ВЕДЬ ЭТО ТАК ПРОСТО!

2-е издание

Python®

для Чайников®

Издательство ДИАЛЕКТИКА

Создайте и запустите свое первое приложение на Python

Научитесь выполнять отладку и устранять ошибки

Освойте работу с Anaconda и Jupyter Notebook

Джон Пол Мюллер

Python®

для чайников®

Beginning Programming with Python®

by John Paul Mueller

Python®

Джон Пол Мюллер

для чайников®

Диалектика

Москва ◆ Санкт-Петербург
2019

ББК 32.973.26-018.2.75
М98
УДК 681.3.07

Компьютерное издательство "Диалектика"

Перевод с английского *А.П. Сергеева*
Под редакцией *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Мюллер, Джон Пол

М98 Python для чайников, 2-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 416 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-26-2 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства John Wiley & Sons, Inc.

Copyright © 2019 by Dialektika Computer Publishing.
Original English edition Copyright © 2018 by John Wiley & Sons, Inc.
All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with John Wiley & Sons, Inc.

Научно-популярное издание
Джон Пол Мюллер

Python для чайников,
2-е издание

Подписано в печать 29.04.2019. Формат 70×100/16
Гарнитура Times.
Усл. печ. л. 33,54. Уч.-изд. л. 21,1
Тираж 500 экз. Заказ № 4201

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-26-2 (рус.)
ISBN 978-1-119-45789-3 (англ.)

© 2019 ООО "Диалектика"
© 2018 by John Wiley & Sons, Inc.

Оглавление

Введение 15

Часть 1. Знакомство со средой Python 19
Глава 1. Разговор с компьютером 21
Глава 2. Установка Python 35
Глава 3. Взаимодействие с Python 53
Глава 4. Создание первого приложения 73
Глава 5. Работа с Anaconda 101

Часть 2. Основы программирования на Python 121
Глава 6. Хранение и изменение информации 123
Глава 7. Управление данными 133
Глава 8. Принятие решений 155
Глава 9. Выполнение повторяющихся действий 171
Глава 10. Обработка ошибок 185

Часть 3. Профессиональные методики программирования на Python 217
Глава 11. Пакеты 219
Глава 12. Работа со строками 251
Глава 13. Управление списками 269
Глава 14. Коллекции данных 289
Глава 15. Создание и использование классов 309

Часть 4. Решение сложных задач 333
Глава 16. Хранение данных в файлах 335
Глава 17. Отправка сообщений электронной почты 353

Часть 5. Великолепные десятки 371
Глава 18. Десять ценных ресурсов для программистов 373
Глава 19. Десять утилит, улучшающих работу с Python 385
Глава 20. Десять (или около того) библиотек, о которых стоит знать 397
Предметный указатель 406

Содержание

Об авторе 14

Введение 15
Особенности книги 15
Исходные предположения 16
Пиктограммы, используемые в книге 17
Дополнительные материалы 17
Что дальше 18
Ждем ваших отзывов! 18

Часть 1. Знакомство со средой Python 19

Глава 1. Разговор с компьютером 21
Зачем нужно общаться с компьютером 22
Приложение — это тоже средство общения 23
Повседневные процедуры 23
Создание процедур 24
Приложения — это те же процедуры 25
Компьютеры воспринимают все буквально 25
Что же такое приложение 25
Компьютеры общаются на специальном языке 26
Как наладить общение человека с компьютером 27
В чем секрет популярности Python 28
Почему именно Python 29
Преимущества Python 30
В каких организациях используется Python 31
Где найти полезные приложения Python 32
Python и другие языки программирования 33

Глава 2. Установка Python 35
Загрузка нужной версии 35
Инсталляция Python 38
В Windows 39
В Mac 42
В Linux 43
Получение доступа к копии Python на своем компьютере 45
В Windows 46
В Mac 49

В Linux 49
Тестирование установки 50

Глава 3. Взаимодействие с Python 53
Открытие командной строки 54
Запуск Python 54
Как извлечь максимум пользы из командной строки 55
Использование переменных среды Python 58
Ввод команд 59
Скажите компьютеру, что ему нужно делать 59
Расскажите компьютеру, что вы сделали 60
Просмотр результатов 60
Справочная система 62
Переход в режим справки 63
Получение справки 64
Выход из режима справки 66
Непосредственное получение справки 67
Выход из режима командной строки 69

Глава 4. Создание первого приложения 73
Преимущества IDE 74
Создание оптимального кода 74
Контроль ошибок 75
Преимущества блокнота 75
Получаем собственную копию Anaconda 76
Где можно скачать Anaconda 76
Установка Anaconda в Linux 77
Установка Anaconda в Mac OS 78
Установка Anaconda в Windows 80
Загрузка наборов данных и файлов примеров 83
Работа с Jupyter Notebook 83
Создание репозитория кода 84
Создание приложения 90
Знакомство с ячейками 90
Добавление документирующих ячеек 92
Другое содержимое ячеек 93
Использование отступов 94
Добавление комментариев 95
Знакомство с комментариями 96
Применение комментариев в качестве напоминаний 98
Применение комментариев для предотвращения выполнения строк кода 98
Как закрыть Jupyter Notebook 98

Глава 5. Работа с Anaconda 101
Загрузка кода 102
Контрольные точки 103
Методика использования контрольных точек 103
Сохранение контрольных точек 104
Восстановление контрольной точки 104
Работа с ячейками 105
Добавление различных типов ячеек 105
Разделение и объединение ячеек 106
Перемещение ячеек 106
Выполнение содержимого ячеек 107
Отключение вывода 108
Изменение внешнего вида Jupyter Notebook 109
Поиск команд в окне Command Palette 110
Отображение номеров строк 110
Панель инструментов ячейки 111
Взаимодействие с ядром 113
Получение справки 114
Применение магических функций 116
Обзор выполняющихся процессов 120

Часть 2. Основы программирования на Python 121

Глава 6. Хранение и изменение информации 123
Хранение данных 124
Использование переменных 124
Правильный выбор переменных 124
Основные типы данных 125
Добавление информации в переменные 125
Знакомство с числовыми типами данных 125
Булевы значения 130
Строки 130
Дата и время 131

Глава 7. Управление данными 133
Представление данных в Python 134
Выполнение сравнений 134
Как компьютеры выполняют операции сравнения 135
Работа с операторами 135
Виды операторов 136
Приоритет операторов 142
Создание и использование функций 143

Функции как средство упаковки кода 144
Повторное использование кода 144
Определение функции 145
Получение доступа к функциям 146
Передача информации функциям 147
Возврат информации из функций 150
Сравнение результатов выполнения функций 152
Получение пользовательского ввода 153

Глава 8. Принятие решений 155
Принятие простых решений с помощью инструкции `if` 156
Знакомство с инструкцией `if` 156
Использование инструкции `if` 156
Выбор альтернативных вариантов с помощью инструкции `if...else` 161
Знакомство с инструкцией `if...else` 161
Использование инструкции `if...else` 162
Использование инструкции `if...elif` 163
Использование вложенных инструкций принятия решений 166
Использование нескольких инструкций `if` или `if...else` 166
Комбинирование различных инструкций принятия решений 167

Глава 9. Выполнение повторяющихся действий 171
Обработка данных с помощью цикла `for` 172
Знакомство с циклом `for` 172
Создание базового цикла `for` 173
Управление выполнением цикла с помощью инструкции `break` 174
Управление выполнением цикла с помощью инструкции `continue` 176
Управление выполнением цикла с помощью инструкции `pass` 177
Управление выполнением цикла с помощью предложения `else` 178
Обработка данных с помощью цикла `while` 179
Знакомство с циклом `while` 180
Использование цикла `while` в приложении 181
Вложенные циклы 182

Глава 10. Обработка ошибок 185
Почему Python вас не понимает 186
Возможные причины появления ошибок 188
Этапы появления ошибок 188
Типы ошибок 189
Перехват исключений 191
Базовая обработка исключений 192
Иерархическая обработка менее специфичных исключений 203

Обработка вложенных исключений 206
Генерирование исключений 208
Генерирование исключений в нештатных ситуациях 209
Передача информации об ошибке вызывающему коду 211
Создание и использование настраиваемых исключений 212
Использование предложения `finally` 213

Часть 3. Профессиональные методики программирования на Python 217

Глава 11. Пакеты 219
Создание блоков кода 220
Типы пакетов 222
Кеш пакетов 223
Импорт пакетов 225
Использование инструкции `import` 226
Использование инструкции `from...import` 228
Поиск пакетов на диске 231
Загрузка пакетов из других источников 232
Открытие командной строки Anaconda 233
Работа с пакетами `conda` 233
Установка пакетов с помощью утилиты `pip` 240
Просмотр содержимого пакета 240
Просмотр документации по пакетам 244
Открытие приложения Pydoc 244
Использование ссылок быстрого доступа 246
Ввод поискового термина 246
Просмотр результатов 247

Глава 12. Работа со строками 251
Различия между строками 252
Определение символа с помощью чисел 252
Создание строк на основе символов 253
Создание строк, включающих специальные символы 254
Выбор отдельных символов 257
Обработка строк 259
Поиск значения в строке 262
Форматирование строк 265

Глава 13. Управление списками 269
Упорядочение информации в приложении 270
Знакомство со списками 270
Как компьютер просматривает списки 271

Создание списков 272
Доступ к спискам 274
Циклический обход списков 276
Изменение списков 276
Поиск в списках 280
Сортировка списков 282
Вывод списков 283
Работа с объектом `Counter` 285

Глава 14. Коллекции данных 289
Понятие коллекции 290
Работа с кортежами 291
Работа со словарями 294
Создание и использование словаря 294
Замена инструкции `switch` словарем 297
Создание стеков с помощью списков 299
Работа с очередями 303
Работа с двухсторонней очередью 305

Глава 15. Создание и использование классов 309
Класс как средство упаковки кода 310
Компоненты класса 312
Создание определения класса 312
Встроенные атрибуты класса 314
Работа с методами 314
Работа с конструкторами 317
Работа с переменными 319
Использование методов с переменным числом аргументов 321
Перегрузка операторов 323
Создание класса 325
Определение класса `MyClass` 325
Сохранение класса на диске 326
Использование класса в приложении 327
Расширение классов для создания новых классов 328
Создание дочернего класса 329
Тестирование класса в приложении 330

Часть 4. Решение сложных задач 333

Глава 16. Хранение данных в файлах 335
Принципы работы долговременного хранилища 336
Создание содержимого для долговременного хранения 338
Создание файла 341

Чтение содержимого файла 344
Изменение содержимого файла 347
Удаление файла 351

Глава 17. Отправка сообщений электронной почты 353
Что происходит при отправке электронной почты 355
Структура электронного письма 355
Компоненты конверта электронного сообщения 356
Компоненты сообщения 361
Создание электронного сообщения 365
Работа с текстовым сообщением 366
Работа с HTML-сообщением 367
Просмотр электронного сообщения 368

Часть 5. Великолепные десятки 371

Глава 18. Десять ценных ресурсов для программистов 373
Онлайн-документация Python 374
Руководство на сайте `LearnPython.org` 375
Веб-программирование на Python 376
Дополнительные библиотеки 377
Быстрое создание приложений с помощью IDE 378
Удобная проверка синтаксиса 379
Использование XML 379
Устранение типичных ошибок, присущих новичкам 381
Знакомство с Unicode 382
Учимся создавать быстродействующие приложения 382

Глава 19. Десять утилит, улучшающих работу с Python 385
Отслеживание ошибок с помощью Roundup Issue Tracker 386
Создание виртуальной среды с помощью VirtualEnv 387
Инсталляция приложений с помощью PyInstaller 389
Создание документации для разработчиков с помощью утилиты `pdoc` 390
Разработка кода приложения с помощью Komodo Edit 391
Отладка приложения с помощью утилиты `pydbgr` 392
Переход в интерактивную среду с помощью IPython 393
Тестирование приложений Python с помощью PyUnit 393
Приведение кода в порядок с помощью утилиты `isort` 394
Управление версиями с помощью Mercurial 395

Глава 20. Десять (или около того) библиотек, о которых стоит знать 397
Создание безопасной среды с помощью PyCrypto 398
Взаимодействие с базами данных с помощью SQLAlchemy 399
Смотрим на мир с помощью Google Maps 399

Добавление графического интерфейса пользователя с помощью TkInter 400
Создание презентации с табличными данными с помощью PrettyTable 400
Добавление звука с помощью PyAudio 401
Построение графиков с помощью PyQtGraph 402
Поиск информации с помощью IRLib 403
Создание интероперабельной среды Java с помощью JPype 404
Доступ к локальным сетевым ресурсам с помощью Twisted Matrix 404
Доступ к интернет-ресурсам с помощью библиотек 405

Предметный указатель 406

Об авторе

Джон Пол Мюллер — писатель и технический редактор, автор более 100 книг и 600 статей, тематика которых варьируется от сетевых технологий и искусственного интеллекта до управления базами данных и программирования на Python. Как технический редактор помог издать книги более чем 70 авторам. Ведет персональный блог по адресу `http://blog.johnmuellerbooks.com/`. Можете отправить ему электронное письмо по адресу `John@JohnMuellerBooks.com` или посетить его сайт `www.johnmuellerbooks.com`.

Введение

Python — универсальный язык программирования с широчайшими возможностями. Причем это не просто мое личное мнение. По итогам последнего голосования, проведенного среди разработчиков, Python стал третьим по популярности языком программирования в мире (`https://www.tiobe.com/tiobe-index/`). Замечательная особенность Python заключается в том, что с его помощью можно создать приложение на одной платформе, а запускать — на любых других. В отличие от других языков программирования, которые только обещают независимость от платформы, Python эту независимость реализует. В данном случае обещания и реальность совпадают.

Для Python характерны читабельность кода и краткость синтаксиса. Это позволяет создавать более компактные приложения, чем при использовании других языков программирования. А поскольку в Python поддерживаются функциональный, императивный, объектно-ориентированный и процедурный стили программирования, вы сможете выбрать тот стиль, который лучше всего вам подходит (глава 3). Python находит применение в любых сферах деятельности. Книга призвана помочь всем, включая непрофессионалов, быстро установить Python и начать программировать.

Некоторые пользователи относятся к Python как к языку написания сценариев, хотя в действительности его возможности намного шире. Более того, Python отлично подходит для образовательных и иных целей, для которых не годятся другие языки программирования. В примерах книги используется среда Jupyter Notebook, которая основана на парадигме грамотного программирования. Эта парадигма была сформулирована компьютерным ученым из Стэнфордского университета Дональдом Кнутом (глава 4). В итоге примеры выглядят как наглядные отчеты, которые легко поймет любой читатель.

Особенности книги

В книге вы найдете все, что нужно для установки и запуска Python. Вы сможете быстро освоить этот язык программирования и воспользоваться им для написания любых приложений. В отличие от большинства книг подобной тематики, здесь с самого начала рассматриваются характерные особенности Python, отличающие его от других языков программирования. Вы поймете,

какие инструменты вам нужны, чтобы начать работу, и как установить Python в конкретной системе.

Далее вы приступите к изучению языка, начав с азов и постепенно продвигаясь к более сложным темам. После успешного освоения примеров книги вы сможете создавать простые приложения и выполнять такие задачи, как отправка электронной почты с помощью Python. Конечно, вы не станете экспертом программирования, но вполне сможете использовать Python для решения производственных задач.

В книге применяются следующие условные обозначения.

- Текст, вводимый пользователем, выделяется в книге **полужирным моноширинным** шрифтом.
- Если в строке ввода есть слова, выделенные *курсивом*, то их нужно заменить фактическим значением.
- Адреса сайтов и программный код выделяются моноширинным шрифтом.
- Если нужно выбрать последовательность команд в меню, то они разделяются стрелкой, например File⇨New File (Файл⇨Создать файл). В данном случае вы сначала переходите в меню File, а затем в этом меню выбираете пункт New File. В результате будет создан новый файл.

Исходные предположения

Возможно, вам будет трудно поверить в то, что я что-то предполагаю о вас. В конце концов, мы даже ни разу с вами не встречались! И хотя большинство подобных предположений выглядит надуманно, я все же рискну их сформулировать, чтобы на что-то опереться.

Большое значение имеет знакомство с применяемой вами платформой, поскольку сама книга не дает каких-либо указаний на этот счет. Разве что в главе 2 приведены инструкции по установке Python на разных платформах, а в главе 4 описывается установка Anaconda, которая включает Jupyter Notebook. Это интегрированная среда разработки (Integrated Development Environment — IDE), применяемая для выполнения примеров книги. Чтобы вы получили максимум информации о Python, в книге не рассматриваются вопросы, зависящие от платформы. Прежде чем приступать к чтению книги, убедитесь в том, что знаете, как устанавливать приложения и запускать их, а также понимаете общие принципы работы на выбранной вами платформе.

В книге также предполагается, что вы умеете искать информацию в Интернете. Поэтому повсеместно приводятся ссылки на материал, доступный

в Интернете, который позволит вам повысить уровень знаний. Учтите, что эти источники информации будут полезны только в том случае, если вы их действительно найдете и изучите.

Пиктограммы, используемые в книге

В процессе чтения книги вы будете видеть пиктограммы на полях, которые обозначают определенный материал. Эти пиктограммы кратко описаны ниже.

Следуя советам, вы сэкономите время или сможете решить поставленную задачу с меньшими усилиями. Советы, приведенные в книге, — это в основном способы экономии времени или ссылки на ресурсы, которыми следует воспользоваться, чтобы эффективнее работать с Python.

Если вы увидите такую пиктограмму, то знайте, что не следует выполнять соответствующие действия. Если вы все же рискнете сделать это, то полученный результат, скорее всего, вас разочарует.

Эта пиктограмма обозначает какую-либо техническую подсказку или продвинутую методику. Порой эти крупицы полезной информации довольно скучны, но они же могут описывать решение, столь необходимое для запуска программы. В принципе, эти абзацы можно пропустить без особого ущерба для понимания материала.

Даже если вы не почерпнете для себя ничего полезного из определенного раздела, все равно запомните материал, отмеченный этой пиктограммой. Здесь обычно приводится информация, которую следует знать, чтобы успешно создавать программы на Python.

Дополнительные материалы

Вряд ли вам захочется вручную набирать код примеров. Большинство читателей предпочтут потратить больше времени на работу с примерами. К счастью, все примеры кода, рассмотренные в книге, доступны для загрузки. Более того, каждый пример содержит сведения о проекте, в котором он используется. Чтобы загрузить файлы примеров, воспользуйтесь следующей ссылкой:

`http://go.dialektika.com/PythonFD`

Что дальше

Итак, пришло время начать путешествие в увлекательный мир Python! Если вы вообще не имеете опыта программирования, начните с главы 1, а затем продвигайтесь в удобном для себя темпе.

Если вы новичок в программировании, но хотите освоить Python как можно быстрее, перейдите непосредственно к главе 2, только учтите, что некоторые темы могут оказаться не совсем понятными. Если у вас уже установлена копия Python, можете сразу перейти к главе 3. Но все же бегло просмотрите главу 2, чтобы знать о том, какие предположения были сделаны при написании книги.

Читатели, которые уже имеют определенный опыт работы с Python, могут сразу же перейти к главе 4. Только не забудьте установить Anaconda, чтобы получить доступ к среде Jupyter Notebook, которая в этой книге используется в качестве интегрированной среды разработки. В противном случае вы не сможете запускать скачанные файлы примеров. Anaconda — бесплатный пакет, так что вам не придется ни за что платить.

Если вы уже установили Jupyter Notebook и знаете, как использовать эту среду, можете перейти непосредственно к главе 6. Вы всегда сможете вернуться к предыдущим главам, если в процессе чтения книги у вас возникнут вопросы. Очень важно разобраться в том, как работает каждый пример, прежде чем переходить к следующему примеру. Все примеры являются весьма полезными и поучительными, поэтому не следует пропускать какой-либо из них.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

1 Знакомство со средой Python

В ЭТОЙ ЧАСТИ...

- Общение с компьютером
- Установка Python в Linux, Mac и Windows
- Взаимодействие с инструментами Python
- Установка Anaconda и создание первого приложения

Глава 1

Разговор с компьютером

В ЭТОЙ ГЛАВЕ...

- Поговорим с компьютером
- Создание программ для общения с компьютером
- Знакомство с программами и процессом их создания
- Зачем нужен Python

Беседа с компьютером может показаться похожей на сценарий из научно-фантастического фильма. Если вы смотрели сериал "Звездный путь", то, наверное, помните, как экипаж корабля "Энтерпрайз" общался со своим корабельным компьютером. Причем зачастую инициатором общения был сам компьютер. С появлением таких программ-помощников, как Apple Siri (`http://www.apple.com/ios/siri/`), Amazon Echo (`https://www.amazon.com/dp/B00X4WHP5E/`) и т.п., беседа человека с компьютером стала вполне реальной.

ЗАПОМНИ!

Одно дело — запрашивать компьютер о предоставлении необходимой информации, а другое — самому предоставлять ему требуемые инструкции. В этой главе вы узнаете, зачем нужно давать инструкции компьютеру и какую выгоду можно от всего этого получать. Вы также узнаете о том, зачем нужен специальный язык для такого общения и почему в качестве такого языка следует использовать Python. Но главный вывод, к которому вы придете после прочтения главы, — программирование является лишь одной из разновидностей общения между вами и компьютером.

Зачем нужно общаться с компьютером

На первый взгляд, идея общения с компьютером может показаться странной, но это не совсем так. Более того, это просто необходимо, ведь компьютер пока еще не умеет читать наши мысли. Но даже если бы он владел телепатией, это все равно было бы общением. На самом деле ничто не может произойти без обмена информацией между компьютером и пользователем. Такие действия, как

- чтение электронной почты,
- публикация в социальной сети,
- поиск подарка в Интернете,

являются примерами общения между компьютером и человеком. А последующее взаимодействия компьютера с другими компьютерами или людьми при обработке тех или иных запросов еще раз доказывает тот факт, что для получения какого-либо результата необходимо общение.

В большинстве случаев общение с компьютером происходит так, что вы его практически не замечаете, по крайней мере до тех пор, пока не задумываетесь об этом. Например, когда вы посещаете комнату чата, у вас может возникнуть ощущение, будто вы общаетесь с человеком. Но в действительности вы обмениваетесь информацией с компьютером, а тот в свою очередь обменивается данными с компьютером другого посетителя комнаты чата. Компьютер же другого посетителя комнаты чата напрямую общается со своим владельцем. Описанная ситуация проиллюстрирована на рис. 1.1.

Рис. 1.1. Общение с компьютером может быть незаметным до тех пор, пока вы не задумаетесь об этом

Обратите внимание на облако, показанное в центре рис. 1.1. В этом облаке может находиться все что угодно, но в данном случае в нем находятся другие компьютеры, на которых выполняются иные приложения. Благодаря этим компьютерам вы и ваш собеседник можете общаться в чате. И самое главное заключается в том, что при использовании чата все процессы выполняются в фоновом режиме. Создается впечатление, будто вы просто обмениваетесь со своим собеседником сообщениями, которые как по мановению волшебной палочки появляются на экране.

Приложение — это тоже средство общения

Общение между компьютерами выполняется благодаря *приложениям*. Одни приложения применяются для ответа на сообщения электронной почты, другие — для приобретения товаров, ну а третьи — для создания презентаций. Благодаря приложениям мы можем транслировать человеческие идеи в понятную компьютерам форму и осуществлять обмен данными. Но данные, демонстрируемые в ходе презентации, отличаются от данных, которые мы вводим при совершении покупок в Интернете. Способы представления данных различны для каждой задачи, поэтому при работе за компьютером приходится использовать разные приложения, в зависимости от конкретной ситуации.

В настоящее время существуют приложения, способные удовлетворить практически любую потребность, которую только можно себе вообразить. За много лет программисты создали миллионы различных приложений, поэтому даже трудно себе представить, что можно придумать какой-либо новый метод общения с компьютером, реализуемый с помощью приложения. В действительности это вполне возможно, если отталкиваться от того, какие данные имеются в вашем распоряжении. Порой данные слишком специфичны либо требуются в формате, который не поддерживается ни одним из имеющихся приложений. В результате приходится писать собственные приложения, чтобы передать эти данные компьютеру.

Далее мы поговорим о создании приложений, которые предназначены для обработки уникальных данных тем или иным нестандартным способом. К примеру, у вас есть коллекция видеофайлов и вам нужно работать с ними определенным образом. Поскольку данные уникальны и ваши потребности в доступе к ним тоже специфичны, вам требуется создать приложение, которое позволит вам обрабатывать данные в соответствии с вашими задачами.

Повседневные процедуры

Процедура — это набор действий, которые нужно совершить для решения определенной задачи. Например, чтобы поджарить тосты, можно воспользоваться следующей процедурой.

1. Откройте хлебницу и достаньте два тоста.
2. Включите тостер в розетку.
3. Вставьте тосты в отдельные слоты тостера.
4. Опустите рычаг тостера, чтобы начать процесс приготовления тостов.
5. Подождите, пока тосты не поджарятся.
6. Извлеките тосты из тостера.

7. Положите тосты на тарелку.
8. Достаньте из холодильника масло.
9. Намажьте тосты маслом.

Конечно, выполняемая вами процедура может отличаться от описанной здесь, но вряд ли вы будете смазывать тосты маслом перед помещением в тостер. Просто большинство людей не задумываются о процедуре приготовления тостов. Но даже если мы не осознаем этого, мы все равно следуем определенной процедуре.

ЗАПОМНИ!

Компьютеры могут решать задачи исключительно в рамках заранее оговоренной процедуры. Вам придется рассказать компьютеру о том, какие действия он должен выполнить, каков порядок выполнения этих действий и какие ситуации могут привести к сбою. Вся эта информация (и многое другое) появляется в приложении. По сути, приложение — это процедура, которая содержит указания компьютеру о том, что ему нужно делать, когда делать и как делать. А поскольку все мы сталкиваемся с процедурами в повседневной жизни, нам лишь остается применить имеющиеся знания в компьютерной среде, сообщив компьютеру необходимые сведения о выполняемых задачах.

Создание процедур

Однажды, когда я учился в школе, учительница попросила каждого из нас записать процедуру приготовления тостов. После того как мы завершили выполнение этого задания, она принесла тостер и несколько буханок хлеба. Каждая процедура была зачитана, а указанные в ней инструкции — продемонстрированы на практике. Ни одна из процедур не сработала так, как ожидалось, и все они привели к анекдотичным результатам. В моем случае я забыл указать на необходимость предварительно вынуть хлеб из упаковки, поэтому учительница попыталась засунуть в тостер хлеб прямо в упаковке. Этот урок я запомнил на всю жизнь. Как видите, процесс написания процедур не такой простой, как кажется. Зачастую мы слишком хорошо знаем всю последовательность действий и предполагаем, что человек, выполняющий процедуру, сделает все в точности так, как мы хотим, даже если мы не будем все расписывать в мельчайших подробностях.

Многие события нашей жизни связаны с процедурами. Например, перед взлетом самолета пилот должен выполнить определенную последовательность действий. И без тщательно продуманной процедуры взлета самолет может потерпеть крушение. Безусловно, овладение искусством написания хороших

процедур требует времени и сил, но это вполне решаемая задача. Впрочем, написать процедуру — это только полдела. Ее еще нужно проверить, поручив эту задачу тому, кто не знаком с самой процедурой. При работе с компьютерами главным тестировщиком становится сам компьютер.

Приложения — это те же процедуры

Компьютер ведет себя во многом подобно школьному учителю из предыдущего раздела. При разработке приложения вы фактически пишете процедуру, определяющую набор действий, которые должен выполнить компьютер, чтобы решить ту или иную задачу. Если же вы пропустите какое-то действие, то результаты будут вовсе не такими, как ожидается. Компьютер не умеет читать ваши мысли и вряд ли сможет решать задачи автоматически, без подсказок с вашей стороны. Единственное, что знает компьютер, — это то, что вы предоставили ему определенную процедуру и ему нужно ее выполнить.

Компьютеры воспринимают все буквально

Со временем люди ко всему привыкают. Они начинают автоматически устранять недостатки в ваших процедурах или делать пометки, описывая пропущенные этапы.

ЗАПОМНИ!

Другими словами, люди способны устранять недостатки, присущие процедурам. Когда вы начнете создавать компьютерные программы, вы столкнетесь с тем, что компьютеры точно выполняют предписанные действия и буквально воспринимают каждую инструкцию. Например, если вы говорите компьютеру о том, что переменная равна 5, он присвоит ей значение, в точности равное 5. Для человека значение 4,9 может оказаться вполне приемлемым приближением, но компьютер не способен на подобные умозаключения. Он решит, что значение 4,9 не равно 5. Короче говоря, компьютерам не присущи такие качества, как гибкость, интуиция и воображение. Если вы написали процедуру, предназначенную для компьютера, то он будет каждый раз выполнять ее в буквальном виде, ничего не меняя и не делая никаких скрытых предположений.

Что же такое приложение

Как уже было сказано, приложения позволяют выражать человеческие идеи в виде, понятном компьютерам. Для этого приложение содержит одну или несколько процедур, которые сообщают компьютеру о том, как выполнять

задачи, связанные с обработкой и представлением данных. К примеру, то, что вы видите на экране, — это текст, отображаемый текстовым редактором. Но для отображения этого текста на экране компьютеру требуются процедуры, которые считывают данные с диска, преобразуют их в понятный формат, а затем выводят на экран.

Компьютеры общаются на специальном языке

Человеческий язык сложен и труден для понимания. Даже такие современные голосовые помощники, как Siri и Alexa, имеют серьезные ограничения в понимании человеческой речи. За много лет компьютеры научились обрабатывать потоки аудиоданных и воспринимать отдельные четко произнесенные слова в качестве команд, но по-прежнему испытывают серьезные затруднения в восприятии речи. Трудности понимания речи станут наглядными, если познакомиться с юридической практикой. Большинство людей, далеких от юриспруденции, просто не понимают юридические тексты. Однако они специально написаны таким образом, чтобы не допускать свободную интерпретацию. Проблема в том, что человеческой речи всегда присуща нечеткость формулировок.

Учитывая вышесказанное, можно сказать, что компьютеры никогда не будут понимать человеческую речь настолько хорошо, чтобы осознать процедуры, предназначенные для людей. Компьютеры всегда все воспринимают буквально, поэтому вы получите совершенно непредсказуемые результаты, если будете использовать человеческий язык для написания приложений. Вот почему для общения с компьютерами люди придумывают специальные языки, называемые *языками программирования*. Эти языки позволяют писать процедуры, которые являются с одной стороны предельно конкретными, а с другой — вполне понятными и людям, и компьютерам.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Компьютеры не говорят ни на каком языке. Они используют двоичные коды для установки различных переключателей и для выполнения математических вычислений. Компьютеры даже не понимают букв — они воспринимают только цифры. Специальное приложение превращает компьютерный язык, используемый для написания процедуры, в двоичные коды. К счастью, вам не нужно беспокоиться об особенностях работы компьютеров на двоичном уровне. Тем не менее важно знать, что компьютеры говорят на языке математики и цифр, а не на человеческом языке.

Как наладить общение человека с компьютером

Создавая любое приложение, важно знать, для чего оно нужно. Благодаря приложениям люди могут общаться с компьютером. Каждое приложение работает с определенными типами данных, которые оно может вводить, сохранять, обрабатывать и выводить, что позволяет пользователям получать требуемые результаты. Это справедливо в отношении любых приложений, будь то игровые программы, электронные таблицы и др. Компьютер обрабатывает предоставленные человеком данные для достижения нужного результата.

Создавая приложение, вы придумываете новый способ общения человека с компьютером, давая возможность пользователям просматривать данные новыми способами. Причем интерфейс между человеком и компьютером должен быть настолько простым, чтобы наличие приложения-посредника вообще не ощущалось. Наилучшими являются приложения, в которых вы целиком фокусируетесь на данных. Например, когда речь заходит об игровых приложениях, наилучшими из них будут те, которые позволяют сосредоточиться на спасении планеты от инопланетных захватчиков или пилотировании космического корабля, а не на деталях работы самого приложения.

СОВЕТ

Чтобы лучше понять, как создаются приложения, как они должны выглядеть и функционировать, стоит изучить опыт разработчиков. Попробуйте составить список преимуществ и недостатков приложений, написанных другими людьми. В процессе изучения приложений задайте себе следующие вопросы.

- Что мне не нравится в этом приложении?
- С чем было легко работать?
- С чем было сложно работать?
- Облегчило ли приложение работу с данными?
- Можно ли было бы еще больше упростить работу с данными?
- В чем планируемые преимущества моего приложения по сравнению с данным приложением?

В процессе создания приложений профессиональные программисты задаются и другими вопросами, но для начала достаточно и этих. Благодаря им вы начнете понимать, что приложения являются средством общения человека с компьютером. Если вы когда-либо испытывали разочарование от неудобной программы, то понимаете, что другие пользователи могут испытать то же самое при работе с вашим приложением. Поэтому очень важно задать самому себе правильные вопросы. Эффективное взаимодействие — ключевой элемент любого создаваемого вами приложения.

Старайтесь продумывать порядок работы. Сформулируйте последовательность выполняемых действий. Разбейте процедуру на этапы и запишите все, что знаете о каждом этапе. Затем попросите кого-нибудь протестировать эту процедуру, чтобы понять, как она работает на самом деле. В итоге может оказаться, что, несмотря на все приложенные усилия и старания, вы все же забыли учесть какие-то важные действия.

ВНИМАНИЕ!

Провальные проекты обычно начинались с того, что разработчик не понял цель, аудиторию и особенности своего приложения. Приступая к написанию программы, подумайте о том, для чего и для кого она создается. Составьте четкий план работы, и тогда процесс создания приложения станет увлекательным. Поэтапно достигайте поставленных целей, а в конце представьте результаты своей работы друзьям, которые наверняка начнут считать вам крутым программистом.

В чем секрет популярности Python

В настоящее время существует множество языков программирования. Можно целый семестр изучать языки программирования, но так и не узнать о существовании некоторых из них. Казалось бы, этого более чем достаточно, чтобы выбрать язык, пригодный для общения с компьютером. Почему же продолжают появляться все новые и новые языки?

ЗАПОМНИ!

Новые языки программирования создаются по разным причинам. В каждом из них есть своя изюминка, характерная особенность, выделяющая его среди других языков. Более того, развитие компьютерных технологий постоянно стимулирует создание новых и совершенствование имеющихся языков программирования. Разработка любого приложения подразумевает эффективное решение поставленных задач. Программистам, владеющим многими языками, проще подобрать оптимальный язык для конкретной задачи. Одни языки лучше приспособлены для работы с базами данных, другие упрощают создание элементов пользовательского интерфейса.

Как и любой язык программирования, Python обладает целым рядом преимуществ. Давайте познакомимся с ними поближе, прежде чем начнем работу с Python. Вы даже не представляете, сколько всего интересного можно сделать с помощью Python. Знание сильных и слабых сторон языка программирования

поможет не только правильно его использовать, но и избежать разочарования, связанного с неудачной попыткой сделать то, для чего он не предназначен.

Почему именно Python

При создании всех языков программирования преследовались вполне определенные цели. Именно эти цели облегчают определение характеристик и способов использования языка. Ведь невозможно создать универсальный язык программирования, который позволял бы решать любые задачи. Что касается Python, то главная цель его разработки заключалась в создании языка, повышающего эффективность работы программистов. С учетом сказанного, рассмотрим основные причины выбора Python для создания приложений.

- **Ускорение разработки приложений.** Код на языке Python обычно в 2–10 раз короче по сравнению с аналогичным кодом, написанным на C/C++ или Java, что позволяет сократить время разработки приложения.
- **Простота чтения.** Язык программирования подобен обычному языку: нужно уметь читать тексты, написанные на нем. Код на языке Python проще читать, чем код, написанный на других языках. Это позволяет сократить время на интерпретацию кода и уделить больше времени внесению существенных изменений.
- **Быстрота изучения.** Разработчики Python ставили своей целью создание простого языка, не перегруженного правилами, чтобы не усложнять его изучение. Ведь, в конце концов, задача программистов — писать приложения, а не изучать сложные языки программирования.

Несмотря на популярность, Python все же не является самым популярным языком программирования. Например, согласно данным организации TIOBE, которая отслеживает статистику использования языков, Python в настоящий момент занимает третье место:

```
https://www.tiobe.com/tiobe-index/
```

В то же время, согласно данным сайта Tech Rapidly, Python занимает первое место по популярности:

```
http://techrapidly.com/top-10-best-programming-languages-learn-2019/
```

СОВЕТ

Если вы ищете работу программиста, то Python подойдет как нельзя лучше. Также неплохим вариантом будут Java, C/C++ и C#, в зависимости от того, чем именно вы будете заниматься. Visual Basic в настоящее время уже не так популярен, как Python. Выбирайте тот язык, который вам больше нравится и который будет максимально

эффективным для создания требуемых приложений. Интересно отметить, что во многих вузах Python используется в качестве первого языка для изучения программирования. В этом отношении он один из самых популярных языков.

Преимущества Python

По большому счету, создавать приложения можно на любом языке программирования. Даже если выбрать не самый удачный язык, вы все равно сможете решить поставленную задачу, просто работа затянется надолго и будет сопряжена с большим количеством ошибок. Разумеется, никому этого не хочется, поэтому важно знать, какие именно приложения чаще всего создаются с помощью Python. Ниже приведены наиболее популярные варианты (это далеко не полный перечень).

- **Создание прототипов приложений.** Разработчики часто создают *прототипы*, т.е. черновые варианты приложений, прежде чем приступить к написанию реальных программ. Python позволяет быстро создавать рабочие прототипы.
- **Создание сценариев для веб-приложений.** JavaScript по праву считается самым популярным языком написания веб-приложений, однако Python не уступает ему в популярности, занимая почетное второе место. Python обеспечивает доступ к большему набору функций, чем JavaScript, и позволяет ускорить разработку веб-приложений (что в наши дни является большим преимуществом).
- **Создание математических, научных и инженерных приложений.** Python предоставляет доступ к специализированным библиотекам, которые упрощают создание указанных приложений. Две наиболее популярные библиотеки — это NumPy (`www.numpy.org`) и SciPy (`www.scipy.org`). Благодаря им можно значительно сократить время на создание соответствующего кода.
- **Работа с XML.** Расширяемый язык разметки (XML) широко применяется для хранения данных в Интернете и во многих настольных приложениях. В отличие от многих других языков программирования, в Python это базовый компонент. Таким образом, Python станет оптимальным вариантом при работе с веб-службами.
- **Взаимодействие с базами данных.** Все бизнес-процессы основаны на использовании баз данных. Python не является языком запросов, как SQL или LINQ, но позволяет легко взаимодействовать с базами данных, упрощая подключения к ним.
- **Разработка пользовательских интерфейсов.** В отличие от C#, в Python нет встроенного дизайнера, позволяющего перетаскивать

элементы из панели инструментов. Тем не менее он располагает обширным набором графических фреймворков, представляющих собой расширения, которые упрощают создание графических интерфейсов. Их список доступен по следующей ссылке:

```
https://wiki.python.org/moin/GuiProgramming
```

Некоторые из фреймворков поставляются с дизайнерами, облегчающими процесс создания пользовательских интерфейсов. Важно то, что Python не привязан к какому-то одному методу создания пользовательских интерфейсов. Вы вольны выбрать любой вариант, какой вам удобен.

В каких организациях используется Python

Python отлично подходит для решения широкого круга задач. Именно поэтому он применяется во многих крупных компаниях, которые вкладывают огромные средства в его усовершенствование. В табл. 1.1 перечислены крупные организации, активно использующие Python.

Таблица 1.1. Крупные организации, использующие Python

Организация	Сайт	Тип приложения
Alice Educational Software (Университет Карнеги — Меллона)	`www.alice.org`	Обучающие приложения
Fermilab	`www.fnal.gov`	Научные приложения
Go.com	`go.com`	Веб-приложения
Google	`www.google.com`	Поисковая система
Industrial Light & Magic	`www.ilm.com`	Различные приложения
Ливерморская национальная лаборатория им. Лоуренса	`www.llnl.gov`	Научные приложения
Национальное управление по аэронавтике и исследованию космического пространства (НАСА)	`www.nasa.gov`	Научные приложения
Нью-Йоркская фондовая биржа	`nyse.nyx.com`	Веб-приложения
Red Hat	`www.redhat.com`	Инструменты установки Linux

Окончание табл. 1.1

Организация	Сайт	Тип приложения
Yahoo!	`www.yahoo.com`	Компоненты почтовой системы Yahoo!
YouTube	`www.youtube.com`	Графический движок
Zope Community	`http://www.zope.org/en/latest/`	Издательское приложение

СОВЕТ

Это лишь небольшой перечень компаний, широко применяющих язык Python. Полный список таких компаний доступен по адресу `http://www.python.org/about/success/`. Вы легко убедитесь в том, насколько этот список огромен.

Где найти полезные приложения Python

Вы можете даже не догадываться о том, что приложение, с которым вы сейчас работаете, написано на Python. Множество современных приложений написано на этом языке, от консольных утилит до полноценных САПР-программ. Одни приложения выполняются на мобильных устройствах, другие — на корпоративных платформах. По сути, не существует таких задач, которые нельзя было бы решить на Python, и вы можете воспользоваться плодами трудов разработчиков приложений. Существует немало сайтов, на которых публикуются списки приложений, созданных на Python. Один из таких списков доступен по следующей ссылке:

`https://wiki.python.org/moin/Applications`

Любой программист, работающий на Python, знает, насколько облегчают жизнь инструменты разработки, позволяющие достичь определенного уровня автоматизации в процессе написания процедур. Чем больше у разработчика инструментов, тем меньше времени у него уходит на создание приложения. В Интернете часто публикуют списки наиболее полезных инструментов. Вот один из примеров:

`http://www.python.org/about/apps/`

ЗАПОМНИ!

В этой главе уже упоминались некоторые инструменты, например NumPy и SciPy (две научные библиотеки). В остальных главах будут описаны и другие важные инструменты. Подготовьте для себя перечень инструментов, которые могут пригодиться в работе.

