — текст отображается под иконкой;
  - `ToolButtonFollowStyle` — `4` — зависит от используемого стиля;
- `toolButtonStyle()` — возвращает стиль кнопок на панели инструментов;
- `setIconSize(<QSize>)` — задает размеры иконок. Метод является слотом с сигнатурой `setIconSize(const QSize&)`;
- `iconSize()` — возвращает размеры иконок (экземпляр класса `QSize`);
- `toggleViewAction()` — возвращает объект действия (экземпляр класса `QAction`), с помощью которого можно скрыть или отобразить панель.

Класс `QToolBar` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

- `actionTriggered(QAction *)` — генерируется при нажатии кнопки на панели. Внутри обработчика через параметр доступен экземпляр класса `QAction`;
- `visibilityChanged(bool)` — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение `True`, если панель видима, и `False` — если скрыта;
- `topLevelChanged(bool)` — генерируется при изменении положения панели. Внутри обработчика через параметр доступно значение `True`, если панель отображается в отдельном окне, и `False` — если прикреплена к области.

28.3.2. Класс *QToolButton*

При добавлении действия на панель инструментов автоматически создается кнопка, реализуемая классом `QToolButton`. Получить ссылку на кнопку позволяет метод `widgetForAction(<QAction>)` из класса `QToolBar`. Иерархия наследования для класса `QToolButton` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QToolButton
```

Конструктор класса `QToolButton` имеет следующий формат:

```
<Объект> = QToolButton([parent=<Родитель>])
```

Класс QToolButton наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- setDefaultAction(<QAction>) — связывает объект действия с кнопкой. Метод является слотом с сигнатурой setDefaultAction(QAction *);
- defaultAction() — возвращает ссылку на объект действия (экземпляр класса QAction), связанный с кнопкой;
- setToolButtonStyle(<Стиль>) — задает стиль кнопки. Метод является слотом с сигнатурой setToolButtonStyle(Qt::ToolButtonStyle). Допустимые значения параметра <Стиль> см. в *разд. 28.3.1* в описании метода setToolButtonStyle();
- toolButtonStyle() — возвращает стиль кнопки;
- setMenu(<QMenu>) — добавляет меню;
- menu() — возвращает ссылку на меню (экземпляр класса QMenu) или значение None;
- showMenu() — отображает меню, связанное с кнопкой. Метод является слотом;
- setPopupMode(<Режим>) — задает режим отображения связанного с кнопкой меню. В качестве параметра указываются следующие атрибуты из класса QToolButton:
  - DelayedPopup — 0 — меню отображается при удержании кнопки нажатой некоторый промежуток времени;
  - MenuButtonPopup — 1 — справа от кнопки отображается кнопка со стрелкой, нажатие которой приводит к немедленному отображению меню;
  - InstantPopup — 2 — нажатие кнопки приводит к немедленному отображению меню. Сигнал triggered() при этом не генерируется;
- popupMode() — возвращает режим отображения связанного с кнопкой меню;
- setArrowType(<Тип иконки>) — позволяет вместо стандартной иконки действия установить иконку в виде стрелки, указывающей в заданном направлении. В качестве параметра указываются атрибуты NoArrow (значение по умолчанию), UpArrow, DownArrow, LeftArrow или RightArrow из класса QtCore.Qt;
- setAutoRaise(<Флаг>) — если в качестве параметра указано значение False, то кнопка будет отображаться с рамкой. По умолчанию кнопка сливается с фоном, а при наведении указателя мыши становится выпуклой.

Класс QToolButton содержит сигнал triggered(QAction *), который генерируется при нажатии кнопки или комбинации клавиш, а также при выборе пункта в связанном меню. Внутри обработчика через параметр доступен экземпляр класса QAction.

28.4. Прикрепляемые панели

Если возможностей панелей инструментов недостаточно и необходимо добавить компоненты, занимающие много места (например, таблицу или иерархический список), то можно воспользоваться прикрепляемыми панелями. Прикрепляемые панели реализуются с помощью класса QDockWidget. Иерархия наследования для класса QDockWidget выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDockWidget
```

Форматы конструктора класса `QDockWidget`:

```
<Объект> = QDockWidget([parent=<Родитель>][, flags=<Тип окна>])
<Объект> = QDockWidget(<Название>[, parent=<Родитель>][,
                       flags=<Тип окна>])
```

В параметре `<Название>` задается название панели, которое отображается в заголовке панели и в контекстном меню, при щелчке правой кнопкой мыши в области меню, панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить прикрепляемую панель.

Класс `QDockWidget` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setWidget(<QWidget>)` — устанавливает компонент, который будет отображаться на прикрепляемой панели;
- `widget()` — возвращает ссылку на компонент, расположенный внутри панели;
- `setTitleBarWidget(<QWidget>)` — позволяет указать пользовательский компонент, отображаемый в заголовке панели;
- `titleBarWidget()` — возвращает ссылку на пользовательский компонент, расположенный в заголовке панели, или значение `None`, если компонент не был установлен;
- `setAllowedAreas(<Области>)` — задает области, к которым можно прикрепить панель. В качестве параметра указываются атрибуты (или их комбинация через оператор `|`) `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху), `BottomDockWidgetArea` (снизу) или `AllDockWidgetAreas` из класса `QtCore.Qt`;
- `setFloating(<Флаг>)` — если в качестве параметра указано значение `True`, то панель будет отображаться в отдельном окне, а если указано значение `False`, то панель будет прикреплена к какой-либо области;
- `isFloating()` — возвращает значение `True`, если панель отображается в отдельном окне, и `False` — в противном случае;
- `setFeatures(<Свойства>)` — устанавливает свойства панели. В качестве параметра указывается комбинация (через оператор `|`) следующих атрибутов из класса `QDockWidget`:
  - `DockWidgetClosable` — панель можно закрыть;
  - `DockWidgetMovable` — панель можно перемещать с помощью мыши;
  - `DockWidgetFloatable` — панель может отображаться в отдельном окне;
  - `DockWidgetVerticalTitleBar` — заголовок панели отображается с левой стороны, а не сверху;
  - `NoDockWidgetFeatures` — панель нельзя закрыть, перемещать и она не может отображаться в отдельном окне;
- `features()` — возвращает комбинацию установленных свойств панели;
- `toggleViewAction()` — возвращает объект действия (экземпляр класса `QAction`), с помощью которого можно скрыть или отобразить панель.

Класс `QDockWidget` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

- `dockLocationChanged(Qt::DockWidgetArea)` — генерируется при изменении области. Внутри обработчика через параметр доступна новая область, к которой прикреплена панель;
- `visibilityChanged(bool)` — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение `True`, если панель видима, и `False` — если скрыта;
- `topLevelChanged(bool)` — генерируется при изменении положения панели. Внутри обработчика через параметр доступно значение `True`, если панель отображается в отдельном окне, и `False` — если прикреплена к области.

28.5. Управление строкой состояния

Класс `QStatusBar` реализует строку состояния, в которую можно выводить различные сообщения. Помимо текстовой информации в строку состояния можно добавить различные компоненты, например, индикатор хода выполнения процесса. Строка состояния состоит из трех секций:

- *секция для временных сообщений.* Секция реализована по умолчанию. В эту секцию, например, выводятся сообщения, сохраненные в объекте действия с помощью метода `setStatusTip()`, при наведении указателя мыши на пункт меню или кнопку на панели инструментов. Вывести пользовательское сообщение во временную секцию можно с помощью метода `showMessage()`;
- *обычная секция.* При выводе временного сообщения содержимое обычной секции скрывается. Чтобы отображать сообщения в обычной секции, необходимо предварительно добавить туда компоненты с помощью метода `addWidget()` или `insertWidget()`. Добавленные компоненты выравниваются по левой стороне строки состояния;
- *постоянная секция.* При выводе временного сообщения содержимое постоянной секции не скрывается. Чтобы отображать сообщения в постоянной секции, необходимо предварительно добавить туда компоненты с помощью метода `addPermanentWidget()` или `insertPermanentWidget()`. Добавленные компоненты выравниваются по правой стороне строки состояния.

Получить ссылку на строку состояния, установленную в главном окне, позволяет метод `statusBar()` из класса `QMainWindow`, а установить пользовательскую панель вместо стандартной можно с помощью метода `setStatusBar(<QStatusBar>)`. Иерархия наследования для класса `QStatusBar` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QStatusBar
```

Формат конструктора класса `QStatusBar`:

```
<Объект> = QStatusBar([parent=<Родитель>])
```

Класс `QStatusBar` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `showMessage(<Текст>[, msecs=0])` — выводит временное сообщение в строку состояния. Во втором параметре можно указать время в миллисекундах, на которое показывается сообщение. Если во втором параметре указано значение `0`, то сообщение показывается, пока не будет выведено новое сообщение или вызван метод `clearMessage()`. Метод является слотом с сигнатурой `showMessage(const QString&,int = 0)`;

- currentMessage() — возвращает временное сообщение, отображаемое в строке состояния;
- clearMessage() — удаляет временное сообщение из строки состояния. Метод является слотом;
- addWidget(<QWidget>[, stretch=0]) — добавляет указанный компонент в конец обычной секции. В параметре stretch может быть указан фактор растяжения;
- insertWidget(<Индекс>, <QWidget>[, stretch=0]) — добавляет компонент в указанную позицию обычной секции и возвращает индекс позиции. В параметре stretch может быть указан фактор растяжения;
- addPermanentWidget(<QWidget>[, stretch=0]) — добавляет указанный компонент в конец постоянной секции. В параметре stretch может быть указан фактор растяжения;
- insertPermanentWidget(<Индекс>, <QWidget>[, stretch=0]) — добавляет компонент в указанную позицию постоянной секции и возвращает индекс позиции. В параметре stretch может быть указан фактор растяжения;
- removeWidget(<QWidget>) — удаляет компонент из обычной или постоянной секций. Обратите внимание на то, что сам компонент не удаляется, а только скрывается и лишается родителя. В дальнейшем компонент можно добавить в другое место;
- setSizeGripEnabled(<Флаг>) — если в качестве параметра указано значение True, то в правом нижнем углу строки состояния будет отображаться маркер изменения размера. Значение False скрывает маркер.

Класс QStatusBar содержит сигнал messageChanged(const QString&), который генерируется при изменении текста во временной секции. Внутри обработчика через параметр доступно новое сообщение или пустая строка.

28.6. MDI-приложения

MDI-приложения (Multiple Document Interface) позволяют отображать сразу несколько документов одновременно в разных вложенных окнах. Чтобы создать MDI-приложение, следует в качестве центрального компонента установить компонент QMdiArea с помощью метода setCentralWidget() из класса QMainWindow. Отдельное окно внутри MDI-области реализуется с помощью класса QMdiSubWindow.

28.6.1. Класс *QMdiArea*

Класс QMdiArea реализует MDI-область, внутри которой могут располагаться вложенные окна (экземпляры класса QMdiSubWindow). Иерархия наследования для класса QMdiArea выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QMdiArea
```

Конструктор класса QMdiArea имеет следующий формат:

```
<Объект> = QMdiArea([parent=<Родитель>])
```

Класс QMdiArea содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- addSubWindow(<QWidget>[, flags=<Тип окна>]) — создает вложенное окно, добавляет в него компонент <QWidget> и возвращает ссылку на созданное окно (экземпляр класса QMdiSubWindow). Пример:

```
w = MyWidget()
sWindow = self.mdi_area.addSubWindow(w)
sWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose)
# ... Производим настройку свойств окна
sWindow.show()
```

  Чтобы окно автоматически удалялось при закрытии, необходимо установить атрибут WA_DeleteOnClose, а чтобы отобразить окно, следует вызвать метод show().

  В параметре <QWidget> можно также указать ссылку на существующее вложенное окно (экземпляр класса QMdiSubWindow). Пример:

```
w = MyWidget()
sWindow = QtGui.QMdiSubWindow()
self.mdi_area.addSubWindow(sWindow)
sWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose)
# ... Производим настройку свойств окна
sWindow.setWidget(w)
sWindow.show()
```

  Также можно добавить вложенное окно в MDI-область, указав MDI-область в качестве родителя при создании объекта вложенного окна. Пример:

```
sWindow = QtGui.QMdiSubWindow(self.mdi_area)
```

- activeSubWindow() — возвращает ссылку на активное вложенное окно (экземпляр класса QMdiSubWindow) или значение None;
- currentSubWindow() — возвращает ссылку на текущее вложенное окно (экземпляр класса QMdiSubWindow) или значение None. Результат выполнения этого метода аналогичен результату выполнения метода activeSubWindow(), если MDI-область находится в активном окне;
- subWindowList([order=CreationOrder]) — возвращает список со ссылками на все вложенные окна (экземпляры класса QMdiSubWindow), добавленные в MDI-область, или пустой список. В параметре order указываются следующие атрибуты из класса QMdiArea:
  - CreationOrder — 0 — окна в списке расположены в порядке добавления в MDI-область;
  - StackingOrder — 1 — окна в списке расположены в порядке расположения. Последний элемент в списке будет содержать ссылку на самое верхнее окно, а последний элемент — ссылку на самое нижнее окно;
  - ActivationHistoryOrder — 2 — окна в списке расположены в порядке истории получения фокуса. Последний элемент в списке будет содержать ссылку на окно, получившее фокус последним;
- removeSubWindow(<QWidget>) — удаляет вложенное окно из MDI-области;
- setActiveSubWindow(<QMdiSubWindow>) — делает указанное вложенное окно активным. Если в качестве параметра указано значение None, то все окна станут неактивными. Метод является слотом с сигнатурой setActiveSubWindow(QMdiSubWindow *);

- `setActivationOrder(<order>)` — задает режим передачи фокуса при использовании методов `activatePreviousSubWindow()`, `activateNextSubWindow()` и др. В параметре `<order>` указываются такие же атрибуты, как и в параметре `order` метода `subWindowList()`;
- `activationOrder()` — возвращает режим передачи фокуса;
- `activatePreviousSubWindow()` — делает активным предыдущее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`;
- `activateNextSubWindow()` — делает активным следующее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`;
- `closeActiveSubWindow()` — закрывает активное вложенное окно. Метод является слотом;
- `closeAllSubWindows()` — закрывает все вложенные окна. Метод является слотом;
- `cascadeSubWindows()` — упорядочивает вложенные окна, располагая их каскадом. Метод является слотом;
- `tileSubWindows()` — упорядочивает вложенные окна, располагая их мозаикой. Метод является слотом;
- `setViewMode(<Режим>)` — задает режим отображения документов в MDI-области. В параметре `<Режим>` указываются следующие атрибуты из класса `QMdiArea`:
  - `SubWindowView` — `0` — в отдельном окне с рамкой (по умолчанию);
  - `TabbedView` — `1` — на отдельной вкладке панели с вкладками;
- `viewMode()` — возвращает режим отображения документов в MDI-области;
- `setTabPosition(<Позиция>)` — задает позицию отображения заголовков вкладок при использовании режима `TabbedView`. По умолчанию заголовки вкладок отображаются сверху. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты из класса `QTabWidget`:
  - `North` — `0` — сверху;
  - `South` — `1` — снизу;
  - `West` — `2` — слева;
  - `East` — `3` — справа;
- `tabPosition(<Область>)` — возвращает позицию отображения заголовков вкладок при использовании режима `TabbedView`;
- `setTabShape(<Форма>)` — задает форму углов ярлыков вкладок при использовании режима `TabbedView`. Могут быть указаны следующие атрибуты из класса `QTabWidget`:
  - `Rounded` — `0` — скругленные углы (значение по умолчанию);
  - `Triangular` — `1` — треугольная форма;
- `tabShape()` — возвращает форму углов ярлыков вкладок в области заголовка при использовании режима `TabbedView`;
- `setBackground(<QBrush>)` — задает кисть для заполнения фона MDI-области;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, то производит установку опции, а если `False` — то сбрасывает опцию. В параметре `<Опция>` может быть указан атрибут `DontMaximizeSubWindowOnActivation` из класса `QMdiArea`. Если

эта опция установлена, то при передаче фокуса из максимально раскрытого окна отображаемое окно не будет максимально раскрываться;

- `testOption(<Опция>)` — возвращает значение `True`, если указанная опция установлена, и `False` — в противном случае.

Класс `QMdiArea` содержит сигнал `subWindowActivated(QMdiSubWindow *)`, который генерируется при изменении активности вложенных окон. Внутри обработчика через параметр доступна ссылка на активное вложенное окно (экземпляр класса `QMdiSubWindow`) или значение `None`.

28.6.2. Класс *QMdiSubWindow*

Класс `QMdiSubWindow` реализует окно, которое может быть отображено внутри MDI-области. Иерархия наследования для класса `QMdiSubWindow` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMdiSubWindow
```

Формат конструктора класса `QMdiSubWindow`:

```
<Объект> = QMdiSubWindow([parent=<Родитель>][, flags=<Тип окна>])
```

Класс `QMdiSubWindow` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setWidget(<QWidget>)` — добавляет компонент в окно;
- `widget()` — возвращает ссылку на компонент внутри окна;
- `mdiArea()` — возвращает ссылку на MDI-область (экземпляр класса `QMdiArea`) или значение `None`;
- `setSystemMenu(<QMenu>)` — позволяет установить пользовательское системное меню окна вместо стандартного;
- `systemMenu()` — возвращает ссылку на системное меню окна (экземпляр класса `QMenu`) или значение `None`;
- `showSystemMenu()` — вызывает отображение системного меню окна. Метод является слотом;
- `setKeyboardSingleStep(<Значение>)` — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками. Чтобы изменить размеры окна или его положение с помощью клавиатуры, необходимо в системном меню окна выбрать пункт **Move** или **Size**. Значение по умолчанию — `5`;
- `setKeyboardPageStep(<Значение>)` — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками при удержании нажатой клавиши <Shift>. Значение по умолчанию — `20`;
- `showShaded()` — сворачивает содержимое окна, оставляя только заголовок. Метод является слотом;
- `isShaded()` — возвращает значение `True`, если отображается только заголовок, и `False` — в противном случае;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, то производит установку опции, а если `False` — то сбрасывает опцию.

В параметре <Опция> могут быть указаны следующие атрибуты из класса QMdiSubWindow:

- RubberBandResize — если опция установлена, то при изменении размеров окна будут изменяться размеры вспомогательного компонента, а не самого окна. По окончании изменения размеров будут изменены размеры окна;
- RubberBandMove — если опция установлена, то при изменении положения окна будет перемещаться вспомогательный компонент, а не само окно. По окончании перемещения будет изменено положение окна;

- testOption(<Опция>) — возвращает значение True, если указанная опция установлена, и False — в противном случае.

Класс QMdiSubWindow содержит следующие сигналы:

- aboutToActivate() — генерируется перед отображением вложенного окна;
- windowStateChanged(Qt::WindowStates,Qt::WindowStates) — генерируется при изменении статуса окна. Внутри обработчика через первый параметр доступен старый статус, а через второй параметр — новый статус.

28.7. Добавление иконки приложения в область уведомлений

Класс QSystemTrayIcon позволяет добавить иконку приложения в область уведомления, расположенную в правой части **Панели задач** в Windows. Иерархия наследования для класса QSystemTrayIcon выглядит так:

```
QObject — QSystemTrayIcon
```

Форматы конструктора класса QSystemTrayIcon:

```
<Объект> = QSystemTrayIcon([parent=<Родитель>])
<Объект> = QSystemTrayIcon(<QIcon>[, parent=<Родитель>])
```

Класс QSystemTrayIcon содержит следующие основные методы:

- isSystemTrayAvailable() — возвращает значение True, если можно отобразить иконку в области уведомлений, и False — в противном случае. Метод является статическим;
- setIcon(<QIcon>) — устанавливает иконку. Установить иконку можно также в конструкторе класса;
- icon() — возвращает иконку (экземпляр класса QIcon);
- setContextMenu(<QMenu>) — устанавливает контекстное меню, отображаемое при щелчке правой кнопкой мыши на иконке;
- contextMenu() — возвращает ссылку на контекстное меню (экземпляр класса QMenu);
- setToolTip(<Текст>) — задает текст всплывающей подсказки;
- toolTip() — возвращает текст всплывающей подсказки;
- setVisible(<Флаг>) — если в качестве параметра указано значение True, то отображает иконку, а если False — то скрывает иконку. Метод является слотом с сигнатурой setVisible(bool);
- show() — отображает иконку. Метод является слотом;
- hide() — скрывает иконку. Метод является слотом;

- `isVisible()` — возвращает значение `True`, если иконка видна, и `False` — в противном случае;
- `geometry()` — возвращает экземпляр класса `QRect` с размерами и координатами иконки на экране;
- `showMessage()` — позволяет отобразить сообщение в области уведомлений. Формат метода:

```
showMessage(<Заголовок>, <Текст сообщения>[, icon=Information][,
            msecs=10000])
```

  Необязательный параметр `icon` задает иконку, которая отображается слева от заголовка сообщения. В качестве значения можно указать атрибуты `NoIcon`, `Information`, `Warning` или `Critical` из класса `QSystemTrayIcon`. Необязательный параметр `msecs` задает промежуток времени, на который отображается сообщение. Обратите внимание на то, что сообщение может не показываться вообще, кроме того, значение параметра `msecs` в некоторых операционных системах игнорируется;
- `supportsMessages()` — возвращает значение `True`, если вывод сообщений поддерживается, и `False` — в противном случае. Метод является статическим.

Класс `QSystemTrayIcon` содержит следующие сигналы:

- `activated(QSystemTrayIcon::ActivationReason)` — генерируется при щелчке мышью на иконке. Внутри обработчика через параметр доступна причина в виде значения следующих атрибутов из класса `QSystemTrayIcon`:
  - `Unknown` — `0` — неизвестная причина;
  - `Context` — `1` — нажата правая кнопка мыши;
  - `DoubleClick` — `2` — двойной щелчок мышью;
  - `Trigger` — `3` — нажата левая кнопка мыши;
  - `MiddleClick` — `4` — нажата средняя кнопка мыши;
- `messageClicked()` — генерируется при щелчке мышью в области сообщения.

Заключение

Вот и закончилось наше путешествие в мир Python и PyQt. Материал книги описывает лишь базовые возможности этих замечательных технологий. В этом заключении мы рассмотрим, где найти дополнительную информацию и продолжить изучение.

Самыми важными источниками информации являются официальные сайты **http://www.python.org/** и **http://www.riverbankcomputing.co.uk/**. На них вы найдете дистрибутивы, новости, а также ссылки на все другие ресурсы в Интернете. Не следует также забывать, что библиотека PyQt является надстройкой над библиотекой Qt, поэтому посещение сайта **http://qt.nokia.com/** лишним не будет.

На сайте **http://docs.python.org/** расположена документация по Python, которая обновляется в режиме реального времени. Язык постоянно совершенствуется, появляются новые функции, изменяются параметры, добавляются модули и т. д. Регулярно посещайте этот сайт, и вы получите самую последнюю информацию.

В пакет установки Python входит большое количество модулей, позволяющих решить наиболее часто встречающиеся задачи. Однако на этом возможности Python не заканчиваются. Мир Python включает множество самых разнообразных модулей и целых библиотек, созданных сторонними разработчиками и доступных для свободного скачивания. На странице **http://pypi.python.org/pypi?%3Aaction=index** и сайте **http://sourceforge.net/** вы сможете найти довольно большой список различных модулей. При выборе модуля необходимо учитывать версию Python. Обычно версия указывается в составе названия дистрибутива.

Особенно необходимо отметить библиотеку PySide (**http://www.pyside.org/**), разрабатываемую специалистами компании Nokia. Эта библиотека является полным аналогом PyQt и распространяется по лицензии LGPL. Не забывайте также о существовании других библиотек для создания графического интерфейса: Tkinter, wxPython (**http://www.wxpython.org/**), PyGTK (**http://www.pygtk.org/**), PyWin32 (**http://sourceforge.net/projects/pywin32/**) и pyFLTK (**http://pyfltk.sourceforge.net/**). Обратите также внимание на библиотеку pygame (**http://www.pygame.org/**), позволяющую разрабатывать игры, и на фреймворк Django (**http://www.djangoproject.com/**), с помощью которого можно создавать Web-приложения.

Если в процессе изучения возникнут вопросы, то не следует забывать, что Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, **http://www.google.com/**) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобной проблемой и описал решение на каком-либо сайте.

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства по адресу: **ftp://85.249.45.166/9785977507974.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Структура архива представлена в табл. П.1.

Таблица П.1. *Структура электронного архива*

Файл	Описание
Listings.doc	Содержит все пронумерованные листинги из книги
PyQt.doc	Содержит более 750 дополнительных листингов, которые демонстрируют возможности библиотеки PyQt
Readme.doc	Описание электронного архива

Предметный указатель

@

@abstractmethod 243
@classmethod 242
@staticmethod 242

_

__abs__() 240
__add__() 239
__all__ 221, 226
__and__() 240
__annotations__ 215
__bases__ 235
__bool__() 238
__call__() 236
__cause__ 257
__complex__() 239
__conform__() 333
__contains__() 241
__debug__ 258
__del__() 232
__delattr__() 238
__delitem__() 237
__dict__ 213, 219, 237
__doc__ 34, 35, 79
__enter__() 252, 253
__eq__() 241
__exit__() 252, 253
__file__ 262
__float__() 238
__floordiv__() 240
__ge__() 241
__getattr__() 237, 244
__getattribute__() 237, 244
__getitem__() 236
__gt__() 241
__hash__() 239
__iadd__() 239
__iand__() 241
__ifloordiv__() 240
__ilshift__() 241
__imod__() 240
__import__() 219
__imul__() 240
__index__() 239
__init__() 231
__int__() 238
__invert__() 240
__ior__() 241
__ipow__() 240
__irshift__() 241
__isub__() 239
__iter__() 238
__itruediv__() 240
__ixor__() 241
__le__() 241
__len__() 238
__lshift__() 241
__lt__() 241
__mod__() 240
__mro__ 236
__mul__() 239
__name__ 216
__ne__() 241
__neg__() 240
__next__() 40, 207, 238, 256, 268, 275, 324, 355
__or__() 241
__pos__() 240
__pow__() 240
__radd__() 239
__rand__() 241
__repr__() 239
__rfloordiv__() 240

__rlshift__() 241
__rmod__() 240
__rmul__() 240
__ror__() 241
__round__() 239
__rpow__() 240
__rrshift__() 241
__rshift__() 241
__rsub__() 239
__rtruediv__() 240
__rxor__() 241
__setattr__() 237, 244
__setitem__() 236
__slots__ 245
__str__() 239
__sub__() 239
__truediv__() 240
__xor__() 241

A

Abc 243
ABCMeta 243
Abort 611, 613
about() 618
aboutQt() 618
aboutToActivate() 663
aboutToHide() 648
aboutToShow() 648
abs() 37, 71, 179, 240
abspath() 260, 262, 281
Accept 353, 625
accept() 386, 425, 442, 448, 455, 456, 458, 463, 464, 601, 604, 608, 610
Accept-Charset 353
Accepted 608, 609, 612, 619, 624
Accept-Encoding 353
acceptHoverEvents() 603
Accept-Language 353
AcceptOpen 624
acceptProposedAction() 463, 464, 604
acceptRichText() 501
AcceptRole 611, 612, 614
AcceptSave 624
access() 278
AccessibleDescriptionRole 528
AccessibleTextRole 528
accumulate() 158
acos() 72
actionChanged() 461
actionGroup() 651
ActionRole 61–1, 614
actions() 647, 652, 654
actionTriggered() 520, 655
activated() 517, 526, 538, 664
activateNextSubWindow() 661
activatePreviousSubWindow() 661
activateWindow() 415, 609
ActivationChange 444
ActivationHistoryOrder 660
activationOrder() 661
Active 420
activeAction() 645, 647
ActivePython 20
activeSubWindow() 660
activeWindow() 581
ActiveWindowFocusReason 449
actualSize() 577
add() 171
addAction() 645, 646, 652, 654
addActions() 646, 654
addButton() 611, 612, 615
addDockWidget() 642
addEllipse() 580
addFile() 576
addItem() 484, 523, 579
addLayout() 470, 473
addLine() 579
addMenu() 645, 647
addPage() 634
addPath() 580
addPermanentWidget() 658, 659
addPixmap() 576, 580
addPolygon() 580
addRect() 580
addRow() 474
addSeparator() 647, 654
addSimpleText() 580
addSpacing() 471
addStretch() 471
addSubWindow() 660
addTab() 481
addText() 580
addToGroup() 597
addToolBar() 641, 653
addToolBarBreak() 642
addWidget() 366, 429, 470—473, 476, 486, 580, 654, 658, 659
Adjust 412, 539
adjustSize() 402
AdjustToContents 525
AdjustToContentsOnFirstShow 525
AdjustToMinimumContentsLength 525

AdjustToMinimumContentsLengthWithIcon 525
AeroStyle 635
AlignAbsolute 471
AlignBottom 396, 471, 565
AlignCenter 369, 396, 471, 565
AlignHCenter 396, 471, 479, 565
AlignJustify 471
AlignLeft 396, 471, 479, 565
alignment() 505
AlignRight 396, 471, 479, 565
AlignTop 396, 471, 565
AlignVCenter 396, 471, 565
all() 147
AllDockWidgetAreas 657
AllEditTriggers 537
AllFonts 527
AllLayers 584
AllNonFixedFieldsGrow 475
AllowNestedDocks 642, 643
AllowTabbedDocks 642
AllToolBarAreas 654
alpha() 553, 554
alphaF() 553—555
AltModifier 454
AmPmSection 515
anchorClicked() 512
and 57
animateClick() 493
AnimatedDocks 642
answerRect() 465
Antialiasing 565
any() 147
AnyFile 624
AnyKeyPressed 537
Apilevel 318
append() 110, 133, 134, 144, 196, 222, 500, 559
appendColumn() 531, 534
appendRow() 531, 534
ApplicationActivate 444
ApplicationDeactivate 444
ApplicationModal 417, 418, 609
ApplicationShortcut 452, 650
Apply 611, 613
ApplyRole 611, 614
Argv 32, 365, 366
ArithmeticError 255
Arraysize 324
ArrowCursor 458
as 218, 220, 225, 253
as_string() 351
AscendingOrder 530, 533, 536, 542, 544, 549, 581
ASCII 115, 118, 119
ascii() 91, 94
asctime() 176
asin() 72
assert 255, 258
AssertionError 255, 258
Assistant 361
astimezone() 188
atan() 72
atBlockEnd() 510
atBlockStart() 510
atEnd() 509
atStart() 509
AttributeError 217, 229, 237, 245, 255
AutoAll 503
AutoBulletList 503
AutoCompatConnection 430
autoCompletion() 525
AutoConnection 429, 430, 435
AutoNone 503
AutoText 490, 615, 635
availableGeometry() 405, 406
availableRedoSteps() 506
availableSizes() 576
availableUndoSteps() 506

B

back() 634
BackButton 635
Background 420
BackgroundLayer 583
BackgroundPixmap 636
BackgroundRole 527
backspace() 496
BacktabFocusReason 449
backward() 511
backwardAvailable() 512
backwardHistoryCount() 511
BannerPixmap 636
BaseException 255
basename() 282
baseSize() 402
Batched 540
begin() 562
beginEditBlock() 510
BevelJoin 556
Bin 518
bin() 71, 239
Black 396, 504, 507, 552, 554, 560

BlankCursor 458
block() 509
blockCount() 507
blockCountChanged() 507
BlockingQueuedConnection 430
blockNumber() 509
blockSignals() 430
BlockUnderCursor 510
Blue 552, 553
blurRadius() 598, 599
blurRadiusChanged() 599
Bold 504, 507, 560
BOM 23, 264
bool() 38, 44, 55, 238
bottom() 412
BottomDockWidgetArea 642, 643, 657
bottomLeft() 413
BottomLeftCorner 643
bottomRight() 413
BottomRightCorner 643
BottomToolBarArea 641, 654
BottomToTop 472, 519
boundedTo() 409
boundingRect() 559, 561, 565, 566, 588
BoundingRectShape 595
Box 480
break 62, 66—68
bspTreeDepth() 579
BspTreeIndex 579
Buffer 270
Builtins 34
BusyCursor 458
button() 456, 602, 612, 615, 635
buttonClicked() 615
buttonDownPos() 602
buttonDownScenePos() 602
buttonDownScreenPos() 602
buttonRole() 612, 615
buttons() 456, 458, 602, 603, 605, 612, 615
buttonText() 636, 637
Byte Order Mark 23, 264
Bytearray 38, 45, 46, 77, 105, 108, 109
Bytes 38, 45, 77, 105, 106, 263, 327, 332
BytesIO 276

C

CacheBackground 585
Cache-Control 353
CacheNone 585
Calendar 173, 189, 192, 194
Cancel 610, 613, 633
CancelButton 635
CancelButtonOnLeft 636
canceled() 633
canPaste() 501
capitalize() 98
cascadeSubWindows() 661
CaseInsensitive 506, 525, 549, 550
CaseSensitive 525, 549, 550
ceil() 73
center() 89, 413
centerOn() 587
centralWidget() 640
cgi 348
chain() 158
changed() 584, 652
changeEvent() 445
changeOverrideCursor() 459
characterCount() 507
chardet 357
chdir() 262, 290
Checked 494, 528, 535
checkedAction() 652
checkOverflow() 518
checkState() 494, 535
CheckStateRole 528
child() 529, 534
ChildAdded 443
childItems() 591
ChildPolished 443
ChildRemoved 443
chmod() 278
choice() 74, 75, 148
chr() 98
Class 228, 229
ClassicStyle 635, 636
cleanText() 513
cleanupPage() 634, 636, 639
clear() 166, 171, 289, 463, 465, 482, 491, 496, 501, 505, 513, 524, 531, 548, 559, 573, 580, 612, 645, 647, 654
ClearAndSelect 548
clearEditText() 525
clearFocus() 449, 582, 590
clearHistory() 511
clearMessage() 396, 659
clearSelection() 510, 537, 548, 583
clearSpans() 541
clearUndoRedoStacks() 506
click() 434, 493
clicked() 367, 384, 426, 427, 431, 433, 479, 492, 493, 517, 538, 612
clickedButton() 615

ClickFocus 450
Clipboard 443, 465
clone() 535
close() 265, 272, 273, 288, 319, 349, 355, 358, 424, 425, 443, 448, 610, 613, 650
closeActiveSubWindow() 661
closeAllSubWindows() 661
closeAllWindows() 424
Closed 270, 273
ClosedHandCursor 458
closeEvent() 386, 425, 442, 445, 448
Cmath 72
Cmyk 555
Code 355
collapse() 544
collapseAll() 544
collapsed() 544
collidesWithItem() 592
collidingItems() 581, 591
color() 598, 599
color0 552, 572, 573
color1 552, 572
colorChanged() 599, 600
colorNames() 552
column() 529, 533
columnCount() 473, 531, 533
columnIntersectsSelection() 547
Columns 548
columnSpan() 541
columnWidth() 541, 543
combinations() 154
combinations_with_replacement() 155
combine() 186
commit() 321, 327, 328, 339
CommitButton 635
compile() 114, 123, 124, 127, 128
complete_statement() 338
completeChanged() 638
complex 38, 69
complex() 239
compress() 157
Confidence 357
connect() 319, 328, 334, 367, 369, 383, 384, 426, 428, 429, 431, 435, 438
contains() 413, 414
ContainsItemBoundingRect 582, 583, 585
ContainsItemShape 581, 583, 585
Content-Length 350, 353
contentsChange() 507
contentsChanged() 508
Content-Type 350, 352—354
Context 664
ContextMenu 444, 663
contextMenuEvent() 497, 501, 644, 648, 649
ContiguousSelection 537
Continue 67
ControlModifier 454
convertFromImage() 571
convertTo() 555
convertToFormat() 575