Python и другие языки программирования

Некорректно сравнивать разные языки программирования, поскольку выбор языка — это дело вкуса и личных предпочтений каждого программиста. Прежде чем подвергнуться нападкам со стороны приверженцев рассмотренных ниже языков, отмечу, что в своей практике я использую разные языки программирования и нахожу, что многие из них в какой-то степени пересекаются. Самого лучшего языка программирования в мире не существует, есть лишь язык, который лучше подходит для решения той или иной задачи. Ниже вашему вниманию предлагается сравнение некоторых языков программирования и Python. Сравнение с другими языками доступно по следующей ссылке:

`https://wiki.python.org/moin/LanguageComparisons`

C#

Многие пользователи обвиняют компанию Microsoft в том, что она создала C# полностью на основе Java. Тем не менее по сравнению с Java язык C# имеет определенные преимущества (и недостатки тоже). Основной и бесспорной целью создания C# является получение усовершенствованной версии языка C/C++ , которая была бы проще в использовании и изучении. Но не будем отвлекаться от нашей главной цели. По сравнению с C# язык Python имеет следующие преимущества:

- гораздо проще в изучении;
- имеет более компактный код;
- полностью поддерживается в качестве языка с открытым исходным кодом;
- имеет более надежную многоплатформенную поддержку;
- обеспечивает возможность использования многочисленных средств разработки;
- проще в расширении при использовании Java и C/C++;
- имеет улучшенную научную и инженерную поддержку.

Java

В течение многих лет программисты пытались создать такой язык программирования, на котором можно было бы один раз написать приложение и потом везде его использовать. К достоинствам Java можно отнести то, что он хорошо работает на любой платформе. Java настолько успешно применяется в любых сферах, что другие языки даже пытаются его эмулировать (с разной степенью успешности). Тем не менее Python имеет ряд преимуществ перед Java, а именно:

- гораздо проще в изучении;
- более компактный код;
- усовершенствованные переменные (блоки памяти), позволяющие хранить различные типы данных в зависимости от текущих потребностей приложения (динамический контроль типов);
- более быстрая разработка.

Perl

Изначально название языка Perl представляло собой сокращение от "Practical Extraction and Report Language" (практический язык для извлечения данных и составления отчетов). Сейчас его называют просто Perl. Основное назначение этого языка сводится к извлечению информации из баз данных и ее выводу в формате отчета. Но и помимо этого спектр применения Perl достаточно широк. Мне даже доводилось использовать его для создания веб-приложений. Вот основные преимущества Python перед Perl:

- проще в изучении;
- легче в чтении;
- улучшенная защита данных;
- более высокий уровень интеграции с Java;
- меньше проблем, связанных с платформенной зависимостью.

R

Специалистам по обработке данных довольно сложно выбирать между языками R и Python, поскольку оба они предназначены для статистического анализа и построения научных диаграмм. Оба языка имеют открытый исходный код и поддерживают широкий спектр платформ. Тем не менее язык R более специфичен по сравнению с Python в том плане, что ориентирован на разработку научных приложений. Преимущества Python по сравнению с R таковы:

- повышение производительности и разборчивости кода;
- ориентированность на корпоративное применение;
- простота отладки;
- использование более совершенных методов кодирования;
- большая степень гибкости;
- проще в изучении.

Глава 2

Установка Python

В ЭТОЙ ГЛАВЕ...

- Как получить копию Python для своей системы
- Установка Python
- Поиск и использование копии Python в своей системе
- Проверка корректности установки Python

Если только вы не собираетесь писать программы на низком уровне (в машинных кодах), что довольно сложно даже для опытных программистов, то учтите, что для создания приложений Python вам понадобятся другие приложения. Они помогают работать с Python, создавая программный код, предоставляя справочную информацию по мере необходимости и позволяя запускать на выполнение написанный вами код. Прочитав эту главу, вы сможете получить копию Python, установить ее на свой жесткий диск, найти установленные приложения, которые можно использовать, и протестировать установленную копию, чтобы увидеть, как она работает.

Загрузка нужной версии

Для каждой *платформы*, представляющей собой сочетание аппаратных и программных средств, существуют специальные правила, которым нужно следовать при выполнении приложений. Python скрывает соответствующие детали от пользователя. Вы просто вводите код, который выполняется на любой поддерживаемой платформе, и Python преобразует этот код во что-то, что может быть запущено на текущей платформе. Но для того, чтобы подобное преобразование было успешным, нужно располагать версией Python, которая работает в вашей конкретной системе.

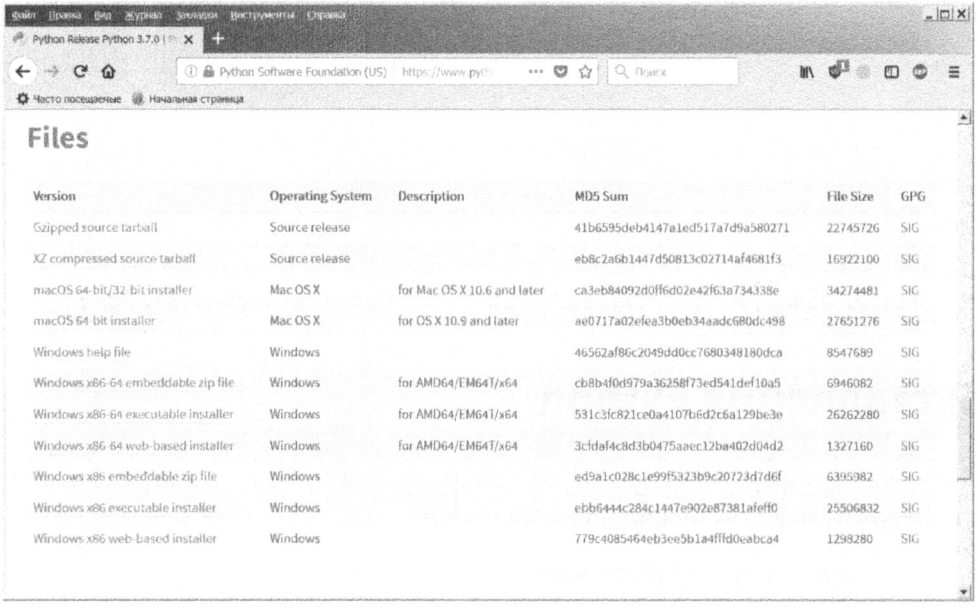
Ниже перечислены платформы, поддерживаемые в Python (хотя это, вероятно, далеко не полный перечень):

- AIX (Advanced IBM Unix);
- Android;
- BeOS;
- Berkeley Software Distribution (BSD)/FreeBSD;
- HP-UX (Hewlett-Packard Unix);
- IBM i (ранее AS/400, iSeries и System i);
- iOS (iPhone Operating System);
- Linux;
- Mac OS X (Python предустановлен в этой системе);
- MS-DOS (Microsoft Disk Operating System);
- MorphOS;
- OS/2 (Operating System 2);
- OS/390 и z/OS;
- PalmOS;
- PlayStation;
- Psion;
- QNX;
- RISC OS (изначально Acorn);
- Series 60;
- Solaris;
- VMS (Virtual Memory System);
- 32-разрядные версии Windows (XP и более поздние);
- 64-разрядные версии Windows;
- Windows CE/Pocket PC.

СОВЕТ

Как видите, количество поддерживаемых платформ просто поражает воображение! Примеры, рассмотренные в этой книге, были проверены в среде Windows, Mac OS X и Linux. Но они могут хорошо работать и в других системах, поскольку не используют код, специфичный для платформы. Если по какой-то причине пример не работает в вашей системе, отправьте мне письмо по адресу `John@JohnMuellerBooks.com`, описав суть проблемы. На момент выхода книги текущая версия Python — 3.7.0.

Чтобы получить версию Python, подходящую для вашей платформы, перейдите по адресу `https://www.python.org/downloads/release/python-370/`. Раздел загрузки обычно скрыт, поэтому нужно выполнить прокрутку, чтобы получить к нему доступ. После этого на экране появится страница, подобная показанной на рис. 2.1. Основной раздел этой страницы содержит ссылки на загрузочные файлы для Windows, Mac OS X и Linux. Эти ссылки позволяют выполнить установку по умолчанию, используемую в книге. В левой части страницы находятся ссылки для разных платформ, отображающие альтернативные конфигурации Python, которые можно использовать в случае необходимости. Например, если вам понадобится редактор с расширенными функциями, отличающийся от редактора, предлагаемого стандартным пакетом Python, можете воспользоваться одной из альтернативных конфигураций.



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		41b6595deb4147a1ed517a7d9a580271	22745726	SIG
XZ compressed source tarball	Source release		eb8c2a6b1447d50813c02714af4681f3	16922100	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	ca3eb84092d0ff6d02e42f63a734338e	34274481	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	ae0717a02efea3b0eb34aadc680dc498	27651276	SIG
Windows help file	Windows		46562af86c2049dd0cc7680348180dca	8547689	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	cb8b4f0d979a36258f73ed541def10a5	6946082	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	531c3fc821ce0a4107b6d2c6a129be3e	26262280	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	3cfdaf4c8d3b0475aaec12ba402d04d2	1327160	SIG
Windows x86 embeddable zip file	Windows		ed9a1c028c1e99f5323b9c20723d7d6f	6395982	SIG
Windows x86 executable installer	Windows		ebb6444c284c1447e902e87381afeff0	25506832	SIG
Windows x86 web-based installer	Windows		779c4085464eb3ee5b1a4fffd0eabca4	1298280	SIG

Рис. 2.1. На странице загрузки Python приведены ссылки для всех версий

Если вы используете другую платформу, перейдите по адресу `https://www.python.org/download/other/`. На экране появится список файлов установки Python для других систем (рис. 2.2). Многие из этих файлов поддерживаются волонтерами, а не разработчиками, создающими версии Python для Windows, Mac OS X и Linux. Обращайтесь к ним, если у вас возникли вопросы по установке, поскольку они лучше знают, каким образом установить Python в вашей системе.

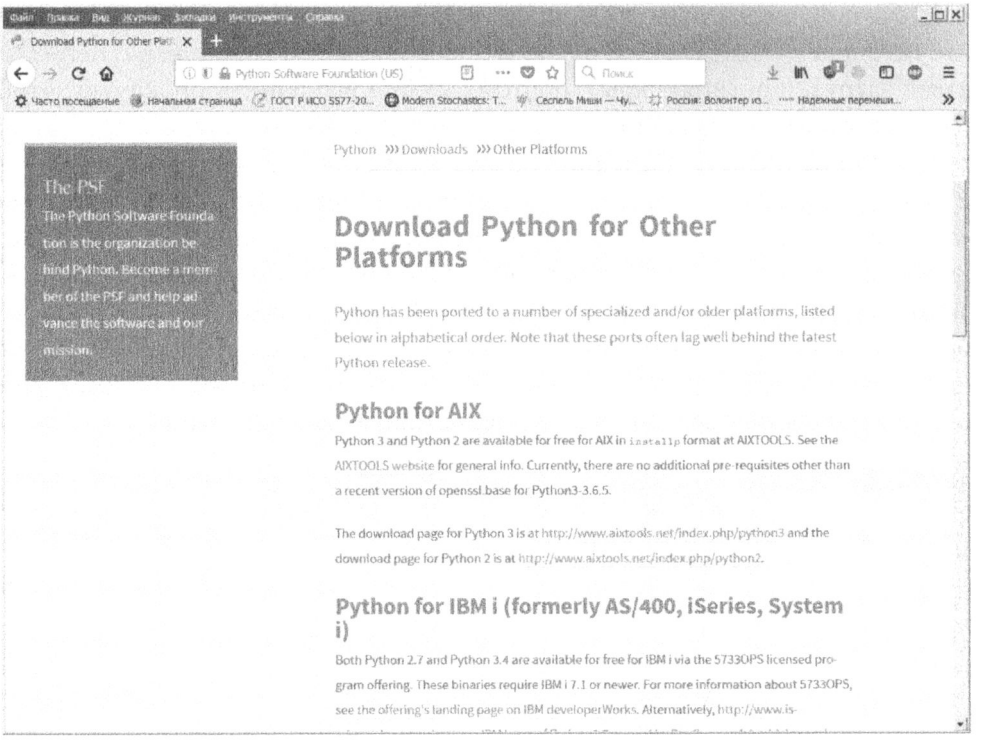


Рис. 2.2. Благодаря волонтерам Python доступен для любых платформ

Инсталляция Python

После загрузки Python нужно установить файлы в системе. Загрузочный пакет содержит все, что нужно для установки:

- интерпретатор Python;
- файлы справки (документация);
- утилиты командной строки;
- интегрированная среда разработки (Integrated DeveLopment Environment — IDLE);
- программа установки (Preferred Installer Program — pip);
- деинсталлятор (только для платформ, в которых он требуется).

В книге предполагается, что вы используете файлы установки Python, доступные по адресу `https://www.python.org/downloads/release/python-370/`. Если же вы работаете с версией, которая отличается от 3.7.0, то некоторые примеры могут оказаться неработоспособными. В следующих

разделах описывается установка Python на трех основных платформах: Windows, Mac OS X и Linux.

В Windows

Процесс установки Python в Windows осуществляется традиционным образом. Главное — найти загрузочный файл, с которого начинается процесс установки. Рассмотренная ниже процедура подходит для любой версии Windows, независимо от того, что вы устанавливаете: 32- или 64-разрядную версию Python.

1. **Найдите загрузочный файл Python.**

   Имя этого файла обычно выглядит как `python-3.7.0.exe` для 32-разрядных систем и `python-3.7.0-amd64.exe` для 64-разрядных систем. Номер версии является частью имени файла. В данном случае имя файла указывает на версию 3.7.0, которая используется в книге.

2. **Дважды щелкните на файле установки.**

   Возможно, на экране появится диалоговое окно Open File — Security Warning (Открыть файл — предупреждение системы безопасности) с вопросом, хотите ли вы запустить файл на выполнение. Щелкните на кнопке Run (Выполнить), если увидите такое окно. После этого на экране появится диалоговое окно Python Setup (Установка Python), подобное показанному на рис. 2.3. Вид этого окна зависит от версии Python.

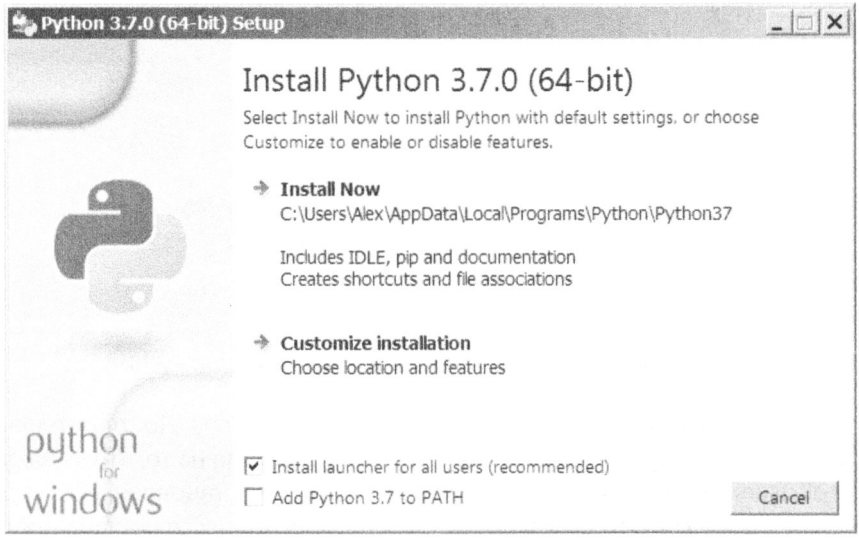


Рис. 2.3. Процесс установки начинается с выбора типа инсталляции

3. **Выберите вариант установки. В книге используется настройка, заданная по умолчанию: Install launcher for all users (Установить для всех пользователей).**

   Благодаря персонализированной установке облегчается инсталляция приложения в системе с несколькими пользователями. Иногда это также уменьшает количество появляющихся предупреждений системы безопасности.

4. **Установите флажок Add Python 3.7 to PATH (Добавить Python 3.7 в переменную PATH).**

   Эта настройка обеспечивает доступ к Python с любого места на жестком диске. В противном случае придется добавлять путь к установочному каталогу Python вручную.

5. **Щелкните на ссылке Customize Installation (Настроить установку).**

   В процессе установки можно будет выбрать, какие функции должны поддерживаться в Python, как показано на рис. 2.4. Установите все флажки, пусть даже не все они вам сейчас понятны.

6. **Щелкните на кнопке Next (Далее).**

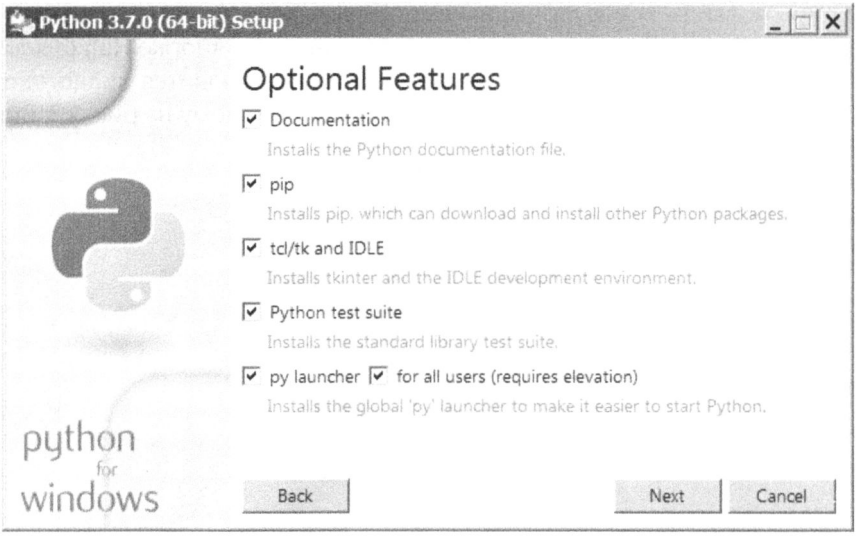


Рис. 2.4. Выберите устанавливаемые функции Python

На экране появится диалоговое окно Advanced Options (Дополнительные параметры), показанное на рис. 2.5. Обратите внимание на то, что флажок Install for all users (Установить для всех пользователей) не установлен, несмотря на то что эта возможность была запрошена раньше. В процессе инсталляции нужно также указать путь к каталогу, в котором будет установлен Python. Проще использовать каталог, заданный по умолчанию, чтобы сэкономить время и силы (хотя можно установить Python в любой каталог).

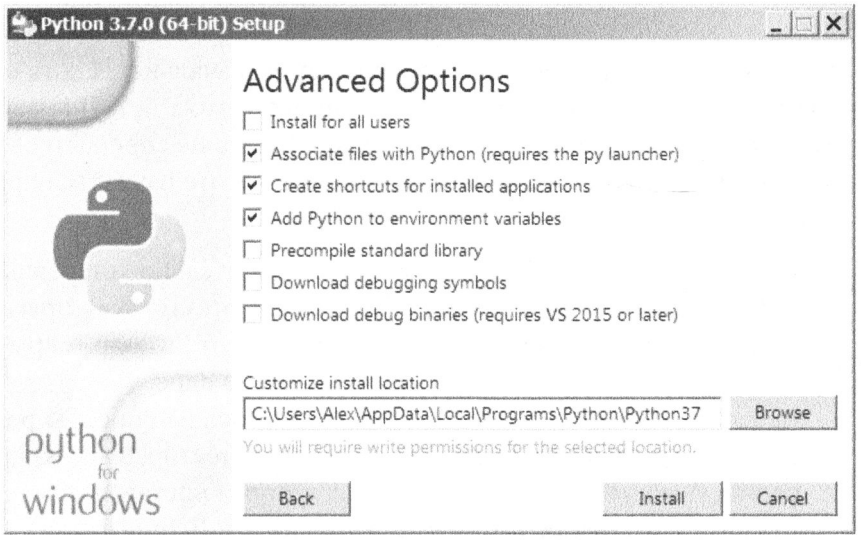


Рис. 2.5. Выбор места для установки Python

ВНИМАНИЕ!

Использование папки `\Program Files` или `\Program Files (x86)` может вызвать проблемы в силу двух причин. Во-первых, имя папки содержит пробел, что затрудняет доступ к ней из некоторых приложений. Во-вторых, для доступа к этим папкам обычно требуются права администратора, поэтому после установки Python в одну из этих папок вам придется постоянно иметь дело со всплывающими окнами системы контроля учетных записей Windows.

7. **Чтобы сделать Python доступным для всех пользователей, установите флажок Install for all users (Установить для всех пользователей).**

   Учтите, что выбор этого параметра приводит к автоматической установке флажка Precompile standard library (Предварительная компиляция стандартной библиотеки), который нужно оставить.

8. **При необходимости введите название целевой папки.**

   В этой книге используется каталог установки `Python37`.

9. **Щелкните на кнопке Install (Установить).**

   После этого начнется процесс установки. В какой-то момент может появиться окно системы контроля учетных записей с запросом о продолжении установки. В таком случае щелкните на кнопке Yes (Да). Процесс установки продолжится, а после его завершения появится диалоговое окно Installation Complete (Установка завершена).

10. **Щелкните на кнопке Close (Закрыть).**

    Теперь можно начинать работать с Python.

В Mac

Как правило, в этой системе Python установлен изначально. И хотя обычно его версия не самая последняя, ничего страшного в этом нет. Для работы с примерами книги последней версии и не нужно. Вы же не собираетесь тестировать все новейшие средства и функции, а просто хотите научиться программировать на Python.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В OS X Leopard (10.5) используется действительно устаревшая версия Python под номером 2.5.1, в которой отсутствует прямой доступ к приложению IDLE. В результате некоторые примеры книги будут выполняться некорректно. В статье по адресу `https://wiki.python.org/moin/MacPython/Leopard` описаны способы решения данной проблемы. Примеры книги были протестированы на предмет совместимости с OS X версии 10.12, которая поставляется вместе с Python 2.7.10. Более поздние версии OS X и Python также должны работать нормально, хотя и могут появляться предупреждения об использовании некорректной библиотеки или о других потенциальных проблемах совместимости.

Скорее всего, в какой-то момент вам потребуется обновить версию Python, в частности, установить утилиты GCC (GNU Compiler Collection), обеспечивающие доступ к низкоуровневым ресурсам. Чтобы установить новую версию Python в Mac OS X 10.6 или выше, выполните следующие действия.

1. **В адресной строке браузера введите ссылку `https://www.python.org/downloads/release/python-370/`.**

   На экране отобразится информация, относящаяся к последней версии Python (см. рис. 2.1).

2. **Щелкните на ссылке `Mac OS X 64-bit/32-bit installer`.**

   После этого начнет загружаться образ диска. Наберитесь терпения, так как загрузка может занять несколько минут. Большинство браузеров показывает ход загрузки, поэтому вы сможете видеть, сколько времени осталось до завершения загрузки.

3. **Дважды щелкните на файле `python-3.7.0-macosx10.6.pkg`, находящемся в папке загрузки.**

   На экране появится диалоговое окно приветствия, в котором отображаются сведения о данной версии Python.

4. **Трижды щелкните на кнопке Continue (Продолжить).**

   Программа установки отобразит последние новости о Python, сведения о лицензии, которую нужно будет принять, щелкнув на кнопке Agree (Согласиться), и, наконец, финальное диалоговое окно.

5. **Щелкните на кнопке Install (Установить).**

   Программа установки может запросить ваш пароль администратора. Введите его и щелкните на кнопке OK. После этого отобразится диалоговое окно Installing Python (Установка Python). В процессе установки содержимое этого окна изменяется, благодаря чему вы будете знать, какая именно часть программы установки выполняется в данный момент.

   После завершения установки появится диалоговое окно Install Succeeded (Установка успешно завершена).

6. **Щелкните на кнопке Close (Закрыть).**

   Теперь ваша копия Python готова к применению. (Можете закрыть образ диска и удалить его из системы.)

В Linux

Некоторые версии Linux поставляются с предустановленной копией Python. Например, если вы располагаете дистрибутивом, основанным на RPM (Red Hat Package Manager), таким как SUSE, Red Hat, Yellow Dog, Fedora Core или CentOS, то Python у вас уже есть, и ничего делать не нужно.

В разных версиях Linux будут установлены разные версии Python, и в некоторых системах может не оказаться приложения IDLE (Interactive DeveLopment Environment). Если у вас старая версия Python (2.5.1 или еще более ранняя), установите версию поновее, где есть IDLE. Многие из примеров, рассматриваемых в книге, рассчитаны на использование именно IDLE.

Python можно установить в Linux двумя способами, которые будут рассмотрены в следующих разделах. Первая методика подходит для любого дистрибутива Linux, тогда как вторая требует выполнения определенных условий.

Стандартная установка

Стандартная установка может применяться в любой системе. Но вам придется работать в окне терминала и вводить команды вручную. Причем некоторые команды могут отличаться в зависимости от версии Linux. На сайте `http://docs.python.org/3/install/` вы найдете ряд полезных советов, которыми сможете воспользоваться при выполнении описанных ниже действий.

1. **В адресной строке браузера введите `https://www.python.org/downloads/release/python-370/`.**

   Отобразится окно с информацией о последней версии Python (см. рис. 2.1).

2. **Щелкните на ссылке, соответствующей вашей версии Linux:**
   - `Gzipped source tarball` (подойдет для любой версии Linux);
   - `XZ compressed source tarball` (лучшая степень сжатия архива и меньшее время загрузки).
3. **Когда появится окно с вопросом о том, что вы хотите сделать — открыть или сохранить файл, — щелкните на кнопке Сохранить.**

   После этого начнут загружаться файлы исходного кода Python. Будьте терпеливы: процесс загрузки файлов может занять несколько минут.
4. **Дважды щелкните на загруженном файле.**

   Откроется окно менеджера архивов. Когда файлы будут извлечены из архива, в окне менеджера появится папка `Python 3.7.0`.
5. **Дважды щелкните на папке `Python 3.7.0`.**

   Менеджер архивов извлечет файлы в подпапку `Python 3.7.0` домашней папки.
6. **Откройте приложение Terminal.**

   На экране появится окно терминала. Если вам не приходилось ранее выполнять сборку программных пакетов, то сначала нужно установить необходимые компоненты сборки, а также утилиты SQLite и `bzip2`, в противном случае установка Python завершится неудачей. Если же соответствующие приложения уже установлены, переходите к п. 10, чтобы незамедлительно начать работать с Python.
7. **Введите команду `sudo apt-get install build-essential`.**

   Linux установит поддержку Build Essential, требуемую для сборки пакетов.
8. **Введите команду `sudo apt-get install libsqlite3-dev`.**

   Linux установит поддержку SQLite, необходимую для работы с базами данных.
9. **Введите команду `sudo apt-get install libbz2-dev`.**

   Linux установит поддержку `bzip2`, необходимую для выполнения операций с архивами.
10. **В окне терминала введите команду `CD Python 3.7.0`.**

    Текущим каталогом станет папка `Python 3.7.0`.
11. **Введите команду `./configure`.**

    Сценарий проверит тип сборки, после чего выполнит ряд действий на основе используемой вами системы. Этот процесс может занять несколько минут.
12. **Введите команду `make`.**

    Linux выполнит сценарий `make`, который создает среду Python. Выполнение сценария занимает около минуты в зависимости от быстродействия системы.
13. **Введите команду `sudo make altinstall`.**

Система может запросить ввод пароля администратора. Введите пароль и нажмите клавишу <Enter>. На этом установка Python в вашей системе завершается.

Графическая установка

Все версии Linux поддерживают стандартную установку, которая была рассмотрена в предыдущем разделе. Также есть несколько дистрибутивов Linux, основанных на Debian, например Ubuntu 12.*x* и более поздние, которые поддерживают графическую установку. Для этого требуется знать пароль администратора (команда `sudo`), поэтому держите его под рукой. Ниже приведена пошаговая инструкция графической установки в Ubuntu; в других версиях Linux процедура будет аналогичной.

1. **Откройте папку `Ubuntu Software Center`. (На других платформах она может называться `Synaptics`.)**

   На экране появится список наиболее популярных программ, доступных для загрузки и установки.

2. **В раскрывающемся списке All Software (Все программы) выберите пункт Developer Tools (Инструменты разработчика) или Development (Разработка).**

   Отобразится список инструментов разработчика, включая Python.

3. **Дважды щелкните на записи `Python 3.7.0`.**

   Программа Ubuntu Software Center отобразит подробные сведения о версии Python 3.7.0 и предложит установить ее.

4. **Щелкните на кнопке Install (Установить).**

   Начнется процесс установки Python и появится индикатор хода выполнения. По завершении установки кнопка Install превратится в кнопку Remove (Удалить).

5. **Закройте папку `Ubuntu Software Center`.**

   Значок Python появится на рабочем столе. Python готов к использованию!

Получение доступа к копии Python на своем компьютере

После завершения установки Python нужно понять, где его можно найти. В определенном смысле Python делает все возможное, чтобы облегчить этот процесс, например добавляет каталог установки в переменную среды `PATH`. И тем не менее полезно знать, где конкретно установлен Python.

НЕСКОЛЬКО СЛОВ ОБ ЭКРАННЫХ СНИМКАХ В КНИГЕ

В первых примерах книги вы будете работать с Python, используя интегрированную среду разработки (IDLE) или командную строку. Затем вы начнете использовать Anaconda, поскольку эта графическая среда существенно упрощает работу с Python. Обе среды, IDLE и Anaconda, называются одинаково на всех трех платформах, да и внешние различия практически отсутствуют. Различия между платформами несущественны, поэтому в процессе чтения книги их можно игнорировать. Учитывая этот факт, все экранные снимки для книги были сделаны в Windows, чтобы все выглядело согласованно.

Командная строка также работает практически идентично на всех трех платформах. Ее внешний вид может отличаться чуть сильнее, чем в случае IDLE или Anaconda, поскольку на разных платформах используются разные командные оболочки. Но сами вводимые команды будут одними и теми же. Также идентичным будет результат выполнения команд на разных платформах. Поэтому при просмотре экранных снимков обращайте внимание на результаты работы программы, а не на различия в интерфейсе.

В Windows

В процессе установки Python в Windows создается новая папка в меню Пуск, с помощью которой обеспечивается доступ к установленным компонентам Python. Чтобы перейти в эту папку, выберите Пуск⇨Все программы⇨Python 3.7. Нас больше всего интересуют пункты IDLE (графическая среда Python) и Python (командная строка). В главе 4 мы поговорим о том, как использовать графическую оболочку Anaconda для создания первого реального приложения, но пока мы будем работать в IDLE и командной строке Python.

После щелчка на пункте IDLE появится окно интерактивной графической среды, подобное показанному на рис. 2.6. IDLE автоматически отображает определенную информацию, свидетельствующую о том, что вы открыли нужное приложение. В частности, программа сообщает номер используемой версии Python (в данном случае 3.7.0). Также выводятся сведения о системе, в которой выполняется Python.

При выборе пункта меню Python (командная строка) откроется окно командной строки и будет выполнена команда `Python` (рис. 2.7). И снова программная среда автоматически отображает определенную информацию, такую как номер версии Python и сведения о платформе.

Третий способ обращения к Python заключается в том, чтобы открыть окно командной строки, ввести **`Python`** и нажать клавишу <Enter>. Это может потребоваться в случае, когда нужно запустить Python в среде с более высокими

привилегиями (где вы получаете дополнительные права доступа). Python поддерживает большое количество параметров командной строки, для просмотра которых нужно ввести **`Python /?`**. На рис. 2.8 показан типичный результат. Пока что особо не переживайте о параметрах командной строки, поскольку при работе с примерами книги они вам не понадобятся; просто полезно знать об их наличии.

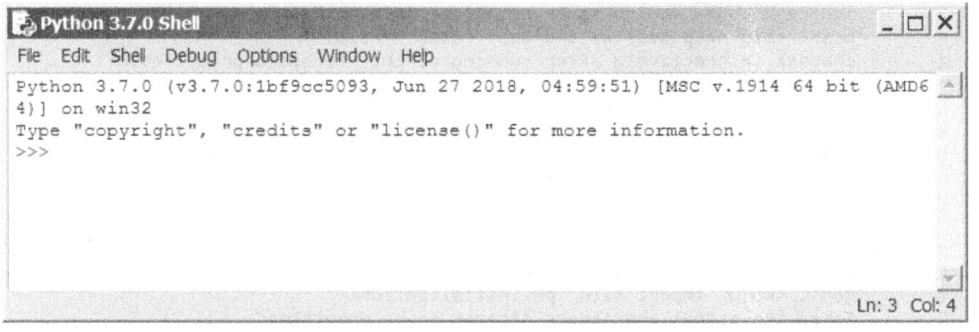


Рис. 2.6. Благодаря IDLE можно работать в удобной графической среде

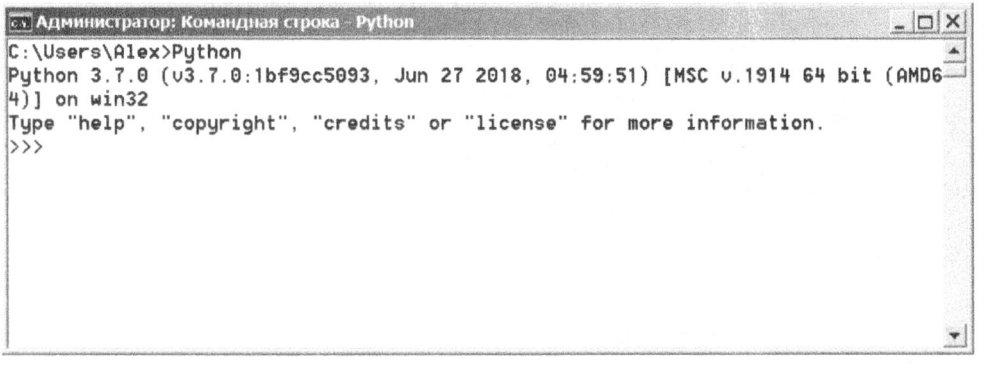


Рис. 2.7. Используйте командную строку для быстрого выполнения требуемых команд

Чтобы третий способ обращения к Python стал доступен, необходимо добавить каталог установки Python в системный путь Windows. Для этого достаточно установить флажок Add Python 3.7 to PATH (Добавить Python 3.7 в переменную PATH) в процессе инсталляции Python. Если же это не было сделано, воспользуйтесь инструкциями, приведенными в моей статье "Adding a Location to the Windows Path", которая опубликована по следующему адресу:

```
http://blog.johnmuellerbooks.com/2014/02/17/
adding-a-location-to-the-windows-path/
```

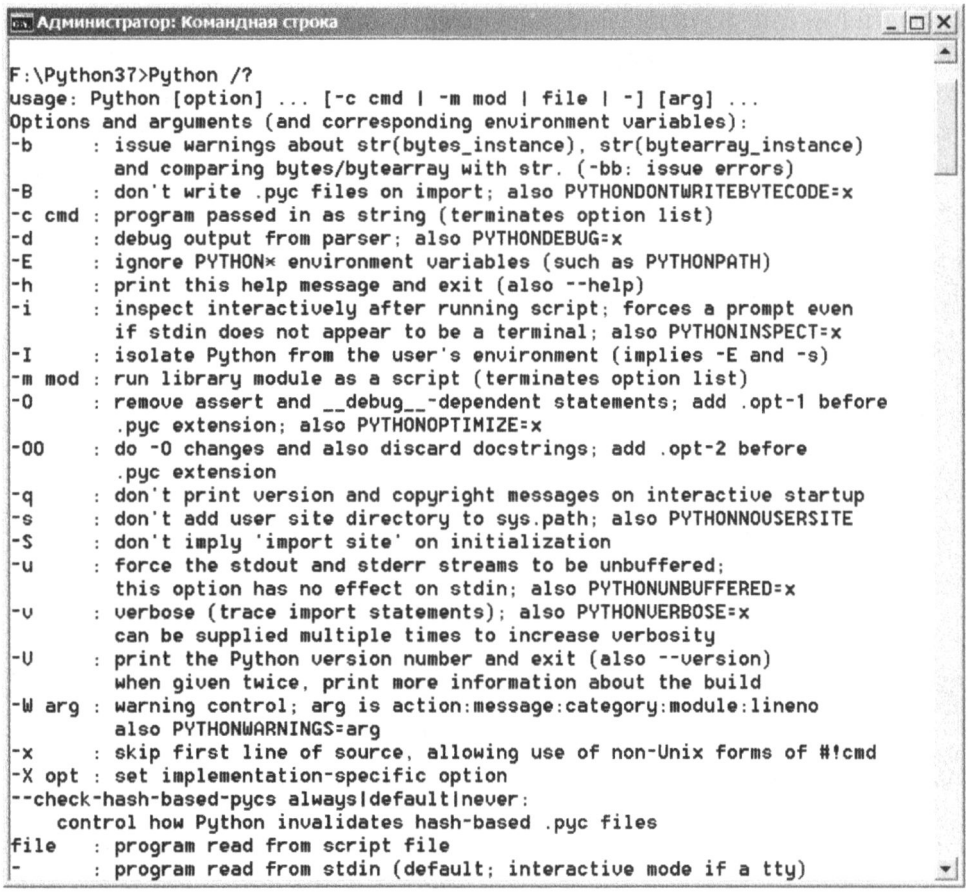


Рис. 2.8. Python можно гибко настраивать с помощью параметров командной строки

Аналогичная методика применяется для добавления переменных среды, специфичных для Python, а именно:

- PYTHONSTARTUP;
- PYTHONPATH;
- PYTHONHOME;
- PYTHONCASEOK;
- PYTHONIOENCODING;
- PYTHONFAULTHANDLER;
- PYTHONHASHSEED.

Ни одна из этих переменных не будет использоваться в примерах книги. Дополнительные сведения о них можно найти по следующему адресу:

`https://docs.python.org/3.7/using/cmdline.html#environment-variables`

В Mac

Работая в Mac, учтите, что Python, скорее всего, уже установлен и вам не нужно устанавливать его повторно. Но в любом случае полезно знать, где он находится. В следующих разделах рассказывается, как можно получить доступ к Python в зависимости от типа выполненной установки.

Поиск установки, заданной по умолчанию

В большинстве случаев стандартная установка OS X не включает папку для Python. Поэтому нужно открыть окно терминала с помощью команды Applications⇨Utilities⇨Terminal, ввести **`Python`** и нажать клавишу <Enter> для получения доступа к командной строке Python. Результат будет напоминать рис. 2.7. Как и в Windows, командная строка поддерживает целый ряд параметров, с помощью которых можно настраивать способ работы Python.

Поиск обновленной версии Python, которую вы установили

Если вы устанавливали Python в Mac, откройте папку `Applications`. В ней находится папка `Python 3.7`, в которой содержится следующее:

- папка `Extras`;
- приложение IDLE (графическая среда разработки);
- Python Launcher (интерактивная командная среда);
- команда Update Sh.

После двойного щелчка на значке IDLE запустится графическая среда, напоминающая ту, которая была показана на рис. 2.6. Разумеется, будут небольшие косметические отличия, но содержимое окна останется таким же. После двойного щелчка на значке Python Launcher откроется окно командной строки, подобное показанному на рис. 2.7. В этой среде используются все заданные по умолчанию настройки Python, обеспечивающие стандартную среду выполнения.

ЗАПОМНИ!

Даже если вы устанавливаете новую версию Python в Mac, вы не обязаны использовать среду, заданную по умолчанию. По-прежнему можно открыть окно терминала, чтобы получить доступ к параметрам командной строки Python. Только следует убедиться в том, что вы не обращаетесь к установке, заданной по умолчанию. Не забудьте добавить каталог `/usr/local/bin/Python3.7` в системный путь.

В Linux

После завершения процесса установки в вашей домашней папке появится подпапка `Python 3.7`. Физически Python 3.7 в Linux обычно находится в папке `/usr/local/bin/Python3.7`. Это важно, поскольку может понадобиться

поменять путь вручную. В Linux при работе в окне терминала, чтобы получить доступ к установке Python 3.7, нужно ввести **`Python3.7`**, а не просто **`Python`**.

Тестирование установки

Чтобы убедиться в наличии рабочей среды, нужно ее протестировать. По сути, процесс тестирования означает написание первого приложения Python. Начните с открытия IDLE. Как уже упоминалось, в окне IDLE автоматически отображаются версия Python и сведения о системе.

Чтобы проверить работоспособность Python, введите **`print("Моя первая программа на языке Python.")`**. Python выдаст только что введенное сообщение, как показано на рис. 2.9. Команда `print()` отобразит на экране все, что вы ей скажете; она является одной из наиболее часто используемых.

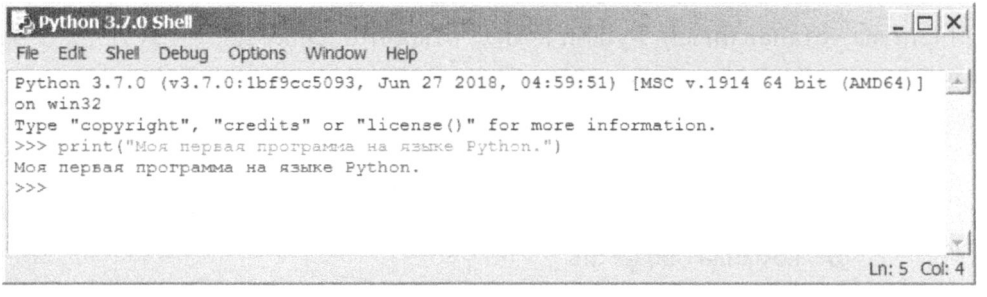


Рис. 2.9. Команда `print()` отображает любую информацию, которую нужно вывести на экран

В IDLE применяется цветовая разметка кода, чтобы его было легче понимать. По цветовой разметке можно сразу же определить, все ли вы сделали правильно. Чаще всего вам встретятся четыре цвета.

- **Фиолетовый.** Указывает на то, что вы ввели команду.
- **Зеленый.** Соответствует аргументам команды.
- **Синий.** Указывает на вывод команды.
- **Черный.** Код, не относящийся к команде.

Итак, вы проверили работоспособность Python, отправив ему команду и получив на нее ответ. А теперь попробуем ввести еще одну команду. Введите **`3 + 4`** и нажмите клавишу <Enter>. Python ответит, отобразив значение `7`, как показано на рис. 2.10. Обратите внимание на то, что выражение `3 + 4` будет выделено черным цветом, так как это не команда. В то же время цифра `7` выделяется синим цветом, поскольку это результат выполнения команды.

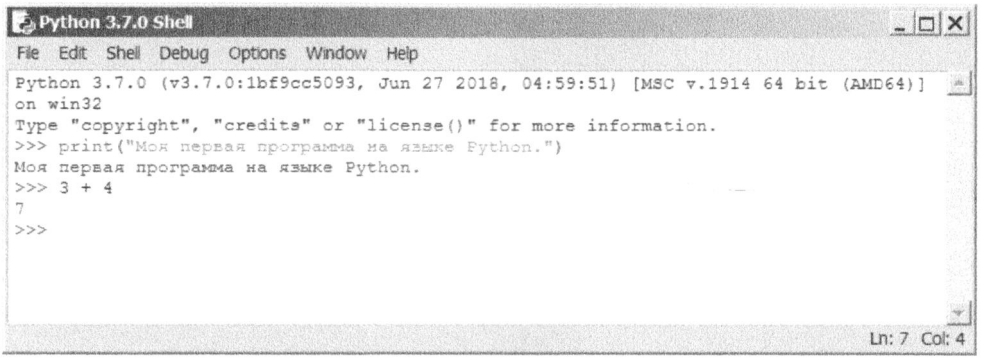


Рис. 2.10. Python позволяет выполнять математические вычисления в интерактивной среде

Ну а теперь пришло время завершить сеанс IDLE. Введите **`quit()`** и нажмите клавишу <Enter>. Появится сообщение, показанное на рис. 2.11. Щелкните на кнопке OK, что приведет к завершению сеанса.

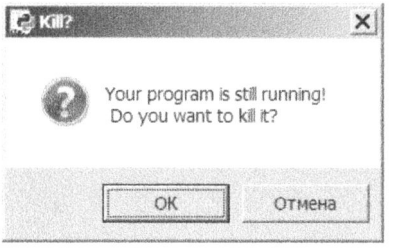


Рис. 2.11. Предупреждение при попытке завершить сеанс

Заметьте, что команда `quit()` снабжена круглыми скобками, как и команда `print()`. В Python все команды дополняются круглыми скобками, это их отличительный признак. Если вы не передаете команде никаких параметров, то между скобками ничего вводить не нужно.

Глава 3

Взаимодействие с Python

В ЭТОЙ ГЛАВЕ...

- Доступ к командной строке
- Использование команд для выполнения задач
- Получение помощи
- Завершение сеанса командной строки

В конечном счете любое приложение, созданное пользователем, взаимодействует с компьютером и содержащимися в нем данными. Основную роль здесь играют данные, поскольку без них нет смысла использовать приложения. Любое приложение, даже такое простое, как пасьянс Косынка, каким-то образом манипулирует данными. То, что делает большинство приложений, хорошо описывается акронимом CRUD:

- Create (Создание);
- Read (Чтение);
- Update (Обновление);
- Delete (Удаление).

Просто запомните акроним CRUD, и вы будете знать, как работает с данными большинство приложений. Но прежде чем приложение сможет общаться с компьютером, нужно составить для него список задач на языке программирования, понятном компьютеру. Об этом и пойдет речь в данной главе. Мы начнем общаться с Python, передавая ему команды, которые он будет превращать в инструкции, воспринимаемые компьютером.

Открытие командной строки

Пользователи могут взаимодействовать с Python несколькими способами. Например, вы уже успели немного поработать в графической среде IDLE (Integrated DeveLopment Environment), которая описывалась в главе 2. (В главе 4 вы познакомитесь с полнофункциональной интегрированной средой разработки под названием Anaconda.) Благодаря IDLE упрощается разработка сложных приложений. Но зачастую нужно просто немного поэкспериментировать или запустить существующее приложение. В таком случае проще воспользоваться командной строкой Python, поскольку она обеспечивает лучший контроль над средой выполнения с помощью различных параметров командной строки и использует меньше ресурсов. Благодаря предельно простому интерфейсу можно сосредоточиться на выполнении кода и не тратить время на разбирательство с графическим интерфейсом.

Запуск Python

Командную строку можно запускать по-разному, в зависимости от платформы. Вот наиболее распространенные варианты.

- Перейдите в папку `Python37` и выберите элемент Python (Командная строка). Это приведет к запуску сеанса командной строки с использованием настроек, заданных по умолчанию.
- Откройте командную строку или окно терминала и введите **`Python`**. Используйте этот способ в том случае, когда хотите получить возможность настраивать среду Python с помощью параметров командной строки.
- Найдите папку Python, которая может называться `C:\Python37` в Windows, и непосредственно запустите файл `Python.exe`. В результате откроется сеанс командной строки, использующий настройки по умолчанию. Можно также открыть этот сеанс с повышенными привилегиями (если планируется запускать приложения, которые нуждаются в доступе к защищенным ресурсам) либо изменить свойства исполняемого файла (добавив параметры командной строки).

В любом случае в командной строке появится подсказка, подобная показанной на рис. 3.1. (Ваше окно может отличаться от окна, показанного на рис. 3.1, если вы работаете не в Windows, если вместо командной версии Python применяется IDLE, ваша система имеет иную конфигурацию либо у вас установлена другая версия Python.) В командной строке отображается информация о версии Python, операционной системе и способах получения дополнительных сведений.

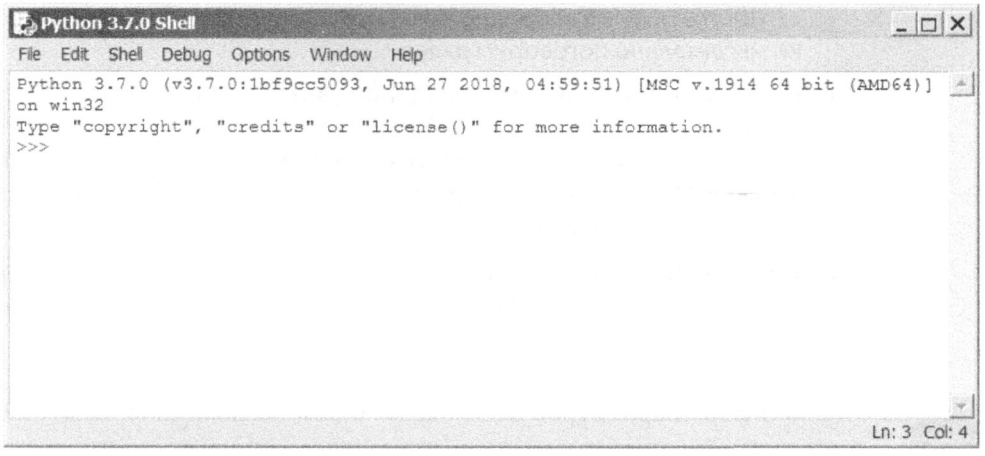


Рис. 3.1. Благодаря командной строке можно больше узнать о среде Python

НЕ ЗАБЫВАЙТЕ О ФАЙЛЕ

Многие приложения включают файл `README`. К сожалению, большинство пользователей игнорируют его, а некоторые даже не подозревают о его существовании. В результате они упускают важные сведения о своих приложениях. В Python имеется файл `NEWS.txt`, который находится в папке `\Python37`. В нем содержится полезная информация, в основном о новинках текущей версии Python, о которых стоит знать.

Не поленитесь открыть и прочитать файл `NEWS.txt`, чтобы потом поражать друзей глубиной своих познаний языка.

Как извлечь максимум пользы из командной строки

Этот раздел может поначалу показаться немного сложным, да и не особо полезным. Но уверен, в будущем вам пригодится эта информация. А пока можете просто пробежать раздел глазами, чтобы понять, какие варианты есть в вашем распоряжении.

Чтобы запустить Python из командной строки, достаточно ввести **`Python`** и нажать клавишу <Enter>. Однако это далеко не все, что можно сделать. В командной строке можно ввести дополнительную информацию, которая позволит изменить способ работы Python.

» **Параметры.** Параметр, или переключатель, командной строки начинается знаком "минус", за которым следует одна или несколько букв. Например, для получения справки о Python введите **`Python -h`**. На экране появится информация о порядке работы

с Python в режиме командной сроки. О параметрах командной строки мы детально поговорим далее.

- **Имя файла.** Если при вызове Python указать имя файла, то Python запустит его на выполнение. Подобным образом можно запустить любой пример книги, просто указав в командной строке имя его файла. Пусть, к примеру, файл называется `SayHello.py`. Чтобы запустить пример, введите **`Python SayHello.py`**.
- **Аргументы.** Работу многих приложений можно контролировать с помощью аргументов командной строки. К этой теме мы вернемся в других главах.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Большинство параметров вам пока ни о чем не скажут. Поэтому можете спокойно пропустить данный раздел и вернуться к нему позже, когда в этом возникнет необходимость.

ЗАПОМНИ!

В Python параметры командной строки чувствительны к регистру символов. Например, параметр `-s` принципиально отличается от параметра `-S`. Параметры запуска Python перечислены ниже.

- `-b`. Вывод предупреждения, когда приложение выполняет сравнение байтов или байтовых массивов с помощью функции `str()`.
- `-bb`. Вывод сообщения об ошибке, когда приложение выполняет сравнение байтов или байтовых массивов с помощью функции `str()`.
- `-B`. Не записывать файлы *.py* или *.pyco* при импорте модулей.
- `-c` *cmd*. Запускать программу с использованием информации, заданной в аргументе *cmd*. Все последующие данные трактуются как часть этого аргумента.
- `-d`. Запуск отладчика (применяется для поиска ошибок в приложении).
- `-E`. Игнорирование всех переменных среды Python, таких как `PYTHONPATH`, применяемых для конфигурирования среды выполнения.
- `-h`. Отображение справки по параметрам и основным переменным среды. В результате выполнения этой команды Python всегда завершает свою работу, чтобы вы смогли прочитать справку.
- `-i`. Заставляет Python перейти в интерактивный режим выполнения кода после запуска сценария. Запрос на ввод появляется даже в том случае, когда `stdin` (стандартное устройство ввода) не является терминалом.

- `-m mod`. Запуск библиотечного модуля, указанного с помощью аргумента `mod`, в качестве сценария. Все последующие данные трактуются как часть аргумента `mod`.
- `-O`. Небольшая оптимизация генерируемого байт-кода (ускоряет его выполнение).
- `-OO`. Дополнительная оптимизация за счет удаления строк документации.
- `-q`. Указывает на то, что не нужно выводить сообщения о версии и авторском праве при интерактивном запуске.
- `-s`. Указывает на то, что не нужно добавлять каталог местоположения пользователя в переменную `sys.path` (эта переменная сообщает Python о том, где искать модули).
- `-S`. Не запускать импорт местоположений при инициализации. Это означает, что Python не будет искать каталоги, в которых могут содержаться необходимые модули.
- `-u`. Разрешает небуферизованный двоичный ввод для устройств `stdout` (стандартный вывод) и `stderr` (стандартный поток ошибок). Устройство `stdin` всегда буферизуется.
- `-v`. Переключение в режим вывода подробных сведений, позволяющий просматривать все инструкции импорта. Многократное использование этого параметра увеличивает уровень подробностей.
- `-V`. Вывод номера версии Python и завершение работы.
- `--version`. Вывод номера версии Python и завершение работы.
- `-W arg`. Изменение уровня предупреждений, которое приводит к тому, что Python отображает больше или меньше предупреждений. Допустимые значения `arg` приведены в ниже.
  - `action`
  - `message`
  - `category`
  - `module`
  - `lineno`
- `-x`. Пропуск первой строки файла исходного кода, благодаря чему можно использовать синтаксис `#!команда`, несовместимый с Unix.
- `-X opt`. Установка параметра, зависящего от реализации. (Информацию о таких параметрах вы сможете найти в документации к своей версии Python.)

Использование переменных среды Python

Переменные среды — это специальные настройки, которые являются частью командной строки или среды терминала операционной системы. Они предназначены для согласованной настройки Python. Переменные среды выполняют те же функции, что и параметры, указываемые при запуске Python. Но благодаря тому, что значения переменных среды не изменяются, можно сохранять постоянную конфигурацию Python при каждом запуске. Это позволит избежать настройки параметров вручную.

Как и в случае с параметрами, переменные среды вряд ли будут для вас полезны прямо сейчас. Это просто информация, которой вы сможете воспользоваться в случае необходимости. Некоторые из переменных среды будут использованы в примерах книги. Сделайте закладку на этом разделе, чтобы вернуться к нему позже.

В большинстве операционных систем переменные среды можно конфигурировать временно, в течение определенного сеанса, или на постоянной основе, как часть конфигурации операционной системы. Как конкретно это делается, зависит от платформы. Например, в Windows можно воспользоваться командой `Set` (см. мой пост в блоге `http://blog.johnmuellerbooks.com/2014/02/24/using-the-set-command-to-your-advantage`) либо специальной утилитой конфигурации Windows (см. мой пост в блоге `http://blog.johnmuellerbooks.com/2014/02/17/adding-a-location-tothe-windows-path/`, где описан пример настройки переменной `Path`).

Переменные среды имеет смысл использовать в том случае, когда приходится регулярно конфигурировать Python однотипным образом. Ниже перечислены переменные среды Python.

- `PYTHONCASEOK=x`. Заставляет Python игнорировать регистр символов при синтаксическом анализе инструкций `import`. Эта переменная среды предназначена только для Windows.
- `PYTHONDEBUG=x`. Выполняет ту же функцию, что и параметр `-d`.
- `PYTHONDONTWRITEBYTECODE=x`. Выполняет ту же функцию, что и параметр `-B`.
- `PYTHONFAULTHANDLER=x`. Заставляет Python выводить дамп стека вызовов в случае возникновения фатальных ошибок.
- `PYTHONHASHSEED=arg`. Определяет начальное значение, используемое для генерирования различных видов хеш-значений. Если этой переменной присвоено значение `random`, Python использует случайное число в качестве начальной величины для генерирования

хеш-значений объектов `str`, `bytes` и `datetime`. Допустимый диапазон целочисленных значений — от 0 до 4 294 967 295. Использование фиксированного числа позволяет получать прогнозируемые хеш-значения в целях тестирования.

- `PYTHONHOME=arg`. Определяет заданный по умолчанию путь поиска, применяемый в Python для поиска модулей.
- `PYTHONINSPECT=x`. Выполняет ту же функцию, что и параметр `-i`.
- `PYTHONIOENCODING=arg`. Задает кодировку (например, UTF-8), используемую для устройств `stdin`, `stdout` и `stderr`.
- `PYTHONNOUSERSITE`. Выполняет ту же функцию, что и параметр `-s`.
- `PYTHONOPTIMIZE=x`. Выполняет ту же функцию, что и параметр `-O`.
- `PYTHONPATH=arg`. Содержит разделенный точками с запятой (`;`) список каталогов, применяемых для поиска модулей. Это значение хранится в переменной `sys.path` в Python.
- `PYTHONSTARTUP=arg`. Задает имя файла, вызываемого при запуске Python. У этой переменной среды нет значения по умолчанию.
- `PYTHONUNBUFFERED=x`. Выполняет ту же функцию, что и параметр `-u`.
- `PYTHONVERBOSE=x`. Выполняет ту же функцию, что и параметр `-v`.
- `PYTHONWARNINGS=arg`. Выполняет ту же функцию, что и параметр `-W`.

Ввод команд

После запуска командной строки Python можно начинать вводить команды. Это позволяет проверять различные идеи, возникающие по ходу написания приложений, и получать представление о том, как фактически работает Python. Соответствующие детали зачастую скрыты в интерактивной среде, такой как IDLE.

Скажите компьютеру, что ему нужно делать

Python, как и любой другой язык программирования, опирается на использование команд. *Команда* — это просто шаг процедуры, как, например, вызов функции `print()`.

Чтобы проинструктировать компьютер о выполняемых действиях, нужно передать ему одну или несколько команд на Python. Затем Python переведет эти команды в инструкции, понятные компьютеру, после чего появится результат.

Команда типа `print()` может практически мгновенно отображать результаты на экране. В Python имеется множество команд, многие из которых не отображают на экране ничего, хотя и делают что-то важное.

По мере чтения книги вы будете использовать команды для решения самых разных задач. Каждая из этих команд поможет вам достичь нужной цели, подобно шагам процедуры. Если вам кажется, что команды Python становятся слишком сложными, просто относитесь к ним, как к отдельным шагам. Любая, даже самая сложная, процедура становится проще, если рассматривать ее пошагово. То же самое можно сказать и о командах Python. Рассматривайте их по очереди и фокусируйтесь на конкретном шаге.

Расскажите компьютеру, что вы сделали

В какой-то момент времени выполнение процедуры завершается. В примере с приготовлением тостов это происходит, когда вы заканчиваете намазывать масло на хлеб. Компьютерные процедуры работают точно так же. Все они имеют точку начала и конца. При вводе команд конечной точкой для конкретного шага является нажатие клавиши <Enter>. После ввода команды вы сообщаете об этом компьютеру нажатием клавиши <Enter>. По мере чтения книги вы обнаружите, что в Python существует несколько способов показать, что команда, процедура или даже целое приложение завершены.

Просмотр результатов

Теперь вы знаете, что команда является шагом процедуры и что каждая команда имеет точку начала и конца. Кроме того, группы команд и целые приложения также имеют начало и конец. Давайте рассмотрим, как все это работает. Следующая процедура поможет вам увидеть результат выполнения команды.

1. **Запустите командную строку Python.**

   На экране появится окно, в котором можно вводить команды, как было показано на рис. 3.1.

2. **В командной строке введите `print("Текстовая строка.")`.**

   Обратите внимание: ничего не происходит. Да, вы ввели команду, но вы не указали, что она завершена.

3. **Нажмите клавишу <Enter>.**

   Теперь команда завершена, поэтому вы увидите результат, подобный показанному на рис. 3.2.

Это упражнение показывает, как все работает в Python. Каждая команда, вводимая пользователем, выполняет определенное действие, но только после того, как вы сообщите Python о том, что команда завершена. К примеру,

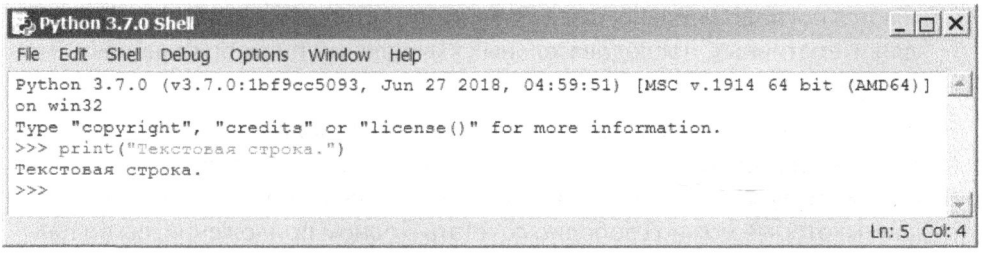


Рис. 3.2. С помощью команд вы говорите Python о том, что должен сделать компьютер

команда `print()` выводит данные на экране. В данном случае отображается текст, вводимый пользователем. Результат, показанный на рис. 3.2, появляется непосредственно после ввода команды, поскольку используется интерактивная среда. Позже, когда вы начнете создавать приложения, вы поймете, что иногда результат появляется не сразу из-за задержек со стороны приложения. Как бы там ни было, команда начинает обрабатываться, как только приложение сообщает Python о том, что команда завершена.

СТИЛИ ПРОГРАММИРОВАНИЯ НА PYTHON

Большинство языков программирования ориентировано на использование одного стиля программирования, что существенно снижает их гибкость. Но о Python такого не скажешь. В нем можно использовать разные стили программирования. Ниже перечислены четыре наиболее часто используемых стиля программрования.

- **Функциональный.** Каждая инструкция является своего рода математическим уравнением. Этот стиль хорош для выполнения параллельной обработки инструкций.
- **Императивный.** Вычисления происходят при изменении состояния программы. Этот стиль чаще всего используется для управления структурами данных.
- **Объектно-ориентированный.** Этот стиль обычно применяется для упрощения кода. В данном случае реальный мир моделируется с помощью парадигмы объектов. Python не полностью реализует такой стиль программирования, поскольку не поддерживает некоторые возможности, например инкапсуляцию. Тем не менее элементы объектно-ориентированного стиля используются достаточно широко. Примеры будут показаны в последующих главах.
- **Процедурный.** Весь код, который вы написали до сих пор (как и большая часть кода примеров книги), является процедурным: задачи в нем выпол-

няются последовательно, шаг за шагом. Этот стиль больше всего подходит для итеративных, последовательных вычислений и выборки данных. Это простейший стиль программирования.

Несмотря на то что наша книга не охватывает все стили программирования, полезно знать, что вы не являетесь заложником одного конкретного стиля. Благодаря тому что в Python поддерживается несколько стилей программирования, которые можно свободно сочетать в одном приложении, вы вольны использовать Python тем способом, который лучше всего подходит для вашего конкретного случая. Дополнительные сведения о различных стилях программирования можно найти в статье по адресу `https://blog.newrelic.com/2015/04/01/python-programming-styles/`.

Справочная система

Не следует забывать о том, что Python — это компьютерный язык, а не человеческий. Поэтому никто не может изначально на нем говорить. И даже освоив Python, вы не начнете свободно говорить "на языке Python". Освоение команд Python требует времени, сравнимого с затратами времени на изучение другого человеческого языка. Например, если вы хорошо говорите по-английски и хотите выучить другой язык, тот же немецкий, вам понадобится соответствующее руководство, иначе ваши попытки могут оказаться безуспешными, а все, что вы будете говорить на другом языке, будет звучать как минимум странно. И даже если вы скажете что-то осмысленное, оно может восприниматься совсем не так, как вы думаете.

Аналогично при изучении языка программирования Python вам понадобится руководство, которое поможет в затруднительных ситуациях. К счастью, о наличии такого руководства позаботились разработчики Python. Справка Python работает на двух уровнях:

- режим справки, позволяющий просматривать сведения по всем доступным командам;
- непосредственная справка, предоставляющая сведения о конкретной команде.

Не существует единственного верного способа использования справки, поскольку все зависит от конкретной ситуации. Способы получения справки описаны в следующих разделах.

Переход в режим справки

После первого запуска Python появляется экран, подобный показанному на рис. 3.1. Python сразу же предлагает вам следующие четыре команды (собственно говоря, это первая часть справочной информации):

- `help();`
- `copyright();`
- `credits();`
- `license()`.

Все они позволяют получить справку о Python. Например, команда `copyright()` сообщает о тех, кто имеет право копировать, лицензировать или иным образом распространять Python. Команда `credits()` сообщает о разработчиках Python. Команда `license()` выводит соглашение об использовании, заключенное между вами и владельцем авторских прав. Однако самая интересная команда — это, конечно, `help()`.

Чтобы перейти в режим справки, введите **`help()`**. Учтите, что нужно указывать круглые скобки после команды, даже если они не отображаются в тексте справки. Каждая команда Python завершается круглыми скобками. После ввода этой команды Python переходит в режим справки, и на экране появляется окно, подобное показанному на рис. 3.3.

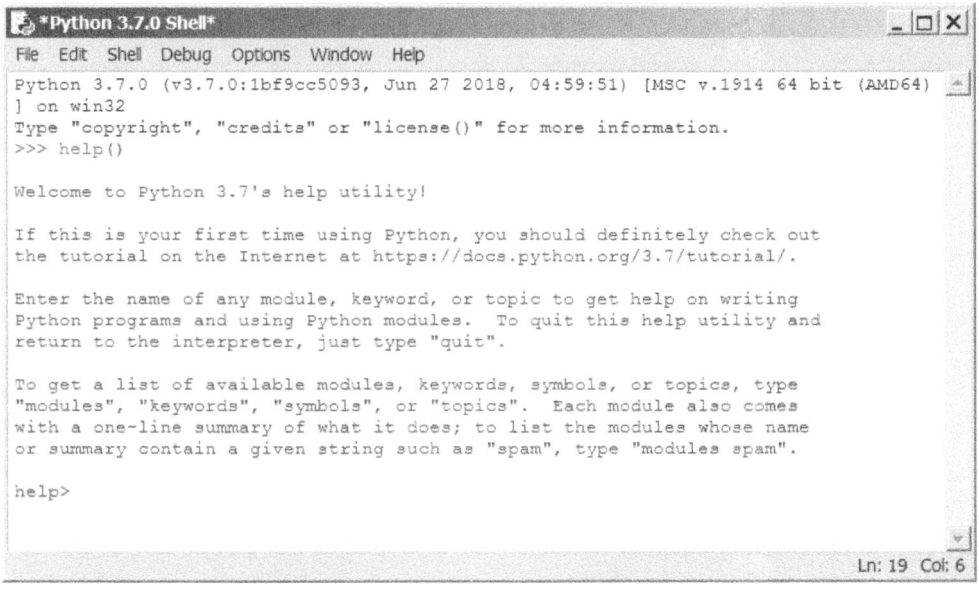


Рис. 3.3. Переход в режим справки

Признаком режима справки служит приглашение `help>` в командной строке.

Получение справки

Чтобы получить справку, нужно знать, какой именно вопрос задать. Начальное справочное сообщение, отображаемое после перехода в режим справки (см. рис. 3.3), содержит несколько полезных советов о типах вопросов, которые можно задать. Если хотите изучить Python, обратите внимание на следующие четыре основные темы:

- `modules`;
- `keywords`;
- `symbols`;
- `topics`.

Первые две темы на данный момент вам неизвестны. Тема `modules` (модули) не понадобится вам до главы 10. Ну а тема `keywords` (ключевые слова) пригодится, начиная с главы 4. Зато темы `symbols` (специальные символы) и `topics` (разделы справки) будут полезны прямо сейчас, поскольку помогут лучше понять, с чего начать путешествие в мир Python. Введите **`symbols`**, и вы увидите список специальных символов, используемых в Python (рис. 3.4). О них мы поговорим в главе 7, при рассмотрении операторов Python.

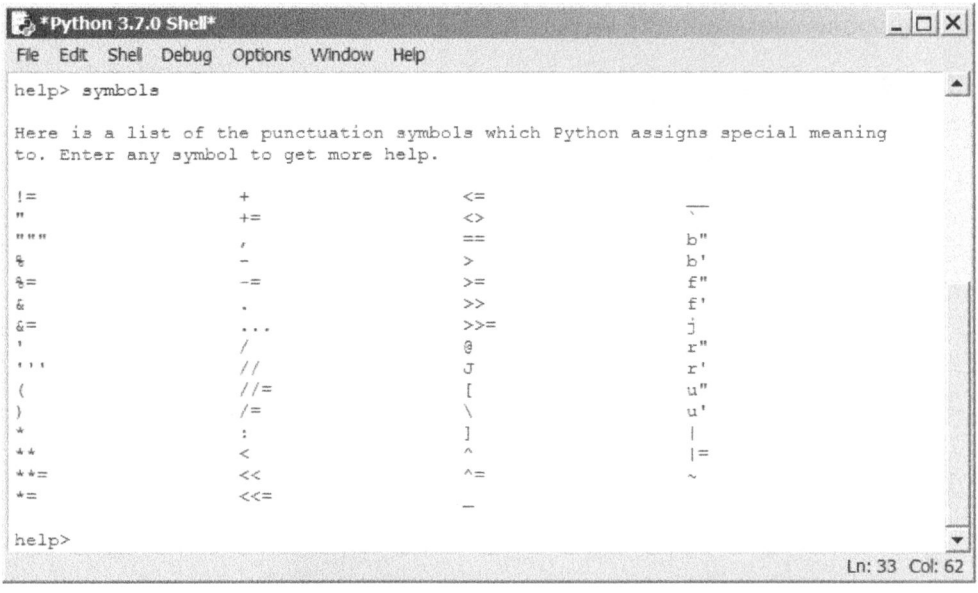


Рис. 3.4. Специальные символы, поддерживаемые в Python

Чтобы просмотреть список доступных разделов, по которым можно получить справку, введите **`topics`**. Если вас заинтересует определенный раздел,

например FUNCTIONS, просто введите его название и нажмите клавишу <Enter>. На экране появится справочная информация, подобная показанной на рис. 3.5.

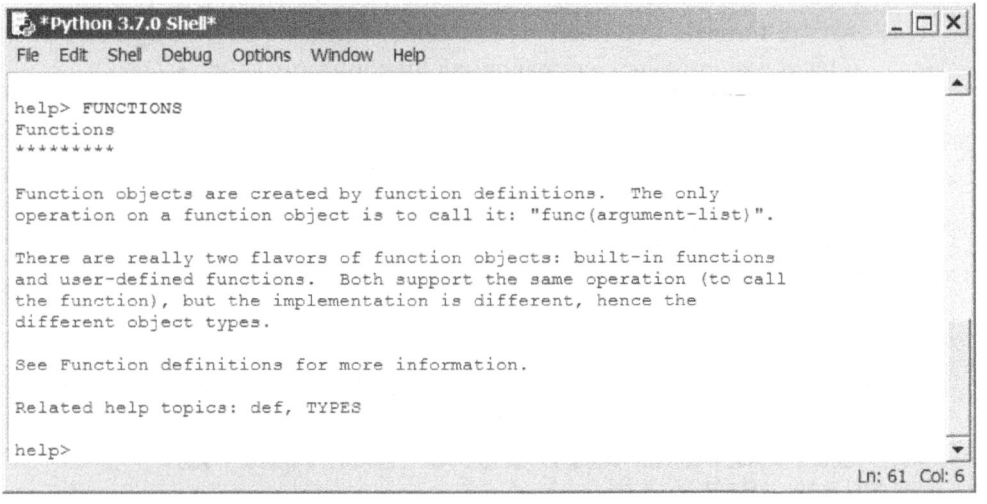


Рис. 3.5. Получение справки по выбранной теме

По мере выполнения примеров книги вы будете использовать различные команды и, возможно, захотите узнать о них больше. Например, в разделе "Просмотр результатов" вы познакомились с командой `print()`. Чтобы просмотреть дополнительные сведения о ней, введите **`print`**. (Учтите, что на сей раз круглые скобки не нужны, поскольку мы не выполняем команду `print()`, а запрашиваем справку по ней.) На рис. 3.6 показана типичная справочная информация по команде `print()`.

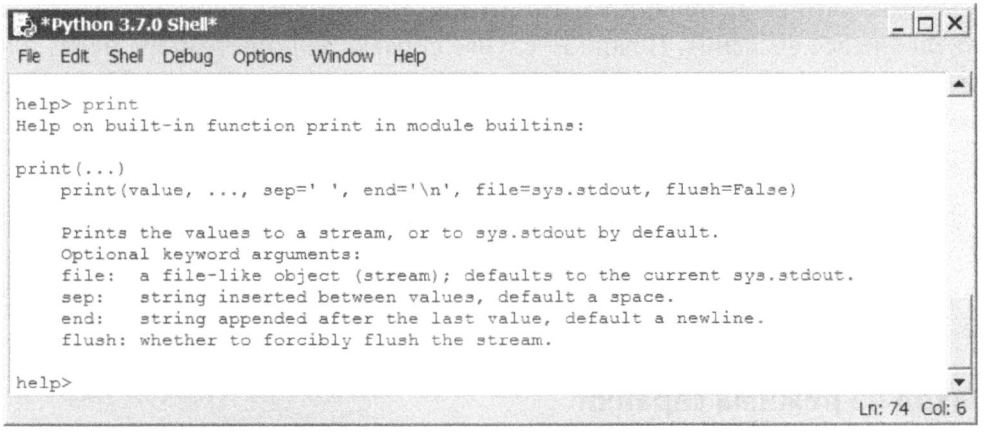


Рис. 3.6. Получение справочной информации о команде

СОВЕТ

Конечно, пока вы практически ничего не знаете о Python, вряд ли вам поможет чтение справочной информации. Но в некоторых ситуациях можно получить дополнительные сведения. Предположим, вы хотите понять, что такое `sys.stdout`. Просто введите **`sys.stdout`**, и на экране появится справочная информация (рис. 3.7).

```
Python 3.7 (64-bit)
Help on TextIOWrapper in sys object:

sys.stdout = class TextIOWrapper(_TextIOBase)
 |  sys.stdout(buffer, encoding=None, errors=None, newline=None, line_buffering=
False, write_through=False)
 |
 |  Character and line based layer over a BufferedIOBase object, buffer.
 |
 |  encoding gives the name of the encoding that the stream will be
 |  decoded or encoded with. It defaults to locale.getpreferredencoding(False).
 |
 |  errors determines the strictness of encoding and decoding (see
 |  help(codecs.Codec) or the documentation for codecs.register) and
 |  defaults to "strict".
 |
 |  newline controls how line endings are handled. It can be None, '',
 |  '\n', '\r', and '\r\n'.  It works as follows:
 |
 |  * On input, if newline is None, universal newlines mode is
 |    enabled. Lines in the input can end in '\n', '\r', or '\r\n', and
 |    these are translated into '\n' before being returned to the
 |    caller. If it is '', universal newline mode is enabled, but line
 |    endings are returned to the caller untranslated. If it has any of
 |    the other legal values, input lines are only terminated by the given
-- Далее --
```

Рис. 3.7. Можно запросить дополнительную справку по уже полученной справке

Не факт, что вы получили то, что искали. Но обратите внимание на то, что это еще не все сведения. В данном случае справка занимает много места и не помещается на экране, поэтому в нижней части экрана отображается следующая запись:

```
-- Далее --
```

Для просмотра следующей страницы справки нажмите клавишу пробела, а если и это еще не конец, то снова нажмите клавишу пробела для перехода к очередной странице. При необходимости всегда можно вернуться к предыдущей странице справки.

Выход из режима справки

Наступает момент, когда нужно выйти из режима справки. Для этого достаточно нажать <Enter>, не вводя дополнительной информации. Появится

сообщение о выходе из режима справки, а приглашение превратится в стандартную командную строку Python, как показано на рис. 3.8.

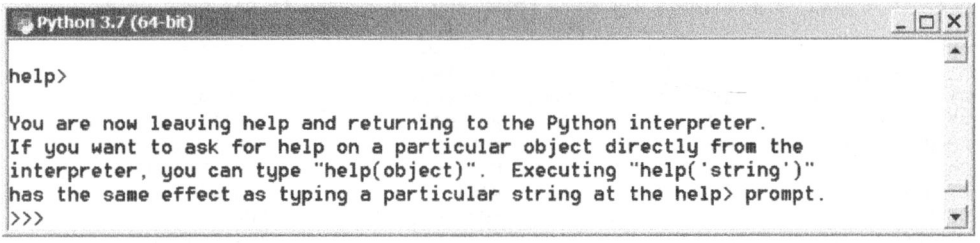


Рис. 3.8. Для выхода из режима справки достаточно нажать клавишу <Enter>

Непосредственное получение справки

Переход в режим справки не слишком-то и нужен, если вы не планируете просматривать несколько разделов справки или же хорошо знаете, что именно хотите найти. В Python есть удобная возможность непосредственного получения справки по конкретной команде. Вместо того чтобы переходить в режим справки, достаточно ввести слово `help`, а затем в круглых скобках указать команду, для которой нужно получить справку. Например, чтобы узнать больше о команде `print()`, введите **`help(print)`**. На рис. 3.9 показан типичный результат при использовании этого вида справки.

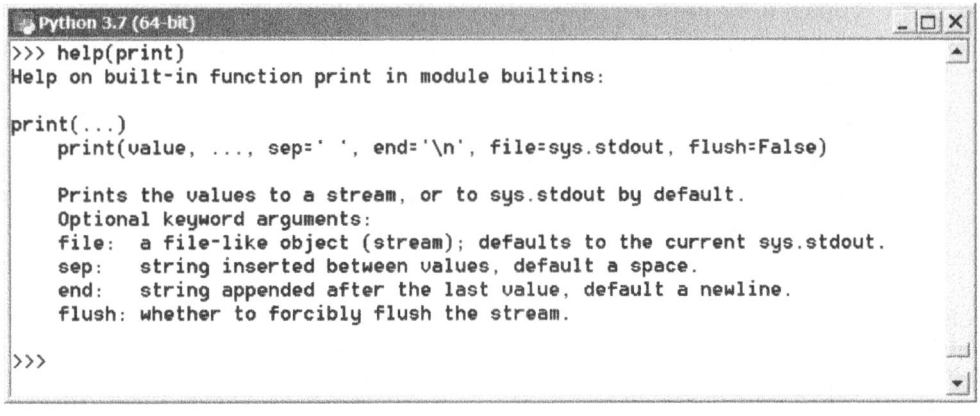


Рис. 3.9. Можно получить справку, не выходя из режима командной строки Python

Можно также просматривать разделы справки в режиме командной строки Python. Если ввести **`help('topics')`**, то на экране появится список тем, как показано на рис. 3.10.

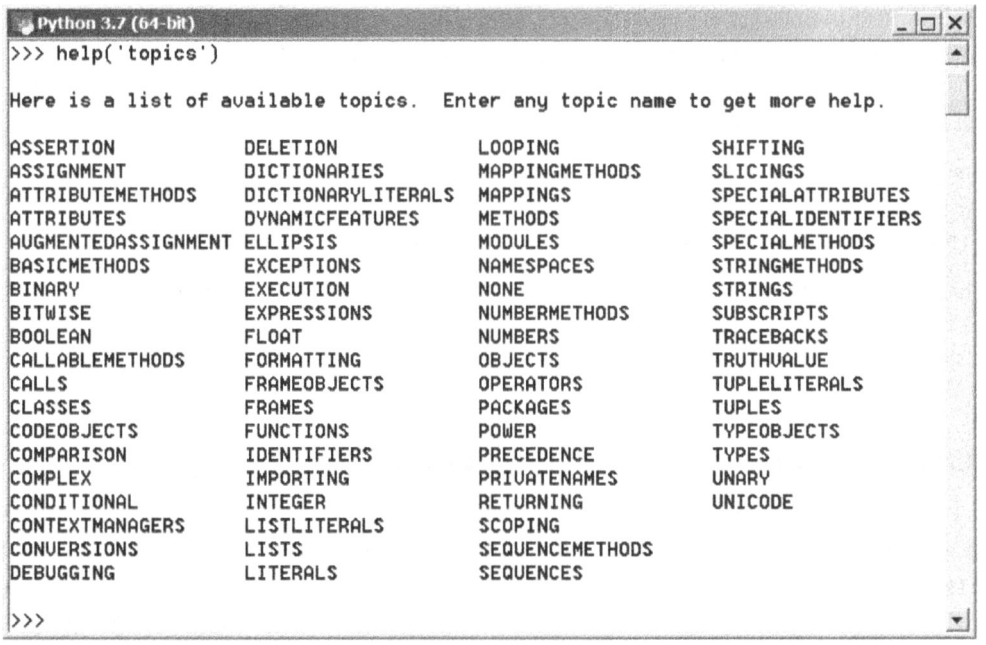


Рис. 3.10. При желании можно просматривать разделы справки в командной строке Python

СОВЕТ

У многих читателей наверняка возник вопрос: зачем нужен режим справки, если те же самые результаты можно получить из командной строки Python? В действительности режим справки просто удобен. В нем легче просматривать информацию. Кроме того, в нем приходится вводить меньше команд. К тому же в нем можно получать дополнительные сведения, как было показано на рис. 3.3. Таким образом, у вас есть все основания для перехода в режим справки, если вы планируете задать Python много вопросов.

ЗАПОМНИ!

Независимо от того, как вы обращаетесь за помощью, нужно придерживаться корректного регистра символов в названиях разделов справки. Например, чтобы получить общую информацию о функциях, нужно ввести **`help('FUNCTIONS')`**, а не **`help('Functions')`** или **`help('functions')`**. Если регистр символов задан неправильно, Python сообщит о том, что не знает, чего вы от него хотите, или не может найти раздел справки. Когда-нибудь компьютеры научатся читать мысли, но пока что до этого далеко.

Выход из режима командной строки

Рано или поздно вы захотите покинуть гостеприимную среду Python, тем более что есть и другие дела, кроме как целый день возиться с Python. Для выхода из среды Python можно воспользоваться двумя стандартными методами и целой кучей нестандартных способов. Стандартные методы гарантируют корректное завершение работы Python. Но и нестандартные способы тоже хороши, особенно если вы просто экспериментировали с Python, не занимаясь написанием готового кода. Вот два стандартных метода:

- `quit();`
- `exit().`

Любая из указанных функций осуществляет выход из интерактивной среды Python (ее еще называют *оболочкой*).

Обе функции поддерживают необязательный аргумент. Например, можно ввести **`quit(5)`** или **`exit(5)`** и нажать клавишу <Enter> для выхода из оболочки. Обычно, если приложение завершилось без сбоев, выполняют команду `quit()` и `exit()` без аргумента. Числовой аргумент устанавливает переменную среды `ERRORLEVEL`, которую можно перехватить в командной строке или в сценарии. Чтобы увидеть, как это работает, выполните следующие действия.

1. **Откройте командную строку или окно терминала.**

   На экране появится приглашение командной строки.

2. **Введите `Python`, чтобы запустить Python.**

   На экране появится приглашение Python.

3. **Введите `quit(5)`.**

   Снова появится стандартное приглашение командной строки.

4. **Введите `echo %ERRORLEVEL%`.**

   Отобразится код ошибки, как показано на рис. 3.11. Если вы работаете не в Windows, нужно будет ввести какую-то другую команду. Например, при работе с bash-сценарием нужно ввести **`echo $`**.

Один из наиболее распространенных нестандартных способов выхода заключается в щелчке на кнопке Close (Закрыть) окна командной строки или терминала. В данном случае у приложения может не быть времени на выполнение необходимой очистки, что может привести к странному поведению. Всегда лучше закрыть Python, используя стандартный метод, если вы делали что-то большее, чем обычный просмотр справки.