Cookie 353
Copy 135, 161, 166, 170, 172, 279, 454, 497, 501, 571, 575, 650
copy2() 279
CopyAction 460, 461
copyAvailable() 502
copyfile() 278
corner() 643
cornerWidget() 488
cos() 72
count() 65, 100, 147, 153, 476, 483, 485, 487, 524, 545, 559
create_aggregate() 331
create_collation() 328
create_function() 330, 331
createItemGroup() 580, 597
createPopupMenu() 641
createStandardContextMenu() 497, 501
CreationOrder 660
Critical 613, 617, 618, 664
CrossCursor 458
CrossPattern 421, 557
cssclasses() 191
ctime() 176, 182, 188
Current 548
currentChanged() 476, 477, 483, 485, 537, 548, 625
currentCharFormat() 505
currentCharFormatChanged() 502
currentColumnChanged() 549
currentFont() 504, 527
currentFontChanged() 527
currentId() 634
currentIdChanged() 637
currentIndex() 476, 483, 485, 524, 529, 536, 548
currentIndexChanged() 526
currentMessage() 659
currentPage() 634
currentPageChanged() 517
currentRowChanged() 548
currentSection() 515
currentSectionIndex() 515
currentSubWindow() 660

currentText() 524
currentWidget() 476, 483, 485
cursor() 319, 459
cursorBackward() 497
cursorForPosition() 508
cursorForward() 496
cursorPosition() 496
cursorPositionChanged() 497, 502, 507
cursorWordBackward() 497
cursorWordForward() 497
CustomButton1 635, 636, 637
CustomButton2 635, 637
CustomButton3 635, 637
customButtonClicked() 637
CustomDashLine 555
customEvent() 467
CustomizeWindowHint 400
cut() 497, 501, 650
cyan() 552, 554
cyanF() 554
cycle() 154

D

DarkBlue 552
DarkCyan 552
darker() 553
DarkGray 552
DarkGreen 552
DarkMagenta 552
DarkRed 552
DarkYellow 552
DashDotDotLine 542, 555
DashDotLine 542, 555
DashLine 542, 555
data() 463, 529, 530, 532, 535, 651
DatabaseError 337
dataChanged() 465
DataError 337
Date 178, 180—182, 186, 187, 336, 514—517
dateChanged() 515
Datetime 173, 178, 180, 183, 184, 186—188, 336, 514, 515
dateTimeChanged() 515
Day 181, 186
Day_abbr 194
Day_name 194
Days 178, 179
DaySection 515
DB-API 318
Dbm 288, 293
Dec 518
Decimal 50, 70
DecorationRole 527
deepcopy() 135, 161, 166
Def 197, 199, 229
defaultAction() 647, 656
defaultSectionSize() 545
degrees() 73
Del 47, 111, 146, 163, 289
del_() 496
delattr() 230
DelayedPopup 656
deleteChar() 510
deletePreviousChar() 510
deleter() 246
delta() 458, 603
DemiBold 504, 507, 560
Dense1Pattern 421, 557
Dense2Pattern 421, 557
Dense3Pattern 421, 557
Dense4Pattern 421, 557
Dense5Pattern 421, 557
Dense6Pattern 421, 557
Dense7Pattern 421, 557
depth() 571, 574
DescendingOrder 530, 533, 536, 542, 544, 549, 581
Description 323
Deselect 496, 548
Designer 361, 371, 373
Desktop 400, 405
destroyItemGroup() 580, 597
DestructiveRole 611, 614
Detail 624
detect() 357, 358
Detect_types 334
Dialog 400
dict() 39, 159, 161, 351
Dict_items 164
Dict_keys 62, 164
Dict_values 164
difference() 168, 172
difference_update() 168
digest() 113
dir() 35, 219
DirectConnection 429
Directory 624, 625
directoryEntered() 626
dirname() 262, 283
Disabled 420
DisabledBackButtonOnLastPage 636
Discard 171, 611, 613

disconnect() 430, 431, 435
display() 518
DisplayRole 527, 532, 549, 550
displayText() 496
divmod() 72
dockLocationChanged() 658
dockOptions() 643
dockWidgetArea() 642
DockWidgetClosable 657
DockWidgetFloatable 657
DockWidgetMovable 657
DockWidgetVerticalTitleBar 657
document() 505, 510, 596
documentMargin() 507
documentTitle() 500
done() 358, 608, 610
DontConfirmOverwrite 624
DontMaximizeSubWindowOnActivation 661
DontUseNativeDialog 629, 631
DontWrapRows 475
DOTALL 114, 116
DotLine 542, 555
DoubleClick 664
doubleClicked() 537, 539
doubleClickInterval() 455
DoubleInput 619, 620, 621
doubleValue() 620
doubleValueChanged() 621
doubleValueSelected() 621
Down 508
DownArrow 656
DragDrop 538
DragEnter 443
dragEnterEvent() 463, 464, 604
DragLeave 443
dragLeaveEvent() 463, 604
dragMode() 585
DragMove 443
dragMoveEvent() 463, 604
DragOnly 538
draw() 597, 598
drawArc() 564
drawBackground() 579
drawChord() 564
drawEllipse() 564
Drawer 400
drawForeground() 579
drawImage() 566, 567, 573
drawLine() 563
drawLines() 563
drawPicture() 569
drawPie() 564
drawPixmap() 566, 570, 572
drawPoint() 563
drawPoints() 563
drawPolygon() 564
drawPolyline() 563
drawRect() 563
drawRoundedRect() 564
drawText() 565
Drop 443
dropAction() 464, 604
dropEvent() 463, 464, 604
DropOnly 538
dropwhile() 156
dst() 184, 188
dump() 286, 287
dumps() 112, 287
dup() 272
dx() 558
dy() 558

E

East 482, 643, 661
Eclipse 14, 365
editingFinished() 497, 513
EditKeyPressed 537
EditRole 527, 532
editTextChanged() 526
ElideLeft 482, 537
ElideMiddle 482, 538
ElideNone 482, 538
ElideRight 482, 537
Elif 60
Ellipsis 39
Ellipsis 39
Else 62, 66, 251
emit() 382, 383, 388, 433—435
Empty 389, 390
enabledChanged() 598
Encoding 270, 357
End 508
end() 126, 497, 562
endEditBlock() 510
EndOfBlock 508
EndOfLine 508
EndOfWord 508
Endpos 125
endswith() 100
ensureCursorVisible() 501
EnsureVisible 538
ensureVisible() 487, 587, 592
ensureWidgetVisible() 487

Enter 443
entered() 539
enterEvent() 457
enumerate() 65, 140
Env 25
EOFError 255, 284
Error 337
escape() 131, 347, 348
eval() 32
event() 444, 452, 455, 467, 651
eventFilter() 465, 466
exc_info() 249, 253
except 248—251, 256
Exception 255, 257, 337
exec_() 367, 379, 386, 388, 460, 608, 609, 614, 619, 624, 644, 648, 649
execute() 320—322, 331
executemany() 322
executescript() 319, 322
ExistingFile 624
ExistingFiles 624
exists() 279
exit() 365, 367, 379, 386, 424
exp() 73
expand() 126, 544
expandAll() 544
expanded() 544
expandedTo() 409
Expanding 475, 477
ExpandingFieldsGrow 475
expandtabs() 88
expandToDepth() 544
extend() 110, 145
ExtendedSelection 537
ExtendedWatermarkPixmap 636
extension() 608

F

F_OK 278
fabs() 73
factorial() 73
False 38, 55
families() 561
family() 560
FastTransformation 572, 575, 595
fdopen() 272
features() 657
feed() 358
fetchall() 325
fetchmany() 324
fetchone() 324
field() 635, 638
FieldsStayAtSizeHint 475
FileName 625
fileno() 268, 283
fileSelected() 626
filesSelected() 626
FileType 625
fill() 571, 574
Filled 518
fillRect() 564
filter() 143, 144
filterfalse() 156
filterSelected() 626
finalize() 331
Finally 251
find() 99, 501, 506
findall() 127
FindBackward 501, 506
findBlock() 507
findBlockByNumber() 507
FindBuffer 465
FindCaseSensitively 501, 506
findData() 526
findItems() 532
finditer() 128
findText() 526
FindWholeWords 501, 506
finish() 396
FinishButton 635
finished() 382, 384
Firebug 354
firstBlock() 507
firstweekday() 192
fitInView() 587
Fixed 477, 539, 545
FixedColumnWidth 503
FixedPixelWidth 503
flags() 529, 535, 589
Flat 518
FlatCap 556
float() 38, 44, 69, 70, 238
floor() 73, 220
flush() 268, 275, 286
fmod() 73
FocusIn 443, 450
focusInEvent() 450, 601
focusItem() 582
focusNextChild() 449
focusNextPrevChild() 449
FocusOut 443, 450
focusOutEvent() 450, 601
focusPolicy() 450

focusPreviousChild() 449
focusProxy() 449, 601
focusWidget() 449, 450
font() 595, 596, 651
fontFamily() 504
fontItalic() 504
fontPointSize() 504
FontRole 527
fontUnderline() 504
fontWeight() 504
for 30, 40, 61—63, 65, 83, 139—141, 163, 167, 168, 172, 196, 207, 268, 275, 324
ForbiddenCursor 458
ForceTabbedDocks 642
Foreground 420
ForegroundLayer 584
ForegroundRole 527
format() 84, 90, 91, 574
Format_ARGB32 574
Format_ARGB32_Premultiplied 573, 574
format_exception() 249
format_exception_only() 249
Format_Indexed8 574
Format_Invalid 573
Format_Mono 573
Format_MonoLSB 574
Format_RGB32 574
formatmonth() 190, 191
formats() 463
formatyear() 191
formatyearpage() 192
forward() 511
forwardAvailable() 512
forwardHistoryCount() 511
Fractions 70
Fragment 341, 342
frameGeometry() 403, 404
FramelessWindowHint 400
frameSize() 403, 405
Free 539
Friday 517
FRIDAY 190
from 219, 220, 224—227, 258
from_iterable() 158
fromCmyk() 554
fromCmykF() 554
fromData() 574
fromhex() 106, 109
fromHsv() 554
fromHsvF() 554
fromImage() 571, 573
fromkeys() 160
fromordinal() 181, 186
fromRgb() 553
fromRgba() 553
fromRgbF() 553
fromtimestamp() 181, 185
frozenset 39, 171, 172
frozenset() 171
full() 389, 390
function 39, 199
functools 144, 438

G

geometry() 403, 404, 664
GET
get() 162, 165, 289, 348, 349, 352, 353, 355, 356, 389, 391
get_all() 352
get_content_maintype() 352
get_content_subtype() 352
get_content_type() 352
get_nowait() 389
getatime() 280
getattr() 217, 229
getbuffer() 276
getCmyk() 554
getCmykF() 554
getColor() 629
getCoords() 413
getctime() 280
getcwd() 289
getDouble() 622
getExistingDirectory() 626
getFont() 630
getheader() 350
getheaders() 350
getHsv() 554
getHsvF() 555
getInt() 621
getInteger() 621
getItem() 623
getlocale() 97
getmtime() 280
getOpenFileName() 627
getOpenFileNameAndFilter() 627
getOpenFileNames() 628
getOpenFileNamesAndFilter() 628
getparam() 352
getRect() 413
getrefcount() 42
getresponse() 349
getRgb() 553

getRgba() 630
getRgbF() 553
getSaveFileName() 628
getSaveFileNameAndFilter() 628
getsize() 279
getter() 246
getText() 621
geturl() 342, 355
getvalue() 273
glob() 292
global() 211
globalPos() 456, 458, 648, 649
globals() 212
globalX() 456, 458, 648
globalY() 456, 458, 648
gmtime() 173, 174, 194
gotFocus() 450
grabKeyboard() 450, 455, 590, 601
grabMouse() 456, 457, 590, 602
grabShortcut() 452
grabWidget() 572
grabWindow() 572
graphicsEffect() 597
green() 553
greenF() 553
group() 125, 597
groupdict() 125
groupindex 124
groups() 124, 126
GroupSwitchModifier 454

H

hasAcceptableInput() 498, 499
hasattr() 217, 230
hasChildren() 533, 534
hasComplexSelection() 510
hasFocus() 449, 582, 590
hasFormat() 463
hashlib 112
hasHtml() 462
hasImage() 462
hasSelectedText() 491, 496
hasSelection() 510, 547
hasText() 462
hasTracking() 520
hasUrls() 462
hasVisitedPage() 634
HaveCustomButton1 635, 636
HaveCustomButton2 635, 637
HaveCustomButton3 635, 637
HaveFinishButtonOnEarlyPages 636
HaveHelpButton 635, 636
HaveNextButtonOnLastPage 636
HEAD 348, 349, 352
header() 543, 545
headerData() 532
height() 402, 405, 408, 412, 561, 571, 574, 579
heightForWidth() 478
Help 611, 613
help() 33, 34, 35, 37
HelpButton 635
HelpButtonOnRight 636
helpRequested() 612, 637
HelpRole 611, 614
HeuristicMaskShape 595
Hex 518
hex() 71, 239
hexdigest() 113
hiddenSectionCount() 546
hide() 386, 399, 443, 590, 609, 610, 663
hideColumn() 542, 543
hideEvent() 445
HideNameFilterDetails 625
hidePopup() 525
hideRow() 541
hideSection() 546
hideTearOffMenu() 647
HideToParent 443
HighestPriority 382
highlighted() 512, 526
HighPriority 382, 651
historyChanged() 512
historyTitle() 511
historyUrl() 511
HLine 480
home() 497, 511
Horizontal 458, 486, 519, 520, 522, 532, 545, 603, 609—611, 642, 644
horizontalHeader() 541, 545
horizontalHeaderItem() 532
horizontalScrollBar() 487
Host 353
hostname 341
hotSpot() 461
hour 183, 185, 186
hours 178
HourSection 515
hover() 652
hovered() 645, 648, 652, 653
hoverEnterEvent() 602
hoverLeaveEvent() 602
hoverMoveEvent() 602
Hsl 555

Hsv 555
hsvHue() 554
hsvHueF() 555
hsvSaturation() 554
hsvSaturationF() 555
html 348, 462
HTMLCalendar 189, 191
http.client 340, 348, 355
http.client.HTTPMessage 351
HTTPConnection 348
HTTP-заголовки 353

I

IBeamCursor 458
icon() 647, 649, 663
IconMode 539
iconSize() 642, 655
IDLE 14, 20, 23, 28, 364
IdlePriority 382
ieHTTPHeaders 354
if…else 58, 59, 61
ignore() 386, 425, 442, 448, 455, 456, 458, 464, 601, 611, 613
IgnoreAction 460
IgnoreAspectRatio 409, 571, 575
IGNORECASE 114
Ignored 477
IgnoreSubTitles 636
imageData() 462
Imp 223
Import 24, 31, 216, 218—221, 225—227
ImportError 255
in 52, 56, 76, 84, 147, 152, 162, 164, 169, 241, 289
Inactive 420
IndentationError 255
IndependentPages 634, 636, 639
index() 65, 99, 147, 153, 247, 529, 530, 532, 536
IndexError 81, 125, 137, 146, 255
indexesMoved() 540
indexFromItem() 532
indexOf() 477, 483, 485, 487
indexWidget() 536
info() 355
information() 613, 616, 664
InheritPriority 382
initializePage() 634, 636, 638
input() 24, 31—33, 35, 46, 255, 284
insert() 110, 111, 145, 222, 496, 559
insertAction() 647, 654
insertActions() 647, 654
InsertAfterCurrent 524
InsertAlphabetically 524
InsertAtBottom 524
InsertAtCurrent 524
InsertAtTop 524
InsertBeforeCurrent 524
insertBlock() 511
insertColumn() 531, 534
insertColumns() 531, 534
insertFragment() 511
insertFrame() 511
insertHtml() 500, 510
insertImage() 511
insertItem() 484, 523
insertItems() 524
insertLayout() 470
insertList() 511
insertMenu() 645, 647
insertPermanentWidget() 658, 659
insertPlainText() 500
insertRow() 474, 531, 534
insertRows() 530, 531, 534
insertSeparator() 524, 647, 654
insertSpacing() 471
insertStretch() 471
insertTab() 482
insertTable() 511
insertText() 510
insertToolBar() 641, 653
insertToolBarBreak() 642
insertWidget() 470, 471, 476, 486, 654, 658, 659
installEventFilter() 466
installSceneEventFilter() 605
InstantPopup 656
int()38, 44, 69, 70, 238
IntegrityError 337, 339
Interactive 545
InterfaceError 337
InternalError 337
InternalMove 538
intersect() 414
intersected() 414
intersection() 169, 172
intersection_update() 169
intersects() 414
IntersectsItemBoundingRect 582, 583, 585
IntersectsItemShape 581, 583, 585