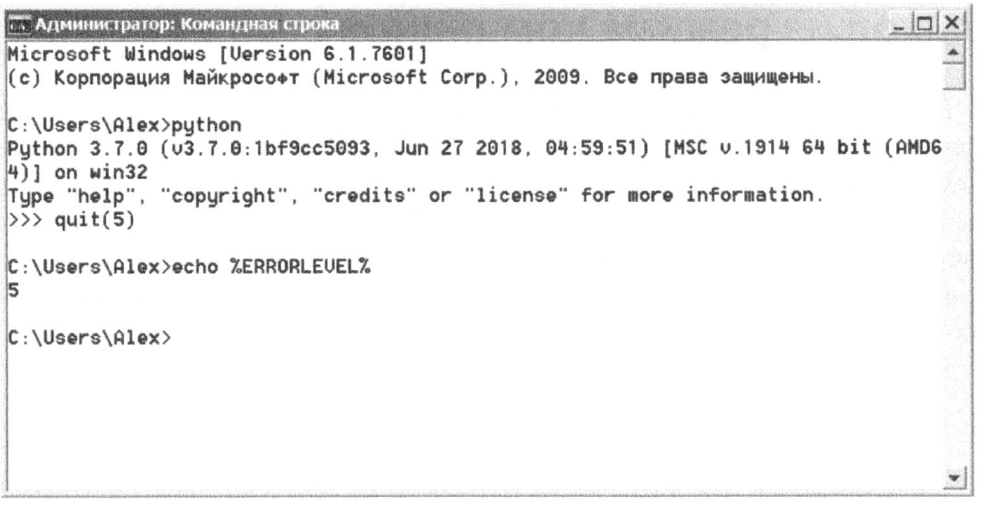


Рис. 3.11. Укажите код ошибки, чтобы уведомить других пользователей о статусе завершения Python

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

При необходимости можно также воспользоваться рядом других команд для закрытия командной строки. В большинстве случаев эти специальные команды не нужны, поэтому можно пропустить оставшуюся часть раздела.

При использовании команды `quit()` или `exit()` Python выполняет ряд действий, гарантирующих корректное завершение сеанса. Если есть подозрение, что сеанс все-таки завершается некорректно, вы всегда можете положиться на одну из двух команд, осуществляющих закрытие командной строки:

- `sys.exit()`;
- `os._exit()`.

Обе они используются только в аварийных ситуациях. Первая команда, `sys.exit()`, поддерживает специальные средства обработки ошибок, которые будут рассмотрены в главе 9. Вторая команда, `os._exit()`, обеспечивает выход из среды Python без выполнения каких-либо стандартных операций по очистке. В обоих случаях следует предварительно импортировать специальный модуль, `sys` или `os`. Таким образом, команда `sys.exit()` фактически вызывается так.

```
import sys
sys.exit()
```

В случае команды `os._exit()` следует указать код ошибки, поскольку эта команда используется только при возникновении серьезной ошибки. Если код

ошибки не указан, вызов команды завершится неудачей. Схема вызова команды `os._exit()` такова (код ошибки в данном случае равен 5).

```
import os
os._exit(5)
```

В главе 10 мы подробно рассмотрим импорт модулей. А пока вам достаточно знать, что эти две команды предназначены для специального применения и в приложениях обычно не используются.

Глава 4

Создание первого приложения

В ЭТОЙ ГЛАВЕ...

- Использование Jupyter Notebook в качестве интегрированной среды разработки
- Создание и выполнение первого приложения
- Форматирование кода приложения
- Комментарии в коде
- Управление приложениями с помощью Anaconda

Многие люди считают разработку приложений своего рода магическим ритуалом, доступным лишь избранным, тем, кто умеет с помощью клавиатуры создавать различные программы. Хотя на самом деле все гораздо проще.

Как однажды отметил Артур Кларк, "любая достаточно продвинутая технология неотличима от магии". Разработка приложений — довольно строгая дисциплина и, безусловно, далекая от магии. В этой главе завеса тайны будет убрана, и читатель познакомится с технологией разработки приложений. После прочтения главы вы сможете написать простую программу. (Безо всякой магии, уж поверьте!)

Для написания приложений нужен соответствующий инструментарий. В случае с Python вам не обязательно использовать сложные инструменты, но они настолько упрощают решение задач, что лучше все же воспользоваться ими. В этой главе применяется общедоступная интегрированная среда разработки (Integrated Development Environment — IDE) под названием Jupyter

Notebook, которая входит в пакет Anaconda. Благодаря IDE значительно упрощается процесс написания, тестирования и отладки кода. В предыдущей главе мы использовали утилиту командной строки, что позволило немного поэкспериментировать с Python. Однако возможности Anaconda намного шире по сравнению с командной строкой и позволяют существенно упростить создание приложений.

СОВЕТ

Для создания приложений Python можно использовать огромное количество других инструментов. В книге они не рассматриваются, поскольку пакет Anaconda свободно доступен и подходит для решения всего спектра требуемых задач. Можете самостоятельно ознакомиться с возможностями других инструментов, например Komodo Edit (`http://www.activestate.com/komodo-edit/downloads`). Обширный перечень подобных инструментов можно найти по следующему адресу:

`https://wiki.python.org/moin/IntegratedDevelopmentEnvironments`

Преимущества IDE

У многих читателей возникнет вполне закономерный вопрос: зачем при работе с Python нужна IDE, если утилиты командной строки вполне достаточно? К тому же в комплект поставки Python входит IDLE (Integrated DeveLopement Environment), более продвинутая среда, чем командная строка. Казалось бы, для целей обучения и даже разработки приложений ничего другого и не нужно. Конечно, инструменты, поставляемые с Python, интересны и полезны на ознакомительном этапе, но с их помощью довольно сложно создавать полнофункциональные приложения. Если вы собираетесь работать с Python серьезно, то в силу многих причин вам понадобится действительно универсальный инструмент.

Создание оптимального кода

Хорошая IDE предлагает определенный уровень интеллектуальной поддержки. Например, IDE предложит альтернативы при вводе неправильного ключевого слова или подскажет, если введенная строка кода не будет работать. Чем обширнее такого рода поддержка со стороны IDE, тем проще создавать хороший код. А это важно, поскольку плохой код чреват появлением ошибок, называемых *багами*, на поиск и устранение которых уходит много времени.

СОВЕТ

Различные IDE сильно отличаются по уровню интеллектуальной поддержки и разнообразию предоставляемой информации. Может, поэтому их так много. И если вас в какой-то момент времени перестала устраивать имеющаяся IDE, не стоит отчаиваться, поскольку всегда можно выбрать другую. У каждого разработчика свои потребности и, следовательно, разные требования к IDE. Вся проблема заключается в выборе IDE, которая поможет быстро и легко писать понятный и эффективный код.

Контроль ошибок

Поиск ошибок в коде называется *отладкой*. Даже самый опытный разработчик массу времени тратит на отладку. Создать совершенный код с первой попытки практически невозможно. Если вам это удастся, можете себя поздравить с редкой удачей. Следовательно, возможности IDE по отладке имеют критически важное значение. К сожалению, входящие в комплект поставки Python инструменты имеют весьма скромные возможности в плане отладки. Вам придется тратить массу времени и сил на отладку, поэтому лучше заранее выбрать другой инструментарий.

СОВЕТ

Наиболее универсальные IDE могут выступать в качестве обучающих инструментов. Обладая достаточным потенциалом, IDE поможет вам изучить код, написанный настоящими профессионалами. Трассировка приложений — проверенный временем способ получения новых навыков и совершенствования имеющихся. Как известно, даже небольшое увеличение багажа знаний может обернуться в будущем серьезной экономией ресурсов. Если вы выбираете IDE, не только уделяйте внимание функциям отладки как средству для контроля ошибок, но и обращайте внимание на возможности, предоставляемые в IDE для изучения Python.

Преимущества блокнота

Большинство IDE выглядят как текстовые редакторы, и это не удивительно. Они предоставляют вам различные интеллектуальные возможности, подсказки, советы, цветовые коды и прочее, но, в конце концов, это всего лишь текстовые редакторы, от которых трудно ожидать многого. Учитывая тот факт, что пользователи Python часто сосредоточены на разработке научных приложений и нуждаются в соответствующей поддержке, которая отличается от правки текста, лучше перейти на использование блокнотов.

Блокнот отличается от текстового редактора тем, что основан на методологии грамотного программирования, разработанной компьютерным ученым из Стэнфордского университета Дональдом Кнутом. *Грамотное программирование* применяется для создания смешанного представления кода, заметок, математических уравнений и графиков. Короче говоря, вы получаете в свое распоряжение блокнот исследователя, заполненный массой информации, которая требуется для всестороннего понимания кода. Методы грамотного программирования часто применяются в дорогостоящих пакетах, таких как Mathematica и MATLAB. Разработка в среде блокнота обеспечивает следующие возможности:

- удобное представление примеров;
- совместная разработка проектов;
- научные расчеты;
- обучение программированию;
- наглядные презентации.

В книге используется пакет Anaconda, который не только предлагает прекрасные возможности по разработке приложений Python, но и позволяет применять принципы грамотного программирования. Если вы решаете в основном научные задачи, то без пакетов типа Anaconda вам не обойтись. Кроме того, Anaconda доступен бесплатно.

Получаем собственную копию Anaconda

Как уже упоминалось, Anaconda не входит в состав Python. Конечно, при желании вы сможете выполнять примеры книги в IDLE, но желательно перейти на Anaconda. В следующих разделах будет показано, как установить Anaconda на трех основных платформах.

Где можно скачать Anaconda

Базовый пакет Anaconda можно бесплатно загрузить по следующему адресу:

```
https://www.anaconda.com/download/
```

На сайте можно выбрать одну из трех платформ и загрузить бесплатно распространяемый пакет. Anaconda поддерживает следующие платформы:

- Windows — 32- и 64-разрядная версии (программа установки предложит загрузить 32- или 64-разрядную версию пакета в зависимости от версии Windows);
- Linux — 32- и 64-разрядная версии;
- Mac OS X — 64-разрядная версия.

ВНИМАНИЕ!

В книге используется версия Anaconda 5.3.0, которая поддерживает Python 3.7.0. Если у вас другая версия Anaconda, может оказаться, что некоторые примеры будут работать некорректно, причем то, что отображается у вас на экране, может не соответствовать содержанию рисунков в книге, даже если вы работаете в Windows. Копии экранов, приведенные в книге, получены в 64-разрядной среде Windows, и они должны быть близки к тому, что отображается на экране при использовании Anaconda 5.3.0 на других платформах.

СОВЕТ

Можно загрузить версию Anaconda, предназначенную для работы с ранними версиями Python. Для этого щелкните на ссылке, соответствующей архиву программы установки, которая отображается в нижней части страницы сайта Anaconda. К ранней версии Python следует прибегать лишь в том случае, если в этом есть необходимость.

Программа установки Miniconda позволяет сэкономить время инсталляции, ограничив количество устанавливаемых функций. Но лучше выполнить полную установку, чтобы иметь под рукой все необходимое для выполнения проектов. Полная установка не займет много времени и не отнимет много ресурсов.

Бесплатной версии Anaconda вполне достаточно для освоения примеров книги. На сайте Anaconda доступен также ряд других продуктов, которые позволят создавать надежные приложения. Эти продукты не рассматриваются в книге, но на сайте Anaconda можно найти подробную информацию об их использовании.

Установка Anaconda в Linux

Для установки Anaconda на Linux придется использовать режим командной строки, поскольку возможность графической установки не предоставляется. Прежде чем выполнить установку, нужно загрузить копию дистрибутива с сайта Anaconda. Необходимую информацию о загрузке можно найти в предыдущем разделе. Следующая процедура должна работать в любой Linux-системе независимо от того, какую версию Anaconda вы используете: 32- или 64-разрядную.

1. **Откройте приложение Terminal.**

   Отобразится окно терминала.

2. **Перейдите в каталог, куда была загружена Anaconda.**

   Имя загружаемого файла может быть разным, но обычно это `Anaconda3-5.3.0-Linux-x86.sh` для 32-разрядных систем и `Anaconda3-5.3.0-Linux-x86_64.sh` — для 64-разрядных. Номер версии закодирован в имени файла. В данном случае имя файла указывает на версию 5.3.0, используемую в книге. Если применяется другая версия, могут возникнуть проблемы с исходным кодом примеров, и в них придется вносить коррективы.

3. **Введите `bash Anaconda3-5.3.0-Linux-x86.sh` (для 32-разрядной версии) или `bash Anaconda3-5.3.0-Linux-x86_64.sh` (для 64-разрядной версии).**

   Запустится мастер установки с просьбой принять условия лицензионного соглашения для использования Anaconda.

4. **Ознакомьтесь с условиями лицензионного договора и примите их.**

   Мастер попросит предоставить место для установки Anaconda. В книге предполагается, что по умолчанию используется местоположение `~/anaconda`. Если выбирается другое место, может возникнуть необходимость внести ряд изменений в некоторые примеры.

5. **Укажите каталог установки (если это необходимо) и нажмите клавишу <Enter> или щелкните на кнопке Next (Далее).**

   Начнется процесс установки приложения. После его завершения появится соответствующее сообщение.

6. **Добавьте путь установки в переменную среды `PATH`.**

   Теперь можно использовать Anaconda.

Установка Anaconda в Mac OS

Для платформы Mac OS X существует только 64-разрядная версия Anaconda. Прежде чем приступить к установке, следует скачать копию файла установки для Mac с сайта Anaconda. Необходимая информация о загрузке была приведена выше.

Файлы установки представлены в двух форматах. Во-первых, имеется графический инсталлятор, а во-вторых — утилита командной строки. Последняя работает так же, как и версия для Linux, которая была описана в главе 2. Ниже приведена пошаговая инструкция, которая поможет установить 64-разрядную версию Anaconda в Mac в графическом режиме.

1. **Найдите в системе загруженную копию файла установки Anaconda.**

   Имя этого файла может быть разным, но обычно это `Anaconda3-5.3.0-MacOSX-x86_64.pkg`. Номер версии закодирован в имени файла. В данном случае имя файла указывает на версию 5.3.0, используемую в книге. Если применяется другая версия, могут возникнуть проблемы с исходным кодом примеров, и в них придется вносить коррективы.

2. **Дважды щелкните на файле установки.**

   Появится начальное диалоговое окно.

3. **Щелкните на кнопке Continue (Продолжить).**

   Мастер установки спросит, желаете ли вы просмотреть файл Read Me. Можно ознакомиться с ним позже.

4. **Щелкните на кнопке Continue.**

   Мастер отобразит текст лицензионного соглашения. Обязательно ознакомьтесь с лицензионным договором, содержащим условия использования программы.

5. **Щелкните на кнопке I Agree (Я согласен), если вы согласны с условиями лицензионного соглашения.**

   Мастер попросит указать место для установки. Выбор местоположения зависит от того, для кого выполняется установка: для отдельного пользователя или для группы пользователей.

ВНИМАНИЕ!

На данном этапе может появиться сообщение об ошибке, в котором будет сказано, что установка Anaconda в вашей системе невозможна. Это следствие ошибки в инсталляторе и не имеет отношения к вашей системе. Для того чтобы сообщение исчезло, установите флажок Install Only for Me (Установить только для меня). Пакет Anaconda невозможно установить в Mac для группы пользователей.

6. **Щелкните на кнопке Continue.**

   Программа установки отобразит диалоговое окно, содержащее варианты по изменению типа установки. Щелкните на кнопке Change Install Location (Изменить местоположение установки), если необходимо изменить каталог Anaconda в системе. (В книге предполагается использование заданного по умолчанию пути `~/anaconda`.) Щелкните на кнопке Customize (Настроить), если необходимо внести изменения в работу программы установки. Например, можно установить флажок, который позволит не добавлять Anaconda в переменную среды `PATH`. Но в книге предполагается выбор параметров установки по умолчанию, и если только в системе не установлена другая копия Python 3.7.0, то нет веских причин для изменения этих параметров.

7. **Щелкните на кнопке Install (Установить).**

Начнется процесс установки. По завершении появится соответствующее диалоговое окно.

8. **Щелкните на кнопке Continue.**

   Все готово к использованию Anaconda.

Установка Anaconda в Windows

В комплект поставки Anaconda входит графический инсталлятор для Windows, поэтому установку можно выполнить с помощью мастера. Конечно, предварительно нужно загрузить копию установочного файла; о том, как это сделать, говорилось выше. Следующая процедура должна работать в любой Windows-системе независимо от того, какую версию Anaconda вы используете: 32- или 64-разрядную.

1. **Найдите в системе загруженную копию Anaconda.**

   Имя загружаемого файла может быть разным, но обычно это `Anaconda3-5.3.0-Windows-x86.exe` для 32-разрядных систем и `Anaconda3-5.3.0-Windows-x86_64.exe` — для 64-разрядных. Номер версии закодирован в имени файла. В данном случае имя файла указывает на версию 5.3.0, используемую в книге. Если применяется другая версия, могут возникнуть проблемы с исходным кодом примеров, и в них придется вносить коррективы.

2. **Дважды щелкните на файле установки.**

   Может появиться предупреждение подсистемы безопасности с вопросом о том, хотите ли вы запустить этот файл. Щелкните на кнопке Run (Выполнить). На экране появится диалоговое окно Anaconda3 5.3.0 Setup (Установка Anaconda3 5.3.0), подобное показанному на рис. 4.1. Точный вид диалогового окна зависит от загружаемой версии программы установки Anaconda. Если у вас 64-разрядная операционная система, то используйте 64-разрядную версию Anaconda, что позволит добиться максимальной производительности. В нашем случае мы имеем дело с 64-разрядной версией продукта.

3. **Щелкните на кнопке Next (Далее).**

   Мастер отобразит лицензионное соглашение. Обязательно прочтите его для ознакомления с условиями использования пакета.

4. **Щелкните на кнопке I Agree (Я согласен), чтобы принять лицензионное соглашение.**

   Появится запрос о типе выполняемой установки (рис. 4.2). В большинстве случаев программа устанавливается для личных целей. Исключение имеет место в случае, когда в системе работают несколько пользователей и всем им необходим доступ к Anaconda.

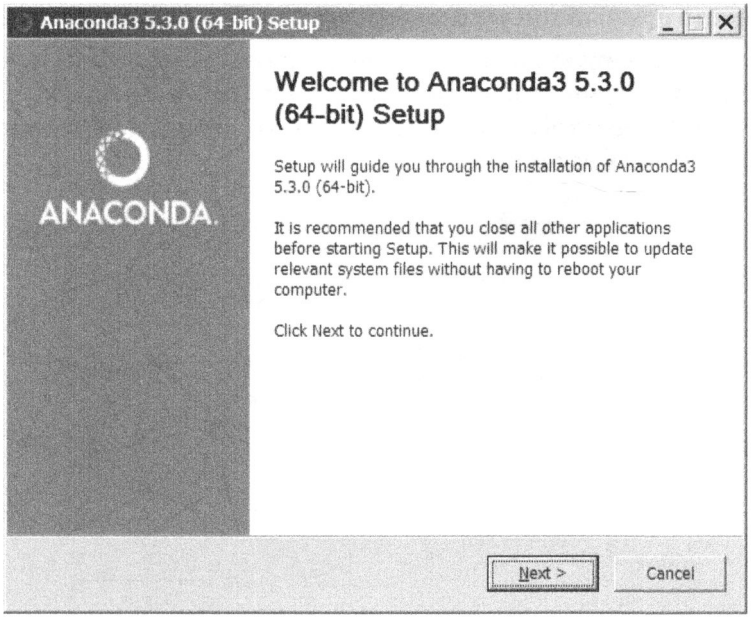


Рис. 4.1. Первое окно программы установки 64-разрядной версии Anaconda для Windows

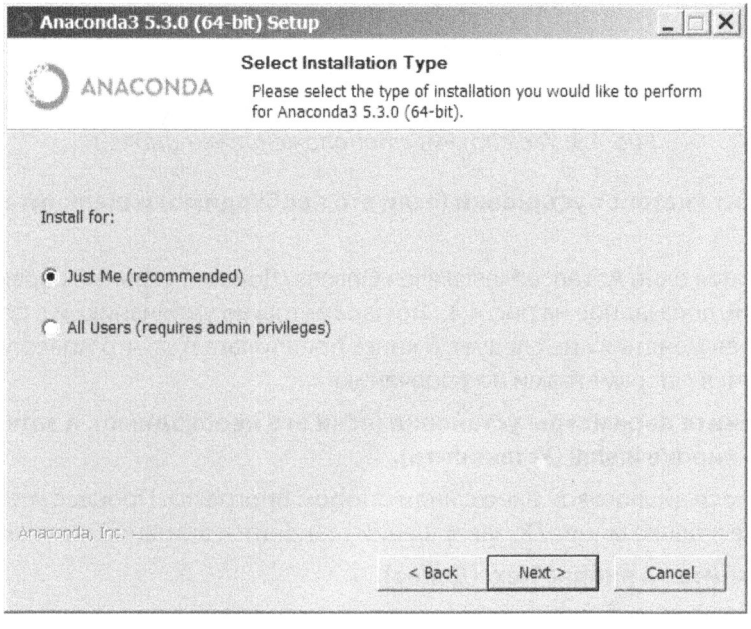


Рис. 4.2. Сообщите программе установки, каким образом следует установить Anaconda в системе

5. **Выберите нужный тип установки и щелкните на кнопке Next (Далее).**

   Мастер попросит указать каталог для установки Anaconda (рис. 4.3). В книге предполагается, что используется местоположение, выбранное по умолчанию. Если выбирается другое место, может возникнуть необходимость внести ряд изменений в некоторые примеры.

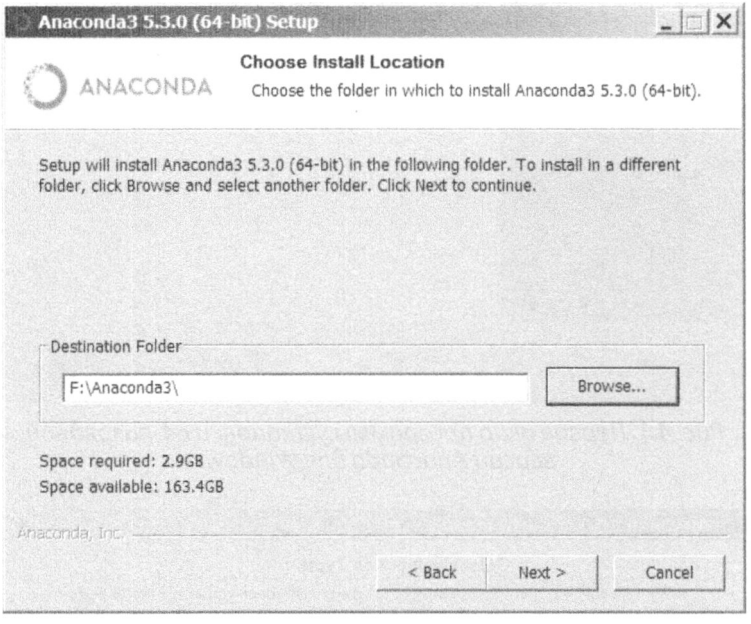


Рис. 4.3. Укажите местоположение установки

6. **Укажите каталог установки (если это необходимо) и щелкните на кнопке Next.**

   Появится окно Advanced Installation Options (Дополнительные параметры установки), показанное на рис. 4.4. Это параметры по умолчанию, и в большинстве случаев менять их не следует. В книге предполагается, что Anaconda устанавливается с параметрами по умолчанию.

7. **Измените параметры установки (если это необходимо), а затем щелкните на кнопке Install (Установить).**

   Появится диалоговое окно с индикатором прогресса. Процесс установки займет несколько минут. По завершении становится активной кнопка Next.

8. **Щелкните на кнопке Next (Далее).**

   Мастер сообщит о том, что установка завершена.

9. **Щелкните на кнопке Finish (Готово).**

   Все готово к использованию Anaconda.

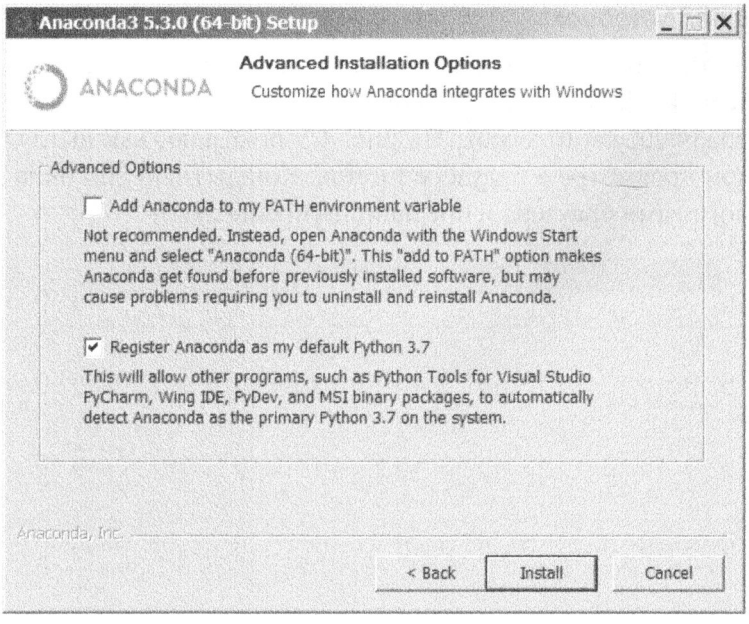


Рис. 4.4. Выбор дополнительных параметров установки

Загрузка наборов данных и файлов примеров

Книга посвящена использованию Python для решения основных задач программирования. Конечно, можно написать код с нуля, выполнить его отладку и затем попытаться понять, как все это работает, но проще скачать готовый код с сайта книги, о чем говорилось во введении, и сразу приступить к работе. В следующих разделах демонстрируется работа с Jupyter Notebook — интегрированной средой разработки Anaconda. Будут показаны различные возможности работы с кодом приложения, включая импорт загружаемого исходного кода и экспорт разработанных вами приложений.

Работа с Jupyter Notebook

Для облегчения работы с примерами в книге используется среда Jupyter Notebook. Данная среда позволяет легко создавать файлы блокнотов Python, содержащие произвольное число примеров, причем каждый из них может выполняться индивидуально. Программа выполняется в браузере, поэтому не имеет значения, какая именно платформа используется для разработки: если в системе имеется браузер, все будет в порядке.

Запуск Jupyter Notebook

Большинство систем снабжено значком, который обеспечивает доступ к Jupyter Notebook. Например, в Windows выберите команду Пуск⇨Все программы⇨Anaconda3⇨Jupyter Notebook. На рис. 4.5 показано, как выглядит Jupyter Notebook при просмотре в браузере Firefox. Конкретный вид окна зависит от используемого вами браузера и операционной системы.

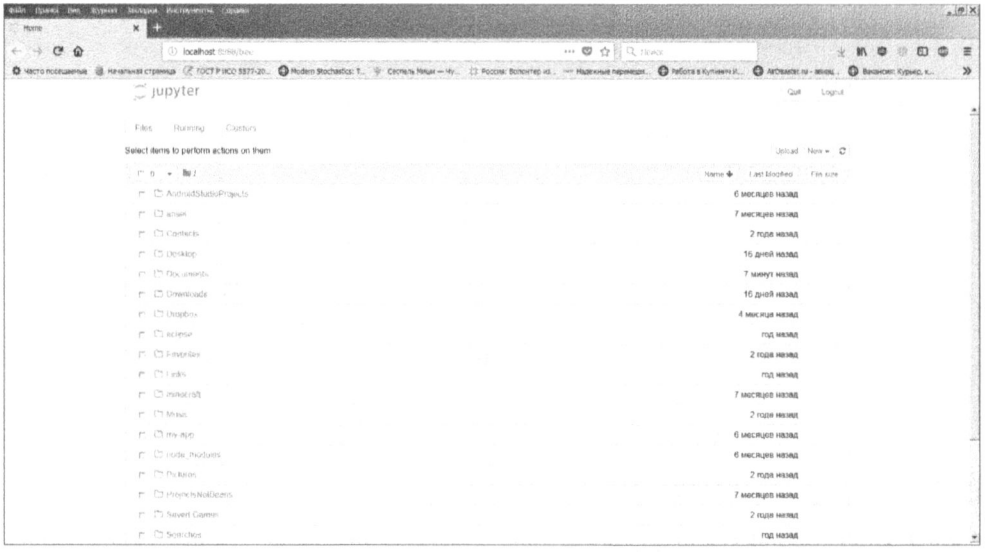


Рис. 4.5. Jupyter Notebook упрощает работу с примерами

Завершение работы сервера Jupyter Notebook

Независимо от того, как вы запустили Jupyter Notebook (в дальнейшем мы будем писать просто Notebook), для его поддержки система обычно открывает окно командной строки или терминала. В данном окне выполняется сервер, управляющий работой приложения. После закрытия окна браузера при завершении сеанса выберите окно сервера и нажмите <Ctrl+C> или <Ctrl+Break>, чтобы остановить сервер.

Создание репозитория кода

Код примеров книги располагается в репозитории, находящемся на жестком диске. Можно рассматривать *репозиторий* как своего рода шкаф, в который заносится код. Notebook открывает ящик, достает папку и показывает код пользователю. Можно изменять код, запускать отдельные примеры в папке, добавлять новые примеры и просто взаимодействовать с кодом. В следующих

разделах мы начнем работать с Notebook, чтобы познакомиться с концепцией репозитория.

Создание папки для примеров книги

Желательно организовать файлы примеров таким образом, чтобы облегчить к ним доступ в дальнейшем. Все примеры книги будут находиться в папке `BPPD` (сокращение от "Beginning Programming with Python For Dummies"). Для создания новой папки выполните следующие действия в Notebook.

1. **Выполните команду New⇨Folder (Создать⇨Папку).**

   Notebook создаст новую папку под названием `Untitled Folder` (рис. 4.6). Названия папок обычно отсортированы по алфавиту, поэтому найти нужную папку не составит особого труда. Достаточно прокрутить список к нужному месту.

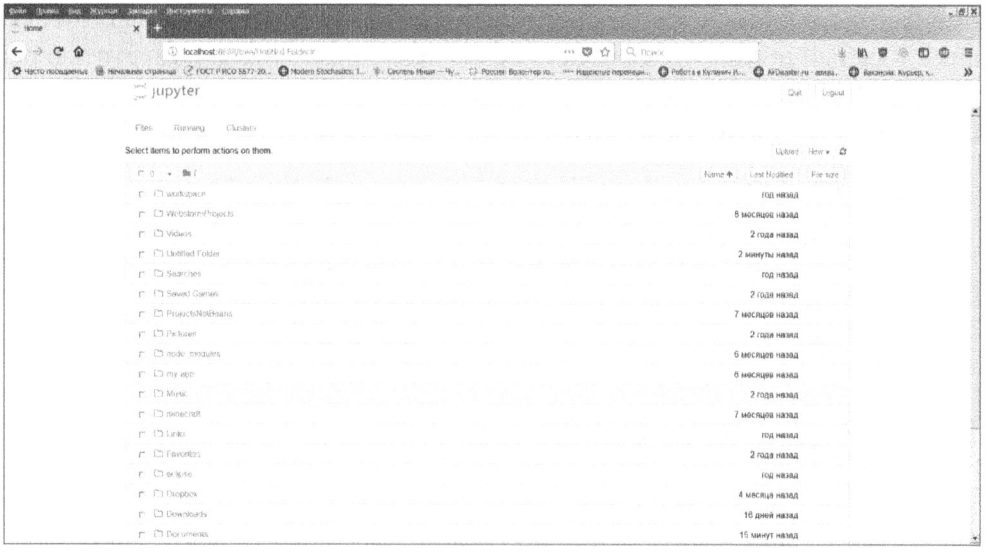


Рис. 4.6. Новая папка получает название `Untitled Folder`

2. **Установите флажок рядом с названием `Untitled Folder`.**
3. **Щелкните на кнопке Rename (Переименовать), находящейся в верхней части страницы.**

   Появится диалоговое окно Rename directory (Переименование каталога), как показано на рис. 4.7.

4. **Введите название `BPPD` и щелкните на кнопке Rename.**

   Notebook изменит название папки.

5. **Щелкните в списке на новой записи `BPPD`.**

   Вы перейдете в только что созданную папку `BPPD`.

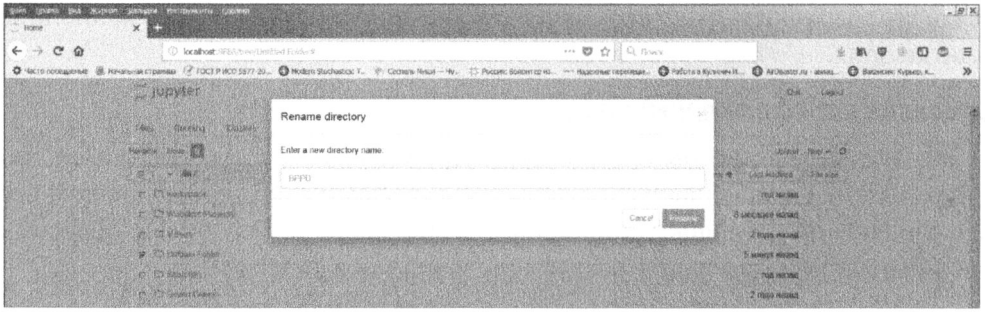


Рис. 4.7. Выбор подходящего названия для папки

Создание нового блокнота

Каждый новый блокнот напоминает папку с файлами. Отдельные примеры помещаются в папку с файлами так же, как листы бумаги — в физическую папку. Каждый пример отображается в ячейке. Также можно поместить в папку и другие файлы. По мере изучения книги вы увидите, как это происходит на практике. Чтобы создать новый блокнот, выполните следующие действия.

1. **Выполните команду New⇨Python 3 (Создать⇨Python 3).**

   В браузере откроется новая вкладка с только что созданным блокнотом (рис. 4.8). Обратите внимание на то, что блокнот содержит ячейку, которая была выделена для ввода в нее кода. Название нового блокнота — `Untitled` (Без названия). Это название не слишком информативно и подлежит изменению.

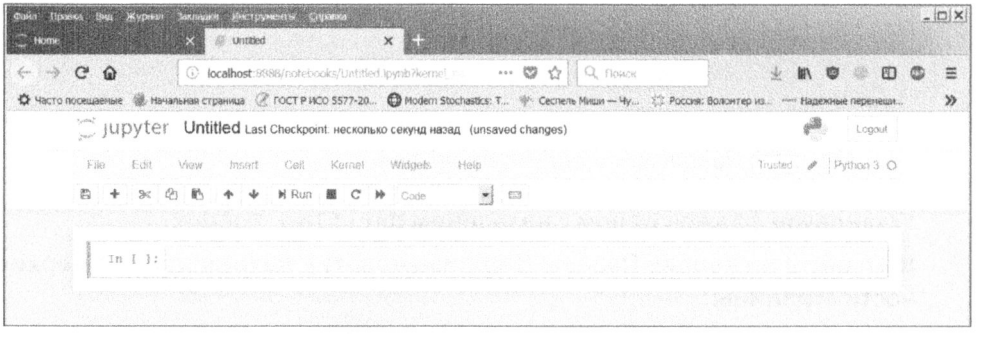


Рис. 4.8. Блокнот содержит ячейки с кодом

2. **Щелкните на названии `Untitled`.**

   Блокнот запросит ввод нового названия, как показано на рис. 4.9.

3. **Введите название `BPPD_04_Sample`.**

Новое название указывает на то, что в данной папке находятся файлы примеров из главы 4 книги *Beginning Programming with Python For Dummies*. Этот файл имеет расширение `.ipynb`. Использование подобного соглашения об именах позволяет легко различать файлы нашего репозитория.

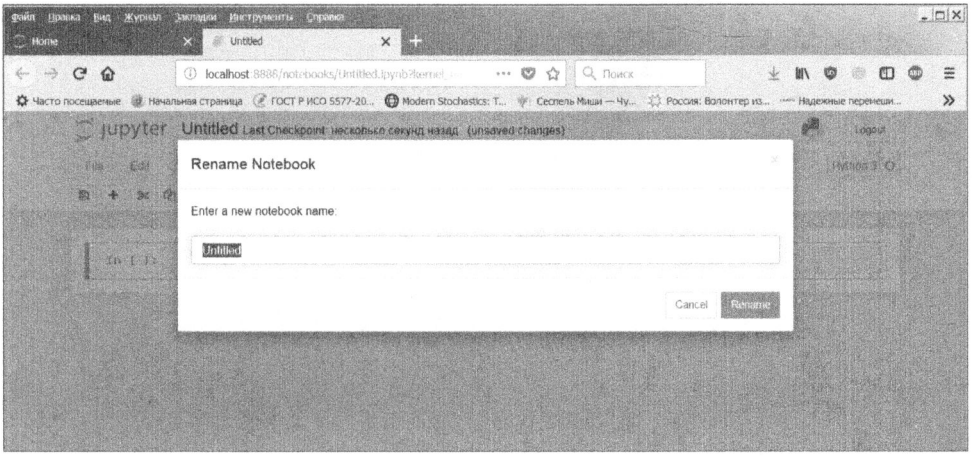


Рис. 4.9. Запрос ввода нового названия блокнота

Конечно, только что созданный блокнот пока что пуст. Поместите курсор в ячейку, введите команду **`print('Python — самая лучшая среда программирования!')`** и щелкните на кнопке Run (Выполнить). Эта кнопка обозначена указывающей вправо стрелкой и находится на панели инструментов. Отобразится результат, показанный на рис. 4.10. Результат выводится в той же ячейке, что и исходный код. Несмотря на то что код находится в прямоугольном поле, а выводимый результат — за пределами этого поля, все это относится к одной и той же ячейке. Заметьте, что блокнот визуально отделяет выходные данные от кода, поэтому можно рассматривать их отдельно. Новые ячейки блокнота создаются автоматически.

По завершении работы с блокнотом следует его закрыть. Для этого выберите команду File⇨Close and Halt (Файл⇨Закрыть и остановить). В результате вы вернетесь на начальную страницу, где заметите, что только что созданный блокнот добавлен в список, как показано на рис. 4.11.

Экспорт блокнота

Пользователи не всегда создают блокноты только для себя. Может так случиться, что вы захотите отправить блокнот другому пользователю. Для этого следует экспортировать блокнот из репозитория в файл. Далее можно отправить этот файл другому пользователю, который импортирует его в свой репозиторий.

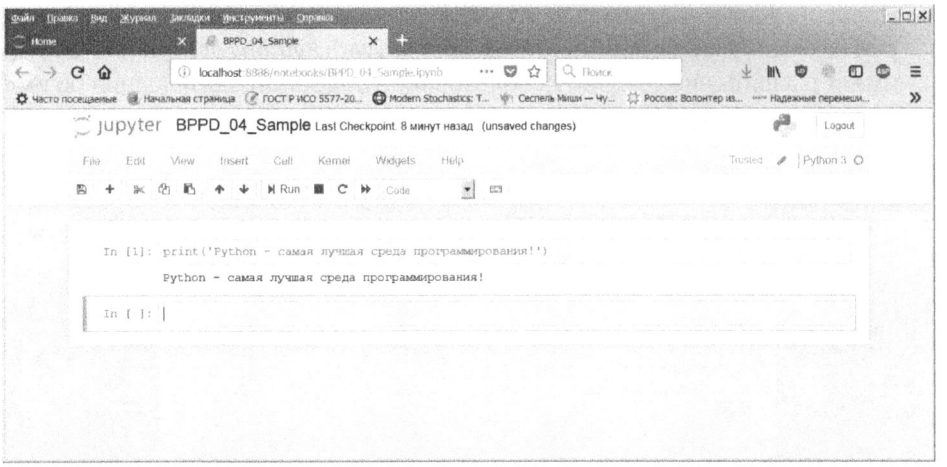


Рис. 4.10. Блокнот хранит код в ячейках

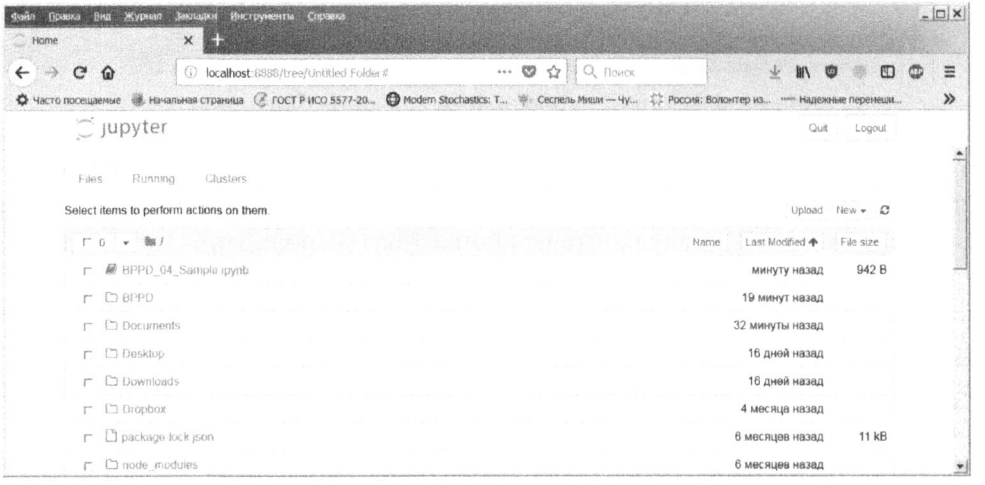


Рис. 4.11. Все созданные блокноты отображаются в списке репозитория

В предыдущем разделе мы создали блокнот `BPPD_04_Sample.ipynb`. Для открытия блокнота следует щелкнуть на его записи в списке репозитория, после чего на экране появится написанный вами код. Для экспорта этого кода выполните команду File⇨Download As⇨Notebook (.ipynb) (Файл⇨Загрузить как⇨Блокнот (.ipynb)). Последующая процедура зависит от браузера. Обычно появляется диалоговое окно с запросом на сохранение блокнота в виде файла. По окончании не забудьте закрыть приложение, выбрав команду File⇨Close and Halt (Файл⇨Закрыть и остановить).

Удаление блокнота

Иногда блокноты устаревают или просто становятся ненужными. Чтобы репозиторий не засорялся ненужными файлами, следует удалить их из списка. Для удаления файла выполните следующие действия.

1. **Установите флажок слева от записи `BPPD_04_Sample.ipynb`.**
2. **Щелкните на пиктограмме корзины (Delete) в верхней части страницы.**

   На экране появится предупреждающее сообщение об удалении блокнота (рис. 4.12).
3. **Щелкните на кнопке Delete (Удалить).**

   Файл будет удален из списка.

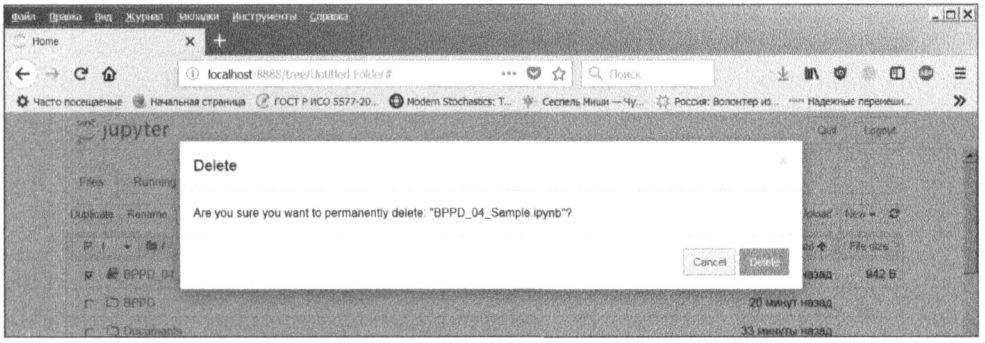


Рис. 4.12. Перед удалением файла из репозитория появляется предупреждающее сообщение

Импорт блокнота

Для работы с исходными кодами примеров книги следует импортировать загруженные файлы в репозиторий. Исходные коды содержатся в архиве, который извлекается в определенный каталог на жестком диске. В архив входят файлы `.ipynb` (IPython Notebook), содержащие исходные коды примеров книги (см. введение). Чтобы импортировать исходный код в репозиторий, выполните следующие действия.

1. **Щелкните на кнопке Upload (Загрузить), находящейся в верхней части страницы.**

   Последующая процедура зависит от применяемого браузера. В большинстве случаев появляется диалоговое окно File Upload (Загрузка файла), обеспечивающее доступ к файлам на жестком диске.
2. **Перейдите в каталог файлов, которые нужно импортировать в блокнот.**

3. **Выделите для импорта один или несколько файлов и щелкните на кнопке Open (Открыть), чтобы начать процесс загрузки.**

   Файл вносится в список загрузки, как показано на рис. 4.13. Файл еще не стал частью репозитория, а просто выбран для загрузки.

4. **Щелкните на кнопке Upload (Загрузить).**

   Notebook помещает файл в репозиторий, и теперь его можно использовать.

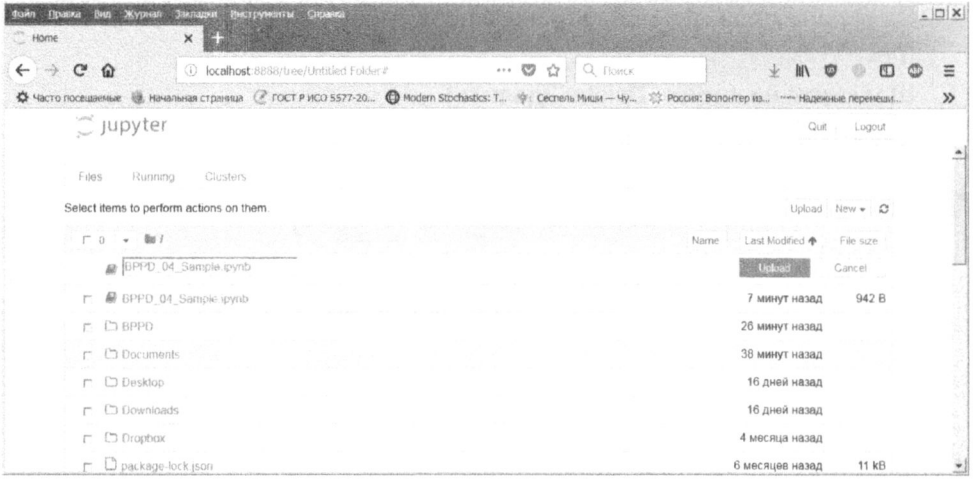


Рис. 4.13. Файлы, добавляемые в репозиторий, включаются в список загрузки

Создание приложения

Фактически, выполняя действия из раздела "Создание нового блокнота", вы создали первое приложение Anaconda. Оно содержит лишь команду `print()`, которая в том или ином виде применяется довольно часто. Однако поддерживаемая в Anaconda методология грамотного программирования требует большего объема знаний, чем у вас есть в данный момент. Следующие разделы помогут получить представление о том, насколько обширны функциональные возможности грамотного программирования. Прежде чем приступить к работе, удостоверьтесь, что файл `BPPD_04_Sample.ipynb` открыт для использования, поскольку он необходим нам для изучения возможностей приложения Notebook.

Знакомство с ячейками

Если бы приложение Notebook было стандартной средой разработки, в нем не было бы ячеек. Вместо этого в вашем распоряжении оказался бы документ, содержащий одну непрерывную последовательность инструкций. Для

структурирования элементов кода пришлось бы использовались отдельные файлы. В случае же ячеек каждая из них независима от остальных, поэтому они различаются между собой. Конечно, результаты операций, выполняемых в предыдущих ячейках, имеют значение, но если ячейка предназначена для автономного использования, можно просто перейти к ней и выполнить соответствующий код. Чтобы увидеть, как все это работает, введите следующий код в ячейку файла `BPPD_04_Sample`.

```
myVar = 3 + 4
print(myVar)
```

После этого щелкните на кнопке Run (обозначена стрелкой, указывающей вправо). В результате выполнения кода отобразится результат, показанный на рис. 4.14. В данном случае результатом будет 7, чего и следовало ожидать. Однако обратите внимание на запись `In [1]:`. Она указывает на то, что выполняется код, находящийся в первой ячейке.

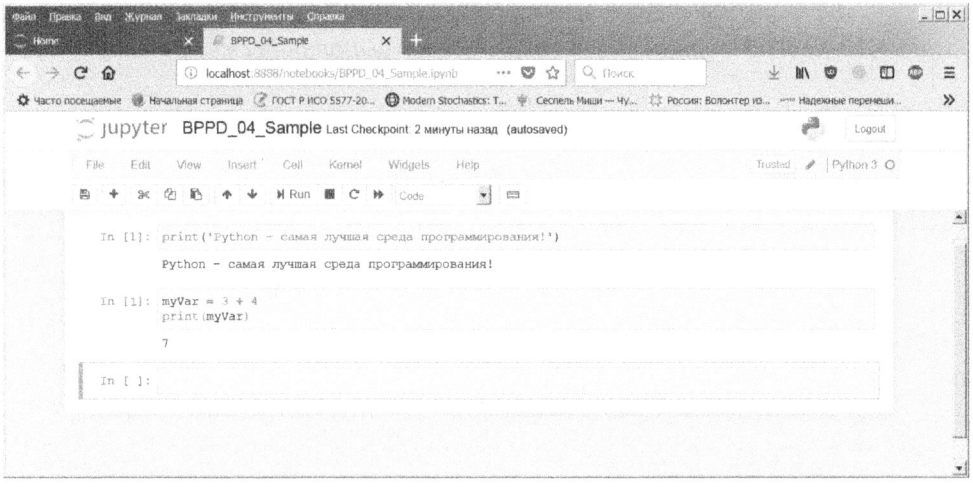


Рис. 4.14. Ячейки обрабатываются в блокноте по отдельности

Первая ячейка также помечена записью `In [1]:`. Она осталась после предыдущего сеанса. Поместите указатель мыши в данную ячейку и щелкните на кнопке Run (Выполнить). Теперь ячейка будет включать запись `In [2]:`. Но заметьте, что следующая ячейка по-прежнему содержит запись `In [1]:` (рис. 4.15).

Теперь поместите указатель мыши в третью ячейку, которая в данный момент пуста, и введите команду **`print("Переменная myVar: ", myVar)`**. Щелкните на кнопке Run (Выполнить). Показанный на рис. 4.16 результат свидетельствует о том, что код, находящийся в ячейках, выполняется не в строгом порядке, а переменная `myVar` является глобальной для блокнота. Действия,

выполняемые над данными, влияют на каждую из ячеек независимо от порядка их выполнения.

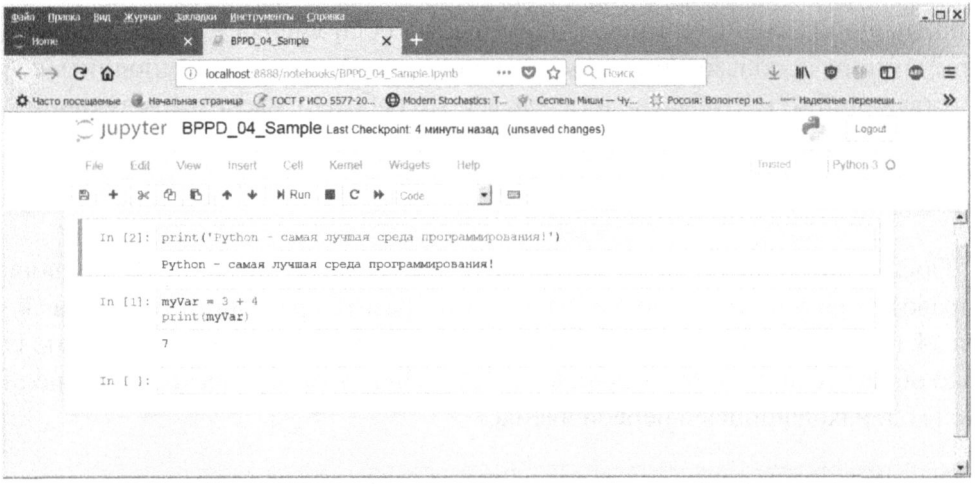


Рис. 4.15. Код, находящийся в ячейках, может выполняться в произвольном порядке

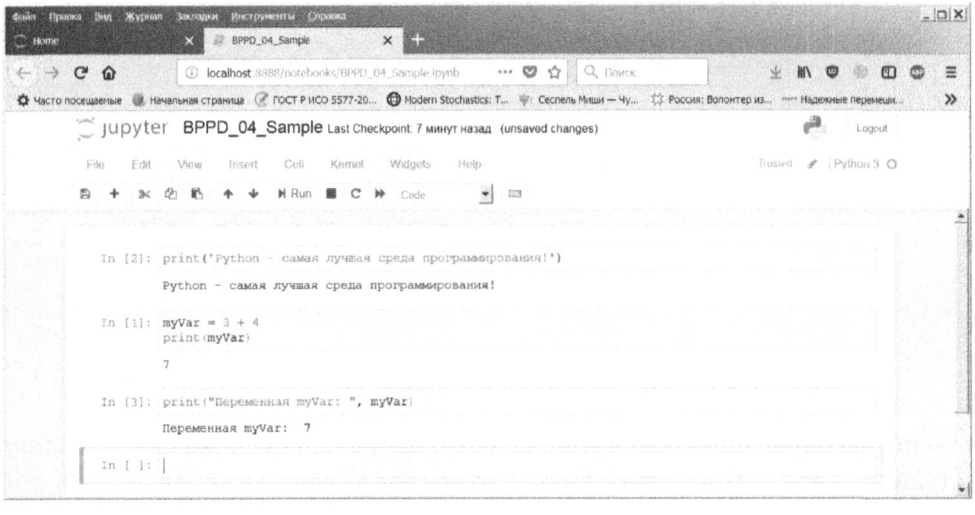


Рис. 4.16. Изменения в данных оказывают влияние на каждую ячейку, в которой используется модифицированная переменная

Добавление документирующих ячеек

Ячейки бывают разных видов, и далеко не все из них будут использоваться в книге. Тем не менее вам может пригодиться знакомство с документирующими ячейками. Выберите первую ячейку (помечена номером 2) и выполните

команду Insert⇨Insert Cell Above (Вставить⇨Вставить ячейку сверху). В блокнот будет вставлена новая ячейка. Обратите внимание на раскрывающийся список панели инструментов, в котором выбран пункт Code (Код). Данный список позволяет указывать тип создаваемой ячейки. Выберите в списке параметр Markdown (Разметка), введите название **`# Заголовок 1-го уровня.`** и щелкните на кнопке Run. В результате текст ячейки превратится в заголовок, как показано на рис. 4.17.

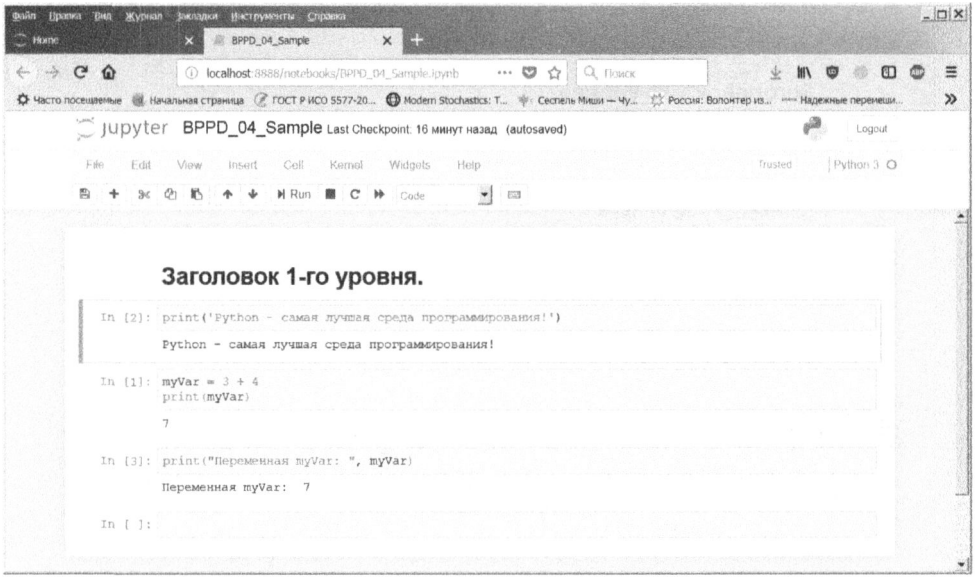


Рис. 4.17. Заголовки помогают визуально разделять и документировать код

Вы, возможно, подумали, что это напоминает HTML-разметку, и будете правы. Выполните команду Insert⇨Insert Cell Below (Вставить⇨Вставить ячейку внизу), выберите в раскрывающемся списке параметр Markdown, введите **`## Заголовок 2-го уровня.`** и щелкните на кнопке Run. Как показано на рис. 4.18, от количества начальных символов # зависит уровень заголовка, хотя в самом заголовке эти символы не отображаются.

Другое содержимое ячеек

В этой главе (и книге) рассматриваются далеко не все виды ячеек, которые можно создавать с помощью приложения Notebook. В блокнот можно добавлять всевозможные объекты, например графические изображения. Со временем вы научитесь выводить на печать содержимое блокнота в виде отчета и использовать его в ходе презентаций. Методология грамотного программирования отличается от всего, с чем вам ранее доводилось сталкиваться, и она

обладает несомненными преимуществами, о чем мы поговорим в следующих главах.

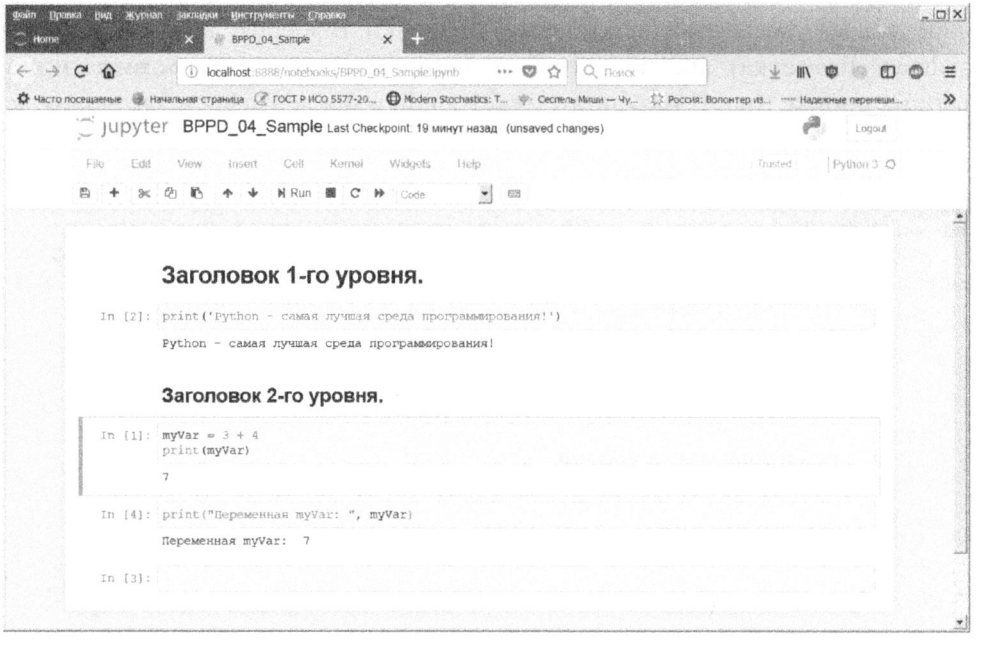


Рис. 4.18. Заголовки разных уровней позволяют структурировать содержимое ячеек

Использование отступов

Работая с примерами книги, вы заметите, что некоторые строки имеют отступы. Помните, что Python игнорирует отступы в коде. Они используются для визуальной разметки кода подобно тому, как в книгах используются отступы красной строки.

Различные способы применения отступов станут понятнее при дальнейшем изучении книги. Но сначала нужно узнать, зачем нужны отступы и как они создаются. Рассмотрим следующий пример, в котором применяется отступ, позволяющий сделать связь между элементами приложения более очевидной.

1. **Выберите команду New⇨Python3.**

   Jupyter Notebook создаст новый блокнот. Соответствующий файл исходного кода называется `BPPD_04_Indentation.ipynb` (это имя можно изменить).

2. **Введите `print("Реально длинная текстовая строка, которая будет"+`.**

Как и ожидалось, на экране отобразится обычный текст. Знак "плюс" (+) указывает Python на наличие дополнительного текста для отображения. Процесс объединения текста из нескольких строк в один фрагмент называется *конкатенацией* (об этом мы поговорим в последующих главах).

3. **Нажмите клавишу <Enter>.**

   Как и следовало ожидать, курсор не возвращается в начало строки. Вместо этого он устанавливается непосредственно под первой двойной кавычкой, как показано на рис. 4.19. В данном случае создается автоматический отступ — особенность, отличающая редактор кода от обычного текстового редактора.

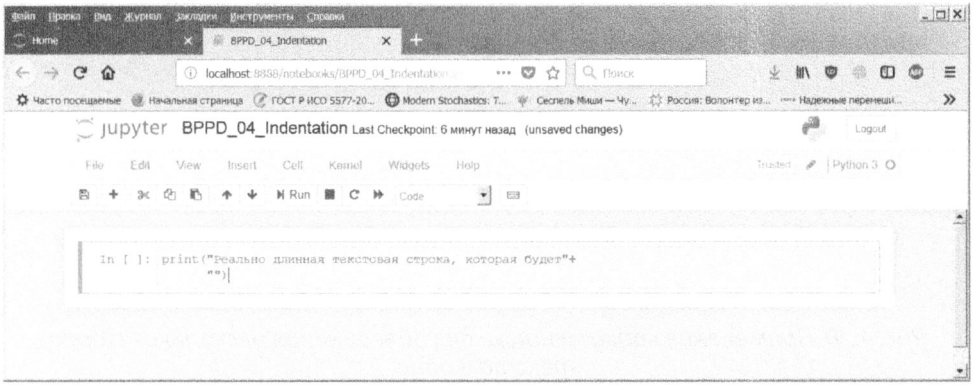


Рис. 4.19. В окне редактирования автоматически создаются необходимые отступы

4. **Введите " занимать несколько строк в файле исходного кода. ").**

   Заметьте, что курсор возвращается в начало строки. По достижении конца кода блокнот автоматически возвращает отступ в исходное положение.

5. **Щелкните на кнопке Run (Выполнить).**

   Отобразится результат, показанный на рис. 4.20. И хотя текст занимает несколько строк в файле исходного кода, программа формирует только одну строку в окне вывода. Разрыв строки появляется из-за недостаточных размеров окна; на самом деле это одна строка.

Добавление комментариев

Людям свойственно постоянно делать для себя заметки. Перед закупкой продуктов проверяется содержимое холодильника, уточняется, что нужно купить, и составляется список. Уже в магазине люди просматривают свои записи, вспоминая, что нужно купить. Заметки делают в разных ситуациях, например, во время деловых переговоров или по ходу лекции. Людям нужны заметки для

того, чтобы позднее можно было освежить воспоминания. Комментарии в исходном коде служат тем же целям. Их добавляют в код, чтобы потом быстро вспомнить, для чего он предназначен. В следующих разделах комментарии описываются более подробно. Соответствующие примеры содержатся в файле исходного кода `BPPD_04_Comments.ipynb`.

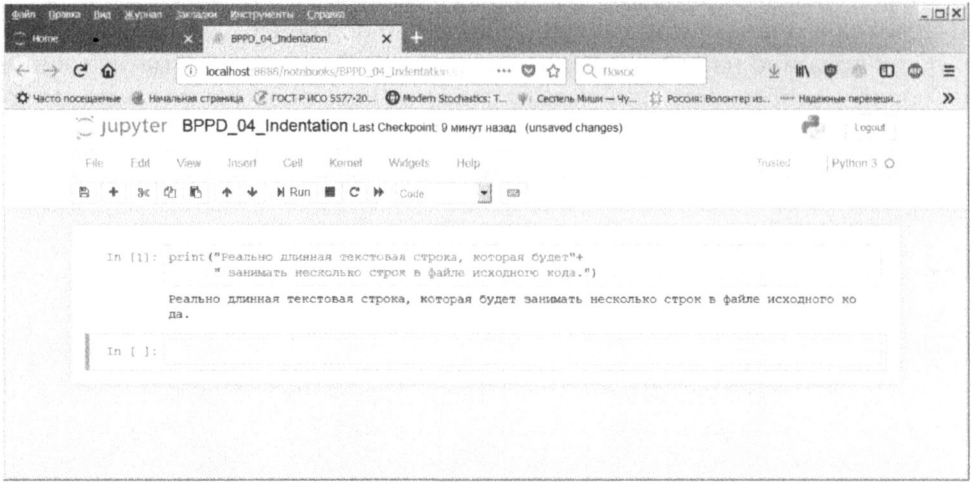


Рис. 4.20. Применение конкатенации для объединения нескольких строк текста в одну

ЗАГОЛОВКИ И КОММЕНТАРИИ

Не нужно путать заголовки и комментарии. Заголовки отображаются в отдельных ячейках, а комментарии — в исходном коде. И служат они разным целям. Заголовки описывают назначение группы строк, а комментарии уточняют назначение отдельных шагов или строк в коде. Впрочем, и то, и то используется для документирования. Комментарии, как правило, являются более многословными по сравнению с заголовками.

Знакомство с комментариями

Важно указать компьютеру, что написанный вами текст является комментарием, а не кодом, который требует выполнения. В Python поддерживаются два способа представления текста в виде комментариев. Первый способ заключается в использовании однострочного комментария. При этом используется символ #.

```
# Пример комментария.
print("Привет от Python!") # Еще один пример комментария.
```

Однострочный комментарий может занимать отдельную строку или располагаться сразу после команды. Обычно такой комментарий содержит короткое описание, например пояснение к инструкции. В окне блокнота комментарии выделяются курсивом и другим цветом (обычно синим).

Многострочные комментарии не поддерживаются в Python напрямую, но их можно сформировать с помощью строки с тремя кавычками. Многострочный комментарий начинается и заканчивается тремя двойными (""") или тремя одинарными кавычками (''''), как показано ниже.

```
"""
   Приложение: Comments.py
   Автор: Алекс
   Назначение: демонстрация применения комментариев
"""
```

Строки комментариев не являются выполняемыми инструкциями. Python ничего не сообщает при их появлении, однако Notebook трактует их по-особому, что и проиллюстрировано на рис. 4.21. Заметьте, что фактические комментарии, которым в ячейке 1 предшествует символ решетки (#), не появляются в выводе. А вот строки, заключенные в тройные кавычки, появляются. Если вы планируете выводить блокнот в виде отчета, избегайте комментариев тремя кавычками. (Некоторые IDE, например IDLE, просто игнорируют тройные кавычки.)

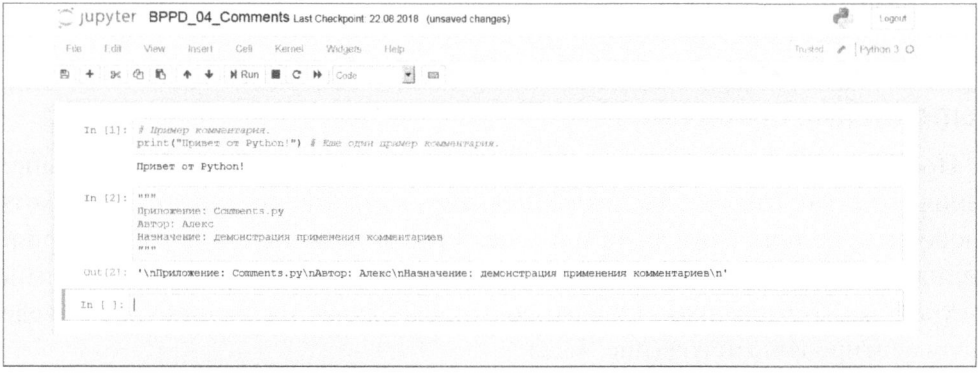


Рис. 4.21. Многострочные комментарии распознаются, но при этом отображаются в выводе

В отличие от стандартных комментариев текст, содержащийся между тройными кавычками, выделяется красным, а не синим цветом, и не выделяется курсивом. Обычно многострочные комментарии содержат информацию о том,

кто автор приложения, для чего оно было создано и какие действия оно выполняет. Разумеется, не существует строгих правил использования комментариев. Нужно просто указать компьютеру, какой текст является комментарием, а какой — не является.

Применение комментариев в качестве напоминаний

Многие пользователи не видят особой ценности в комментариях; более того, им кажется, что комментарии излишни. Просто они не учитывают, что спустя много месяцев или даже лет после написания программы будет трудно восстановить в памяти нюансы ее работы. Перечислим основные причины использования комментариев в коде:

- они позволяют вспомнить, с какой целью создавался код и какую задачу он решает;
- они содержат указания о том, как правильно запускать программу;
- они упрощают работу с кодом другим программистам;
- они содержат подсказки относительно будущих обновлений кода;
- они содержат ссылки на документацию;
- они содержат список изменений, которые были внесены в код.

Конечно, можно применять комментарии и в других целях, но указанные варианты наиболее популярны. В последующих главах вы увидите различные примеры использования комментариев. По мере усложнения кода приходится добавлять все больше и больше комментариев, чтобы не упустить из виду никаких важных моментов.

Применение комментариев для предотвращения выполнения строк кода

Программисты иногда используют комментарии, чтобы избежать выполнения кода (как говорят, *закомментировать код*). Так поступают, когда хотят проверить, не приводит ли та или иная строка кода к сбою в работе приложения. Для подобных целей можно применять как однострочные, так и многострочные комментарии. Однако во втором случае на экран будет выведен закомментированный код (рис. 4.22).

Как закрыть Jupyter Notebook

После выполнения команды File⇨Close and Halt для закрытия каждого из открытых блокнотов нужно закрыть окно браузера, чтобы завершить текущий

сеанс. Однако сервер продолжит выполняться в фоновом режиме (рис. 4.23). Окно Jupyter Notebook останется открытым до тех пор, пока вы не остановите сервер. Чтобы завершить сеанс работы с сервером, нажмите комбинацию клавиш <Ctrl+C>, после чего окно закроется.

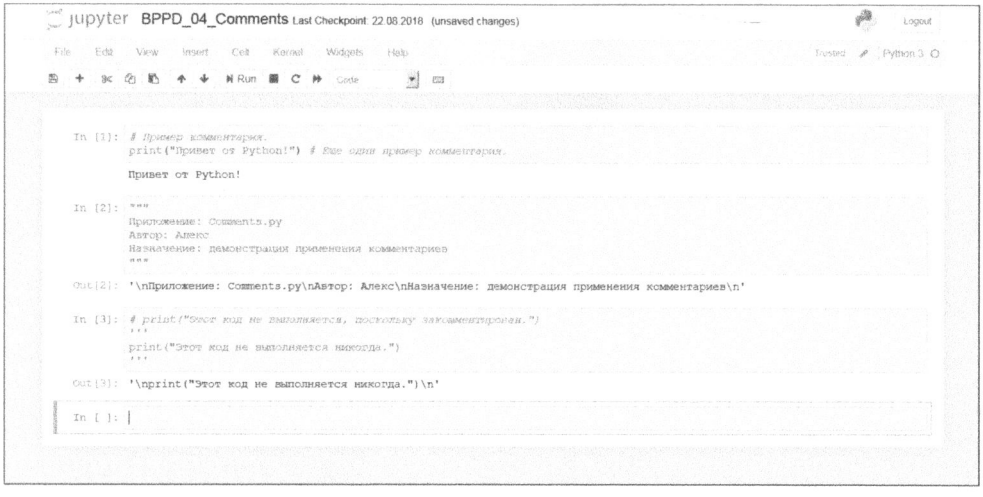


Рис. 4.22. Комментарии, предотвращающие выполнение кода

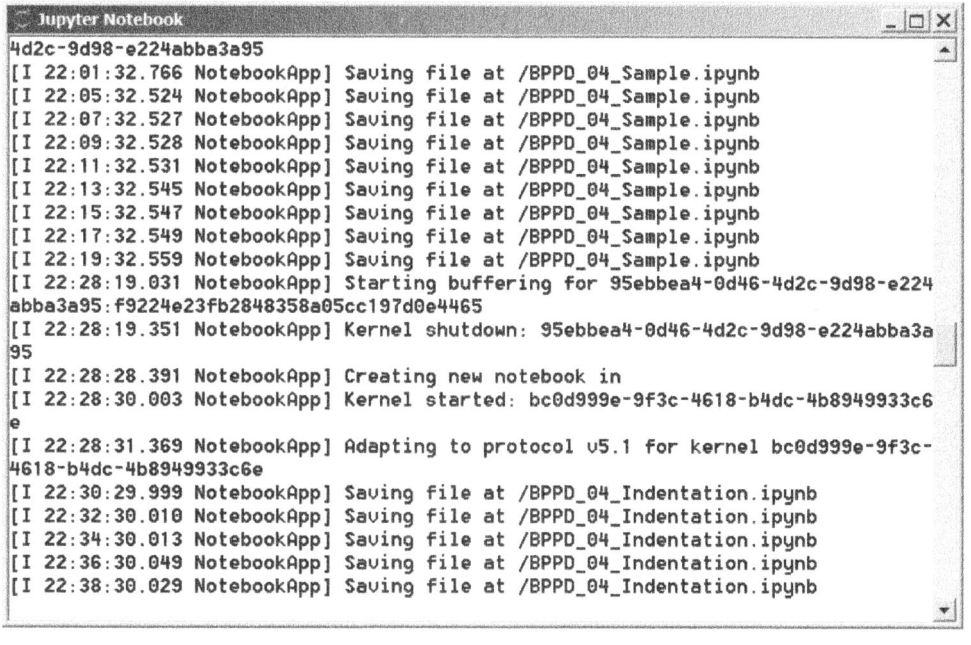


Рис. 4.23. Убедитесь, что окно сервера закрыто

Еще раз взгляните на рис. 4.23. Команды, отображаемые в данном окне, сообщают о том, какие действия выполняются в пользовательском интерфейсе. Это позволяет узнать, что происходит во время сеанса. Вряд ли вы будете часто заглядывать сюда, тем не менее полезно помнить о такой возможности.

Глава 5

Работа с Anaconda

В ЭТОЙ ГЛАВЕ...

- Работа с файлами кода
- Работа с ячейками
- Конфигурирование пользовательского интерфейса
- Изменение внешнего вида кода

В качестве интегрированной среды разработки в Anaconda используется Jupyter Notebook. Фактически все программистские задачи, рассматриваемые в книге, можно решить с помощью одной лишь этой утилиты. Вот почему в данной главе основное внимание уделяется Jupyter Notebook (в дальнейшем просто Notebook). В отличие от большинства IDE, в Notebook используется принцип грамотного программирования, описанный в главе 4. Прочитав данную главу, вы поймете, каким образом методология грамотного программирования позволяет вести продуктивную разработку приложений на Python.

По ходу знакомства с возможностями Notebook вы узнаете о том, как загружать код и как создавать контрольные точки, упрощающие процесс восстановления после ошибок. Важной частью процесса разработки является эффективная работа с файлами. В главе 4 мы рассмотрели лишь основы, ну а в данной главе мы остановимся на этой теме более подробно.

Кроме того, в главе 4 рассматривались ячейки. Благодаря этим структурам существенно упрощается процесс кодирования. В данной главе вас ожидает больше информации о них.

Мы также поговорим о том, как эффективно работать с Notebook. Например, вам может не понравиться то, как сконфигурирована рабочая среда, и вы захотите выяснить, как изменить настройки среды. Вы узнаете, как

перезапустить ядро в случае замедления или ухудшения работы Notebook и как получить справку. Кроме того, в Notebook есть ряд функций, которые иногда называются *магическими* (в силу предоставляемых ими возможностей). Они не влияют на работу кода, но влияют на способ его отображения в Notebook и на представление графики. В конце мы поговорим о том, как взаимодействовать с выполняющимися процессами. Иногда нужно выяснить, чем занят конкретный процесс, чтобы определить порядок обращения к нему.

Загрузка кода

Notebook — это особая среда разработки, основанная не на текстах программ, в отличие от других IDE. Если открыть файл IPython Notebook с расширением `.ipynb`, то можно увидеть, что данный файл содержит читабельный текст, но это не листинг программы. Для включения специальных функций, предоставляемых в Notebook, файл должен содержать дополнительную информацию, которая отсутствует в обычном текстовом файле. Соответственно, чтобы можно было открыть код в другой среде, его нужно загрузить.

ЗАПОМНИ!

В главе 4 было показано, каким образом можно экспортировать блокнот в формат, поддерживаемый программой Notebook. Можно также загружать код в других форматах, распознаваемых иными приложениями. Команда File⇨Download As (Файл⇨Загрузить как) обеспечивает доступ к следующим форматам:

- Python (`.py`);
- HTML (`.html`);
- Markdown (`.md`);
- reST (`.rst`);
- LaTeX (`.tex`);
- PDF via LaTeX (`.pdf`).

ВНИМАНИЕ!

Перечисленные форматы доступны не всегда. Например, чтобы создать файл PDF с помощью LaTeX, нужно установить модуль XeTeX. Соответствующие инструкции приведены по следующему адресу:

```
https://nbconvert.readthedocs.io/en/latest/install.
html#installingtex
```

Модуль XeTeX содержит движок рендеринга, который требуется для создания PDF-файлов.

В зависимости от конфигурации системы файлы некоторых форматов можно открывать прямо в браузере. Например, на рис. 5.1 показано, какой вид может иметь один из примеров главы 4 в формате HTML, открываемый в браузере. Все выглядит точно так же, как и при открытии файла примера в Jupyter Notebook.

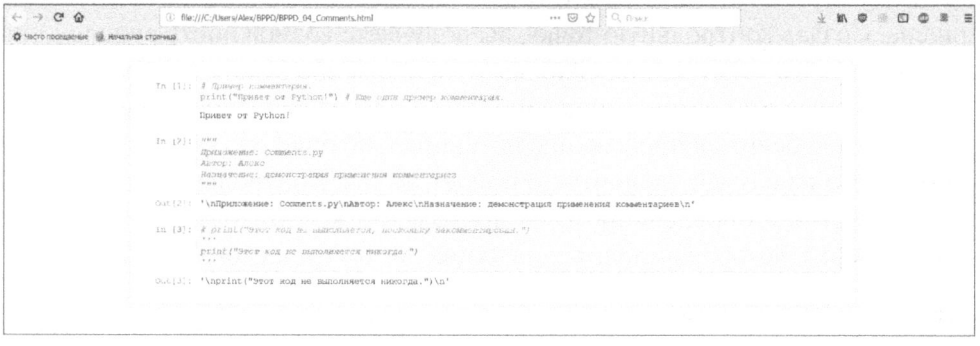


Рис. 5.1. Некоторые форматы файлов можно открывать непосредственно в браузере

Контрольные точки

Благодаря *контрольным точкам* можно сэкономить немало времени и усилий на отладку кода. Они могут применяться как для сохранения промежуточного состояния, так и для контроля кода. С помощью этого средства можно получить снимок состояния приложения в определенный момент времени.

Методика использования контрольных точек

В отличие от операции сохранения приложения контрольная точка приводит к созданию скрытого файла, который помещается в специальную папку проекта. Например, при работе с примерами книги контрольные точки можно найти в папке `\BPPD\.ipynb_checkpoints`. К контрольной точке можно вернуться впоследствии, когда потребуется восстановить предыдущее состояние приложения. Контрольные точки создаются следующим образом.

- **Автоматически.** По умолчанию Notebook автоматически создает контрольную точку каждые 120 секунд, если пользователь не изменит данный интервал с помощью магической функции `%autosave` (об этом мы поговорим в разделе "Применение магических функций").
- **Вручную.** Можно создать отдельный файл контрольной точки в ручном режиме.

Все контрольные точки сохраняются в одном файле. Таким образом, при каждом сохранении предыдущий файл перезаписывается. Это позволяет иметь резервную копию на случай повреждения исходного файла (например, во время запуска или закрытия приложения).

Контрольные точки можно создавать вручную. К примеру, в определенный момент приложение начало работать стабильно, хотя его разработка еще не завершена. Создав контрольную точку, вы получаете возможность впоследствии вернуться к этому состоянию, если вдруг приложение будет повреждено.

Контрольные точки могут пригодиться и в других ситуациях. Например, вы собираетесь добавить в приложение потенциально опасную функцию и хотите защитить его на случай, если что-то пойдет не так. Контрольные точки позволяют вернуться к определенному этапу разработки приложения. Это своего рода страховка, которая дополняет функцию автоматического сохранения.

СОВЕТ

Notebook позволяет видеть только одну контрольную точку, но при желании их можно создать несколько. Просто переименуйте имеющуюся контрольную точку, а затем создайте новую. Например, если имеется контрольная точка `BPPD_04_Comments-checkpoint.ipynb`, то перед новым сохранением переименуйте ее в `BPPD_04_Comments-checkpoint1.ipynb`. Чтобы вернуться к исходной контрольной точке, верните ее прежнее название, `*checkpoint.ipynb`.

Сохранение контрольных точек

Для сохранения контрольной точки выберите команду File⇨Save and Checkpoint (Файл⇨Сохранить файл и контрольную точку). Notebook автоматически сохранит копию имеющегося блокнота в папке `.ipynb_checkpoints` под существующим именем, к которому добавляется суффикс `-checkpoint`. Если не переименовать имеющуюся контрольную точку, то в результате ручного или автоматического сохранения существующий файл будет перезаписан. Следовательно, во всех случаях используется один и тот же файл контрольной точки, если, конечно, не переименовывать файлы в ручном режиме.

Восстановление контрольной точки

Для восстановления контрольной точки выберите команду File⇨Revert to Checkpoint (Файл⇨Вернуться к контрольной точке). Может показаться, будто команда позволяет иметь несколько контрольных точек, но в действительности всегда отображается только одна запись, включающая дату и время создания контрольной точки.

Работа с ячейками

Именно ячейки отличают Notebook от других IDE. Благодаря ячейкам можно выполнять различные операции с приложениями, которые были бы слишком трудоемкими или чреватыми ошибками в других IDE. Например, можно перемещать связанный код не построчно, а целыми фрагментами. В главе 4 было описано несколько приемов, которые удобно применять при работе с ячейками. В следующих разделах рассматриваются дополнительные способы работы с ячейками.

Добавление различных типов ячеек

В Notebook можно использовать ячейки нескольких типов. В главе 4 рассматривались два типа ячеек. Ниже кратко перечислены все типы, поддерживаемые в Notebook.

- **Code (Код).** Содержит интерпретируемый код Python, который поддерживает области ввода и вывода.
- **Markdown (Разметка).** Отображает текст документации с использованием разметки GitHub, описание которой можно найти по следующему адресу:

  `https://help.github.com/categories/writing-on-github/`

  В примерах книги ячейки типа Markdown в основном применяются для заголовков, но они могут включать любую информацию.
- **Raw NBConvert (Исходное содержимое NBConvert).** Позволяет включать в блокнот неинтерпретируемое содержимое, которое оказывает влияние на определенные виды выводимой информации, например при работе с LaTeX. В книге не используются ячейки подобного типа, поскольку формируемый ими вывод довольно специфичен. Дополнительную информацию по этой теме можно найти по следующему адресу:

  `https://ipython.org/ipython-doc/3/notebook/nbconvert.html#nbconvert`
- **Heading (Заголовок), устаревший тип.** Это старый способ выделения заголовков, который в настоящее время не используется.

Всякий раз при выполнении содержимого ячейки с помощью команды Cell⇨Run Cells (Ячейка⇨Выполнить ячейки) или Cell⇨Run Cells and Select Below (Ячейка⇨Выполнить ячейки и выбрать ниже), когда курсор находится в последней ячейке, Notebook автоматически добавляет новую ячейку. Но это не единственный способ добавления новых ячеек. Например, можно вставить

новую ячейку посередине блокнота. Для этого выполните команду Insert⇨Insert Cell Above (Вставка⇨Вставить ячейку выше) или Insert⇨Insert Cell Below (Вставка⇨Вставить ячейку ниже), в зависимости от того, вставляется ли новая ячейка выше или ниже текущей ячейки.

Разделение и объединение ячеек

В Notebook ячейки обрабатываются как отдельные объекты. Любые действия, связанные с содержимым ячейки, оказывают влияние на приложение в целом. Причем обращаться к содержимому отдельной ячейки можно, не изменяя содержимого других ячеек. Более того, можно выполнять ячейки в любом порядке, обращаясь к одним ячейкам чаще, чем к другим. Поэтому следует продумывать структуру ячеек: необходимо определить, является ли ячейка достаточно независимой и в то же время функциональной для решения требуемой задачи. С учетом этого приходится разделять или, наоборот, объединять ячейки.