interval() 440, 441
IntInput 619—621
intValue() 518, 620

intValueChanged() 620
intValueSelected() 621
Invalid 555
invalidate() 583
invalidateScene() 588
InvalidRole 611, 612, 614
invertPixels() 576
InvertRgb 576
InvertRgba 576
invisibleRootItem() 532
io 269, 273, 276
IOError 255, 259, 263, 278, 279
is 41, 56, 134, 135
isabs() 282
isAccepted() 442
isActive() 441, 562, 583
isActiveWindow() 415
isalnum() 103
isalpha() 103
isAmbiguous() 452
isatty() 284
isAutoRepeat() 454
isBackwardAvailable() 511
isCheckable() 479, 535, 650
isChecked() 479, 493—495, 650
isColumnHidden() 542, 543
isColumnSelected() 547
isCommitPage() 638
isComplete() 638
isdecimal() 102
isdigit() 102
isdir() 291
isdisjoint() 170, 172
isDockNestingEnabled() 643
isDown() 493
isEmpty() 408, 413, 505, 559, 647
isEnabled() 432, 493, 590, 598, 651, 653
isExpanded() 544
isfile() 291
isFinalPage() 638
isFinished() 382
isFlat() 479
isFloatable() 655
isFloating() 655, 657
isForwardAvailable() 511
isFullScreen() 415
isHeaderHidden() 543
isHidden() 399
isIndexHidden() 542, 544
isinstance() 44
isInteractive() 585
isItemEnabled() 485
isleap() 193
islink() 292
islower() 103
isMaximized() 415
isMinimized() 415
isModal() 417, 609
isModified() 505
isMovable() 546, 654
isNull() 379, 407, 408, 413, 506, 509, 557, 570, 574, 576
isnumeric() 102
isocalendar() 183, 188
isoformat() 182, 184, 188
isolation_level 328
isoweekday() 182, 188
ISOWeekNumbers 517
isQBitmap() 571
isReadOnly() 496, 503
isRedoAvailable() 497, 506
isRowHidden() 540, 542, 543
isRowSelected() 547
isRunning() 382, 386
isSectionHidden() 546
isSelected() 547, 590
isSeparator() 649
isShaded() 662
isSingleShot() 441
isSizeGripEnabled() 609
isspace() 103
issubset() 170, 172
issuperset() 170, 172
isSystemTrayAvailable() 663
isTabEnabled() 483
isTearOffEnabled() 647
isTearOffMenuVisible() 647
istitle() 103
isTristate() 495, 535
isUndoAvailable() 497, 505
isUndoRedoEnabled() 501, 506
isupper() 103
isValid() 408, 413, 529, 552
isValidColor() 552
isVisible() 399, 590, 651, 653, 664
italic() 560
item() 532
itemAt() 581, 586
itemChange() 605, 606
itemChanged() 533
ItemCursorChange 606
ItemCursorHasChanged 606
itemData 524
ItemDoesntPropagateOpacityToChildren 589

ItemEnabledChange 605
ItemEnabledHasChanged 605
ItemFlagsChange 606
ItemFlagsHaveChanged 606
itemFromIndex() 532
ItemIgnoresParentOpacity 589
ItemIgnoresTransformations 589
ItemIsDragEnabled 529
ItemIsDropEnabled 529
ItemIsEditable 529
ItemIsEnabled 529
ItemIsFocusable 582, 589, 590, 600
ItemIsMovable 589
ItemIsPanel 589
ItemIsSelectable 529, 582, 585, 589, 590
ItemIsTristate 529
ItemIsUserCheckable 529
ItemLayer 583
ItemMatrixChange 605
ItemOpacityChange 606
ItemOpacityHasChanged 606
ItemPositionChange 605
ItemPositionHasChanged 605
items() 164, 288, 351, 581, 586
itemsBoundingRect() 579
ItemScenePositionHasChanged 606
ItemSelectedChange 606
ItemSelectedHasChanged 606
ItemSendsGeometryChanges 605, 606
ItemSendsScenePositionChanges 606
ItemStacksBehindParent 589
itemText() 485, 524
ItemToolTipChange 606
ItemToolTipHasChanged 606
ItemTransformChange 606
ItemTransformHasChanged 606
ItemVisibleChange 606
ItemVisibleHasChanged 606
ItemZValueChange 606
ItemZValueHasChanged 606
iter() 40
itertools 157

J

join() 96, 151, 283, 389
joinPreviousEditBlock() 510

K

KeepAnchor 509
KeepAspectRatio 409, 571, 575
KeepAspectRatioByExpanding 409, 571, 575
key() 452, 454
KeyboardInterrupt 66, 256
keyboardModifiers() 464
KeyError 162, 165, 171, 255, 289
KeypadModifier 454
KeyPress 443, 444, 465
keyPressEvent() 442, 454, 601
KeyRelease 443
keyReleaseEvent() 454, 601
keys() 62, 163, 164, 288, 326, 352
killTimer() 439, 440, 441

L

Lambda 205
lastBlock() 507
Lastgroup 125
Lastindex 125
Last-Modified 353
lastPos() 602, 603
Lastrowid 323
lastScenePos() 602, 603
lastScreenPos() 602, 603
LC_ALL 97
LC_COLLATE 97
LC_CTYPE 97
LC_MONETARY 97
LC_NUMERIC 97
LC_TIME 97
leapdays() 193
Leave 443
leaveEvent() 457
left() 412, 508
LeftArrow 656
LeftButton 456, 458
LeftDockWidgetArea 642, 643, 657
LeftToolBarArea 641, 654
LeftToRight 471, 540
len() 64, 83, 94, 136, 153, 163, 168, 238, 289, 477, 483, 485, 487
LifoQueue 388
Light 504, 507, 560
lighter() 553
LightGray 552
line() 593
lineCount() 507
LineUnderCursor 510
Linguist 361
LinkAction 460, 461
linkActivated() 492, 596
linkHovered() 492, 596

LinksAccessibleByKeyboard 491, 502
LinksAccessibleByMouse 491, 502
list() 38, 46, 96, 132, 134, 135, 148, 624
listdir() 290—292
ListMode 539
ListView 536
ljust() 89
load() 286, 287, 569, 570, 574
loadFromData() 570, 574
loads() 112, 288
loadUi() 373
loadUiType() 374
locale 97
LOCALE 114
localeconv() 98
LocaleHTMLCalendar 189, 191
LocaleTextCalendar 189, 190
locals() 212
localtime() 174, 515
Location 353, 354
lock() 392
log() 73
logicalIndex() 546
LogoPixmap 636
LongDayNames 517
LookIn 625
lostFocus() 450
lower() 98
LowestPriority 382
LowPriority 382, 651
lseek() 272
lstrip() 94

M

MacStyle 635, 636
magenta() 554
magentaF() 554
maketrans() 101
manhattanLength() 407
ManualWrap 503
map() 141, 142, 143
mapFrom() 457
mapFromGlobal() 457
mapFromParent() 457
mapFromScene() 586
mapTo() 457
mapToGlobal() 457
mapToParent() 457
mapToScene() 586
mask() 423, 571
MaskShape 595
match() 123, 124
MatchCaseSensitive 526
MatchContains 526
MatchEndsWith 526
matches() 454
MatchExactly 526
MatchFixedString 526
MatchRecursive 526
MatchRegExp 526
MatchStartsWith 526
MatchWildcard 526
MatchWrap 526
Math 72, 217, 218, 220
matrix() 587, 591
max() 71, 147
Maximum 477
maximumBlockCount() 507
maximumHeight() 402
maximumSize() 402
maximumWidth() 402
MaxUser 444
MAXYEAR 180, 181, 184, 186
md5() 112, 113
mdiArea() 662
MDI-приложения 640, 659
memoryview() 276
menu() 494, 651, 656
menuAction() 647
menuBar() 641, 645
MenuBarFocusReason 449
MenuButtonPopup 656
menuWidget() 641
mergeBlockCharFormat() 511
mergeBlockFormat() 511
mergeCharFormat() 511
messageChanged() 659
messageClicked() 664
MetaModifier 454
microsecond 183, 185, 186
microseconds 178, 179
MidButton 456, 458
MiddleButton 456
MiddleClick 664
milliseconds 178
mimeData() 461, 464, 465, 604
min() 72, 147
Minimum 477
MinimumExpanding 475, 477
minimumHeight() 402
minimumSectionSize() 545
minimumSize() 402
minimumSizeHint() 402

minimumWidth() 402
minute 183, 185, 186
minutes 178
MinuteSection 515
MINYEAR 180, 181, 184, 186
mirrored() 576
MiterJoin 556
mkdir() 290
mktime() 174
mode 270
model() 529, 536, 539, 541, 543
ModernStyle 635, 636
modificationChanged() 508
modifiers() 454, 456, 458, 602, 603, 605
Module 39, 219
Monday 190, 517
MonospacedFonts 527
Month 181, 186, 192
Month_abbr 194
Month_name 194
monthcalendar() 193
monthrange() 193
MonthSection 515
monthShown() 516
MouseButtonDblClick 443
MouseButtonPress 443, 444, 465, 466
MouseButtonRelease 443
mouseButtons() 464
mouseDoubleClickEvent() 455, 601
MouseFocusReason 449
mouseGrabberItem() 583, 602
MouseMove 443
mouseMoveEvent() 456, 460, 601
mousePressEvent() 455, 460, 601, 605
mouseReleaseEvent() 455, 601
move() 279, 403, 443, 468
MoveAction 460, 461
MoveAnchor 509
moveBottom() 412
moveBottomLeft() 412
moveBottomRight() 412
moveBy() 589
moveCenter() 412
moveCursor() 508, 509
moveEvent() 446
movePosition() 509
moveRight() 412
moveSection() 546
moveTo() 411
moveTop() 411
moveTopLeft() 412
moveTopRight() 412
MSecSection 515
Msg 351, 355
msleep() 384
MSWindowsFixedSizeDialogHint 400
MULTILINE 114, 116
Multiprocessing 395
MultiSelection 537

N

Name 270, 271, 552
NameError 256
Netbeans 14
Netloc 341
next() 40, 65, 634
NextBlock 509
NextButton 635
NextCell 509
NextCharacter 509
nextId() 634, 638
NextRow 509
NextWord 509
No 610, 613
NoArrow 656
NoBackButtonOnLastPage 636
NoBackButtonOnStartPage 636
NoBrush 421, 557, 563
NoButton 456, 610, 613
NoButtons 512, 620, 629, 631
NoCancelButton 635, 636
NoDefaultButton 636
NoDockWidgetArea 642
NoDockWidgetFeatures 657
NoDrag 585
NoDragDrop 538
NoEcho 495, 619
NoEditTriggers 537
NoFocus 450
NoFrame 480
NoHorizontalHeader 517
NoIcon 613, 664
NoIndex 579
NoInsert 524
NoItemFlags 529
NoModifier 454
NoMove 508
None 38, 56
NoneType 38
NonModal 417, 609
NonRecursive 392
NonScalableFonts 527
NoPen 542, 555, 563

Normal 420, 495, 504, 507, 560, 619
normalized() 413
NormalPriority 382, 651
normpath() 283
NoRole 611, 612, 614
North 482, 643, 661
NoSection 515
NoSelection 517, 537
Not 57
Notepad++ 14, 24
NoTextInteraction 491, 502
NoTicks 521
NoToAll 610, 613
NoToolBarArea 641
NotSupportedError 338
NoUpdate 548
NoVerticalHeader 517
now() 185
NoWrap 503

O

O_APPEND 271
O_BINARY 271
O_CREAT 271
O_RDONLY 271
O_RDWR 271
O_TEXT 271
O_TRUNC 271
O_WRONLY 271
Object 234
oct() 71, 239
offset() 595, 598
offsetChanged() 599
OffsetFromUTC 515
Ok 610, 613
oldPos() 446
oldSize() 446
oldState() 445
opacity() 589, 600
opacityChanged() 600
opacityMask() 600
opacityMaskChanged() 600
open() 254, 259, 261—263, 265, 271, 272, 288, 609, 610, 613, 650
OpenHandCursor 458
OperationalError 327, 337
or 58
ord() 99
orientation() 458, 603
os 262, 271, 278—281, 289
os.path 260, 262, 279, 281, 282, 291
OSError 271
OtherFocusReason 449
Outline 518
overflow() 518
OverflowError 174
overline() 561
overrideCursor() 459
overwriteMode() 503

P

p1() 558
p2() 558
page() 634
pageAdded() 637
pageIds() 634
pageRemoved() 637
Paint 443
paint() 588
paintEvent() 447, 551, 562
palette() 420, 421
Panel 480
Params 341
parent() 529, 533, 534
parentItem() 591
parentWidget() 399
PARSE_COLNAMES 334
PARSE_DECLTYPES 334
parse_qs() 343, 344
parse_qsl() 344
ParseResult 340, 341
partial() 438
PartiallyChecked 494, 528, 535
partition() 96
Pass 197, 230
Password 342, 495, 496, 619
PasswordEchoOnEdit 495, 619
paste() 497, 501
Path 341
PEP-8 14
permutations() 155
Phonon 377
Pi 72, 217, 220
Pickle 112, 286—288, 293
pixel() 575
pixelSize() 560
pixmap() 396, 461, 577, 595
Plain 480
PlainText 490, 615, 635
PlusMinus 512
point() 559
PointingHandCursor 458

pointSize() 560
pointSizeF() 560
Polish 443
PolishRequest 443
polygon() 594
pop() 111, 146, 165, 171, 289
popitem() 165, 289
popup() 400, 648
PopupFocusReason 449
popupMode() 656
Port 341
pos() 125, 404, 446, 455, 457—459, 464, 589, 602—604, 648
posF() 456
position() 509
PositionAtBottom 538
PositionAtCenter 538
PositionAtTop 538
positionInBlock() 509
possibleActions() 464, 604
POST 348, 349, 353, 355, 356
postEvent() 466
pow() 71, 73
Pragma 353
prcal() 193
Preferred 477
prepareGeometryChange() 589
PrepareProtocol 333
prepend() 559
pressed() 493, 538
PreviousBlock 508
PreviousCell 509
PreviousCharacter 508
PreviousRow 509
PreviousWord 508
print() 28—31, 239, 284
print_() 502, 506
print_exception() 249
print_tb() 249
priority() 382, 651
PriorityQueue 389
prmonth() 190, 193
processEvents() 380, 381, 632
product() 155
ProgrammingError 338
property() 245
ProportionalFonts 527
proposedAction() 464, 604
pryear() 191
put() 389
put_nowait() 389
PyDev 14, 365
Pydoc 33
PyQt_PyObject 434
PYQT_VERSION_STR 362
PyQt4 377
pyqtSignal() 435
pyqtSlot() 429, 437
PyScripter 14, 24
Python Shell 23
python.exe 17, 18
PYTHONPATH 222
pythonw.exe 17
PythonWin 14
pyuic4.bat 375

Q

QAbstractButton 492, 493, 494
QAbstractGraphicsShapeItem 593
QAbstractItemView 529, 536—540, 543, 545, 547
QAbstractListModel 530
QAbstractProxyModel 549
QAbstractScrollArea 487
QAbstractSlider 520—522
QAbstractSpinBox 512—514
QAction 453, 644, 645, 647—649, 651—657
QActionGroup 651—653
QApp 365, 367
QApplication 365, 379, 419, 420, 424, 450, 455, 459, 460, 465, 560
QAxContainer 377
QBitmap 557, 569, 572, 573
QBoxLayout 471
QBrush 421, 556, 557
QButtonGroup 494
QByteArray 378, 419, 569
QCalendarWidget 516, 517
QChar 378
QCheckBox 494
QClipboard 465
QCloseEvent 425, 448
QColor 396, 420, 504, 551, 555, 556, 574, 575
QColorDialog 629, 630
QComboBox 523—527, 529, 536, 620
QCompleter 495, 525
QConicalGradient 557
QContextMenuEvent 648
QCoreApplication 380, 466
QCursor 458, 459
QDate 379, 514—517
QDateEdit 514, 515
QDateTime 379, 514, 515

QDateTimeEdit 514, 515
QDesktopWidget 406
QDial 521
QDialog 368, 370, 398, 418, 419, 607—609
QDialogButtonBox 610—612
QDockWidget 644, 656, 657
QDoubleSpinBox 512, 513
QDoubleValidator 499
QDrag 460, 461, 462
QDragEnterEvent 463, 464
QDragLeaveEvent 463, 464
QDragMoveEvent 463, 464
QDropEvent 464
QErrorMessage 631
QEvent 442, 457, 464, 466, 602—604
QFileDialog 607, 623—626
QFocusEvent 450, 601
QFont 504, 506, 560
QFontComboBox 526, 527
QFontDatabase 561
QFontDialog 630, 631
QFontMetrics 561
QFontMetricsF 561
QFormLayout 473—475
QFrame 368, 398, 479, 480
QGraphicsBlurEffect 599
QGraphicsColorizeEffect 599, 600
QGraphicsDropShadowEffect 598, 599
QGraphicsEffect 597, 598
QGraphicsEllipseItem 578, 580, 592, 594
QGraphicsItem 578, 582, 588—592, 595, 597, 601, 603, 605, 606
QGraphicsItemGroup 580, 597
QGraphicsLineItem 579, 592
QGraphicsOpacityEffect 600
QGraphicsPathItem 580, 592
QGraphicsPixmapItem 580, 592, 594, 595
QGraphicsPolygonItem 580, 592, 593, 594
QGraphicsProxyWidget 580
QGraphicsRectItem 578, 580, 592, 593
QGraphicsScene 578—584, 588, 589, 597
QGraphicsSceneDragDropEvent 604
QGraphicsSceneEvent 602, 603, 604
QGraphicsSceneHoverEvent 602, 603
QGraphicsSceneMouseEvent 601, 602
QGraphicsSceneWheelEvent 602, 603
QGraphicsSimpleTextItem 580, 592, 595
QGraphicsSvgItem 592
QGraphicsTextItem 580, 592, 595, 596
QGraphicsView 578, 584, 585—587
QGraphicsWidget 581
QGridLayout 472, 473
QGroupBox 478, 479, 494
QHBoxLayout 469, 470
QHeaderView 541, 543, 545, 546
QHideEvent 445
QIcon 419, 576
QImage 551, 562, 567, 569, 573, 574
QImageReader 419, 569
QImageWriter 569
QInputDialog 607, 618—621
QIntValidator 499
QItemSelection 548
QItemSelectionModel 547, 548
QKeyEvent 444, 454, 465, 601
QKeySequence 452, 454, 650
QLabel 366, 423, 452, 474, 489, 492, 651
QLCDNumber 517, 518
QLine 557
QLinearGradient 557
QLineEdit 495, 497, 619
QLineF 557
QListView 529, 539, 540, 620
QListWidget 536
QMainWindow 368, 398, 640, 642, 645, 653, 658, 659
QMatrix 568, 573, 587, 591
QMdiArea 640, 659—662
QMdiSubWindow 640, 659, 660, 662, 663
QMenu 644—648, 651, 656, 662, 663
QMenuBar 641, 644, 645
QMessageBox 607, 612—616
QMimeData 461, 462—464, 604
QMimeSource 464
QModelIndex 528—530, 532, 533, 536, 538—540, 542—544, 547, 548
QMouseEvent 444, 455, 456, 465, 466
QMoveEvent 446
QMutex 392
QMutexLocker 392, 394
QObject 367, 369, 381, 398, 426, 430, 431, 433, 439, 465, 489
QPaintDevice 367, 398, 551, 562, 570, 572, 573, 576
QPainter 551, 562, 563, 565—567, 569, 570, 572, 573
QPainterPath 588
QPaintEvent 447
QPalette 420, 421
QPen 555, 556
QPersistentModelIndex 529
QPicture 551, 562, 568
QPixmap 395, 396, 422, 423, 459, 551, 557, 562, 566, 567, 569, 570, 573, 576

QPlainTextEdit 500
QPoint 406, 407, 410
QPointF 407
QPolygon 558, 559
QPolygonF 559
QPrintDialog 607
QPrinter 502, 506, 551, 562
QPrintPreviewDialog 607
QProgressBar 518, 519
QProgressDialog 632, 633
QPushButton 366, 391, 492, 493
QPyNullVariant 379
QRadialGradient 557
QRadioButton 494
QRect 406, 407, 410, 414
QRectF 407
QRegExp 550
QRegExpValidator 499
QRegion 447
QResizeEvent 446
QScrollArea 487, 522
QScrollBar 487, 488, 522
QSemaphore 395
QShortcut 453
QShortcutEvent 452
QShowEvent 445
QSize 390, 406—408, 410
QSizeF 407
QSizePolicy 477
QSlider 519, 520, 521
QSortFilterProxyModel 549
QSpinBox 512, 513
QSplashScreen 395, 396, 400
QSplitter 485, 486
QStackedLayout 475, 476
QStackedWidget 476
QStandardItem 530—535
QStandardItemModel 530, 533
QStatusBar 641, 658, 659
QStatusTipEvent 651
QString 377
QStringList 378
QStringListModel 529, 530
QStyle 419, 577
QSystemTrayIcon 663, 664
Qt 377
QT_VERSION_STR 362
QTableView 530, 536, 540, 545
QTableWidget 536
QTabWidget 480—483, 643, 661
QtCore 365, 368, 377, 392
QtCore4.dll 361
QtDeclarative 377
QtDesigner 377
QTextBlock 507
QTextBrowser 511, 512
QTextCharFormat 505, 517
QTextCursor 506, 508—510
QTextDocument 501, 505—507
QTextDocumentFragment 510
QTextEdit 500, 502—505, 508, 509, 511
QTextOption 503, 565
QTextStream 379
QtGui 365, 366, 368, 377
QtGui4.dll 361
QtHelp 377
QThread 381—384, 395
QTime 379, 514, 515
QTimeEdit 514, 515
QTimer 440—442
QTimerEvent 439
QtMultimedia 377
QtNetwork 377
QToolBar 641, 653—655
QToolBox 484
QToolButton 655, 656
QtOpenGL 377
QTransform 591
QTreeView 530, 536, 543—545
QTreeWidget 536
QtScript 377
QtScriptTools 377
QtSql 377
QtSvg 377
QtTest 377
QtWebKit 377, 512
QtXml 377
QtXmlPatterns 377
Query 341
Question 613, 616
Queue 388, 391
QueuedConnection 384, 429, 430
quit() 367, 379, 386, 424, 428
quote() 345, 346
quote_from_bytes() 346
quote_plus() 344, 345
quoteattr() 348
QUrl 379
QValidator 499, 525
QVariant 378, 379
QVBoxLayout 469, 470
QWaitCondition 395
QWheelEvent 457, 458

QWidget 366—369, 391, 398, 399, 415—417, 419, 420, 424, 429, 432, 447, 449, 450, 452, 457, 458, 463, 469, 471, 477, 479, 489, 493, 539, 551, 562, 572, 609, 646, 654
QWindowStateChangeEvent 445
QWizard 633, 635, 636, 637
QWizardPage 633—635, 637, 639

R

R_OK 278
radians() 73
Raise 256, 257, 258
raise_() 609
Raised 480
randint() 74
random() 73—75, 148, 150
randrange() 74
range() 63, 64, 74, 140, 150
rangeChanged() 520
raw_input() 32
Re 114, 124
read() 266, 272, 274, 349, 354
readline() 267, 274, 354
readlines() 267, 275, 354
ReadOnly 625
reason() 351, 450
rect() 402, 447, 571, 574, 593, 594
Recursive 392
red() 552, 553
redF() 553
redo() 497, 501, 505, 510
redoAvailable() 502, 507
RedoStack 506
reduce() 144
Referer 353
region() 447
register_adapter() 332
register_converter() 334
registerEventType() 442, 466
registerField() 634, 635, 637
reject() 608, 610, 625
rejected() 609, 612, 619, 624
RejectRole 611, 612, 614
released() 493
releaseKeyboard() 450, 455
releaseMouse() 456
releaseShortcut() 452
reload() 223, 224, 511
remove() 111, 146, 171, 279, 559
removeAction() 647, 652, 654
removeButton() 612, 615
removeColumn() 534
removeColumns() 531, 534
removeDockWidget() 642
removeEventFilter() 466
removeFormat() 463
removeFromGroup() 597
removeItem() 484, 524, 580
removePage() 634
removeRow() 534
removeRows() 530, 531, 534
removeSceneEventFilter() 605
removeSelectedText() 510
removeSubWindow() 660
removeTab() 482
removeToolBar() 641
removeToolBarBreak() 642
removeWidget() 470, 476, 659
rename() 279
render() 583, 588
repaint() 447, 448, 551
repeat() 154, 196
replace() 101, 182, 184, 187
repr() 91, 94, 239
Request 349, 354, 356
reset() 358, 519, 611, 613, 633
resetCachedContent() 585
resetMatrix() 587, 591
ResetRole 611, 614
resetTransform() 591
resize() 366, 401, 443, 468
resizeColumnsToContents() 541
resizeColumnToContents() 541, 543
resizeEvent() 446
resizeRowsToContents() 541
resizeRowToContents() 541
resizeSection() 545
ResizeToContents 545
restart() 634
restore() 568
RestoreDefaults 611, 613
restoreDockWidget() 644
restoreGeometry() 644
restoreOverrideCursor() 459
restoreState() 486, 546, 625, 644
Result 358, 608
retranslateUi() 375
retrieveData() 463
Retry 611, 613
Return 197, 198
returnPressed() 497
reverse() 111, 148
reversed() 148

RFC 2616 353
rfind() 99
Rgb 555
rgb() 553, 574, 575
rgba() 553, 574, 575
RichText 490, 615, 635
Right 509
RightArrow 656
RightButton 456, 458
RightDockWidgetArea 642, 643, 657
RightToLeft 471
RightToolBarArea 641, 654
rindex() 100
rjust() 89
rmdir() 290, 291
rmtree() 291
rollback() 327, 339
rootIndex() 536
rotate() 568, 587, 590
rotation() 591
round() 71, 239
RoundCap 556
Rounded 483, 643, 661
RoundJoin 556
row() 529, 533
row_factory 325, 326
rowcount 323
rowCount() 473, 530, 533
rowHeight() 541, 543
rowIntersectsSelection() 547
Rows 548
rowSpan() 541
rpartition() 96
rsplit() 95
rstrip() 94
RubberBandDrag 585
RubberBandMove 663
RubberBandResize 663
run() 381, 383, 384, 386, 388, 391

S

sample() 75, 149, 150
Saturday 517
SATURDAY 190
save() 568—570, 574, 610, 613
SaveAll 610, 613
saveGeometry() 644
saveState() 486, 546, 625, 644
ScalableFonts 527
scale() 408, 568, 587, 591
scaled() 571, 575
scaledToHeight() 572, 576
scaledToWidth() 572, 575
scene() 584, 589
sceneBoundingRect() 589
sceneEvent() 605
sceneEventFilter() 605
sceneMatrix() 591
scenePos() 589, 602—604
sceneRect() 579, 584
sceneRectChanged() 584
sceneTransform() 591
scheme 341
ScientificNotation 499
screenGeometry() 405
screenPos() 602—604
ScrollBarAlwaysOff 488
ScrollBarAlwaysOn 488
ScrollBarAsNeeded 488
ScrollHandDrag 585
scrollTo() 538
scrollToBottom() 538
scrollToTop() 538
SDI-приложения 640
search() 123, 124, 126
Second 183, 185, 186
Seconds 178, 179
SecondSection 515
sectionAt() 515
sectionClicked() 546
sectionCount() 515
sectionDoubleClicked() 547
sectionMoved() 547
sectionPressed() 546
sectionResized() 547
sectionsHidden() 546
sectionSize() 545
sectionText() 515
seed() 74
seek() 269, 273
SEEK_CUR 269, 272
SEEK_END 269, 272
SEEK_SET 269, 272
select() 510, 548
selectAll() 496, 501, 513, 537
selectColumn() 542
SelectColumns 537
SelectCurrent 548
selected() 653
SelectedClicked 537
selectedColumns() 548
selectedDate() 516
selectedFiles() 625

selectedIndexes() 540, 542, 544, 547
selectedItems() 583
selectedRows() 548
selectedText() 491, 496, 510
selectFile() 625
Selection 465
selection() 510, 548
selectionArea() 583
selectionChanged() 497, 502, 517, 549, 584
selectionEnd() 510
selectionModel() 537, 547
selectionStart() 491, 496, 510
SelectItems 537
selectRow() 542
SelectRows 537
Self 229
sendEvent() 466
Sep 260, 282
Server 353
set() 39, 167, 171
setAcceptDrops() 463, 603
setAccepted() 442
setAcceptedMouseButtons() 601
setAcceptHoverEvents() 603
setAcceptMode() 624
setAcceptRichText() 501
setActionGroup() 651
setActivationOrder() 661
setActiveAction() 645, 647
setActiveSubWindow() 660
setActiveWindow() 581
setAlignment() 369, 479, 487, 490, 496, 505, 513, 585
setAllowedAreas() 653, 654, 657
setAlpha() 553
setAlphaF() 553
setAlternatingRowColors() 536
setAnimated() 544, 642
setapi() 377
setArrowType() 656
setattr() 229
setAttribute() 418, 425, 456
setAutoClose() 633
setAutoCompletion() 525
setAutoCompletionCaseSensitivity() 525
setAutoDefault() 493, 610
setAutoExclusive() 493
setAutoFillBackground() 420, 422
setAutoFormatting() 503
setAutoRaise() 656
setAutoRepeat() 492, 651
setAutoReset() 633
setAutoScroll() 538
setAutoScrollMargin() 538
setBackground() 535, 661
setBackgroundBrush() 579, 585
setBar() 632
setBaseSize() 402
setBatchSize() 540
setBinMode() 518
setBlue() 553
setBlueF() 553
setBlurRadius() 598, 599
setBold() 560
setBottom() 411
setBottomLeft() 411
setBottomRight() 411
setBrush() 421, 556, 562, 593
setBspTreeDepth() 579
setBuddy() 452, 474, 490
setButton() 635
setButtonLayout() 636
setButtonSymbols() 512
setButtonText() 636, 637
setCacheMode 585
setCalendarPopup() 515
setCancelButton() 632
setCancelButtonText() 619, 633
setCapStyle() 556
setCascadingSectionResizes() 545
setCaseSensitivity() 506
setCenterButtons() 612
setCentralWidget() 640, 659
setCheckable() 479, 493, 535, 650
setChecked() 479, 493—495, 650
setCheckState() 494, 535
setChild() 533
setChildrenCollapsible() 486
setClickable() 546
setCmyk() 553
setCmykF() 554
setCollapsible() 486
setColor() 420, 421, 556, 557, 598, 599
setColumnCount() 530, 533
setColumnHidden() 541, 543
setColumnMinimumWidth() 473
setColumnStretch() 473
setColumnWidth() 541, 543
setComboBoxEditable() 620
setComboBoxItems() 620
setCommitPage() 638
setCompleter() 495, 525
setContextMenu() 663
setCoords() 411

setCorner() 643
setCornerButtonEnabled() 542
setCurrentCharFormat() 505
setCurrentFont() 504, 527
setCurrentIndex() 476, 483, 485, 524, 536, 548
setCurrentPage() 516
setCurrentSection() 515
setCurrentSectionIndex() 515
setCurrentWidget() 476, 483, 485
setCursor() 458, 590
setCursorPosition() 496
setCursorWidth() 503
setData() 462, 530, 532, 535, 651
setDate() 514
setDateRange() 515, 516
setDateTextFormat() 517
setDateTime() 514
setDateTimeRange() 515
setDecimals() 513
setDecMode() 518
setdefault() 162, 165, 289
setDefault() 493, 610
setDefaultAction() 647, 656
setDefaultAlignment() 546
setDefaultButton() 615
setDefaultFont() 506
setDefaultProperty() 638
setDefaultSectionSize() 545
setDefaultStyleSheet() 507
setDefaultSuffix() 625
setDefaultTextColor() 596
setDefaultUp() 645
setDetailedText() 614
setDigitCount() 518
setDirection() 471
setDirectory() 625
setDisabled() 371, 384, 432, 651, 653
setDisplayFormat() 515
setDockNestingEnabled() 643
setDockOptions() 642
setDocument() 505, 596
setDocumentMargin() 507
setDocumentMode() 483
setDocumentTitle() 500
setDoubleClickInterval() 455
setDoubleDecimals() 620
setDoubleMaximum() 620
setDoubleMinimum() 620
setDoubleRange() 620
setDoubleValue() 620
setDown() 493
setDragCursor() 461
setDragDropMode() 538
setDragEnabled() 496, 535, 538
setDragMode() 585
setDropAction() 463, 464, 604
setDropEnabled() 535
setDropIndicatorShown() 538
setDuplicatesEnabled() 525
setDynamicSortFilter() 550
setEchoMode() 495, 496
setEditable() 524, 535
setEditText() 524
setEditTriggers() 537
setElideMode() 482
setEnabled() 432, 493, 535, 590, 598, 645, 647, 651, 653
setEscapeButton() 615
setExclusive() 652
setExpanded() 544
setExtension() 608
setFamily() 560
setFeatures() 657
setField() 634, 638
setFieldGrowthPolicy() 475
setFileMode() 624
setFilterCaseSensitivity() 550
setFilterFixedString() 549
setFilterKeyColumn() 550
setFilterRegExp() 550
setFilterRole() 550
setFiltersChildEvents() 605
setFilterWildcard() 550
setFinalPage() 638
setFirstDayOfWeek() 516
setfirstweekday() 190, 192
setFixedHeight() 401
setFixedSize() 401
setFixedWidth() 401
setFlag() 582, 589
setFlags() 535, 589
setFlat() 479, 493
setFloatable() 653, 655
setFloating() 657
setFlow() 540
setFocus() 449, 450, 582, 590
setFocusItem() 582
setFocusPolicy() 450, 455
setFocusProxy() 449, 601
setFont() 535, 560, 579, 595, 596, 651
setFontFamily() 504
setFontFilters() 527
setFontItalic() 504
setFontPointSize() 504

setFontUnderline() 504
setFontWeight() 504
setForeground() 535
setForegroundBrush() 579, 585
setFormAlignment() 475
setFrame() 496, 513, 525
setFrameShadow() 480
setFrameShape() 480
setFrameStyle() 480
setGeometry() 401, 404, 468
setGraphicsEffect() 597