Разделение ячейки означает создание двух ячеек на основе имеющейся одной. Разделение произойдет внутри ячейки в месте текущего расположения курсора. Для разделения выполните команду Edit⇨Split Cell (Правка⇨Разделить ячейку).

Объединение ячеек означает формирование одной ячейки на основе двух. Ячейки объединяются в том же порядке, в каком отображаются в блокноте, поэтому еще до объединения следует убедиться, что порядок их расположения правильный. Для объединения ячеек выполните команду Edit⇨Merge Cell Above (Правка⇨Объединить с ячейкой выше) или Edit⇨Merge Cell Below (Правка⇨Объединить с ячейкой ниже).

Перемещение ячеек

Порядок расположения ячеек можно изменять. Чтобы переместить ячейку, проще всего выделить ее и воспользоваться командой Edit⇨Move Cell Up (Правка⇨Переместить ячейку вверх) или Edit⇨Move Cell Down (Правка⇨Переместить ячейку вниз). Но иногда требуется выполнить и другие действия с ячейками.

Как и большинство редакторов, Notebook располагает полным набором команд редактирования. Все они доступны в меню Edit (Правка). Ниже описаны команды редактирования, поддерживаемые в Notebook в дополнение к стандартным командам по перемещению ячеек.

- **Cut Cells (Вырезать ячейки).** Удаляет выделенную ячейку, помещая ее в буфер обмена для дальнейшего использования.
- **Copy Cells (Копировать ячейки).** Помещает копию выделенной ячейки в буфер обмена, не удаляя ее.

- **Paste Cells Above (Вставить ячейки выше).** Вставляет копию ячейки из буфера обмена выше выделенной ячейки.
- **Paste Cells Below (Вставить ячейки ниже).** Вставляет копию ячейки из буфера обмена ниже выделенной ячейки.
- **Delete Cells (Удалить ячейки).** Удаляет выделенную ячейку без создания копии.
- **Undo Delete Cells (Отменить удаление ячеек).** Возвращает удаленную ячейку в блокнот (имеется лишь один уровень отмены удаления, поэтому данную команду следует вызвать сразу же после удаления).

Выполнение содержимого ячеек

Чтобы увидеть результат интерпретации ячейки, даже если это ячейка разметки, необходимо выполнить ее содержимое. Самый простой способ — щелкнуть на кнопке Run Cell (Выполнить ячейку), находящейся на панели инструментов (помечена направленной вправо стрелкой). Если же вы не хотите выполнять содержимое ячейки стандартным способом, то выберите один из пунктов в меню Cell (Ячейка).

- **Run Cells (Выполнить ячейки).** Выполнение выделенной ячейки без изменения текущего выделения.
- **Run Cells and Select Below (Выполнить ячейки и выбрать ниже), вариант по умолчанию.** Выполнение выделенной ячейки с последующим выделением следующей ячейки. Если же выделенная ячейка является последней, то Notebook создает новую ячейку для выделения.
- **Run Cells and Insert Below (Выполнить ячейки и вставить ниже).** Выполнение выделенной ячейки с последующей вставкой новой ячейки ниже текущей. Данная команда позволяет добавлять новые ячейки в середину блокнота, поскольку новая ячейка создается, даже если выделенная ячейка не является последней.
- **Run All (Выполнить все).** Выполнение всех ячеек в блокноте сверху вниз. Когда Notebook доходит до конца, выделяется последняя ячейка, но не вставляется новая.
- **Run All Above (Выполнить все выше).** Выполнение начинается с текущей ячейки и продолжается в обратном порядке для всех ячеек, расположенных выше. Данная команда отсутствует в большинстве IDE.
- **Run All Below (Выполнить все ниже).** Выполнение начинается с текущей ячейки и продолжается до конца блокнота. Когда Notebook

обработает все ячейки, выделяется последняя ячейка, но не вставляется новая.

ДОВЕРЯЙТЕ СВОЕМУ БЛОКНОТУ

ВНИМАНИЕ!

В правом верхнем углу окна Notebook можно заметить небольшой квадратик со словами *Not Trusted* (Не доверено). Чаще всего работа с Python не зависит от уровня доверия к Notebook. Однако при использовании некоторых локальных или защищенных ресурсов на сайте может потребоваться установить определенный уровень доверия для Notebook.

Самый быстрый и простой способ настройки уровня доверия — щелкнуть на кнопке Not Trusted. Появится диалоговое окно, в котором можно включить режим доверия к Notebook. К сожалению, это может вызвать проблемы с безопасностью, поэтому поступайте так, только если уверены, что можете доверять исходному коду.

В Anaconda имеются другие способы обеспечения безопасного доступа к защищенным ресурсам. Впрочем, эта тема выходит за рамки книги. Дополнительные сведения доступны по следующему адресу:

`https://jupyter-notebook.readthedocs.io/en/latest/security.html`

Для выполнения примеров книги режим доверия не требуется, поэтому кнопку Not Trusted можно игнорировать.

Отключение вывода

Чаще всего нужно видеть результаты выполнения ячейки, но иногда это просто мешает. Кроме того, бывают ситуации, когда нужно начать с чистого вывода, поэтому старую информацию необходимо стереть. В меню Cell⇨Current Outputs (Ячейка⇨Текущие выводимые данные) находятся команды, относящиеся лишь к выделенной ячейке, тогда как меню Cell⇨All Output (Ячейка⇨Все выводимые данные) содержит команды, управляющие выводом данных из всех ячеек блокнота. Ниже перечислены команды, контролирующие вывод информации.

- **Toggle (Переключение).** Включение или отключение вывода на основе предыдущего состояния. При этом информация выводится в полном объеме.
- **Toggle Scrolling (Прокрутка).** Сокращение вывода до принятого по умолчанию числа строк. Вы увидите достаточно, чтобы понять, как отработала ячейка, но детали останутся скрытыми.

» **Clear (Очистить).** Удаление текущих выводимых данных. После вызова этой команды нужно повторно выполнить ячейку, чтобы сгенерировать новые результаты.

Изменение внешнего вида Jupyter Notebook

Внешний вид Notebook можно изменить. Конечно, по сравнению с другими IDE возможности здесь ограничены, но предлагаемых настроек вполне достаточно. Все они доступны в меню View (Вид).

» **Toggle Header (Заголовок).** Заголовок в верхней части рабочей области содержит название блокнота. (О том, как поменять название блокнота, рассказывалось в главе 4.) Кроме того, если щелкнуть на находящейся в левом верхнем углу ссылке Jupyter, можно будет получить доступ к панели управления блокнота. Также в заголовке отображается текущий статус сохранения и содержится кнопка Logout (Выход), предназначенная для выхода из Notebook.

» **Toggle Toolbar (Панель инструментов).** Панель инструментов содержит ряд кнопок для быстрого выполнения команд. Эти кнопки описаны ниже в порядке расположения слева направо.

- **Save and Checkpoint (Сохранить файл и создать контрольную точку).** Сохраняет текущий блокнот и создает для него контрольную точку.
- **Insert Cell Below (Вставить ячейку ниже).** Добавляет новую ячейку ниже выделенной ячейки.
- **Cut Selected Cells (Вырезать выделенные ячейки).** Удаляет текущую ячейку, занося ее в буфер обмена.
- **Copy Selected Cells (Копировать выделенные ячейки).** Создает копию текущей ячейки в буфере обмена, не удаляя ее.
- **Paste Cells Below (Вставить ячейки ниже).** Вставляет ячейку из буфера обмена ниже текущей ячейки.
- **Move Selected Cells Up (Переместить выбранные ячейки вверх).** Перемещает выделенные ячейки на одну позицию вверх.
- **Move Selected Cells Down (Переместить выделенные ячейки вниз).** Перемещает выделенные ячейки на одну позицию вниз.
- **Run Cell (Выполнить ячейку).** Выполняет содержимое текущей ячейки и выбирает следующую ячейку. Если это последняя ячейка, Notebook создает под ней новую ячейку. Ячейки типа Raw NBConvert не интерпретируются; в их случае ничего не произойдет.

- **Interrupt Kernel (Прервать ядро).** Прекращение выполнения инструкций ядром в текущей ячейке.
- **Restart the Kernel (Перезапуск ядра).** Останов и повторный запуск ядра. В результате будут потеряны значения всех переменных.
- **Cell Type Selection (Выбор типа ячейки)**. Это раскрывающийся список, в котором можно выбрать тип ячейки (см. раздел "Добавление различных типов ячеек").
- **Open the Command Palette (Открыть палитру команд)**. Отображает диалоговое окно Command Palette (Палитра команд), в котором можно искать команды. Это окно более подробно рассматривается в следующем разделе.

» **Toggle Line Numbers (Номера строк).** Добавляет или удаляет нумерацию строк в листингах кодов. Данная настройка не оказывает влияния на другие типы ячеек. Она будет более подробно рассмотрена в разделе "Отображение номеров строк".

» **Cell Toolbar (Панель инструментов ячейки).** Позволяет добавлять команды на панель инструментов ячейки. Эти команды помогают определенным образом взаимодействовать с ячейками. Данная функция более подробно рассматривается в разделе "Панель инструментов ячейки".

Поиск команд в окне Command Palette

Диалоговое окно Command Palette позволяет получить доступ к списку команд, поддерживаемых в Notebook. Щелкните на значке Open the Command Palette (Открыть палитру команд), находящемся на панели инструментов, чтобы открыть список (рис. 5.2).

Для поиска нужной команды начните вводить фразу, которая, по вашему мнению, характеризует команду. Например, чтобы отобразить все связанные с ячейкой команды, введите слово `cell`. После обнаружения нужной команды щелкните на ней, чтобы выполнить ее.

Отображение номеров строк

С длинными листингами бывает неудобно работать, и возможность нумеровать строки оказывается весьма кстати. Для вывода номеров строк выполните команду View⇨Toggle Line Numbers (Вид⇨Номера строк). В результате все строки кода, введенные в ячейки, будут пронумерованы, как показано на рис. 5.3. Заметьте, что строки вывода не нумеруются.

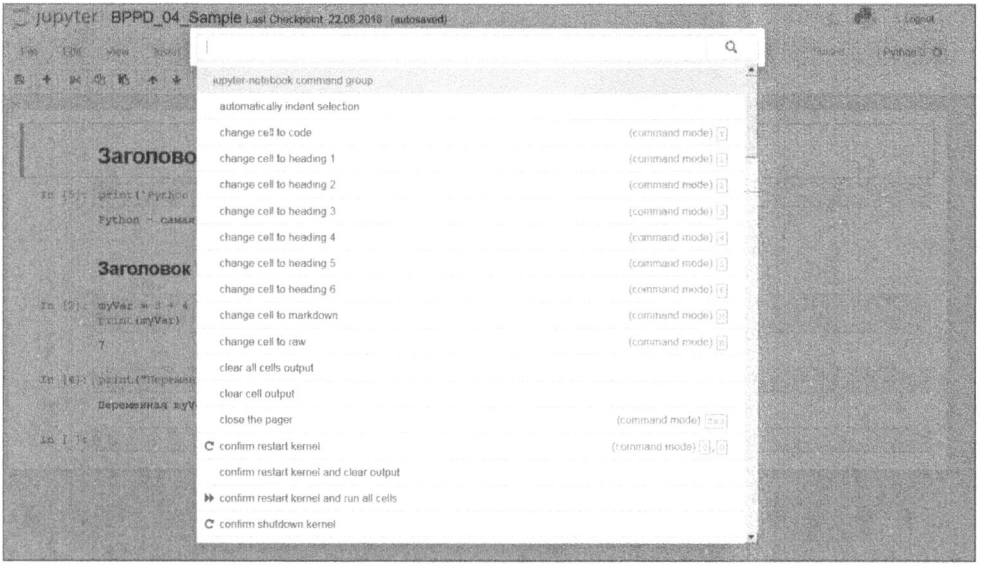


Рис. 5.2. Воспользуйтесь диалоговым окном Command Palette для поиска необходимых команд

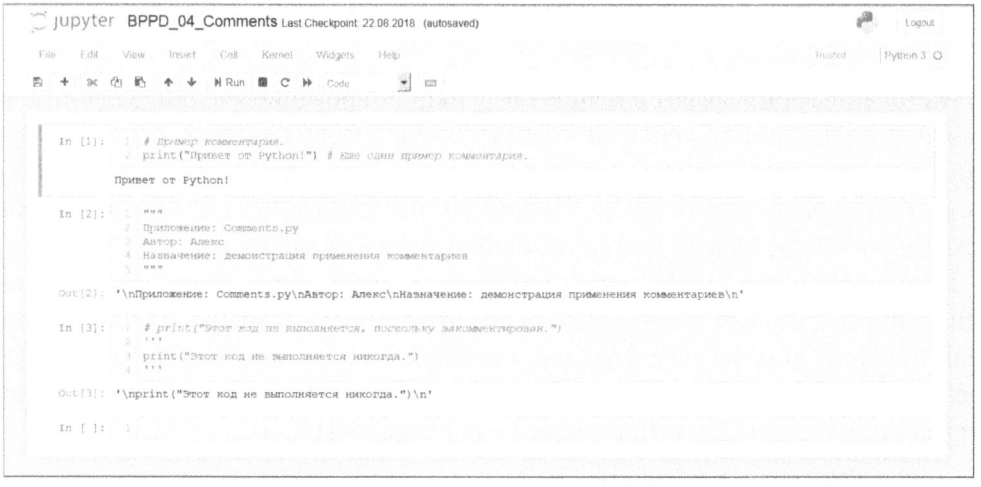


Рис. 5.3. Номера строк значительно облегчают совместную работу над проектом

Панель инструментов ячейки

У каждой ячейки есть определенные функциональные возможности. Чтобы получить к ним доступ, нужно добавить кнопку на панель инструментов ячейки с помощью команд меню View⇨Cell Toolbar (Вид⇨Панель инструментов

ячейки). На рис. 5.4 показана ячейка с кнопкой Edit Metadata (Изменить метаданные).

Рис. 5.4. Кнопки, находящиеся на панели инструментов ячейки, позволяют менять содержимое ячейки

Метаданные оказывают влияние на функционирование ячейки. Заданные по умолчанию настройки управляют как уровнем доверия ячейки, так и прокруткой длинного содержимого. Некоторые настройки оказывают влияние лишь на ячейки определенных типов. Например, настройка Raw Cell Format (Формат исходной ячейки) влияет только на ячейки типа Raw NBConvert.

На панели инструментов ячейки может одновременно отображаться только одна кнопка. Соответственно, нельзя сконфигурировать ячейку для слайд-шоу, если к ней добавлены теги. Придется выбрать что-то одно. Команда View⇨Cell Toolbar⇨None (Вид⇨Панель инструментов ячейки⇨Нет) очищает панель инструментов ячейки. Доступные кнопки перечислены ниже.

- **Edit Metadata (Изменить метаданные).** Позволяет настраивать функциональные свойства ячеек, используя как стандартные, так и пользовательские метаданные.
- **Raw Cell Format (Исходный тип ячейки).** Позволяет выбрать тип данных, хранящихся в ячейках типа Raw NTConvert. Среди доступных вариантов — None, LaTeX, reST, HTML, Markdown, Python и Custom.

» **Slideshow (Слайд-шоу).** Позволяет задать содержащийся в ячейке тип слайда. Доступны следующие варианты: Slide (Слайд), Sub-slide (Подслайд), Fragment (Фрагмент), Skip (Пропустить) и Notes (Заметки).

» **Attachments (Вложения).** Отображает список вложений текущей ячейки. Например, в ячейки типа Markdown можно добавлять изображения или картинки.

СОВЕТ

» **Tags (Теги).** Позволяет управлять тегами, связанными с каждой ячейкой. *Тег* — это информационная метка, которая проясняет особенности ячейки и помогает ее классифицировать. Теги предназначены для пользователя; для Notebook они ничего не означают. Их нужно использовать согласованно, чтобы от них была польза.

Взаимодействие с ядром

Ядро — это сервер, который позволяет выполнять содержимое ячеек в Notebook. Обычно команды ядра отображаются в отдельном окне терминала, как показано на рис. 5.5.

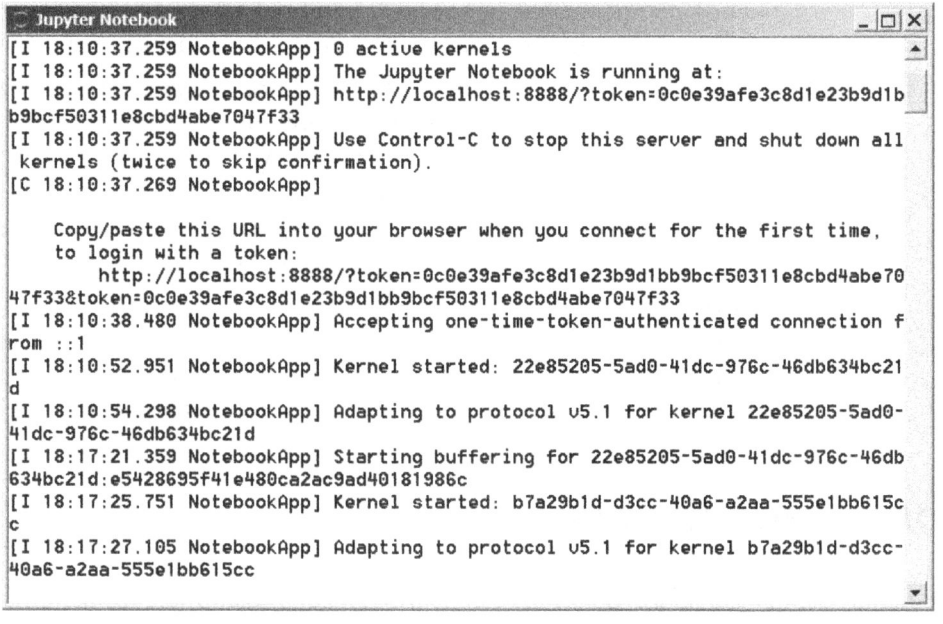


Рис. 5.5. Команды ядра в отдельном окне Jupyter Notebook

Каждая запись содержит следующие сведения: время выполнения задачи ядром, название приложения, которое вызвало данную команду, выполняемая задача и использованные ресурсы. В большинстве случаев с этим окном не

выполняют никаких действий, но оно может оказаться полезным при возникновении ошибок, поскольку отображаемые в нем сообщения помогают идентифицировать проблему.

Управлять ядром можно по-разному. Например, при сохранении файла посылается команда ядру, которое выполняет соответствующую операцию. Ниже перечислены связанные с ядром команды, находящиеся в меню Kernel (Ядро).

- **Interrupt (Прервать).** Ядро прекращает выполнение текущей задачи, но само продолжает выполняться. Данную команду удобно использовать, если нужно приостановить обработку большого набора данных.
- **Restart (Перезапустить).** Выполняется останов и перезапуск ядра. В результате будут потеряны значения всех переменных. Иногда это именно то, что нужно.
- **Restart & Clear Output (Перезапустить и очистить вывод).** Выполняется останов и перезапуск ядра, а также очистка всех выходных данных ячеек.
- **Restart & Run All (Перезапустить и выполнить все).** Выполняется останов и перезапуск ядра, после чего выполняются все ячейки от первой до последней. Как только Notebook достигнет конца, будет выбрана последняя ячейка, но не произойдет вставка новой ячейки.
- **Reconnect (Подключиться повторно).** Восстанавливает подключение к ядру. Иногда по тем или иным причинам подключение пропадает, и данная команда позволяет восстановить подключение без потери значений переменных.
- **Shutdown (Закрыть).** Закрытие ядра. Эту команду удобно применять при подготовке к использованию другого ядра.
- **Change Kernel (Сменить ядро).** Выбор другого ядра из списка установленных ядер. Например, можно тестировать приложение, используя различные версии Python, что позволит убедиться в корректности его функционирования на разных платформах.

Получение справки

Справочная система в Notebook обеспечивает определенный уровень интерактивности. Например, при выборе команды Help⇨User Interface Tour (Справка⇨Обзор интерфейса пользователя) мастер пройдет по различным элементам текущего блокнота и сообщит об их функциональных возможностях. Это позволит понять назначение каждого элемента.

Команда Help⇨Keyboard Shortcuts (Справка⇨Комбинации клавиш) отображает перечень комбинаций клавиш. Для перехода в командный режим нажмите клавишу <Esc> и введите необходимые для выполнения команды данные. Например, комбинация клавиш Esc⇨F отображает диалоговое окно Find and Replace (Найти и заменить). Если выбрать команду Help⇨Edit Keyboard Shortcuts (Справка⇨Изменить комбинации клавиш), появится диалоговое окно Edit Command Mode Shortcuts (Изменить комбинации клавиш командного режима), показанное на рис. 5.6. Оно служит для изменения способа работы Notebook в командном режиме.

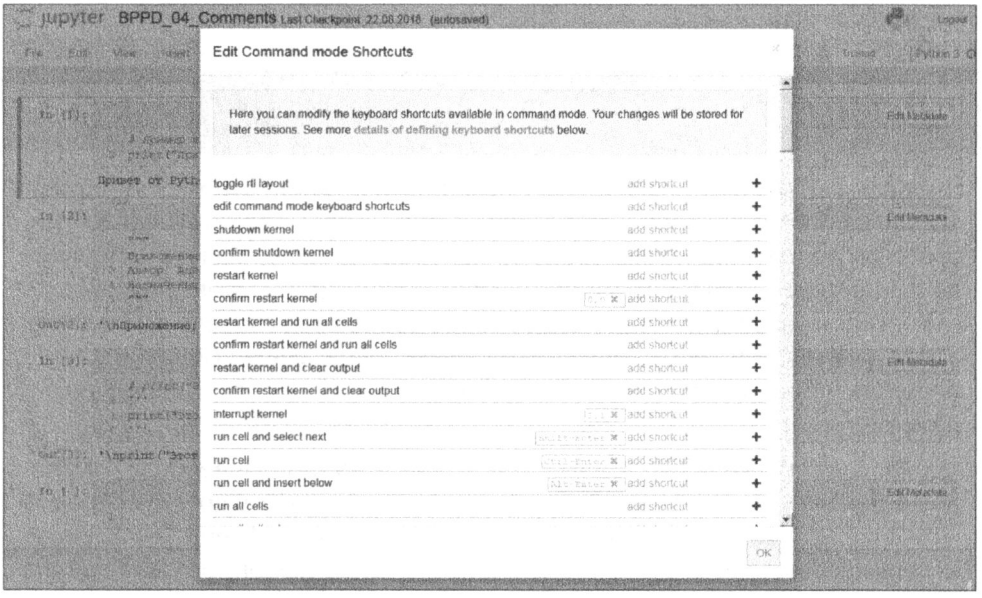


Рис. 5.6. Изменение комбинаций клавиш в командном режиме

Меню Help включает еще две команды. Команда Notebook Help позволяет получить справку в Интернете по следующему адресу:

```
http://nbviewer.jupyter.org/github/ipython/ipython/blob/3.x/
examples/Notebook/Index.ipynb
```

Данный сайт включает учебные пособия и другие сведения о том, как эффективно применять Notebook.

Команда Markdown переадресует вас на следующий сайт:

```
https://help.github.com/articles/
getting-started-with-writing-and-formatting-on-github/
```

На этом сайте находится информация о форматировании содержимого ячеек типа Markdown.

СОВЕТ

В нижней части меню Help отображается традиционная команда About. После ее выбора появляется диалоговое окно со сведениями о текущей установке. Самые важные сведения — это используемые версии Python и Anaconda. Данная информация может понадобиться для получения помощи от других пользователей Anaconda.

Остальные пункты меню Help зависят от того, какие модули установлены в данный момент. Они связаны с конкретными элементами Python (начиная с версии самого языка Python). Также отображаются все популярные библиотеки, такие как NumPy и SciPy. Эти пункты справки помогают получать помощь в процессе создания кода.

НЕМНОГО ОБ АЛЬТЕРНАТИВЕ IPYTHON

В главах 1–3 рассказывалось о применении командной строки Python. Командная оболочка IPython имеет такой же вид, но характеризуется рядом интересных дополнений. Наиболее заметной особенностью является использование цветового кодирования для вводимого кода (тем самым уменьшается вероятность ошибок). Например, команды отображаются зеленым цветом, а текст — желтым. Также выделяются цветом совпадающие скобки и другие блочные элементы, что позволяет видеть, какой именно блок закрывается в данный момент.

Справочная система представляет собой еще одно различие между командной строкой Python и IPython. Вы получаете дополнительный доступ к справке, и сама справка более подробная. Одной из интересных особенностей является использование вопросительного знака после имени любого объекта Python. Например, если ввести **`print?`**, то можно получить краткий обзор возможностей команды `print()`. Если ввести `?`, то вы получите справку по IPython.

В отличие от командной строки Python, IPython поддерживает многие дополнительные возможности Notebook, например магические функции, которые рассматриваются далее. Эти специальные функции позволяют, в частности, изменять вид выводимых в IPython и Notebook данных. Короче говоря, если у вас часто возникает необходимость в использовании командной строки, работайте с IPython.

Применение магических функций

Notebook и его аналог IPython располагают специальными функциями, которые называются *магическими*. Конечно, магия тут ни при чем, просто эти

функции действительно творят чудеса с выходными данными. Например, графический вывод, как по мановению волшебной палочки, можно отобразить не в отдельном окне, а в ячейке (обычно ячейки содержат только текст). Или же можно быстро проверить производительность приложения без написания дополнительного кода, который обычно требуется в подобной ситуации.

Магическая функция предваряется знаком `%` либо `%%`. Функции со знаком `%` действуют в масштабах рабочей среды, а функции со знаком `%%` применяются на уровне ячейки. Для получения списка магических функций введите **`%lsmagic`** в IPython или Notebook. В результате отобразится список, показанный на рис. 5.7. (Заметьте, что в IPython применяются те же подсказки для ввода, `In`, и вывода, `Out`, что и в Notebook.)

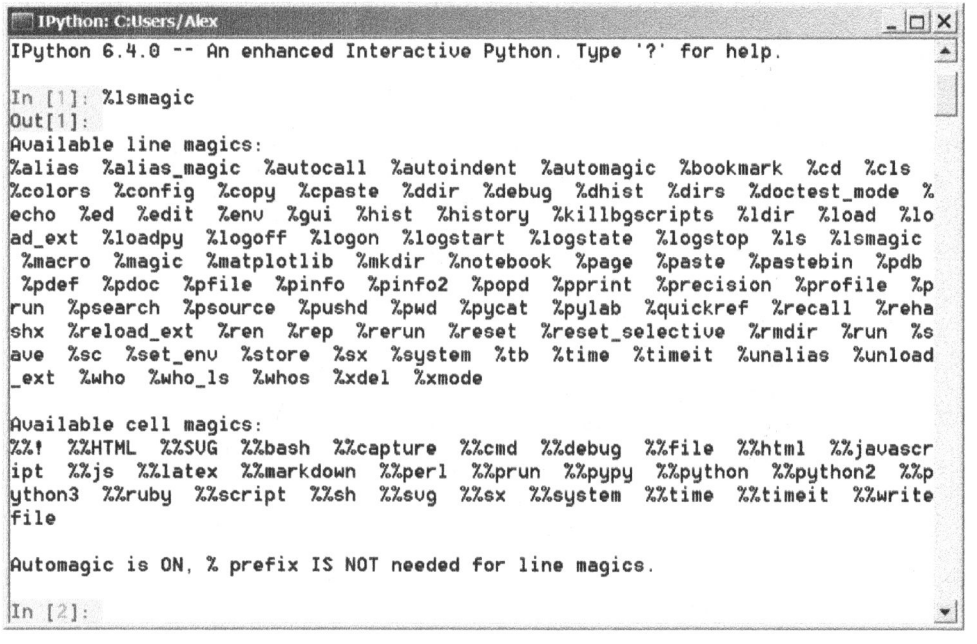


Рис. 5.7. Функция `%lsmagic` отображает список магических функций

ЗАПОМНИ!

В IPython работают не все магические функции. К примеру, функция `%autosave` неактивна в среде IPython, поскольку здесь отсутствует возможность автоматического сохранения.

В табл. 5.1 описаны наиболее распространенные магические функции. Для получения полного перечня введите **`%quickref`** при работе в Notebook и IPython или же обратитесь за по следующему адресу:

`https://damontallen.github.io/IPython-quick-ref-sheets/`

Таблица 5.1. Распространенные магические функции в Notebook и IPython

Магическая функция	Поддерживает ли статус при отдельном вводе	Описание
`%alias`	Да	Назначает или отображает псевдоним для системной команды
`%autocall`	Да	Позволяет вызывать функции без ввода скобок. Имеет следующие настройки: `Off`, `Smart` (по умолчанию) и `Full`. В случае указания параметра `Smart` скобки нужны только при наличии аргумента
`%automagic`	Да	Позволяет вызывать магические функции без ввода знака процента (`%`). Имеет следующие настройки: `False` (по умолчанию) и `True`
`%autosave`	Да	Отображает или изменяет значение интервала между автоматическими сохранениями блокнота. По умолчанию сохранение происходит каждые 120 секунд
`%cd`	Да	Меняет текущий каталог. С помощью этой команды можно перемещаться по истории каталогов или переходить к закладке
`%cls`	Нет	Выполняет очистку экрана
`%colors`	Нет	Задает цветовую схему для отображения текста, который ассоциируется с подсказками, информационной системой и обработчиками исключений. Доступные варианты: `NoColor` (черно-белая схема), `Linux` (по умолчанию) и `LightBG`
`%config`	Да	Позволяет конфигурировать IPython
`%dhist`	Да	Выводит список каталогов, к которым выполнялось обращение в ходе текущего сеанса
`%file`	Нет	Выводит имя файла, содержащего исходный код данного объекта
`%hist`	Да	Отображает список магических функций, которые вызывались во время текущего сеанса
`%install_ext`	Нет	Устанавливает заданный модуль расширения
`%load`	Нет	Загружает код приложения из другого источника, например из Интернета

Окончание табл. 5.1

Магическая функция	Поддерживает ли статус при отдельном вводе	Описание
`%load_ext`	Нет	Загружает расширение Python с указанным именем модуля
`%lsmagic`	Да	Отображает список доступных в настоящий момент магических функций
`%magic`	Да	Отображает экран справки, содержащий информацию о магических функциях
`%matplotlib`	Да	Устанавливает модуль построения диаграмм, позволяя отображать графики внутри ячеек блокнота. Возможные значения: `'gtk'`, `'gtk3'`, `'inline'`, `'nbagg'`, `'osx'`, `'qt'`, `'qt4'`, `'qt5'`, `'tk'` и `'wx'`
`%paste`	Нет	Вставляет содержимое буфера обмена в командную строку IPython
`%pdef`	Нет	Показывает, каким образом вызывается данный объект (при условии, что объект является вызываемым)
`%pdoc`	Нет	Отображает строку документации объекта
`%pinfo`	Нет	Выводит подробную информацию об объекте (часто более подробную, чем предлагает справочная система)
`%pinfo2`	Нет	Выводит дополнительную информацию об объекте (если она доступна)
`%reload_ext`	Нет	Повторно загружает предварительно установленный модуль расширения
`%source`	Нет	Отображает исходный код объекта (при условии, что он доступен)
`%timeit`	Нет	Вычисляет наилучшую производительность инструкции
`%%timeit`	Нет	Вычисляет наилучшую производительность для всех инструкций ячейки, кроме той, которая находится в одной строке с магической функцией (т.е. является инструкцией инициализации)
`%unalias`	Нет	Удаляет из списка ранее созданный псевдоним
`%unload_ext`	Нет	Выгружает указанный модуль расширения
`%%writefile`	Нет	Записывает содержимое ячейки в указанный файл

Обзор выполняющихся процессов

На главной странице Notebook, где выбираются открываемые блокноты, есть три вкладки. Обычно работают с вкладкой Files (Файлы). Вкладка Clusters (Кластеры) больше не используется, поэтому ее мы не рассматриваем. А вот вкладка Running (Выполняются) содержит полезную информацию о терминалах и открытых блокнотах (рис. 5.8).

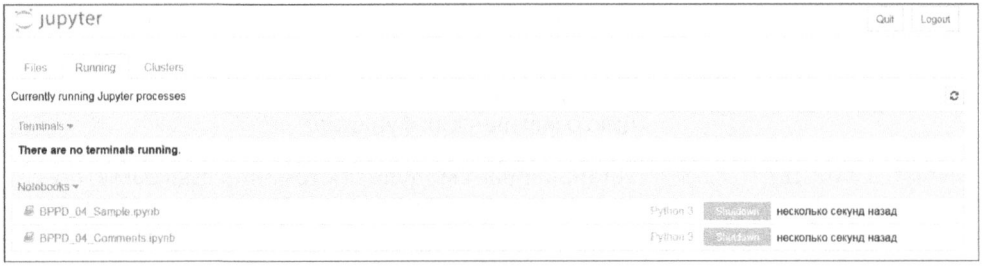


Рис. 5.8. Просмотр списка терминалов, подключенных к системе, и открытых блокнотов

Терминалы появляются, только если сконфигурировать сервер в многопользовательском режиме. В книге данный режим не рассматривается. Информацию о работе с терминалами можно получить по следующей ссылке:

```
http://jupyter-notebook.readthedocs.io/en/latest/examples/
Notebook/Connecting%20with%20the%20Qt%20Console.html
```

Самый важный элемент вкладки Running — список Notebooks (Блокноты). Всякий раз при запуске нового блокнота его запись добавляется в список. В данном случае работают два блокнота, оба на ядре Python 3.

ВНИМАНИЕ!

Здесь же имеется кнопка Shutdown (Остановить), предназначенная для закрытия блокнота. Обычно блокнот закрывают с помощью команды File⇨Close and Halt (Файл⇨Закрыть и остановить), чтобы избежать потери данных. Кнопка Shutdown будет полезна, если блокнот по какой-то причине перестал реагировать на действия пользователя.

2 Основы программирования на Python

В ЭТОЙ ЧАСТИ...

- Хранение данных и управление ими в памяти
- Использование функций
- Выбор среди нескольких вариантов
- Многократное выполнение задач
- Обнаружение и перехват ошибок приложения

Глава 6

Хранение и изменение информации

В ЭТОЙ ГЛАВЕ...

- Понятие о хранилище данных
- Типы хранилищ данных
- Работа со значениями даты и времени

В главе 3 вы познакомились с аббревиатурой CRUD, образованной от слов "Create" (Создание), "Read" (Чтение), "Update" (Обновление) и "Delete" (Удаление). CRUD — это акроним для запоминания основных операций, выполняемых компьютерными программами при работе с данными (или информацией, что одно и то же).

ЗАПОМНИ!

Чтобы информация была полезной, ее следует каким-то образом хранить, иначе при каждом выключении компьютера она будет исчезать. Более того, Python должен обеспечивать определенные правила изменения информации, чтобы предотвратить некорректное выполнение приложений. В этой главе рассматриваются вопросы управления информацией, а именно способы ее постоянного сохранения и обработки пользовательскими приложениями. Исходный код примеров данной главы находится в файле `BPPD_06_Storing_And_Modifying_Information.ipynb` (см. введение).

Хранение данных

Любое приложение должно иметь возможность получать быстрый доступ к информации, иначе выполнение задач будет отнимать много времени. Приложения хранят данные в памяти, однако оперативная память — это временное хранилище, и перед выключением компьютера данные следует переносить на постоянный носитель, например на жесткий диск, USB-флешку, SD-карту или в облачное хранилище в Интернете, такое как Anaconda Cloud. При этом нужно учитывать тип данных — текстовый или числовой. В следующих разделах подробно рассматриваются вопросы хранения данных.

Использование переменных

Приложения хранят информацию в *переменных*, которые можно рассматривать как ящики. Для изменения информации следует получить доступ к соответствующей переменной и сохранить в ней новое значение.

ЗАПОМНИ!

В компьютерах все организовано очень компактно. Каждая переменная хранит один элемент информации, благодаря чему найти необходимые данные не составляет особого труда. В предыдущих главах переменные не использовались, однако на практике большинство приложений работает с данными через переменные.

Правильный выбор переменных

Люди далеко не всегда хранят свои вещи оптимальным образом. Например, туфли порой можно найти в шкафу для одежды, а набор ручек — в коробке из-под обуви. Python во всем любит порядок и аккуратность, поэтому числа хранятся в одних переменных, а текстовые данные — в других. Несмотря на то что в обоих случаях используются переменные, каждый тип переменной предназначен для хранения определенного вида информации. Поэтому просто запомните, что каждый тип данных хранится в специально предназначенной для него переменной.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Python использует различные типы переменных для удобства программирования. Сами по себе компьютеры ничего не знают о типах данных. Все их знания сводятся к единицам и нулям, т.е. к наличию или отсутствию сигнала в соответствующей ячейке памяти. На более высоком уровне компьютеры работают с числами, но это предел их возможностей. Вся информация касательно чисел, букв,

дат, времени и всего остального в компьютерной системе сводится к единицам и нулям. Например, буква A хранится в виде двоичного числа 01000001 или десятичного числа 65.

Основные типы данных

Любой язык программирования, включая Python, определяет переменные для хранения различных видов информации. Конкретная разновидность переменной называется *типом данных*. Тип данных переменной определяет характер содержащейся в ней информации. Чтобы сохранить информацию определенного вида, нужна переменная соответствующего типа. Например, Python не сохраняет текстовую информацию в переменной, предназначенной для хранения чисел. Распространенные типы данных в Python — числа, строки и булевы значения. В следующих разделах будут рассмотрены стандартные типы данных Python.

Добавление информации в переменные

Чтобы присвоить значение любой переменной, следует воспользоваться оператором присваивания (=). Основные операторы Python будут подробно рассмотрены в главе 7, но с данным оператором нужно познакомиться уже сейчас. Например, чтобы присвоить число 5 переменной `myVar`, в строке ввода Python необходимо набрать **`myVar = 5`** и подтвердить действие нажатием клавиши <Enter>. Внешне Python никак не отреагирует на это, но можно проверить значение, содержащееся в переменной, как показано на рис. 6.1.

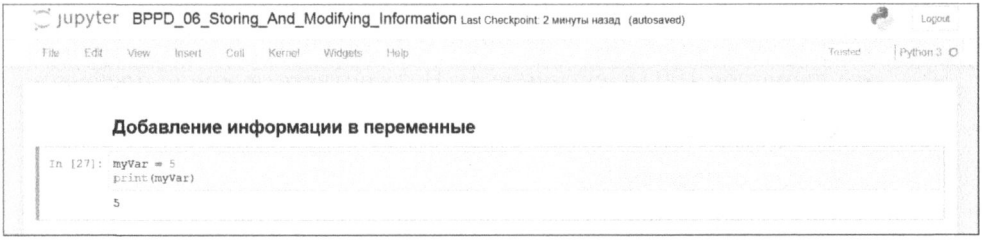


Рис. 6.1. Использование оператора присваивания для занесения информации в переменную

Знакомство с числовыми типами данных

Люди мыслят общими категориями чисел. Для нас 1 и 1,0 — это одно и то же число с той лишь разницей, что во втором случае используется десятичная запятая. Тем не менее данные числа абсолютно одинаковы, а потому

взаимозаменяемы. Но для Python это разные типы чисел, и каждое из них обрабатывается по-разному. В следующих разделах рассматриваются следующие числовые типы, поддерживаемые в Python: целые, с плавающей точкой (здесь используется десятичная точка, а не запятая) и комплексные.

Целые числа

Любое целое число относится к *целочисленному типу*. Например, значение 1 — это целое число, а значит, целочисленный тип. С другой стороны, 1,0 — не целое число, поскольку имеет дробную часть. Целочисленные типы представлены типом данных `int`.

ЗАПОМНИ!

Как и ящики, целочисленные типы имеют ограниченную емкость. При попытке записать слишком большое число в целочисленную переменную возникнет ошибка. На большинстве платформ допускается хранение чисел типа `int` в диапазоне между –9 223 372 036 854 775 808 и 9 223 372 036 854 775 807 (максимальное значение, помещаемое в 64-разрядную переменную). Даже самое большое число не может быть бесконечным.

Типу данных `int` присущ целый ряд интересных особенностей. Одна из них — поддержка различных *систем счисления*.

- **Основание 2**. В качестве чисел используются только 0 и 1.
- **Основание 8**. Используются числа от 0 до 7.
- **Основание 10**. Обычные числа.
- **Основание 16**. Используются числа от 0 до 9 и буквы от A до F.

Чтобы сообщить Python об использовании недесятичной системы счисления, необходимо добавить к числу ведущий 0 и буквенный префикс. Например, `0b100` — это двоичное 100 (десятичное 4). Вот соответствующие префиксы:

- `b` — основание 2;
- `o` — основание 8;
- `x` — основание 16.

Числовые значения могут быть преобразованы в другие системы счисления с помощью команд `bin()`, `oct()` и `hex()`. На рис. 6.2 показаны варианты таких преобразований. Попробуйте выполнить пример с другими числами, чтобы понять, как он работает. Переход в другие системы счисления часто бывает полезен, и в книге вам встретятся соответствующие примеры. А пока достаточно запомнить, что целочисленные переменные могут быть представлены в различных системах счисления.

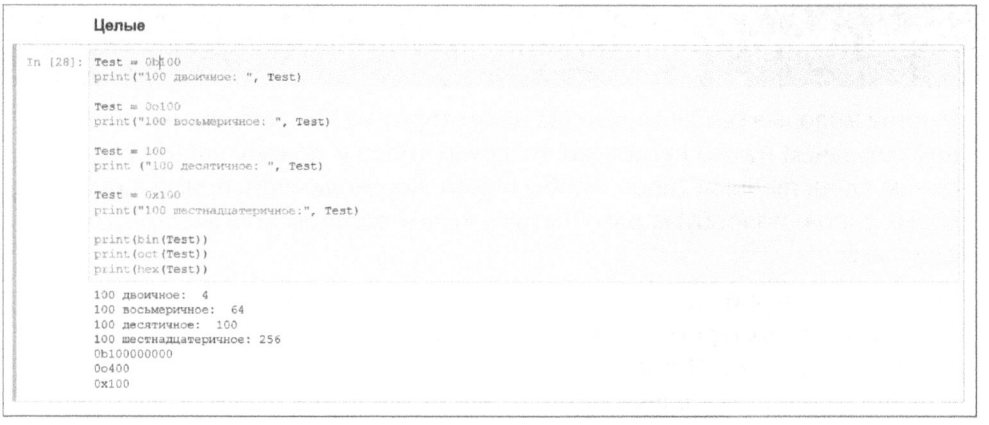


Рис. 6.2. Целочисленные переменные имеют множество интересных свойств, включая возможность использования различных систем счисления

Числа с плавающей точкой

Любое число, содержащее дробную часть, является значением с плавающей точкой, например `1.0`. Python хранит значения с плавающей точкой в типе данных `float`.