setGreen() 553
setGridSize() 539
setGridStyle() 542
setGridVisible() 517
setGroup() 597
setHandleWidth() 486
setHeader() 545
setHeaderData() 532
setHeaderHidden() 543
setHeaderTextFormat() 517
setHeight() 408, 411
setHeightForWidth() 478
setHexMode() 518
setHighlightSections() 546
setHistory() 625
setHorizontalHeader() 545
setHorizontalHeaderFormat() 516
setHorizontalHeaderItem() 532
setHorizontalHeaderLabels() 532
setHorizontalPolicy() 477
setHorizontalScrollBarPolicy() 488
setHorizontalSpacing() 473, 475
setHorizontalStretch() 477
setHotSpot() 461
setHsv() 554
setHsvF() 554
setHtml() 462, 500, 505, 596
setIcon() 492, 535, 614, 647, 649, 663
setIconPixmap() 614
setIconSize() 492, 525, 537, 641, 655
setIconVisibleInMenu() 649
setImageData() 462
setIndent() 491
setIndentation() 544
setIndexWidget() 536
setInformativeText() 614
setInputMask() 498
setInputMode() 619
setInsertPolicy() 524
setInteractive() 585
setInterval() 440
setIntMaximum() 620
setIntMinimum() 620
setIntRange() 620
setIntStep() 620
setIntValue() 620
setInvertedAppearance() 519, 520
setInvertedControls() 520
setItalic() 560
setItem() 531
setItemData() 524
setItemEnabled() 485
setItemIcon() 485, 524
setItemIndexMethod() 579
setItemsExpandable() 544
setItemText() 485, 524
setItemToolTip() 485
setJoinStyle() 556
setKeepPositionOnInsert() 510
setKeyboardPageStep() 662
setKeyboardSingleStep() 662
setLabel() 632
setLabelAlignment() 475
setLabelText() 619, 625, 633
setLayout() 367, 429, 470, 472, 473, 476, 478
setLayoutDirection() 471
setLayoutMode() 540
setLeft() 411
setLine() 558, 593
setLineWidth() 480
setLineWrapColumnOrWidth() 503
setLineWrapMode() 503
setlocale() 97
setMargin() 472, 473, 475, 491
setMask() 423, 571
setMatrix() 568, 587, 591
setMaxCount() 525
setMaximum() 513, 519, 520, 633
setMaximumBlockCount() 507
setMaximumDate() 515, 516
setMaximumDateTime() 515
setMaximumHeight() 402
setMaximumSize() 402
setMaximumTime() 515
setMaximumWidth() 402
setMaxLength() 496
setMaxVisibleItems() 525
setMenu() 494, 651, 656
setMenuBar() 641, 645
setMenuWidget() 641
setMidLineWidth() 480
setMimeData() 461, 462, 465
setMinimum() 513, 519, 520, 633

setMinimumContentsLength() 525
setMinimumDate() 515, 516
setMinimumDateTime() 515
setMinimumDuration() 633
setMinimumHeight() 402
setMinimumSectionSize() 545
setMinimumSize() 401
setMinimumTime() 515
setMinimumWidth() 402
setModal() 609
setMode() 518
setModel() 529, 530, 539, 541, 543
setModelColumn() 539
setMouseTracking() 457, 539
setMovable() 483, 546, 654
setMovement() 539
setMovie() 491
setNamedColor() 552
setNameFilter() 625
setNameFilterDetailsVisible() 625
setNameFilters() 625
setNavigationBarVisible() 516
setNotation() 499
setNotchesVisible() 521
setNotchTarget() 521
setNum() 490
setObjectName() 644
setOctMode() 518
setOffset() 595, 598
setOkButtonText() 619
setOpacity() 589, 600
setOpacityMask() 600
setOpaqueResize() 486
setOpenExternalLinks() 490, 596
setOpenLinks() 511
setOption() 620, 624, 636, 661, 662
setOptions() 620, 625, 637
setOrientation() 486, 519, 520, 609, 611
setOverline() 561
setOverrideCursor() 459
setOverwriteMode() 503
setP1() 558
setP2() 558
setPage() 634
setPageStep() 520
setPalette() 420, 422
setParent() 399
setParentItem() 591
setPen() 562, 593
setPicture() 491
setPixel() 575
setPixelSize() 560
setPixmap() 396, 423, 461, 490, 595, 636, 637
setPlaceholderText() 496
setPlainText() 500, 505, 596
setPoint() 559
setPoints() 557, 559
setPointSize() 560
setPointSizeF() 560
setPolygon() 594
setPopupMode() 656
setPos() 459, 588
setPosition() 509
setPrefix() 513
setPriority() 382, 651
setQuitOnLastWindowClosed() 379
setRange() 513, 519, 520, 633
setReadOnly() 496, 503, 513, 625
setRect() 411, 593, 594
setRed() 553
setRedF() 553
setRenderHint() 565, 566
setResizeMode() 539, 545
setResult() 608
setRgb() 552, 553
setRgba() 553
setRgbF() 553
setRight() 411
setRootIndex() 536
setRootIsDecorated() 544
setRotation() 591
setRowCount() 530, 533
setRowHeight() 541
setRowHidden() 540, 541, 543
setRowMinimumHeight() 473
setRowStretch() 473
setRowWrapPolicy() 475
setRubberBandSelectionMode() 585
setScale() 591
setScaledContents() 491
setScene() 584
setSceneRect() 579, 584
setSectionHidden() 546
setSegmentStyle() 518
setSelectable() 535
setSelected() 582, 590
setSelectedDate() 516
setSelectedSection() 515
setSelection() 491, 496
setSelectionArea() 582
setSelectionBehavior() 537
setSelectionMode() 517, 537
setSelectionModel() 537, 547
setSelectionRectVisible() 540

setSeparator() 649
setSeparatorsCollapsible() 648
setShapeMode() 595
setShortcut() 492, 650
setShortcutContext() 650
setShortcutEnabled() 452
setShortcuts() 650
setShowGrid() 542
setSidebarUrls() 625
setSingleShot() 441
setSingleStep() 513, 520
setSize() 411
setSizeAdjustPolicy() 525
setSizeGripEnabled() 609, 659
setSizePolicy() 477
setSizes() 486
setSliderPosition() 520
setSmallDecimalPoint() 518
setSortCaseSensitivity() 549
setSortingEnabled() 542, 544, 549
setSortLocaleAware() 549
setSortRole() 533, 549
setSource() 511
setSourceModel() 549
setSpacing() 472, 473, 475, 540
setSpan() 541
setSpanAngle() 594
setSpecialValueText() 513
setStackingMode() 476
setStandardButtons() 611, 615
setStartAngle() 594
setStartDragDistance() 460
setStartDragTime() 460
setStartId() 634
setStatusBar() 641, 658
setStatusTip() 650, 658
setStickyFocus() 582
setStrength() 599
setStretchFactor() 486
setStretchLastSection() 545
setStrikeOut() 561
setStringList() 530
setStyle() 556, 557
setStyleSheet() 420
setSubTitle() 637
setSubTitleFormat() 635
setSuffix() 513
setSystemMenu() 662
setTabChangesFocus() 503, 596
setTabEnabled() 483
setTabIcon() 482
setTabKeyNavigation() 538
setTabOrder() 450
setTabPosition() 482, 643, 661
setTabsClosable() 483
setTabShape() 482, 643, 661
setTabStopWidth() 503
setTabText() 482
setTabToolTip() 483
setTabWhatsThis() 483
setTearOffEnabled() 647
setter() 246
setText() 371, 384, 388, 391, 462, 465, 489, 492, 496, 498, 500, 535, 595, 614, 649
setTextAlignment() 535
setTextBackgroundColor() 504
setTextColor() 504
setTextCursor() 508, 596
setTextDirection() 519
setTextEchoMode() 619, 621
setTextElideMode() 537
setTextFormat() 490, 614
setTextInteractionFlags() 491, 502, 596
setTexture() 557
setTextureImage() 557
setTextValue() 619
setTextVisible() 519
setTextWidth() 596
setTickInterval() 521
setTickPosition() 521
setTime() 514
setTimeRange() 515
setTimeSpec() 515
setTitle() 479, 637, 647
setTitleBarWidget() 657
setTitleFormat() 635
setToolButtonStyle() 641, 655, 656
setToolTip() 424, 535, 590, 650, 663
setTop() 411
setTopLeft() 411
setTopRight() 411
setTracking() 520
setTransform() 591
setTransformationMode() 595
setTransformOriginPoint() 591
setTristate() 494, 535
setUnderline() 561
setUndoRedoEnabled() 501, 506
setUniformItemSizes() 540
setUniformRowHeights() 543
setUpdatesEnabled() 447
setupUi() 374, 375, 376
setUrls() 462
setUsesScrollButtons() 483

setValidator() 499, 525
setValue() 513, 519, 520, 632
setVerticalHeader() 545
setVerticalHeaderFormat() 517
setVerticalHeaderItem() 532
setVerticalHeaderLabels() 532
setVerticalPolicy() 477
setVerticalScrollBarPolicy() 488
setVerticalSpacing() 473, 475
setVerticalStretch() 477
setViewMode() 539, 624, 661
setViewport() 567
setViewportMargins() 488
setVisible() 399, 590, 609, 610, 645, 647, 651, 653, 663
setWeekdayTextFormat() 517
setWeight() 560
setWhatsThis() 424, 535, 650
setWidget() 487, 657, 662
setWidgetResizable() 487
setWidth() 408, 411, 556
setWidthF() 556
setWindow() 567
setWindowFlags() 399
setWindowIcon() 419
setWindowModality() 417, 609
setWindowOpacity() 416
setWindowState() 415
setWindowTitle() 366, 399, 581, 614
setWizardStyle() 635
setWordWrap() 490, 540, 542, 544
setWordWrapMode() 503
setWrapping() 513, 521, 540
setX() 407, 411, 589
setXOffset() 598
setY() 407, 411, 589
setYOffset() 599
setZValue() 589
sha1() 112
sha224() 112
sha256() 112
sha384() 112
sha512() 112
shape() 588
shear() 568, 587, 591
Sheet 400
Shelve 286, 288, 293
ShiftModifier 454
Shortcut 452
ShortcutFocusReason 449
shortcutId() 452
ShortDayNames 517
Show 443
show() 396, 399, 405, 590, 609, 663
ShowAlphaChannel 629
showColumn() 542, 543
ShowDirsOnly 624
showEvent() 445
showExtension() 608
showFullScreen() 415
showMaximized() 415
showMenu() 494, 656
showMessage() 396, 632, 658, 664
showMinimized() 415
showNextMonth() 516
showNextYear() 516
showNormal() 415
showPopup() 525
showPreviousMonth() 516
showPreviousYear() 516
showRow() 541
showSection() 546
showSelectedDate() 516
showShaded() 662
showStatusText() 651
showSystemMenu() 662
showToday() 516
ShowToParent 443
shuffle() 75, 148
Shutil 278, 291
sibling() 529
SIGNAL() 426, 433
signalsBlocked() 430
sin() 72
SingleLetterDayNames 517
SinglePass 540
SingleSelection 517, 537
singleShot() 442
Sip 377
size() 402, 412, 446, 559, 571, 574
SizeAllCursor 458
SizeBDiagCursor 458
SizeFDiagCursor 458
sizeHint() 402, 475, 477
SizeHintRole 528
SizeHorCursor 458
sizePolicy() 477
sizes() 486
SizeVerCursor 458
sleep() 177, 379, 383, 384
SliderMove 520
sliderMoved() 521
SliderNoAction 520
SliderPageStepAdd 520

SliderPageStepSub 520
sliderPosition() 520
sliderPressed() 521
sliderReleased() 521
SliderSingleStepAdd 520
SliderSingleStepSub 520
SliderToMaximum 520
SliderToMinimum 520
SLOT() 428
smoothSizes() 561
SmoothTransformation 572, 575, 595
Snap 539
SolidLine 542, 555
SolidPattern 421, 557
sort() 149, 150, 163, 206, 530, 533, 549
sortByColumn() 542, 544
sortChildren() 536
sorted() 63, 150, 163
source() 461, 464, 511, 605
sourceChanged() 512
sourceModel() 549
South 482, 643, 661
span() 126
spanAngle() 594
spec() 555
SplashScreen 400
split() 95, 130, 131, 283
splitDockWidget() 644
splitdrive() 283
splitext() 283
SplitHCursor 458
splitlines() 95
SplitResult 342
splitterMoved() 487
SplitVCursor 458
spontaneous() 442
SQL 293
SQLite 293
Sqlite_version 318
Sqlite_version_info 318
Sqlite3 293, 318
sqrt() 73
SquareCap 556
StackAll 476
stackingMode() 476
StackingOrder 660
StackOne 476
StandardNotation 499
starmap() 157
start() 126, 381, 384, 386, 391, 440, 461
startAngle() 594
startDragDistance() 460
startDragTime() 460
started() 382, 384
startId() 634
StartOfBlock 508
StartOfLine 508
StartOfWord 508
startswith() 100
startTimer() 439, 440, 441
Stat 278
stat() 280
Stat_result 280
stateChanged() 495
Static 539
Status 351
statusBar() 641, 658
StatusTip 651
statusTip() 650
StatusTipRole 527
Stderr 268, 283
Stdin 31, 268, 283, 284
Stdout 29, 268, 270, 283, 285
step() 331
stepBy() 513
stepUp() 513
stickyFocus() 582
stop() 440
StopIteration 65, 207, 238, 256, 268, 275, 324, 355
str() 38, 45, 77, 84, 91, 94, 151, 239, 263, 327
strength() 599
strengthChanged() 600
Stretch 545, 636
strftime() 175—177, 182, 184, 188, 216
strikeOut() 561
String 125
StringIO 273, 276
stringList() 530
strip() 94
StrongFocus 450
strptime() 175, 176, 186
struct_time 173—176, 182, 187, 194
StyledPanel 480
styles() 561
sub() 128, 129
subn() 130
subTitle() 637
SubWindow 400
subWindowActivated() 662
subWindowList() 660, 661
SubWindowView 661
sum() 72, 196
Sunday 190, 517

Sunken 480
super() 233, 234
supportedImageFormats() 419, 569
supportsMessages() 664
SvgMiterJoin 556
swapcase() 98
swapSections() 546
symmetric_difference() 169, 172
symmetric_difference_update() 169
SyntaxError 256
sys 31, 42, 219, 249, 253
sys.argv 32
sys.path 222
sys.stdin 31
sys.stdout 31
systemMenu() 662

T

TabbedView 661
tabCloseRequested() 483
TabFocus 450
TabFocusReason 449
tabifiedDockWidgets() 644
tabifyDockWidget() 644
tabPosition() 643, 661
tabShape() 643, 661
tabStopWidth() 503
tabText() 482
takeChild() 534
takeColumn() 531, 534
takeItem() 531
takeRow() 531, 534
takewhile() 156
takeWidget() 487
tan() 72
target() 461
targetChanged() 461
TargetMoveAction 460
task_done() 389
tee() 158
tell() 269, 273
terminate() 386
testOption() 662, 663
text() 454, 462, 465, 490, 492, 496, 513, 519, 535, 595, 649
Text_factory 327, 332
TextAlignmentRole 527
TextAntialiasing 566
textBackgroundColor() 505
TextBrowserInteraction 491, 503
TextCalendar 189, 190
textChanged() 497, 502
textColor() 504
textCursor() 508, 596
TextDontClip 565
TextEditable 491, 502
textEdited() 498
TextEditorInteraction 491, 502
TextExpandTabs 565
TextHideMnemonic 565
TextInput 619, 620
TextSelectableByKeyboard 491, 502
TextSelectableByMouse 491, 502
TextShowMnemonic 565
TextSingleLine 565
textValue() 620
textValueChanged() 620
textValueSelected() 620
textWidth() 596
TextWordWrap 565
TextWrapAnywhere 565
Threading 395
Thursday 517
THURSDAY 190
TicksAbove 521
TicksBelow 521
TicksBothSides 521
TicksLeft 521
TicksRight 521
tileSubWindows() 661
Time 173, 175, 177, 178, 182—184, 186—188, 194, 216, 217, 379, 514, 515
time() 173, 187, 515
timeChanged() 515
TimeCriticalPriority 382
timedelta 178, 179
timegm() 194
timeit() 173, 195, 196
timeout() 440
Timer 195, 443
timerEvent() 439
timerId() 439, 441
timetuple() 182, 187
timetz() 187
tip() 651
title() 98, 637, 647
titleBarWidget() 657
tobytes() 276