ЗАПОМНИ!

Преимущество чисел с плавающей точкой над целыми числами заключается в том, что они позволяют хранить как чрезвычайно большие, так и невероятно малые значения. При этом их емкость также ограничена. Максимальное (по модулю) значение, которое можно записать в переменную типа `float`, — $\pm 1{,}7976931348623157 \times 10^{308}$, а минимальное — $\pm 2{,}2250738585072014 \times 10^{-308}$ на большинстве платформ.

Существует несколько способов представления значений с плавающей точкой. Самые популярные варианты — непосредственное предоставление числа и использование экспоненциальной записи. Во втором случае число отделяется от экспоненты буквой *e*. На рис. 6.3 показано несколько примеров. Обратите внимание на то, что при использовании отрицательной экспоненты получается дробное значение.

Комплексные числа

Надеюсь, вы еще помните из школьного курса математики, что такое комплексные числа. *Комплексное число* состоит из вещественной и мнимой части. Освежить в памяти сведения о комплексных числах можно по следующей ссылке:

`https://ru.wikipedia.org/wiki/Комплексное_число`

ЗАЧЕМ НУЖНО СТОЛЬКО ЧИСЛОВЫХ ТИПОВ

Многие молодые разработчики (да и некоторые ветераны тоже) не всегда понимают, зачем нужно несколько числовых типов и почему нельзя обойтись одним-единственным типом. Чтобы понять необходимость применения числовых типов, попробуем разобраться, каким образом компьютер работает с числами.

Целое число хранится в виде цепочки битов, которые непосредственно считываются компьютером. Значение 0100 в двоичной системе равно 4 в десятичном эквиваленте. Числа с десятичным разделителем хранятся совершенно по-другому. Вспомните школьные уроки, на которых изучались экспоненты. Надеюсь, вы их не проспали, так как сейчас эти знания вам точно пригодятся. Число с плавающей точкой хранится как комбинация знакового разряда (плюс или минус), мантиссы (значащей части числа) и порядка, или экспоненты (показателя степени 2). Для получения значения с плавающей точкой используется следующее выражение:

значение = мантисса * 2^порядок

Раньше в разных системах использовались разные форматы представления чисел с плавающей точкой, но сейчас применяется единый стандарт IEEE-754. Детальное рассмотрение структуры чисел с плавающей точкой выходит за рамки данной книги. Тем, кому это интересно, рекомендуем обратиться по следующей ссылке:

```
http://www.cprogramming.com/tutorial/floating_point/
understanding_floating_point_representation.html
```

Имеет также смысл самостоятельно поиграть с числами. Удобный конвертер чисел доступен по следующему адресу:

```
http://www.h-schmidt.net/FloatConverter/IEEE754.html
```

Щелкайте на флажках отдельных разрядов, чтобы понять, как установка соответствующего разряда влияет на величину числа.

В силу своей сложности числа с плавающей точкой обычно занимают больше места в памяти, чем целые числа. Кроме того, они обрабатываются медленнее. Наконец, целые числа имеют точное представление, тогда как числа с плавающей точкой иногда можно выразить лишь приближенно. Тем не менее ключевое преимущество чисел с плавающей точкой — возможность хранения огромных значений. В реальном мире никуда не деться от десятичных дробей, поэтому мы и используем числа с плавающей точкой. Целые числа используются там, где необходимо уменьшить объем потребляемой памяти и ускорить работу приложения. В любом случае компромиссы просто неизбежны.

Значения с плавающей точкой

```
In [1]: Test = 255.0
        print("Непосредственное присваивание: ", Test)

        Test = 2.55e2
        print("Экспоненциальная запись: ", Test)

        Test = 2.55e-2
        print("Отрицательная экспонента: ", Test)

        Непосредственное присваивание:  255.0
        Экспоненциальная запись:  255.0
        Отрицательная экспонента:  0.0255
```

Рис. 6.3. Существует несколько способов присваивания чисел с плавающей точкой

Комплексные числа используются в следующих сферах:

- электротехника;
- гидроаэродинамика;
- квантовая механика;
- компьютерная графика;
- динамические системы.

Python — один из немногих языков программирования, предлагающих встроенный тип данных для работы с комплексными числами. Наличие подобных возможностей обеспечило Python популярность в качестве языка научных расчетов.

Мнимая часть комплексного числа всегда содержит суффикс `j`. Если, например, нужно создать комплексное число, в котором 3 будет вещественной частью, а 4 — мнимой, то это будет выглядеть следующим образом:

```
myComplex = 3 + 4i
```

Чтобы просмотреть вещественную часть переменной, просто введите **`myComplex.real`** в командной строке. Таким же образом для отображения мнимой части переменной введите **`myComplex.imag`**.

ОПРЕДЕЛЕНИЕ ТИПА ПЕРЕМЕННОЙ

Иногда тип переменной невозможно определить на основе анализа исходного кода или же исходный код недоступен. В таком случае, чтобы определить тип переменной, воспользуйтесь методом `type()`. Например, если записать значение 5 в переменную `myInt` с помощью выражения `myInt = 5`, а затем ввести **`type(myInt)`**, то мы получим `<class 'int'>`, т.е. переменная содержит значение типа `Int`.

Булевы значения

Компьютеры устроены так, что всегда дают прямые ответы на заданные вопросы. Компьютер никогда не ответит "может быть". Любой ответ компьютера — это "верно" или "неверно". Существует раздел математической логики, названный булевой алгеброй в честь английского математика Джорджа Буля. Именно эта алгебра помогает компьютерам принимать решения. Впрочем, сама булева алгебра появилась еще в 1854 году, задолго до изобретения компьютеров.

В Python булевы значения представляются типом данных `bool`. Переменная этого типа может содержать только два значения: `True` и `False`. Присвоить значение можно с помощью ключевых слов `True` и `False` либо путем создания логического выражения, которое будет истинным или ложным. Например, в результате выполнения выражения `myBool = 1 > 2` переменная `myBool` станет равна `False`, поскольку 1 меньше, чем 2. Если концепция булевых переменных пока не до конца вам понятна, не переживайте: в последующих главах мы будем активно применять переменные типа `bool`.

Строки

Среди всех типов данных строковые значения лучше всего воспринимаются человеком, притом что они совершенно непонятны компьютеру. *Строка* — это группа символов, заключенных в кавычки. Например, выражение

```
myString = "Python – великий язык программирования."
```

присваивает строку символов переменной `myString`.

Компьютер не понимает букв вообще. Каждая используемая буква представлена в памяти определенным числом. Например, буква 'A' хранится в виде числа 65. Чтобы увидеть это, в командной строке Python введите **`print(ord("A"))`**. В результате отобразится число 65. С помощью команды `ord()` можно преобразовать любую букву в ее числовой эквивалент.

Иногда требуется преобразовать строку в число. Для этого используются команды `int()` и `float()`. Например, если ввести **`myInt = int("123")`**, будет создана переменная `myInt` типа `int`, содержащая число 123. На рис. 6.4 показано, как проверить содержимое и тип переменной `myInt`.

В свою очередь, число можно преобразовать в строку с помощью команды `str()`. Например, если ввести **`myStr = str(1234.56)`**, будет создана строковая переменная `myStr`, содержащая значение `"1234.56"`. На рис. 6.5 показано, как выполнить такое преобразование и проверить тип переменной. Как видите, можно достаточно легко переключаться между строками и числами.

В последующих главах вы узнаете, как с помощью таких преобразований решать весьма нетривиальные задачи.

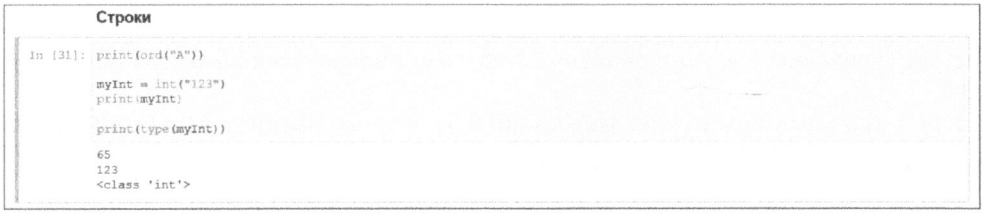


Рис. 6.4. Преобразование строки в число

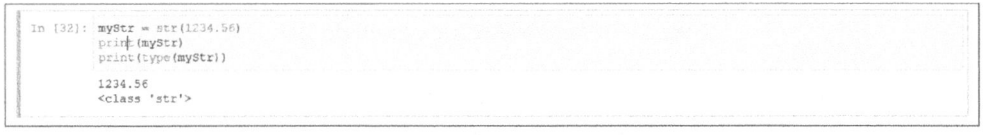


Рис. 6.5. Можно также преобразовать число в строку

Дата и время

Все в нашей жизни основано на датах и времени. Мы планируем события и назначаем встречи на определенные дату и время. Практически весь наш день расписан по минутам. В связи с этим полезно узнать, как Python работает с подобными величинами. Напомним, что компьютеры понимают только числа, поэтому такого понятия, как дата и время, для них не существует.

Для работы со значениями даты и времени в Python нужно подключить специальные средства. В частности, необходимо выполнить команду `import datetime`. Технически это называется *импортом пакета*, о чем будет говориться в главе 11. А пока не беспокойтесь о том, что такое пакеты. Просто используйте эту команду всякий раз, когда возникает необходимость выполнить те или иные действия с датой или временем.

В каждом компьютере есть встроенные часы. От них зависит корректная работа некоторых приложений, но все-таки они больше нужны пользователю, а не компьютеру. Чтобы узнать текущее время, просто введите **`datetime.datetime.now()`**. Вы получите полную информацию о дате и времени в том виде, в каком они представляются в системе (рис. 6.6).

Рис. 6.6. Получение текущих значений даты и времени с помощью команды `now()`

Предположим, вам нужна только дата. Для этого введите **`str(datetime.datetime.now().date())`** (рис. 6.7).

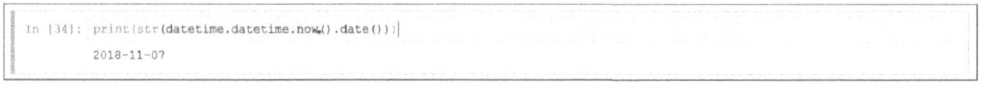


Рис. 6.7. Извлечение текущей даты

В Python также есть команда `time()`, с помощью которой можно определить текущее время. При этом можно получить отдельные значения для каждого компонента с помощью аргументов `day`, `month`, `year`, `hour`, `minute`, `second` и `microsecond`. В последующих главах вы узнаете, как применять эти функции для информирования пользователей о текущей дате и текущем времени.

Глава 7

Управление данными

В ЭТОЙ ГЛАВЕ...

- Просмотр данных в Python
- Использование операторов для присвоения, изменения и сравнения данных
- Структурирование кода с помощью функций
- Взаимодействие с пользователем

При работе с данными нам нужно иметь возможность присваивать значения переменным, определенным образом модифицировать содержимое этих переменных и сравнивать получаемые результаты с тем, что мы хотим получить. В этой главе будут рассмотрены все три аспекта управления данными в приложениях.

Также необходимо позаботиться о понятности кода. Да, можно написать приложение в виде одной длинной процедуры, однако понять такую процедуру будет крайне сложно, к тому же она будет содержать множество повторяющихся фрагментов, поскольку некоторые этапы должны выполняться более одного раза. Функции позволяют разбить код на компактные фрагменты, которые легче понимать и повторно использовать.

Еще одна важная задача — взаимодействие с пользователем. Конечно, существуют полезные приложения, совершенно не взаимодействующие с пользователем, но это исключение из правила. Чаще всего приложению требуется узнать, как пользователь хочет управлять данными. В этой главе мы поговорим об организации такого взаимодействия, а в последующих главах будем неоднократно возвращаться к данной теме.

Представление данных в Python

Как объяснялось в главе 6, компьютер хранит все данные в виде нулей и единиц. Он не понимает, что такое буквы, булевы значения, дата, время и другие виды информации, кроме чисел. К тому же обработка чисел в компьютере носит прямолинейный и упрощенный характер. При обработке строк Python преобразует их в форму, понятную компьютеру. Благодаря контейнерам данных, создаваемым в виде переменных, Python понимает, как обращаться с нулями и единицами, которые хранятся в памяти компьютера. Важно понимать, что Python рассматривает данные не так, как человек или компьютер. Python выступает в качестве посредника, который обеспечивает работу приложений.

ЗАПОМНИ!

Для управления данными приложение должно контролировать способ их представления в Python. Для этого используются операторы и функции, а также процедуры пользовательского ввода. И все они основаны на выполнении сравнений. Чтобы определить, какое действие должно быть выполнено следующим, программа должна проверить текущее состояние переменной и сравнить его с тестовым значением. Если в данный момент переменная содержит имя "Джон", но вы хотите, чтобы это было имя "Мария", то сначала необходимо убедиться, что переменная действительно содержит имя "Джон". Только после этого можно принять решение о замене содержимого переменной именем "Мария".

Выполнение сравнений

Основной способ выполнения сравнений в Python реализуется с помощью операторов. Операторы также играют важную роль в управлении данными. Ниже будет рассмотрено, как работают операторы и каким образом с их помощью можно управлять данными. В следующих главах вы узнаете о том, как с помощью операторов создавать приложения, которые могут принимать решения, выполнять повторяющиеся действия и взаимодействовать с пользователем самыми разными способами. Ну а пока мы сконцентрируемся на применении операторов для выполнения разного рода сравнений.

В некоторых случаях сравнения выполняются довольно нестандартными способами. Например, можно сравнивать результаты работы двух функций (о чем рассказывается в разделе "Сравнение результатов выполнения функций"). В Python можно выполнять операции сравнения на нескольких уровнях, что позволяет сосредоточиться на самой сути сравнения и определить реакцию

приложения на результат операции, а не погружаться в детали. Выбор способа сравнения определяет способ просмотра данных в Python и возможные варианты управления данными после получения результатов сравнения. На первый взгляд, все это кажется сложным и непонятным, но не стоит заранее паниковать. Важно лишь знать, что сравнения необходимы приложениям для корректной обработки данных.

Как компьютеры выполняют операции сравнения

Компьютеры не понимают структуризацию кода с помощью функций и других структур, создаваемых в Python. Все это нужно исключительно для удобства пользователя. Зато компьютеры напрямую поддерживают концепцию операторов. Большинство операторов Python непосредственно связано с командами, которые обрабатывает компьютер. Например, если попросить компьютер сравнить два числа, то он с помощью оператора напрямую выполнит это сравнение.

ЗАПОМНИ!

Не все операции сравнения выполняются напрямую. Это связано с тем, что компьютеры работают только с числами. Поэтому, если попросить Python сравнить две строки, он фактически сравнит числовые значения каждого символа в строке. Например, буква *A* в действительности хранится как число 65. Строчная буква *a* имеет другое числовое значение: 97. Поэтому, даже если кажется, будто выражение `ABC` эквивалентно `abc`, для компьютера это разные объекты, поскольку числовые значения отдельных букв различаются.

Работа с операторами

Операторы служат основой для управления данными в приложениях. С помощью операторов можно сравнивать между собой переменные и модифицировать значения отдельных переменных. Операторы нужны для выполнения математических задач и присвоения значений переменным.

ЗАПОМНИ!

Оператору необходимо передать либо переменную, либо выражение. Как вы уже знаете, переменная — это, по сути, хранилище данных. *Выражение* — это уравнение или формула. Чаще всего результатом оценки выражения является булево значение (`True` или `False`). В следующих разделах будут подробно описаны основные операторы Python.

Виды операторов

Оператор получает на вход один или несколько элементов данных в виде переменных или выражений, выполняет требуемое действие (к примеру, сравнение или сложение) и выдает полученный результат. Операторы классифицируются как по типу операций, так и по числу получаемых элементов данных. Например, унарный оператор работает с одной переменной или выражением; бинарный оператор — с двумя.

ЗАПОМНИ!

Элементы, подаваемые на вход оператора, называются *операндами*. Ниже представлены основные категории операторов, используемых в Python:

- унарные;
- арифметические;
- сравнения;
- логические;
- побитовые;
- присваивания;
- членство;
- тождественность.

Унарные операторы

Унарные операторы в качестве операнда получают единственную переменную или выражение (табл. 7.1).

Таблица 7.1. Унарные операторы Python

Оператор	Описание	Пример
~	Инвертирует биты числа, в результате чего все биты 0 превращаются в биты 1 и наоборот	Результат выражения `~4` будет равен –5
-	Выполняет операцию отрицания для исходного значения, в результат чего положительные значения становятся отрицательными и наоборот	Результат выражения `-(-4)` будет равен 4, а выражения `-4` — –4
+	Поддерживается исключительно ради полноты. Возвращает значение операнда	Результат выражения `+4` будет равен 4

ТЕРНАРНЫЙ ОПЕРАТОР В PYTHON

Для тернарного оператора требуются три операнда. В Python поддерживается лишь один такой оператор, который служит для проверки истинности выражения. Этот оператор имеет следующий синтаксис (вообще говоря, у него отсутствует определенное название, но, в принципе, его можно назвать оператором `if...else`):

ЗначениеИстина `if` *Выражение* `else` *ЗначениеЛожь*

Если результат проверки *Выражения* равен `True`, оператор вернет *ЗначениеИстина*. Если же результат проверки *Выражение* будет `False`, оператор вернет *ЗначениеЛожь*. Например, если ввести

```
"Привет" if True else "Пока"
```

то оператор вернет результат `'Привет'`. Если же ввести

```
"Привет" if False else "Пока"
```

то оператор вернет `'Пока'`. Такой оператор удобен в тех случаях, когда нужно быстро принять решение, но нет желания писать длинный код.

Одним из преимуществ Python является наличие нескольких способов решения одних и тех же задач. Альтернативная форма тернарного оператора в Python имеет еще более короткий синтаксис:

(*ЗначениеЛожь*, *ЗначениеИстина*)[*Выражение*]

Как и раньше, если *Выражение* равно `True`, оператор возвращает *Значение Истина*, в противном случае возвращается *ЗначениеЛожь*. Обратите внимание на то, что в этом случае операнды *ЗначениеИстина* и *ЗначениеЛожь* меняются местами. Пример такого синтаксиса выглядит следующим образом:

```
("Привет", "Пока")[Истина]
```

В данном случае оператор возвращает значение `"Пока"`, поскольку оно соответствует *ЗначениюИстина*. Из двух форм синтаксиса тернарного оператора первая более четкая и понятная, вторая — более краткая.

Арифметические операции

Компьютеры применяются для выполнения сложных математических вычислений. Но любые сложные расчеты всегда основаны на более простых математических операциях, таких как сложение. Python обеспечивает доступ к библиотекам, предназначенным для решения сложных математических задач, но при желании можно создавать собственные библиотеки математических функций с помощью простых операторов, перечисленных в табл. 7.2.

Таблица 7.2. Арифметические операторы Python

Оператор	Описание	Пример
+	Сложение двух операндов	5 + 2 = 7
-	Вычитание правого операнда из левого	5 - 2 = 3
*	Умножение правого операнда на левый	5 * 2 = 10
/	Деление левого операнда на правый	5 / 2 = 2.5
%	Остаток от деления левого операнда на правый	5 % 2 = 1
**	Возведение левого операнда в степень, задаваемую правым операндом	5 ** 2 = 25
//	Целочисленное деление левого операнда на правый (с округлением вниз)	5 // 2 = 2

Операторы сравнения

Операторы этой группы сравнивают два значения между собой и сообщают, является ли проверяемое отношение истинным (табл. 7.3). Например, 1 меньше 2, но 1 никогда не будет больше 2. Операторы сравнения часто используются для принятия решений в приложениях.

Таблица 7.3. Операторы сравнения в Python

Оператор	Описание	Пример
==	Проверяет, будут ли два значения равны. Обратите внимание на удвоение знака равенства. Ошибка многих новичков заключается в использовании одного знака равенства, что приводит к присваиванию одного значения другому	1 == 2 дает `False`
!=	Проверяет, будут ли два значения не равны. В некоторых устаревших версиях Python вместо оператора != допускается использовать оператор <>. В современных версиях Python это приведет к ошибке	1 != 2 дает `True`
>	Проверяет, будет ли значение левого операнда больше значения правого операнда	1 > 2 дает `False`
<	Проверяет, будет ли значение левого операнда меньше значения правого операнда	1 < 2 дает `True`

Окончание табл. 7.3

Оператор	Описание	Пример
`>=`	Проверяет, будет ли значение левого операнда больше или равно значению правого операнда	`1 >= 2` дает `False`
`<=`	Проверяет, будет ли значение левого операнда меньше или равно значению правого операнда	`1 <= 2` дает `True`

Логические операции

Логические операторы комбинируют булевы значения переменных или выражений, возвращая булево значение. Используя логические операторы, можно создавать булевы выражения, которые помогают определить, нужно ли выполнять те или иные действия в программе. Логические операторы описаны в табл. 7.4.

Таблица 7.4. Логические операторы Python

Оператор	Описание	Пример

`and`	Проверяет, являются ли оба операнда истинными (`True`)	`True and True` равно `True`
		`True and False` равно `False`
		`False and True` равно `False`
		`False and False` равно `False`

`or`	Проверяет, является ли один из операндов истинным (`True`)	`True or True` равно `True`
		`True or False` равно `True`
		`False or True` равно `True`
		`False or False` равно `False`

`not`	Инвертирует булево значение операнда: истинное значение становится ложным, а ложное — истинным	`not True` равно `False` `not False` равно `True`

Побитовые операции

Побитовые операторы работают с отдельными битами числа (табл. 7.5). Например, число `6` в двоичном (битовом) представлении выглядит так: `0b0110`.

Побитовый оператор определенным образом воздействует на каждый бит числа. В битовой логике значение 0 рассматривается как ложное, а 1 — как истинное.

Таблица 7.5. Побитовые операторы Python

Оператор	Описание	Пример
& (И)	Выполняет поразрядное сравнение двух операндов, устанавливая бит результата в 1, если оба исходных бита равны 1	`0b1100 & 0b0110 = 0b0100`
| (ИЛИ)	Выполняет поразрядное сравнение двух операндов, устанавливая бит результата в 1, если один из исходных битов равен 1	`0b1100 | 0b0110 = 0b1110`
^ (исключающее ИЛИ)	Выполняет поразрядное сравнение двух операндов, устанавливая бит результата в 1, если ровно один из исходных битов равен 1. Если же оба бита истинны или оба бита ложны, результирующий бит устанавливается равным 0	`0b1100 ^ 0b0110 = 0b1010`
~ (НЕ)	Инвертирует все разряды операнда	`~0b1100 = -0b1101` `~0b0110 = -0b0111`
<< (сдвиг влево)	Сдвигает биты левого операнда влево на число позиций, задаваемых правым операндом. Все новые биты устанавливаются равными 0, а все отсекаемые биты теряются	`0b00110011 << 2 = 0b11001100`
>> (сдвиг вправо)	Сдвигает биты левого операнда вправо на число позиций, задаваемых правым операндом. Все новые биты устанавливаются равными 0, а все отсекаемые биты теряются	`0b00110011 >> 2 = 0b00001100`

Операторы присваивания

Операторы присваивания заносят данные в переменную. Простой оператор присваивания рассматривался в предыдущих главах, но в Python доступен ряд других интересных операторов, которые могут параллельно выполнять математические вычисления, благодаря чему можно комбинировать присваивание с арифметическими операциями. Операторы присваивания описаны в табл. 7.6. В этой таблице начальное значение переменной `MyVar` в столбце "Пример" равно 5.

Таблица 7.6. Операторы присваивания в Python

Оператор	Описание	Пример
=	Присваивает левому операнду значение правого операнда	`MyVar = 5` Переменная `MyVar` содержит `5`

Оператор	Описание	Пример
`+=`	Добавляет значение правого операнда к значению левого операнда и присваивает результат левому операнду	`MyVar += 2` Переменная `MyVar` содержит `7`
`-=`	Вычитает значение правого операнда из значения левого операнда и присваивает результат левому операнду	`MyVar -= 2` Переменная `MyVar` содержит `3`
`*=`	Умножает значение правого операнда на значение левого операнда и присваивает результат левому операнду	`MyVar *= 2` Переменная `MyVar` содержит `10`
`/=`	Делит значение левого операнда на значение правого операнда и присваивает результат левому операнду	`MyVar /= 2` Переменная `MyVar` содержит `2.5`
`%=`	Делит значение левого операнда на значение правого операнда и присваивает остаток от деления левому операнду	`MyVar %= 2` Переменная `MyVar` содержит `1`
`**=`	Возводит значение левого операнда в степень, заданную правым операндом, и присваивает результат левому операнду	`MyVar **= 2` Переменная `MyVar` содержит `25`
`//=`	Делит значение левого операнда на значение правого операнда и присваивает целочисленный результат левому операнду	`MyVar //= 2` Переменная `MyVar` содержит `2`

Проверка членства

Операторы членства проверяют наличие значения в списке и возвращают булево значение, соответствующее результату проверки. Это напоминает поиск в базе данных. Вы вводите значение, которое, по вашему мнению, должно находиться в базе данных, а оператор сообщает о том, есть ли оно там. Операторы членства описаны в табл. 7.7.

Таблица 7.7. Операторы членства

Оператор	Описание	Пример
`in`	Проверяет, содержится ли значение левого операнда в последовательности, заданной правым операндом	`"Привет" in "Привет Пока"` равно `True`
`not in`	Проверяет, отсутствует ли значение левого операнда в последовательности, заданной правым операндом	`"Привет" not in "Привет Пока"` равно `False`

Проверка тождественности

Операторы тождественности проверяют, относится ли значение или выражение к определенному классу или типу. С их помощью вы сможете убедиться в том, что действительно работаете с нужной информацией. Это позволит избежать ошибок, вызванных несовместимостью типов, либо выбрать нужный вид обработки для проверяемого значения. Операторы тождественности описаны в табл. 7.8.

Таблица 7.8. Операторы тождественности в Python

Оператор	Описание	Пример
`is`	Возвращает `True`, если тип значения или выражения правого операнда соответствует типу значения или выражения левого операнда	`type(2) is int` равно `True`
`is not`	Возвращает `True`, если тип значения или выражения правого операнда не соответствует типу значения или выражения левого операнда	`type(2) is not int` равно `False`

Приоритет операторов

При создании простых инструкций, содержащих один оператор, определить результат не составляет труда. Но при наличии нескольких операторов нужно понимать очередность их выполнения. Например, чему равен результат вычисления выражения `1 + 2 * 3`: `7` (операция умножения выполняется первой) или `9` (сначала выполняется сложение)? На основе приоритета операторов можно сказать, что результат равен `7`, и изменить его можно только с помощью скобок. Выражение `(1 + 2) * 3` будет равно `9`, поскольку скобки имеют более высокий приоритет, чем операция умножения. В табл. 7.9 операторы Python перечислены в порядке уменьшения приоритета.

Таблица 7.9. Приоритет операторов Python

Оператор	Описание
`()`	С помощью скобок можно группировать выражения и переопределять заданный по умолчанию приоритет. В результате операция с более низким приоритетом (такая, как сложение) получает преимущество перед операцией с более высоким приоритетом (например, умножением)
`**`	Возведение левого операнда в степень, задаваемую правым операндом

Окончание табл. 7.9

Оператор	Описание
~ + -	Унарные операторы применяются к одной переменной или одному выражению
* / % //	Умножение, деление, остаток от деления (деление по модулю) и целочисленное деление
+ -	Сложение и вычитание
>> <<	Побитовый сдвиг вправо и влево
&	Побитовое И
^ |	Побитовое исключающее ИЛИ и простое ИЛИ
<= < > >=	Операторы сравнения
== !=	Операторы равенства
= %= /= //= -= += *= **=	Операторы присваивания
is is not	Операторы тождественности
in not in	Операторы членства
not or and	Логические операторы

Создание и использование функций

Каждая строка создаваемого кода решает определенную задачу, и мы должны научиться правильно комбинировать эти строки для достижения требуемых результатов. Иногда нужно повторять одни и те же инструкции с другими входными данными, а в некоторых случаях код становится настолько длинным, что отслеживать его работу становится слишком сложно. Функции позволяют упорядочивать код, делая его аккуратным и понятным. К тому же они обеспечивают возможность повторного использования ранее созданных инструкций с другими данными. В этом разделе вы найдете все, что нужно знать о функциях. И что еще более важно, здесь вы начнете создавать первое серьезное приложение, используя те же приемы, что и профессиональные разработчики.

Функции как средство упаковки кода

Представьте себе, что в один прекрасный день вы подходите к шкафу, открываете его дверцу, и на вас обрушивается гора вещей. Вам очень повезло, что вы остались целы и невредимы. Еще немного, и шар для боулинга, который лежал на верхней полке, нанес бы вам серьезные травмы! Тогда вы вспоминаете, что у вас есть выдвижные ящики, по которым можно разложить все вещи. Обувь будет находиться в одном ящике, игрушки — в другом, а старые открытки и письма — в третьем. Наведя порядок, вы сможете найти в шкафу все, что вам нужно, не боясь получить травму. То же самое можно сказать и о функциях: они наводят порядок в коде, позволяя легко видеть, что у вас есть, и понимать, как это работает.

ЗАПОМНИ!

Интернет изобилует комментариями о том, что такое функции и для чего они нужны, но если отбросить словесную шелуху, то все сводится к одной идее: функции являются средством упаковки кода, позволяющим быстро находить нужный код. Это как полки в шкафу. Если вы аккуратно раскладываете вещи по полкам, то легко можете найти то, что вам нужно.

Повторное использование кода

Представьте себе, что вы подходите к шкафу, достаете новые брюки и рубашку, срываете с них этикетки и одеваетесь. В конце дня вы снимаете все это с себя и выбрасываете. Странно, да? В реальности мы поступаем по-другому: мы снимаем с себя одежду, стираем ее, а затем вешаем в шкаф для повторного использования. Функции тоже предназначены для повторного использования. Никому не хочется постоянно писать один и тот же код, поскольку это скучно. Создавая функцию, вы получаете блок кода, который можно использовать множество раз для решения одной и той же задачи. Вы просто указываете компьютеру, какую функцию нужно выполнить. Компьютер послушно выполнит каждую инструкцию в функции всякий раз, когда вы его об этом попросите.

ЗАПОМНИ!

Код, который обращается к функции, называется *вызывающим*. Он должен предоставить функции определенную информацию, а функция, в свою очередь, должна вернуть результат вызывающему коду.

В былые времена компьютерные программы не поддерживали концепцию повторного использования кода, и программистам приходилось писать один и тот же код снова и снова. Но потом нашелся кто-то, которому в голову пришла идея функций. Эта концепция развивалась на протяжении многих лет, пока

функции не стали достаточно гибкими, чтобы решать практически любые задачи. Повторное использование кода позволяет:

- сократить время разработки;
- уменьшить количество программных ошибок;
- повысить надежность и эффективность приложений;
- предоставить программистам возможность использовать результаты работы друг друга;
- упростить код.

Работая с примерами книги, вы увидите, насколько повторное использование кода упрощает жизнь программисту. Без этой технологии нам пришлось бы постоянно набирать огромные фрагменты кода вручную.

Определение функции

Создать функцию не так уж и сложно. Благодаря Python это не потребует от вас много времени и сил. Порядок действий таков.

1. **Создайте новый блокнот в Notebook.**

   Исходный код примеров этой главы находится в файле `BPPD_07_Managing_Information.ipynb` (см. введение).

2. **Введите `def Hello():`.**

   Мы сообщаем Python о том, что создается функция `Hello`. Скобки важны, поскольку они определяют способ использования функции. (В данном случае ей не передаются никакие аргументы.) Двоеточие завершает объявление функции (рис. 7.1). Обратите внимание на то, что указатель вставки теперь имеет отступ. Это служит напоминанием о том, что нужно ввести код функции.

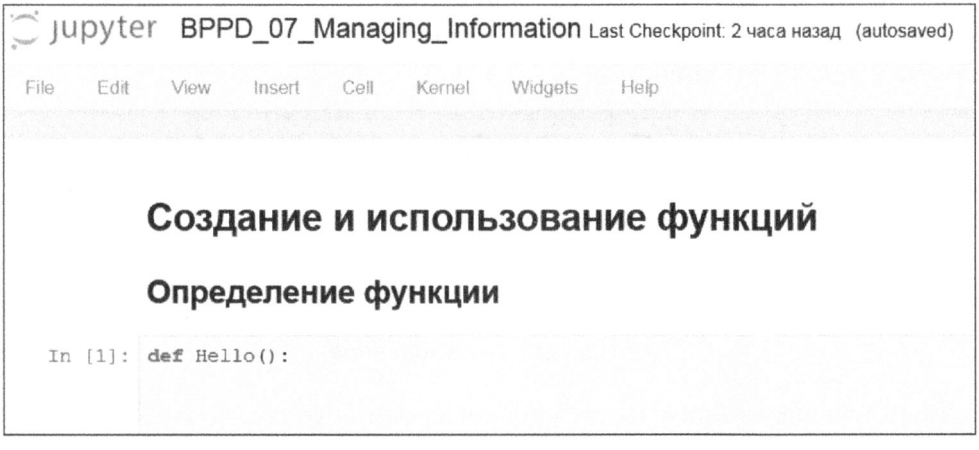


Рис. 7.1. Указание имени функции

3. **Введите `print("Первая функция Python!")`.**

   Указатель вставки по-прежнему имеет отступ (рис. 7.2), поскольку Notebook ожидает, что ввод тела функции еще не закончен.

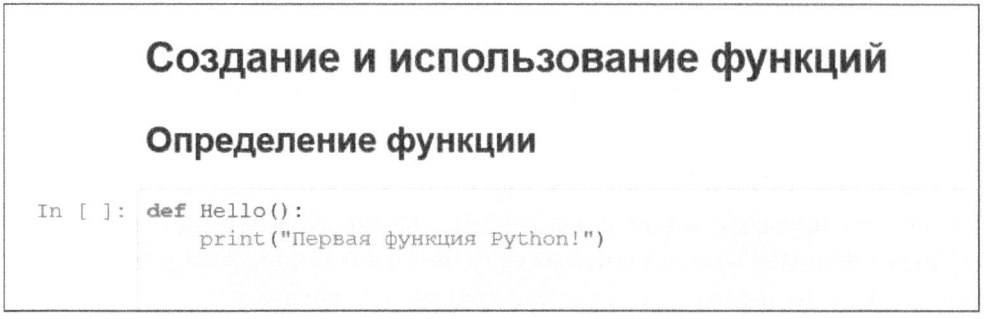


Рис. 7.2. Notebook ожидает следующей инструкции

4. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   На этом создание функции завершено.

Несмотря на простоту эта функция демонстрирует шаблон, по которому создается любая функция Python. Необходимо определить имя функции, указать ее аргументы (в данном случае отсутствуют) и ввести ее код. Определение функции завершится, когда курсор ввода будет находиться слева или произойдет переход к следующей ячейке.

Получение доступа к функциям

После создания функции вам захочется проверить ее в работе. Для начала нужно понять, как получить доступ к функции. В предыдущем разделе была создана новая функция `Hello()`. Чтобы вызвать ее, введите **`Hello()`** и щелкните на кнопке Run Cell (Выполнить ячейку). На рис. 7.3 показан результат выполнения этой функции.

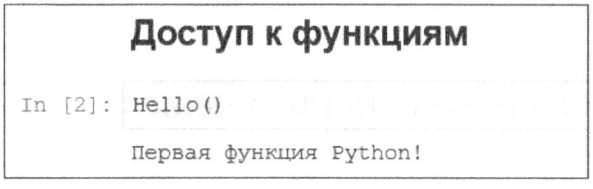


Рис. 7.3. После ввода имени функции отобразится результат ее выполнения

Другие функции применяются аналогичным образом. Нужно ввести имя функции, открывающую скобку, требуемые аргументы и закрывающую скобку,

после чего нажать клавишу <Enter>. В данном случае аргументы отсутствуют, поэтому достаточно ввести **`Hello()`**. В последующих разделах будут рассмотрены другие примеры, для которых требуется ввод аргументов.

Передача информации функциям

Пример функции `Hello()`, рассмотренный в предыдущем разделе, хорош тем, что вам не придется вводить длинную строку каждый раз, когда нужно просто сказать `Hello()`. В то же время возможности этой функции сильно ограничены, т.к. она способна выводить лишь одну строку. В идеале функции должны быть достаточно гибкими и допускать возможность настройки, иначе придется создавать множество функций, которые будут различаться в зависимости от получаемых данных, а не выполняемых операций. Благодаря аргументам можно создавать гибкие функции, работающие с различными данными.

Аргументы функции

Аргументом называется элемент информации, представляемой функции. По сути, это входные данные, с которыми работает функция. Благодаря аргументам функция становится более гибкой.

Функцию `Hello()` трудно назвать гибкой, поскольку она выводит на экран ровно одну строку. Чтобы придать этой функции необходимую степень гибкости, нужно добавить к ней аргумент. Это позволит передавать функции различные строки, задаваемые пользователем.

Чтобы увидеть, как работают аргументы, создайте в блокноте новую функцию `Hello2()`. Она представляет собой модифицированную версию функции `Hello()`, которая требует наличия аргумента.

```
def Hello2( Greeting ):
    print(Greeting)
```

Заметьте, что на этот раз скобки не пусты. В них содержится слово `Greeting`, которое является аргументом функции `Hello2()`. Аргумент `Greeting` фактически является переменной, которую можно передать функции `print()`, чтобы вывести содержимое аргумента на экран.

Передача требуемых аргументов

Итак, у нас появилась новая функция `Hello2()`, для вызова которой требуется задать аргумент. Но что если обойтись без него? Введите **`Hello2()`** и щелкните на кнопке Run Cell (Выполнить ячейку). Это приведет к появлению сообщения об ошибке, показанного на рис. 7.4, в котором говорится о том, что функция `Hello2()` требует наличия аргумента.

```
In [4]: Hello2()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-62b0700617ba> in <module>()
----> 1 Hello2()

TypeError: Hello2() missing 1 required positional argument: 'Greeting'
```

Рис. 7.4. Если не задать аргументы, появится сообщение об ошибке

Как видите, Python сообщает не только о том, что аргумент отсутствует, но и о том, как он называется. Определение созданной нами функции требует обязательного указания аргумента. Введите **`Hello2("Любопытная функция.")`** и щелкните на кнопке Run Cell (Выполнить ячейку). На этот раз результат будет таким, как нам нужно. Но мы еще не знаем, будет ли функция `Hello2()` достаточно гибкой, чтобы выводить на экран разные сообщения. Введите **`Hello2("Другое сообщение...")`** и снова щелкните на кнопке Run Cell. Результат показан на рис. 7.5. Как видите, функция `Hello2()` является улучшенной версией функции `Hello()`.

```
In [27]: Hello2("Любопытная функция.")

         Любопытная функция.

In [28]: Hello2("Другое сообщение...")

         Другое сообщение...
```

Рис. 7.5. Использование функции `Hello2()` *для вывода произвольного сообщения*

Исходя из приведенных примеров можно предположить, что аргумент `Greeting` может быть лишь строкой. Но если ввести **`Hello2(1234)`** и щелкнуть на кнопке Run Cell, то функция выдаст `1234`. А если ввести **`Hello2(5 + + 5)`**, то функция вернет результат вычисления выражения, т.е. `10`.

Передача аргументов по ключевому слову

По мере того как ваши функции будут становиться все более сложными, а способы их использования — более изощренными, вы захотите точнее контролировать, как вызываются функции и как им передаются аргументы. До сих пор мы использовали *позиционные аргументы*, значения которых предоставлялись в порядке их появления в списке аргументов функции. Наряду с этим Python поддерживает передачу аргументов по ключевому слову, или

именованных аргументов. В данном случае указывается имя аргумента, за которым следуют знак равенства (=) и значение аргумента. Чтобы увидеть, как все это работает, введите в блокноте код следующей функции.

```
def AddIt(Value1, Value2):
    print(Value1, " + ", Value2, " = ", (Value1 + Value2))
```

Обратите внимание на то, что функция `print()` включает список выводимых на экран элементов, разделенных запятыми. К тому же эти аргументы разных типов. В Python можно легко сочетать аргументы подобным образом.

А теперь протестируем функцию `AddIt()`. Сначала попробуем выполнить функцию с позиционными аргументами, поэтому введите **`AddIt(2, 3)`** и щелкните на кнопке Run Cell. В результате появится ожидаемый вывод `2 + 3 = 5`. Теперь введите **`AddIt(Value2 = 3, Value1 = 2)`** и снова щелкните на кнопке Run Cell. Вы опять получите результат `2 + 3 = 5`, несмотря на то что аргументы поменялись местами.

Присваивание аргументам функции значений по умолчанию

Независимо от того, с какими аргументами мы имели дело (позиционными или именованными), функции требовали указывать значения аргументов. Но иногда функция вполне может использовать значения, заданные по умолчанию. Это позволяет упростить вызов функций и уменьшить число ошибок, связанных с отсутствием требуемых данных. Значение, используемое по умолчанию, указывается после имени аргумента. Чтобы увидеть, как это работает, введите в блокнот следующую функцию.

```
def Hello3(Greeting = "Значение отсутствует"):
    print(Greeting)
```

Это еще одна версия функции `Hello()`, только на этот раз функция автоматически компенсирует ошибки пользователей, которые не указывают значение аргумента. Если кто-то попытается вызвать функцию `Hello3()` без аргумента, то это не приведет к возникновению ошибки. Введите **`Hello3()`** и убедитесь сами. Далее введите **`Hello3("Строка")`**, чтобы получить обычный отклик. Если полагаете, что функция способна работать только со строками, то введите **`Hello3(5)`** и убедитесь в обратном. Затем введите **`Hello3(2 + 7)`**. Результат выполнения всех этих тестов показан на рис. 7.6.