toCmyk() 555
today() 180, 185
toggle() 493, 548, 652
ToggleCurrent 548
toggled() 479, 493—495, 652

toggleViewAction() 655, 657
toHsl() 555
toHsv() 555
toHtml() 501, 505, 510, 596
toImage() 571
tolist() 276
Tool 400
toolBarArea() 641
toolBarBreak() 642
ToolButtonFollowStyle 641, 655
ToolButtonIconOnly 641, 655
toolButtonStyle() 641, 655, 656
ToolButtonTextBesideIcon 641, 655
ToolButtonTextOnly 641, 655
ToolButtonTextUnderIcon 641, 655
toolTip() 400, 424, 650, 663
ToolTipRole 524, 527
toordinal() 181, 182, 186, 188
top() 412
TopDockWidgetArea 642, 643, 657
toPlainText() 500, 505, 510, 596
topLeft() 413
TopLeftCorner 643
topLevelChanged() 655, 658
topLevelItem() 591
topRight() 413
TopRightCorner 643
TopToBottom 472, 519, 540
TopToolBarArea 641, 654
toPyDate() 514
toPyDateTime() 514
toPyObject() 378
toPyTime() 514
toRgb() 555
total_seconds() 179
traceback 249
transform() 591
transformed() 572, 573, 576
translate() 101, 412, 568, 587, 590
translated() 412
Transparent 552
transpose() 409
Triangular 483, 643, 661
trigger() 652, 664
triggered() 645, 648, 652, 653, 656
True 38, 55
truncate() 268, 275
try 248
tryLock() 392
Tuesday 190, 517
tuple() 38, 46, 151
type() 39, 43, 443
TypeError 96, 146, 151, 256, 379
typeName() 379
tzinfo 178, 183—187
tzname() 184, 188

U

Uic 373
UliPad 14
UnboundLocalError 211, 256
Unchecked 494, 528, 535
underline() 561
undo() 497, 501, 505, 510
UndoAndRedoStacks 506
undoAvailable() 502, 507
undoCommandAdded() 507
UndoStack 506
unescape() 348
ungrabKeyboard() 590, 601
ungrabMouse() 590, 602
UNICODE 115, 119
UnicodeDecodeError 45, 78, 256
UnicodeEncodeError 105, 109, 256
uniform() 74
union() 168, 172
UniqueConnection 430
unite() 414
united() 414
UniversalDetector 358
Unknown 664
unlink() 279
unlock() 392, 394
Unpickler 287
unquote() 346
unquote_plus() 346
unquote_to_bytes() 346
unsetCursor() 459, 590
Up 508
UpArrow 656
UpArrowCursor 458
update() 112, 166, 168, 289, 447, 448, 551, 584, 592, 598
updateScene() 588
updateSceneRect() 588
UpDownArrows 512
upper() 98
urlencode() 344
urljoin() 347
urllib.parse 340, 343—345, 347
urllib.request 340, 354
urlopen() 354, 356
urlparse() 340—342

urls() 462
urlsplit() 342
urlunparse() 342
urlunsplit() 343
URL-адрес 340
UseListViewForComboBoxItems 620
User 444
User-Agent 353
Username 342
UserRole 528
usleep() 384
UTC 515
utcfromtimestamp() 185
utcnow() 185
utcoffset() 184, 188
utctimetuple() 187
utime() 281

V

validateCurrentPage() 634
validatePage() 639
value() 513, 518—520, 554, 633
valueChanged() 513, 519—521
ValueError 65, 99, 100, 111, 146, 147, 153, 175, 180, 183, 185, 186, 247, 256, 344
valueF() 555
values() 164, 288, 352
vars() 213
VERBOSE 115
Version 351
Vertical 458, 486, 519, 520, 522, 532, 545, 603, 609—611, 642, 644
verticalHeader() 541, 545
verticalHeaderItem() 532
verticalScrollBar() 488
VerticalTabs 643
viewMode() 661
viewport() 488, 567
viewportEntered() 539
views() 583
visibilityChanged() 655, 658
visitedPages() 634
visualIndex() 546
VLine 480

W

W_OK 278
WA_DeleteOnClose 418, 425, 660
WA_NoMousePropagation 456
wait() 386
WaitCursor 458, 459
walk() 290, 291
Warning 337, 613, 664
wasCanceled() 633
WatermarkPixmap 636
Wednesday 190, 517
weekday() 182, 188, 194
Weeks 179
weight() 560
West 482, 643, 661
whatsThis() 424, 650
WhatsThisCursor 458
WhatsThisRole 527
Wheel 443
wheelEvent() 457, 602
WheelFocus 450
while 26, 66—68, 140, 196
White 396, 552
Widget 399
widget() 476, 483, 485, 487, 657, 662
widgetForAction() 654, 655
widgetRemoved() 476, 477
WidgetShortcut 452, 650
WidgetWidth 503
WidgetWithChildrenShortcut 452, 650
width() 402, 405, 408, 412, 561, 571, 574, 579
Window 399, 420, 567
WindowActivate 443
WindowActive 415
WindowBlocked 444
WindowCloseButtonHint 401
WindowContextHelpButtonHint 401
WindowDeactivate 443
windowFlags() 401
WindowFullScreen 415
WindowMaximizeButtonHint 400
WindowMaximized 415
WindowMinimizeButtonHint 400
WindowMinimized 415
WindowMinMaxButtonsHint 401
WindowModal 417, 418, 609
windowModality() 417, 609
WindowNoState 415
windowOpacity() 417
WindowsError 279—281
WindowShortcut 452, 650
windowState() 415
WindowStateChange 444, 445
windowStateChanged() 663
WindowStaysOnBottomHint 401
WindowStaysOnTopHint 396, 401
WindowSystemMenuHint 400

WindowText 420
WindowTitleHint 400
windowType() 400
WindowUnblocked 444
winId() 572
WinPanel 480
with 252—254, 339, 394, 395
wizard() 637
WordLeft 508
WordRight 509
WordUnderCursor 510
WordWrap 503
WrapAllRows 475
WrapAnywhere 503
WrapAtWordBoundaryOrAnywhere 503
WrapLongRows 475
write() 31, 265, 272, 273
writelines() 266, 274

X

x() 404, 407, 412, 455, 458, 589, 648
X_OK 278
x1() 558
x2() 558
XButton1 456
XButton2 456
xml.sax.saxutils 347
xOffset() 598

Y

y() 404, 407, 412, 455, 458, 589, 648
y1() 558
y2() 558
Year 181, 186
YearSection 515
yearShown() 516
Yellow 552, 554
yellowF() 554
Yes 610, 613
YesRole 611, 612, 614
YesToAll 610, 613
Yield 206
yOffset() 599

Z

ZeroDivisionError 249, 255, 256
zfill() 89
zip() 142, 143, 160
zip_longest() 157
zoomIn() 501
zoomOut() 501
zValue() 589

Б

Безопасность 322

В

Ввод 31
◊ перенаправление 283
Время 173
Вывод 29
◊ перенаправление 283
Выделение блоков 26
Выражения-генераторы 141

Г

Генераторы
◊ множеств 172
◊ словарей 166
◊ списков 140

Д

Дата 173
◊ текущая 173
◊ форматирование 175
Декораторы классов 246

Деструктор 232
Динамическая типизация 41, 44
Добавление записей в таблицы 301
Документация 33

З

Записи базы данных
◊ вставка 301
◊ добавление 301
◊ извлечение 305
◊ извлечение из нескольких таблиц 308
◊ количество 306
◊ максимальное значение 306
◊ минимальное значение 306
◊ обновление 303
◊ ограничение при выводе 307
◊ сортировка 307
◊ средняя величина 306
◊ сумма значений 307
◊ удаление 303
Запуск программы 24, 32
Засыпание скрипта 177

И

Извлечение записей 305
Изменение структуры таблицы 304
Именование переменных 36
Индекс 132, 152, 312
Индикатор выполнения процесса 285
Исключения 247
◊ возбуждение 256
◊ иерархия классов 254
◊ перехват всех исключений 250
◊ пользовательские 256

К

Календарь 189
◊ HTML 191
◊ текстовый 190
Каталог 289
◊ обход дерева 290
◊ очистка дерева каталогов 291
◊ права доступа 277
◊ преобразование пути 281
◊ создание 290
◊ список объектов 290
◊ текущий рабочий 260, 289
◊ удаление 290
Квантификатор 119
Класс 228
Ключ 312
Ключевые слова 36
Кодировка 23, 25
◊ определение 357
Комментарии 27
Конструктор 231
Кортеж 132, 151
◊ количество элементов 153
◊ объединение 152
◊ повторение 152
◊ поиск элементов 153
◊ проверка на вхождение 152
◊ создание 151
◊ срез 152

Л

Локаль 97

М

Маска прав доступа 277
Множества 167
◊ frozenset 171
◊ set 167
◊ генераторы 172
Модуль 200, 216
◊ импорт модулей внутри пакета 226
◊ импортирование 216
◊ инструкция from 219
◊ инструкция import 216
◊ относительный импорт 226
◊ повторная загрузка 223
◊ получение значения атрибута 217
◊ проверка существования атрибута 217
◊ пути поиска 222
◊ список всех идентификаторов 219

Н

Наследование 232
◊ множественное 234

О

Обновление записей 303
Объектно-ориентированное программирование (JJG) 228
◊ абстрактные методы 243

Объектно-ориентированное программирование (JJG) *(прод.)*
◊ декораторы 246
◊ деструктор 232
◊ класс 228
◊ классические классы 228
◊ конструктор 231
◊ методы класса 242
◊ множественное наследование 234
◊ наследование 232
◊ определение класса 228
◊ перегрузка операторов 239
◊ псевдочастные атрибуты 244
◊ свойства класса 245
◊ создание атрибута класса 229
◊ создание метода класса 229
◊ создание экземпляра класса 229
◊ специальные методы 236
◊ статические методы 242
Операторы 48
◊ break 67
◊ continue 67
◊ for 61
◊ if…else 58, 59, 61
◊ in 56
◊ is 56
◊ pass 197
◊ while 66
◊ двоичные 50
◊ для работы с последовательностями 51
◊ логические 57
◊ математические 48
◊ перегрузка 239
◊ приоритет выполнения 53
◊ присваивания 52
◊ сравнения 56
◊ условные 55
Отображения 40
Ошибка
◊ времени выполнения 247
◊ логическая 247
◊ синтаксическая 247

П

Пакет 224
Переменная 36
◊ глобальная 210
◊ локальная 210
◊ удаление 47
Перенаправление ввода/вывода 283
Перенос строк 27
Последовательности 40
◊ количество элементов 136
◊ объединение 138
◊ операторы 51
◊ перебор элементов 139
◊ повторение 138
◊ преобразование в кортеж 151
◊ преобразование в список 132
◊ проверка на вхождение 138
◊ сортировка 150
◊ срез 137
Права доступа 277
Присваивание 41
◊ групповое 41
◊ позиционное 42
Путь к интерпретатору 25

Р

Регулярные выражения 114
◊ группировка 120
◊ замена 128
◊ квантификаторы 119
◊ классы 118
◊ метасимволы 116
◊ обратная ссылка 120
◊ поиск всех совпадений 127
◊ поиск первого совпадения 123
◊ разбиение строки 130
◊ специальные символы 115
◊ флаги 114
◊ экранирование спецсимволов 131
Редактирование файла 24
Рекурсия 209

С

Сигналы 426
Словарь 159
◊ генераторы 166
◊ добавление элементов 166
◊ количество элементов 163
◊ перебор элементов 163
◊ поверхностная копия 161
◊ полная копия 161
◊ проверка существования ключа 162, 164
◊ создание 159
◊ список значений 164

◊ список ключей 164
◊ удаление элементов 163, 165
Слот 367
События 426
Создание файла с программой 23
Специальный символ 81
Список 132
◊ выбор элементов случайным образом 148
◊ генераторы 140
◊ добавление элементов 144
◊ заполнение числами 150
◊ количество элементов 136
◊ максимальное значение 147
◊ минимальное значение 147
◊ многомерный 139
◊ перебор элементов 139
◊ переворачивание 148
◊ перемешивание 148
◊ поверхностная копия 135
◊ поиск элемента 147
◊ полная копия 135
◊ преобразование в строку 151
◊ соединение двух списков 138
◊ создание 132
◊ сортировка 149
◊ срез 137
◊ удаление элементов 146
Срез 82, 137
Строки 76
◊ длина 83, 94
◊ документирования 28, 34, 79
◊ замена в строке 101
◊ изменение регистра 98
◊ код символа 98
◊ кодирование 112
◊ конкатенация 83
◊ методы 94
◊ неформатированные 80
◊ операции 81
◊ перебор символов 83
◊ повторение 84
◊ поиск в строке 99
◊ преобразование объекта в строку 112
◊ проверка на вхождение 84
◊ проверка типа содержимого 102
◊ разбиение 95
◊ соединение 83
◊ создание 77
◊ специальные символы 81
◊ срез 82
◊ тип данных 76
◊ удаление пробельных символов 94
◊ форматирование 84, 90
◊ функции 94
◊ шифрование 112
◊ экранирование спецсимволов 79
Структура программы 24

Т

Таблица базы данных
◊ изменение структуры 304
◊ создание 295
◊ удаление 317
Текущий рабочий каталог 260
Тип данных 38
◊ преобразование 44
◊ проверка 43

У

Удаление записей 303
Установка
◊ PyQt 361
◊ Python 17

Ф

Файл 259
◊ абсолютный путь 259
◊ время последнего доступа 280
◊ время последнего изменения 280
◊ дата создания 280
◊ дескриптор 268
◊ закрытие 265, 272
◊ запись 265, 272
◊ копирование 278
◊ обрезание 268
◊ открытие 259, 271
◊ относительный путь 260
◊ переименование 279
◊ перемещение 279
◊ перемещение указателя 269
◊ позиция указателя 269
◊ права доступа 277
◊ преобразование пути 281
◊ проверка существования 279
◊ размер 279
◊ режим открытия 263
◊ создание 259

Файл *(прод.)*
◊ сохранение объектов 286
◊ удаление 279
◊ чтение 266, 272
Факториал 209
Функция 197
◊ аннотации функций 215
◊ анонимная 205, 211
◊ вложенная 213
◊ вызов 198
◊ генератор 206
◊ декораторы 207
◊ значения параметров по умолчанию 203
◊ лямбда 205
◊ необязательные параметры 201
◊ обратного вызова 199
◊ переменное число параметров 203
◊ расположение определений 200
◊ рекурсия 209
◊ создание 197
◊ сопоставление по ключам 201

Ц

Цикл
◊ for 61
◊ while 66
◊ переход на следующую итерацию 67
◊ прерывание 66, 67

Ч

Числа 69
◊ абсолютное значение 71, 73
◊ вещественные 69
  ▫ точность вычислений 70
◊ возведение в степень 71, 73
◊ восьмеричные 69
◊ двоичные 69
◊ десятеричные 69
◊ квадратный корень 73
◊ комплексные 69, 70
◊ логарифм 73
◊ модуль math 72
◊ модуль random 73
◊ округление 71, 73
◊ преобразование 70
◊ случайные 73
◊ факториал 73
◊ функции 70
◊ целые 69
◊ шестнадцатеричные 69
◊ экспонента 73

Я

Язык 97
Язык SQL
◊ ABORT 300
◊ ALL 306
◊ ALTER TABLE 304
◊ ANALYZE 314
◊ AUTOINCREMENT 299
◊ AVG() 306
◊ BEGIN 315, 317
◊ CHECK 299, 300
◊ COLLATE 298
◊ COMMIT 316
◊ COUNT() 306
◊ CREATE INDEX 313
◊ CREATE TABLE 295
◊ CROSS JOIN 308
◊ DEFAULT 298
◊ DELETE FROM 303
◊ DISTINCT 306
◊ DROP INDEX 314
◊ DROP TABLE 296, 317
◊ END 316
◊ ESCAPE 311
◊ EXPLAIN 313
◊ FAIL 301
◊ FROM 308
◊ GROUP BY 306
◊ HAVING 306, 309
◊ IGNORE 301
◊ INNER JOIN 308
◊ INSERT 315
◊ INSERT INTO 301
◊ JOIN 308, 309
◊ LEFT JOIN 309
◊ LIKE 310
◊ LIMIT 307
◊ MAX() 306
◊ MIN() 306
◊ ON CONFLICT 300
◊ ORDER BY 307
◊ PRAGMA 295
◊ PRIMARY KEY 299, 300, 313
◊ REINDEX 314

◊ RELEASE 317
◊ REPLACE 301, 303
◊ ROLLBACK 300, 316
◊ SAVEPOINT 317
◊ SELECT 305, 308, 314
◊ SUM() 307
◊ UNIQUE 299, 300
◊ UPDATE 303
◊ USING 308
◊ VACUUM 304, 314
◊ WHERE 305, 308, 309
◊ агрегатные функции 306
◊ вложенные запросы 314
◊ вставка записей 301
◊ выбор записей 305
◊ выбор записей из нескольких таблиц 308
◊ изменение свойств таблицы 304
◊ индексы 312
◊ обновление записей 303
◊ создание базы данных 293
◊ создание таблицы 295
◊ транзакции 315
◊ удаление базы данных 317
◊ удаление записей 303
◊ удаление таблицы 317