Создание функций с переменным числом аргументов

В большинстве случаев количество аргументов функции точно известно. С такими функциями проще работать, и их легче отлаживать. Но иногда количество аргументов невозможно определить заранее. Например, при создании утилиты командной строки мы предполагаем, что пользователь может не

указать ни одного аргумента, задать максимальное число аргументов (при условии, что их больше одного) или произвольное число аргументов.

```
In [25]: def Hello3(Greeting = "Значение отсутствует"):
             print(Greeting)

In [26]: Hello3()
         Hello3("Строка")
         Hello3(5)
         Hello3(2 + 7)

         Значение отсутствует
         Строка
         5
         9
```

Рис. 7.6. Старайтесь предоставлять аргументам функции значения по умолчанию всегда, когда это возможно

К счастью, Python позволяет передавать функции переменное число аргументов. Для этого достаточно поставить звездочку (`*`) перед именем аргумента, например `*VarArgs`. В таких случаях обычно имеется второй аргумент, задающий общее количество переданных аргументов. Ниже приведен пример функции, которая может выводить на экран переменное число элементов. (Не переживайте, если не можете полностью разобраться в работе функции; понимание наступит позже.)

```
def Hello4(ArgCount, *VarArgs):
    print("Передано аргументов: ", ArgCount)
    for Arg in VarArgs:
       print(Arg)
```

В этом примере мы используем цикл `for`, который будет подробно рассмотрен в главе 9. Пока что вам достаточно знать, что аргументы берутся из списка `VarArgs` по одному, значения отдельных аргументов помещаются в переменную `Arg`, а затем значение переменной `Arg` выводится с помощью функции `print()`. Нас больше всего интересует, как обрабатывается переменное число аргументов.

После того как функция будет добавлена в блокнот, введите **`Hello4(1, "Тестовая строка.")`** и щелкните на кнопке Run Cell (Выполнить ячейку). Функция должна сообщить о том, сколько аргументов ей передано, и вывести тестовую строку — ничего необычного. А теперь введите **`Hello4(3, "Один", "Два", "Три")`** и убедитесь в том, что функция корректно обрабатывает переменное число аргументов (рис. 7.7).

Возврат информации из функций

Функции могут выводить данные на экран или же возвращать их вызывающему коду, который будет выполнять дальнейшую обработку. Обычно имеет

место и то и другое. Способ работы функции зависит от задачи, которую она должна решать. Например, математическая функция с большей вероятностью вернет данные вызывающему коду.

Создание функций с переменным числом аргументов

```
In [5]: def Hello4(ArgCount, *VarArgs):
            print("Передано аргументов: ", ArgCount)
            for Arg in VarArgs:
                print(Arg)

In [6]: Hello4(1, "Тестовая строка.")

        Передано аргументов:  1
        Тестовая строка.

In [7]: Hello4(3, "Один", "Два", "Три")

        Передано аргументов:  3
        Один
        Два
        Три
```

Рис. 7.7. Функции с переменным числом аргументов позволяют сделать приложения более гибкими

Чтобы вернуть результат, функция должна включить инструкцию `return`, в которой указываются возвращаемые данные. Не существует ограничений на тип возвращаемого результата. Функция может возвращать следующие результаты.

- **Значение.** Это может быть любое значение: число, например `1` или `2.5`, строка вида `"Привет всем"` или же булево значение, такое как `True` или `False`.
- **Переменная.** Содержимое любой переменной действует так же, как и непосредственно указанное значение. Вызывающий код получит данные, хранящиеся в переменной.
- **Выражение.** Многие программисты используют выражения для сокращения кода. Например, можно напрямую вернуть результат выражения `A + B`, вместо того чтобы вычислять его в отдельной инструкции, присваивать результат переменной, а затем возвращать эту переменную вызывающему коду. Выражение — более короткий путь решения той же задачи.
- **Результат, возвращаемый другой функцией.** Функция может возвращать результаты других функций, вызываемых в ее коде.

Пора узнать, как возвращать значения из функций. В блокноте введите следующий код.

```
def DoAdd(Value1, Value2):
    return Value1 + Value2
```

Функция `DoAdd` получает два значения в качестве входных данных и возвращает сумму этих значений. Понятно, что такую простую задачу можно решить и без помощи функции, но у нас другие цели. Чтобы протестировать функцию, введите **`print("Сумма значений 3 + 4 равна ", DoAdd(3, 4))`** и щелкните на кнопке Run Cell (Выполнить ячейку). Результат выполнения кода показан на рис. 7.8.

Возврат информации из функций

```
In [12]: def DoAdd(Value1, Value2):
             return Value1 + Value2

In [18]: print("Сумма значений 3 + 4 равна ", DoAdd(3,4))

         Сумма значений 3 + 4 равна  7
```

Рис. 7.8. Функции становятся еще более полезными, когда возвращают значения

Сравнение результатов выполнения функций

Функции, возвращающие значения, могут применяться по-разному. Например, в предыдущем примере результат функции становился аргументом другой функции. Функции можно также применять для выполнения сравнений, создавая логические выражения.

Чтобы понять, как это работает на практике, воспользуемся функцией `DoAdd()` из предыдущего раздела. Введите **`print("3 + 4 равно 2 + 5 будет", (DoAdd(3,4) == DoAdd(2,5)))`** и щелкните на кнопке Run Cell (Выполнить ячейку). Как показано на рис. 7.9, `3 + 4` действительно равно `2 + 5`. Цель этого примера — показать, что функции являются достаточно универсальными инструментами, которые можно использовать самыми разными способами. Благодаря функциям код приложения становится гибким и удобным.

Сравнение результатов выполнения функций

```
In [15]: print("3 + 4 равно 2 + 5 будет ", (DoAdd(3,4)==DoAdd(2,5)))

         3 + 4 равно 2 + 5 будет  True
```

Рис. 7.9. С помощью функций можно решать самые разнообразные задачи

Получение пользовательского ввода

Лишь немногие приложения существуют в изолированном компьютерном мире. Большинство приложений активно взаимодействует с пользователями, поскольку именно для этого и нужны компьютеры. Чтобы организовать взаимодействие с пользователем, приложению нужно каким-то образом получать от него данные. К счастью, это достаточно легко реализовать с помощью функции `input()`.

ЗАПОМНИ!

Функция `input()` всегда возвращает строку. Даже если пользователь вводит число, функция `input()` преобразует его в строку. Это означает, что для получения числа результат функции `input()` нужно преобразовать обратно из строки в число. Также функция `input()` позволяет отобразить подсказку ввода. Назначение этой подсказки — сообщить пользователю о том, какая информация от него ожидается. В следующем примере показано, как применять функцию `input()`.

```
Name = input("Укажите свое имя: ")
print("Привет, ", Name)
```

В данном случае функция `input()` запрашивает имя пользователя. После того как пользователь укажет свое имя и нажмет клавишу <Enter>, программа сформирует для него персональное приветствие. На рис. 7.10 показаны результат работы программы.

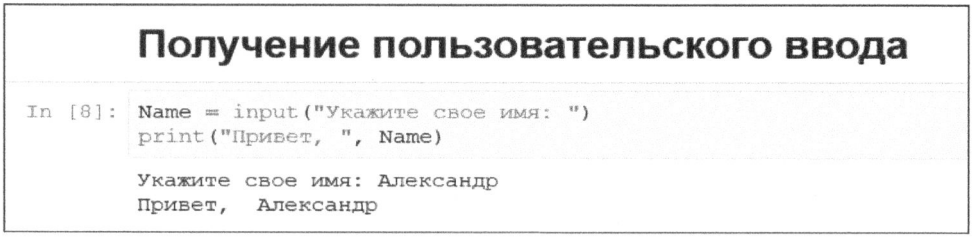


Рис. 7.10. Укажите имя пользователя, чтобы получить персональное приветствие

Функцию `input()` можно использовать для ввода других типов данных, но в таком случае нужно позаботиться о соответствующем преобразовании. Ниже показан соответствующий пример.

```
ANumber = float(input("Введите число: "))
print("Вы ввели: ", ANumber)
```

После запуска этого примера приложение попросит вас ввести число, но фактически функция `input()` принимает строковые данные. С помощью функции `float()` введенные данные преобразуются в число. После выполнения преобразования функция `print()` выводит результат. Если попробовать ввести число, например `5.5`, все будет работать корректно.

Преобразование данных сопряжено с определенным риском. Если вы попытаетесь ввести что-то кроме числа, то получите сообщение об ошибке, как показано на рис. 7.11. В главе 10 будет показано, как обнаружить и исправить ошибки до того, как они вызовут сбой приложения.

```
In [17]: ANumber = float(input("Введите число: "))
         print("Вы ввели: ", ANumber)

         Введите число: Привет

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         <ipython-input-17-3b5a480f85e4> in <module>()
         ----> 1 ANumber = float(input("Введите число: "))
               2 print("Вы ввели: ", ANumber)

         ValueError: could not convert string to float: 'Привет'
```

Рис. 7.11. Некорректное преобразование данных может привести к ошибке

Глава 8

Принятие решений

В ЭТОЙ ГЛАВЕ...

- Принятие простых решений с помощью инструкции `if`
- Принятие сложных решений с помощью блока `if...else`
- Вложенные инструкции

В любом деле важно уметь принять верное решение или выбрать правильный путь решения задачи. С помощью математических вычислений компьютер получает полезную информацию, но для ее обработки нужно принимать различные решения, поэтому в любом языке программирования закладывается возможность принятия решений, реализуемая тем или иным способом. В этой главе рассматриваются способы принятия решений в Python. Исходный код примеров данной главы находится в файле `BPPD_08_Making_Decisions.ipynb` (см. введение).

ЗАПОМНИ!

Процесс принятия решений должен быть хорошо продуманным. Получая фактическое значение той или иной переменной, пользователь сравнивает его с тестовым значением и затем выполняет соответствующие действия. В реальной жизни, видя красный свет светофора, вы понимаете, что это не зеленый свет, а потому останавливаетесь и ждете, пока не загорится зеленый свет. Большинство людей даже не задумываются о подобного рода вещах, поскольку они давно превратились в рутину. Для человека принятие решений становится привычкой, компьютеры же выполняют следующий алгоритм.

1. Получение фактического значения того или иного элемента.
2. Сравнение фактического значения с проверочным.
3. Выполнение действия в соответствии с результатом сравнения.

Принятие простых решений с помощью инструкции `if`

Инструкция `if` является самым простым инструментом принятия решений в Python. Ее суть сводится к тому, что если проверяемое выражение истинно, то Python должен выполнить дальнейшие действия. В следующих разделах показано, как применять инструкцию `if`. Вы удивитесь тому, насколько велики ее возможности.

Знакомство с инструкцией `if`

Мы незаметно для самих себя постоянно применяем инструкцию `if` в повседневной жизни. К примеру, вы говорите себе: "Если сегодня среда, то на обед я съем салат с тунцом". В Python инструкции `if` не столь многословны, хотя и строятся по такому же образцу. К примеру, мы создаем переменную `TestMe` и присваиваем ей значение `6`:

```
TestMe = 6
```

Теперь попросим компьютер проверить, содержит ли переменная `TestMe` значение `6`.

```
if TestMe == 6:
   print("Значение переменной TestMe равно 6!")
```

Видя ключевое слово `if`, Python понимает, что от него требуется принять определенное решение. После слова `if` стоит *условие*, которое конкретизирует, какого рода сравнение должен выполнить Python. В нашем примере Python должен определить, содержит ли переменная `TestMe` значение `6`.

Еще раз обратите внимание на использование оператора равенства ==, а не оператора присваивания =. Довольно часто программисты путают их. Список операторов сравнения приводился в главе 7.

Условие всегда заканчивается двоеточием (`:`). Если не поставить двоеточие, Python не поймет, что это конец условия, и будет продолжать искать дополнительные условия, требуемые для принятия решения. После двоеточия следует перечень инструкций, которые должен выполнить Python. В данном примере Python сообщает, что значение переменной `TestMe` равно `6`.

Использование инструкции `if`

В Python инструкция `if` может использоваться по-разному. Ниже представлены три наиболее распространенных способа.

- Проверка единственного условия и выполнение одной инструкции, если условие истинно.
- Проверка единственного условия и выполнение нескольких инструкций, если условие истинно.
- Проверка нескольких условий и выполнение одной или нескольких инструкций, если комбинированное условие истинно.

В следующих разделах будут рассмотрены все три варианта. Примеры использования инструкции `if` будут неоднократно встречаться на протяжении всей книги, поскольку это ключевой инструмент принятия решений.

Работа с операторами сравнения

Оператор сравнения проверяет, как значение левого операнда соотносится со значением правого операнда. После выполнения сравнения он выдает результат `True` или `False`. Например, выражение `6 == 6` истинно, а выражение `5 == 6` ложно. Чтобы создать инструкцию `if`, выполните следующие действия.

1. **Откройте новый блокнот.**

   Можете также воспользоваться файлом исходного кода `BPPD_08_Making_Decisions.ipynb` (см. введение).

2. **Введите выражение `TestMe = 6`.**

   Переменной `TestMe` будет присвоено значение 6. Заметьте, что вместо оператора равенства здесь используется оператор присваивания.

3. **Введите `if TestMe == 6:`.**

   Будет создана инструкция `if`, которая проверяет значение переменной `TestMe` с помощью оператора равенства. Отметим две особенности блокнота в данный момент:

   - слово `if` выделено другим цветом;
   - следующая строка автоматически выравнивается с помощью отступа.

4. **Введите `print("Значение переменной TestMe равно 6")`.**

   Заметьте, что Python пока не выполняет инструкцию `if`. Вместо этого он выравнивает следующую строку отступом. Слово `print` выделяется другим цветом, поскольку это название функции. Текст в кавычках тоже выделяется другим цветом текст, указывающим на то, что это строковое значение. Цветовая кодировка упрощает работу в среде Python.

5. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Notebook выполнит инструкцию `if` (рис. 8.1). Поскольку переменная `TestMe` содержит значение `6`, результат будет вполне ожидаемым.

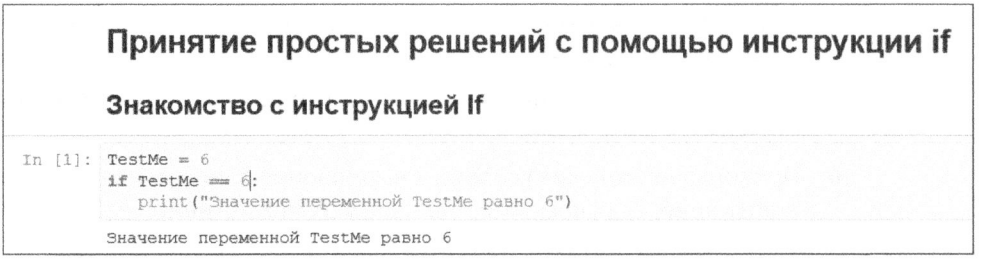


Рис. 8.1. Благодаря простым инструкциям `if` приложение знает, что нужно делать в тех или иных условиях

Выполнение нескольких задач

Иногда после принятия решения может возникнуть необходимость выполнить несколько инструкций. Чтобы понять, где заканчивается блок инструкций, Python ориентируется на отступы в коде инструкции `if`. До тех пор пока следующая строка смещена вправо, она является частью блока `if`. Как только очередная строка смещается влево, она становится первой строкой кода за пределами блока `if`. *Блок кода* включает команды, относящиеся к одной и то же основной инструкции. В рассматриваемом примере мы работаем с инструкцией `if`, которая формирует блок кода. Ниже показано, как выполнить несколько операций внутри инструкции `if`.

1. **Введите следующий код в блокноте.**

```
TestMe = 6
if TestMe == 6:
   print("Значение переменной TestMe равно 6!")
   print("Завершено!")
```

Заметьте, что по мере ввода кода автоматически создаются отступы для строк. Каждая вводимая строка является частью блока кода текущей инструкции `if`.

ЗАПОМНИ!

В Notebook блок кода создается путем ввода строк одна за другой. Если дважды подряд нажать клавишу <Enter>, не вводя текст, то блок кода завершится.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Python выполнит весь блок кода. Результат представлен на рис. 8.2.

Выполнение нескольких сравнений с помощью логических операторов

До сих пор во всех примерах демонстрировалось выполнение единственного сравнения. На практике чаще возникает необходимость во множественных сравнениях для удовлетворения сразу нескольких требований.

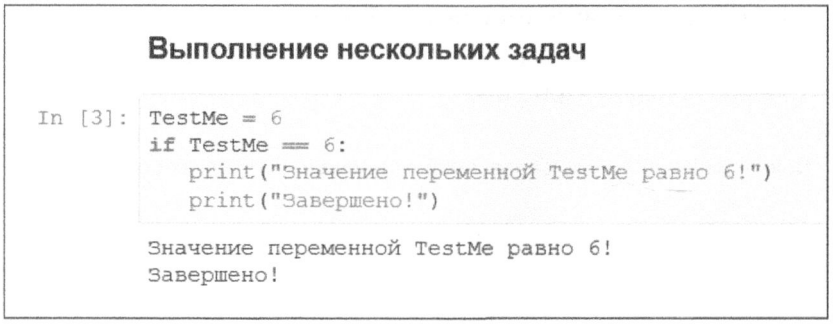


Рис. 8.2. Блок кода может содержать множество строк — по одной для каждой выполняемой задачи

Чтобы выполнить несколько сравнений одновременно, необходимо создать набор условий с помощью операторов сравнения и логических операторов (см. главу 7). *Логический оператор* определяет способы комбинирования условий. Например, `x == 6 and y == 7` можно рассматривать как два условия, объединяемых в одну проверку. Ключевое слово `and` является логическим оператором, требующим, что оба условия были истинными.

Чаще всего множественные сравнения применяют, чтобы установить принадлежность значения переменной определенному диапазону. Выполните следующие действия.

1. **Введите следующий код в блокноте.**

```
Value = int(input("Введите число от 1 до 10: "))
if (Value > 0) and (Value <= 10):
   print("Вы ввели: ", Value)
```

Пример начинается с получения значения от пользователя. Мы не знаем, что именно ввел пользователь, за исключением того, что это какое-то число. Функция `int()` требует ввода целого числа (без дробной части). В противном случае приложение сгенерирует *исключение* (указание на ошибку; исключения будут рассмотрены в главе 10).

Инструкция `if` содержит два условия. Согласно первому из них значение переменной `Value` должно быть больше 0. Это условие можно также представить в виде `Value >= 1`. Согласно второму условию значение переменной `Value` должно быть меньше или равно 10. Только при соблюдении обоих условий блок инструкции `if` успешно отобразит значение, которое ввел пользователь.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение предложит пользователю ввести число в диапазоне от 1 до 10.

3. **Введите 5 и нажмите клавишу <Enter>.**

   Приложение определит, что число находится в нужном диапазоне, и выведет сообщение (рис. 8.3).

Выполнение нескольких сравнений с помощью логических операторов

```
In [4]: Value = int(input("Введите число от 1 до 10: "))
        if (Value > 0) and (Value <= 10):
           print("Вы ввели: ", Value)

        Введите число от 1 до 10: 5
        Вы ввели:  5
```

Рис. 8.3. Приложение подтверждает, что значение находится в требуемом диапазоне, и выдает сообщение

4. **Выберите ячейку еще раз. Повторите пп. 2 и 3, но вместо 5 введите 22.**

   Приложение не отображает ожидаемое сообщение, поскольку число находится за пределами корректного диапазона. Всякий раз при вводе значения, выходящего за рамки установленного диапазона, инструкции, которые являются частью блока `if`, не выполняются.

ЗАПОМНИ!

Заметьте, что номер ячейки увеличивается на единицу. Всякий раз при выполнении ячейки ее номер будет изменяться. Допустим, номер ячейки был равен 4. Тогда после повторного выполнения ячейки на полях блокнота отобразится `In [5]:`.

5. **Выберите ячейку еще раз. Повторите пп. 2 и 3, но вместо 5 введите 5.5.**

   Python выдаст сообщение об ошибке (рис. 8.4). Несмотря на то что `5.5` и `5` — числа, для Python первое число — это значение с плавающей точкой, а второе — целое.

6. **Повторите пп. 2 и 3, но вместо 5 введите `Hello`.**

   Python снова отобразит сообщение об ошибке, поскольку для него не важно, к какому типу принадлежат некорректные данные. Он знает только то, что тип введенных данных некорректен и потому не может быть принят.

СОВЕТ

В профессиональных приложениях используются различные виды проверок диапазонов, обеспечивающие предсказуемость работы приложения. Чем более предсказуемо ведет себя приложение, тем проще пользователю сконцентрироваться на выполнении полезной работы.

Выполнение нескольких сравнений с помощью логических операторов

```
In [1]: Value = int(input("Введите число от 1 до 10: "))
        if (Value > 0) and (Value <= 10):
           print("Вы ввели: ", Value)

        Введите число от 1 до 10: 5
        Вы ввели:  5
```

```
In [1]: Value = int(input("Введите число от 1 до 10: "))
        if (Value > 0) and (Value <= 10):
           print("Вы ввели: ", Value)

        Введите число от 1 до 10: 5.5

        ---------------------------------------------------------------------------
        ValueError                                Traceback (most recent call last)
        <ipython-input-1-873105e77778> in <module>()
        ----> 1 Value = int(input("Введите число от 1 до 10: "))
              2 if (Value > 0) and (Value <= 10):
              3    print("Вы ввели: ", Value)

        ValueError: invalid literal for int() with base 10: '5.5'
```

Рис. 8.4. При вводе некорректной информации появляется сообщение об ошибке

Выбор альтернативных вариантов с помощью инструкции `if...else`

Многие решения основаны на выборе одного из двух вариантов на основании условия. Например, в зависимости от сигнала светофора следует либо нажать на тормоз и остановиться, либо нажать на газ и продолжить движение. Все зависит от конкретных условий. Зеленый свет означает, что можно продолжать движение, а красный — что нужно остановиться. В следующих разделах рассматривается, каким образом Python совершает выбор между двумя возможными вариантами.

Знакомство с инструкцией `if...else`

В Python выбор одного из двух вариантов осуществляется с помощью предложения `else` инструкции `if`. *Предложение* — это дополнительная часть инструкции, изменяющая способ ее работы. Инструкции, содержащие предложения, называются *составными*. В данном случае предложение `else` позволяет выполнить другую ветвь блока `if`. Многие программисты рассматривают инструкцию `if`, включающую предложение `else`, как самостоятельную инструкцию `if...else`. Многоточие в данном случае означает, что между `if` и `else` выполняются другие инструкции.

ВНИМАНИЕ!

Программисты иногда сталкиваются с проблемами при работе с инструкцией `if...else`, поскольку забывают, что ветвь `else` выполняется всегда, когда условие `if` оказывается ложным. Последствия могут оказаться самыми неожиданными.

Использование инструкции `if...else`

Пример из предыдущего раздела был бы более полезен, если бы программа реагировала на ввод значения за пределами диапазона. Ведь даже при вводе некорректных данных пользователь получает сообщение об ошибке, тогда как ввод корректных данных, но лежащих за пределами диапазона, ни к чему не приводит. В следующем примере показано, как это исправить с помощью предложения `else`, которое позволяет выбрать альтернативный вариант в случае ложного условия `if`.

1. **Введите следующий код в блокноте.**

```
Value = int(input("Введите число от 1 до 10: "))
if (Value > 0) and (Value <= 10):
   print("Вы ввели: ", Value)
else:
   print("Введенное значение некорректно!")
```

Как и в предыдущем примере, введенные пользователем данные проверяются на соответствие требуемому диапазону. Но на этот раз, если пользователь вводит данные, выходящие за пределы диапазона, выполняется ветвь `else`, где выводится другое сообщение.

ЗАПОМНИ!

Заметьте, что предложение `else` заканчивается двоеточием, как и условие `if`. Это характерно для большинства предложений в Python. Двоеточия нужны, чтобы Python мог определить, где заканчивается предложение. Если вы вдруг получаете сообщение о синтаксической ошибке, убедитесь в наличии двоеточий.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение предложит ввести число в диапазоне между 1 и 10.

3. **Введите 5 и нажмите клавишу <Enter>.**

Приложение определит, что число находится в корректном диапазоне, и отобразит соответствующее сообщение (см. рис. 8.3).

4. **Повторите пп. 2 и 3, но вместо 5 введите 22.**

На этот раз приложение выдаст сообщение о том, что введенное значение некорректно (рис. 8.5). Теперь пользователь понимает, что совершил ошибку и нужно попробовать ввести данные еще раз.

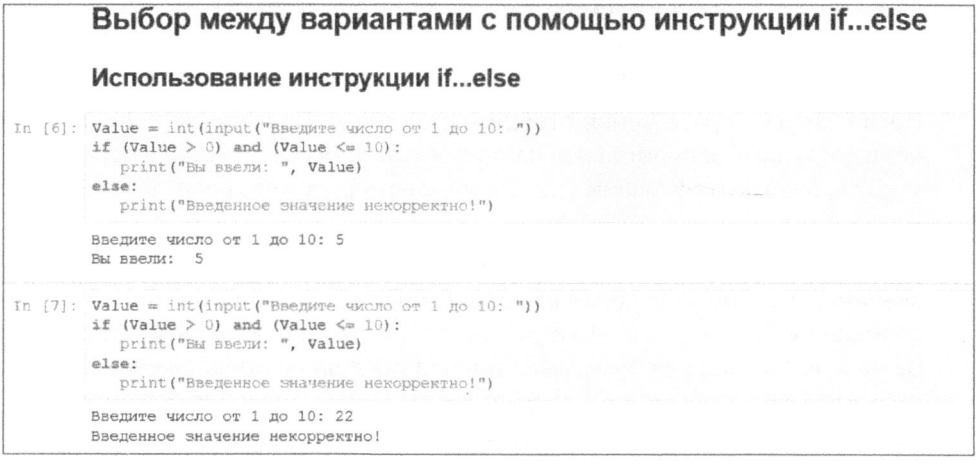


Рис. 8.5. Реакция приложения на ввод некорректных данных

Использование инструкции `if...elif`

Предположим, вы пришли в ресторан и просматриваете меню, в котором на завтрак предлагаются яичница, блинчики с творогом и овсянка. Вы делаете выбор, и официант приносит выбранное блюдо. Для создания такого меню требуется нечто большее, чем простая инструкция `if...else`. В Python имеется предложение `elif`, позволяющее создать другой набор условий. По сути это комбинация предложения `else` и отдельного блока `if`. Ниже показано, как запрограммировать множественный выбор с помощью инструкции `if...elif`.

1. **Введите следующий код в блокноте.**

```
print("1. Красный")
print("2. Оранжевый")
print("3. Желтый")
print("4. Зеленый")
print("5. Синий")
print("6. Пурпурный")
Choice = int(input("Выберите ваш любимый цвет: "))
if (Choice == 1):
   print("Вы выбрали красный цвет!")
elif (Choice == 2):
   print("Вы выбрали оранжевый цвет!")
elif (Choice == 3):
   print("Вы выбрали желтый цвет!")
elif (Choice == 4):
   print("Вы выбрали зеленый цвет!")
elif (Choice == 5):
   print("Вы выбрали синий цвет!")
elif (Choice == 6):
```

```
    print("Вы выбрали пурпурный цвет!")
else:
    print("Вы сделали неверный выбор!")
```

После запуска приложения отображается меню. Пользователь видит перечень доступных вариантов и делает выбор, который заносится в переменную `Choice`. Благодаря функции `int()` пользователь может ввести только число.

Далее приложение ищет введенное значение в списке возможных значений. В каждом случае значение переменной `Choice` сравнивается с конкретным значением в рамках условия выбора. Если ввести **1**, приложение отобразит сообщение `"Вы выбрали красный цвет!"`. Если ни один из вариантов выбора не является корректным, выполняется последнее предложение `else`, где пользователю сообщается о том, что введенное им значение неверное.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение отобразит меню и попросит вас выбрать любимый цвет.

3. **Введите 1 и нажмите клавишу <Enter>.**

   Появится соответствующее сообщение (рис. 8.6).

```
In [7]: print("1. Красный")
        print("2. Оранжевый")
        print("3. Желтый")
        print("4. Зеленый")
        print("5. Синий")
        print("6. Пурпурный")
        Choice = int(input("Выберите ваш любимый цвет: "))
        if (Choice == 1):
            print("Вы выбрали красный цвет!")
        elif (Choice == 2):
            print("Вы выбрали оранжевый цвет!")
        elif (Choice == 3):
            print("Вы выбрали желтый цвет!")
        elif (Choice == 4):
            print("Вы выбрали зеленый цвет!")
        elif (Choice == 5):
            print("Вы выбрали синий цвет!")
        elif (Choice == 6):
            print("Вы выбрали пурпурный цвет!")
        else:
            print("Вы сделали неверный выбор!")

        1. Красный
        2. Оранжевый
        3. Желтый
        4. Зеленый
        5. Синий
        6. Пурпурный
        Выберите ваш любимый цвет: 1
        Вы выбрали красный цвет!
```

Рис. 8.6. Программа предлагает выбрать один вариант из списка

4. **Повторите п. 3, но вместо 1 введите 5.**

   Приложение выдаст другое сообщение, соответствующее запрашиваемому цвету.

5. **Повторите п. 3, но вместо 1 введите 8.**

   Приложение сообщит о некорректном выборе.

6. **Повторите п. 3, но вместо 1 введите Red.**

   Приложение выдаст сообщение об ошибке (рис. 8.7). Любое приложение должно уметь обнаруживать ошибки ввода данных. В главе 10 будет показано, как обрабатывать такие ошибки.

```
In [8]: print("1. Красный")
        print("2. Оранжевый")
        print("3. Желтый")
        print("4. Зеленый")
        print("5. Синий")
        print("6. Пурпурный")
        Choice = int(input("Выберите ваш любимый цвет: "))
        if (Choice == 1):
           print("Вы выбрали красный цвет!")
        elif (Choice == 2):
           print("Вы выбрали оранжевый цвет!")
        elif (Choice == 3):
           print("Вы выбрали желтый цвет!")
        elif (Choice == 4):
           print("Вы выбрали зеленый цвет!")
        elif (Choice == 5):
           print("Вы выбрали синий цвет!")
        elif (Choice == 6):
           print("Вы выбрали пурпурный цвет!")
        else:
           print("Вы сделали неверный выбор!")

        1. Красный
        2. Оранжевый
        3. Желтый
        4. Зеленый
        5. Синий
        6. Пурпурный
        Выберите ваш любимый цвет: Red

        ---------------------------------------------------------------------------
        ValueError                                Traceback (most recent call last)
        <ipython-input-8-4d4caea623e5> in <module>()
              5 print("5. Синий")
              6 print("6. Пурпурный")
        ----> 7 Choice = int(input("Выберите ваш любимый цвет: "))
              8 if (Choice == 1):
              9     print("Вы выбрали красный цвет!")

        ValueError: invalid literal for int() with base 10: 'Red'
```

Рис. 8.7. Каждое приложение должно уметь распознавать некорректно введенные данные

А ГДЕ ЖЕ ИНСТРУКЦИЯ SWITCH

Если вы работали с другими языками программирования, то, должно быть, удивились тому, что в Python отсутствует инструкция `switch`, которую часто применяют для создания меню выбора. В Python для этих целей служит инструкция `if...elif`. Она отличается от инструкции `switch` тем, что не использует одну переменную для выполнения сравнения. В связи с этим для замены инструкции `switch` некоторые разработчики используют средства работы со словарями Python. Подробнее работа со словарями будет рассмотрена в главе 14.

Использование вложенных инструкций принятия решений

Зачастую процесс принятия решений происходит на разных уровнях. Например, выбирая в ресторане на завтрак яичницу, вы принимаете решение первого уровня. Затем официант предлагает выбрать бутерброд к яичнице. Если бы вы выбрали блинчики с творогом, то официант не предложил бы вам бутерброд. В данном случае выбор бутерброда является решением второго уровня. Предположим, вы выбираете бутерброд с рыбой. Тогда официант предлагает вам на выбор несколько вариантов рыбы. Это решение третьего уровня. Если бы вы выбрали бутерброд не с рыбой, то вам не пришлось бы принимать решение третьего уровня. Такой процесс многоуровневого принятия решений, в котором каждый уровень основан на решении предыдущего уровня, называется *вложением*. Вложенные инструкции часто применяются при создании приложений, которые могут принимать сложные решения на основе различных входных данных. В следующих разделах рассказывается о некоторых видах вложений, используемых в Python для принятия сложных решений.

Использование нескольких инструкций `if` или `if...else`

Самая распространенная методика множественного выбора — это комбинация инструкций `if` и `if...else`, называемая *деревом выбора* из-за сходства с ветвями дерева. В этом случае для достижения требуемого результата следует пройти определенный путь. Ниже описан процесс создания дерева выбора.

1. **Введите следующий код в блокноте.**

```
One = int(input("Введите число от 1 до 10: "))
Two = int(input("Введите число от 1 до 10: "))
```

```
if (One >= 1) and (One <= 10):
   if (Two >= 1) and (Two <= 10):
      print("Ваш секретный номер: ", One * Two)
   else:
      print("Второе значение некорректно!")
else:
   print("Первое значение некорректно!")
```

Это расширение примера, который рассматривался в разделе "Использование инструкции `if...else`". Обратите внимание на специфику отступов. Вторая инструкция `if...else` выравнивается внутри первой инструкции `if...else`. Таким образом, Python понимает, что это инструкция второго уровня.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа предложит ввести число в диапазоне от 1 до 10.

3. **Введите 5 и нажмите клавишу <Enter>.**

   Появится запрос на ввод другого числа, находящегося в диапазоне от 1 до 10.

4. **Введите 2 и нажмите клавишу <Enter>.**

   Программа сообщит результат умножения двух чисел (рис. 8.8).

Использование вложенных инструкций принятия решений

Использование нескольких инструкций if или if...else

```
In [1]: One = int(input("Введите число от 1 до 10: "))
        Two = int(input("Введите число от 1 до 10: "))
        if (One >= 1) and (One <= 10):
           if (Two >= 1) and (Two <= 10):
              print("Ваш секретный номер: ", One * Two)
           else:
              print("Второе значение некорректно!")
        else:
           print("Первое значение некорректно!")

        Введите число от 1 до 10: 5
        Введите число от 1 до 10: 2
        Ваш секретный номер:  10
```

Рис. 8.8. Использование нескольких уровней принятия решений позволяет решать более сложные задачи

ЗАПОМНИ!

В этом примере программа тоже контролирует, чтобы вводимые значения попадали в нужный диапазон. При попытке ввести некорректное значение программа выдает предупреждение. Причем пользователь будет точно знать, какое именно значение — первое или второе — оказалось неправильным.

Комбинирование различных инструкций принятия решений

Для получения нужного результата можно использовать любую комбинацию инструкций `if`, `if...else` и `if...elif`, создавая проверки условий на

любых уровнях вложенности. Например, в листинге 8.1 показано, как составить меню на завтрак.

Листинг 8.1. Составление меню на завтрак

```
print("1. Яичница")
print("2. Блинчики с творогом")
print("3. Бельгийские вафли")
print("4. Овсяная каша")
MainChoice = int(input("Выберите блюдо на завтрак: "))
if (MainChoice == 2):
   Meal = "блинчики с творогом"
elif (MainChoice == 3):
   Meal = "бельгийские вафли"
if (MainChoice == 1):
   print("1. Бутерброд с ветчиной")
   print("2. Бутерброд с сыром")
   print("3. Бутерброд с рыбой")
   print("4. Бутерброд с салом")
   Bread = int(input("Выберите бутерброд: "))
   if (Bread == 1):
      print("Вы выбрали яичницу и бутерброд с ветчиной.")
   elif (Bread == 2):
      print("Вы выбрали яичницу и бутерброд с сыром.")
   elif (Bread == 3):
      print("Вы выбрали яичницу и бутерброд с рыбой.")
   elif (Bread == 4):
      print("Вы выбрали яичницу и бутерброд с салом.")
   else:
      print("У нас есть яичница, но нет такого бутерброда.")
elif (MainChoice == 2) or (MainChoice == 3):
   print("1. Сироп")
   print("2. Клубника")
   print("3. Сахарная пудра")
   Topping = int(input("Выберите десерт: "))
   if (Topping == 1):
      print ("Вы выбрали " + Meal + " с сиропом.")
   elif (Topping == 2):
      print ("Вы выбрали " + Meal + " с клубникой.")
   elif (Topping == 3):
      print ("Вы выбрали " + Meal + " с сахарной пудрой.")
   else:
      print ("У нас есть " + Meal + ", но нет такого десерта.")
elif (MainChoice == 4):
   print("Вы выбрали овсяную кашу.")
else:
   print("У нас нет этого блюда на завтрак!")
```

Этот код имеет ряд интересных особенностей. Программисты порой считают, что в инструкции `if...elif` всегда должно быть предложение `else`. В данном примере демонстрируется тот случай, когда оно не нужно. В первой инструкции `if...elif` в переменную `Meal` заносится значение `2` или `3`, при этом остальные варианты нас не интересуют.

Методика выбора здесь такая же, как и в предыдущих примерах. Пользователь вводит число, которое должно находиться в определенном диапазоне. В трех случаях требуется дополнительный выбор, поэтому отображается расширенное меню, подходящее для конкретного варианта. Например, при заказе яичницы десерт не предлагается, тогда как при выборе блинчиков с творогом или бельгийских вафель он нужен.

В приложении есть фрагмент, где особым образом комбинируются переменные и текст. Поскольку десерт может предлагаться как для блинчиков с творогом, так и для бельгийских вафель, необходимо сообщить, какое именно блюдо будет подаваться на стол. Для этого используется переменная `Meal`, которая задается в самом начале.

Чтобы лучше понять пример, запустите его на выполнение. Попробуйте разные варианты меню и посмотрите, как работает приложение.

Глава 9

Выполнение повторяющихся действий

В ЭТОЙ ГЛАВЕ...

- Циклы с заданным числом итераций
- Циклы с условием завершения
- Вложенные циклы

Во всех приведенных выше примерах демонстрировалось однократное выполнение определенных действий. На практике часто требуется совсем иное. Многие задачи, с которыми мы сталкиваемся, имеют повторяющийся характер. Например, доктор может порекомендовать вам увеличить физическую нагрузку, отжимаясь по сто раз в день. Если вы не будете следовать рекомендации врача и делать по одному отжиманию в день, то никакой пользы от этого не будет. В Python можно выполнять подобные повторяющиеся действия с помощью цикла `for`.

К сожалению, мы не всегда знаем, сколько именно раз нужно выполнить требуемое действие. Например, есть стопка монет, которые нужно проверить на предмет их раритетности. Если взять только верхнюю монету из стопки, то не факт, что именно она окажется раритетной. Придется проверять каждую монету, и только после того, как будет найдена нужная, мы сможем сказать, что задача выполнена. Не зная точного количества монет в стопке, невозможно

предсказать количество проверок. Подобные повторяющиеся действия выполняются в Python с помощью цикла `while`.

ЗАПОМНИ!

В программировании повторяющиеся действия называются *циклом*. Программа выполняет один и тот же код до тех пор, пока цикл не закончится.

Иногда возникает необходимость в создании одного цикла внутри другого. Так, например, создается таблица умножения. Внутренний цикл вычисляет значения столбцов, а внешний цикл перемещается между строками. Не беспокойтесь, если это пока что звучит непонятно. Соответствующий пример будет приведен в конце главы. Исходный код примеров данной главы находится в файле `BPPD_09_Performing_Repetitive_Tasks.ipynb` (см. введение).

Обработка данных с помощью цикла `for`

Первый цикл, с которым сталкиваются программисты, — это цикл `for`. Сейчас уже трудно представить язык программирования без такой инструкции. В данном случае цикл выполняется фиксированное число раз, и это число известно заранее, еще до запуска цикла.

Знакомство с циклом `for`

Цикл `for` начинается с инструкции цикла, определяющей, сколько раз он будет выполняться. Циклически просматриваемая последовательность может быть как набором букв в строке, так и коллекцией элементов. С помощью функции `range()` можно даже указать конкретный ряд значений. Вот простой пример инструкции цикла:

```
for Letter in "Привет":
```

Цикл начинается ключевым словом `for`. Следующий элемент — переменная цикла, которая на каждой итерации будет содержать очередной элемент последовательности. В данном случае переменная называется `Letter`. Ключевое слово `in` указывает на то, что далее идет последовательность. Здесь последовательностью является строка `"Привет"`. Инструкция `for` всегда завершается двоеточием, точно так же, как и инструкции принятия решений, рассмотренные в главе 8.

Под инструкцией `for` с отступом вправо расположены инструкции, выполняемые в цикле. В Python каждая смещенная вправо инструкция является частью блока кода, образующего цикл `for`.

Создание базового цикла `for`

Давайте создадим цикл `for`, чтобы понять, как он работает. В данном примере просматриваемой последовательностью будет строка. Цикл `for` по очереди обрабатывает каждый символ в строке до тех пор, пока они не закончатся.

1. **Откройте новый блокнот.**

   Можете использовать файл исходного кода `BPPD_09_Performing_Repetitive_Tasks.ipynb` (см. введение).

2. **Введите следующий код в блокноте.**

```
LetterNum = 1
for Letter in "Привет":
   print("Буква ", LetterNum, " будет ", Letter)
   LetterNum+=1
```

   Пример начинается с создания переменной `LetterNum` для отслеживания количества обработанных букв. В конце каждой итерации цикла значение переменной `LetterNum` увеличивается на 1.

   Цикл `for` последовательно проходит все буквы в строке `"Привет"`. Каждая буква по очереди присваивается переменной `Letter`. Функция `print()` отображает текущее значение переменной `LetterNum` и соответствующий ей символ, находящийся в переменной `Letter`.

3. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение отобразит последовательность букв вместе с их номерами (рис. 9.1).

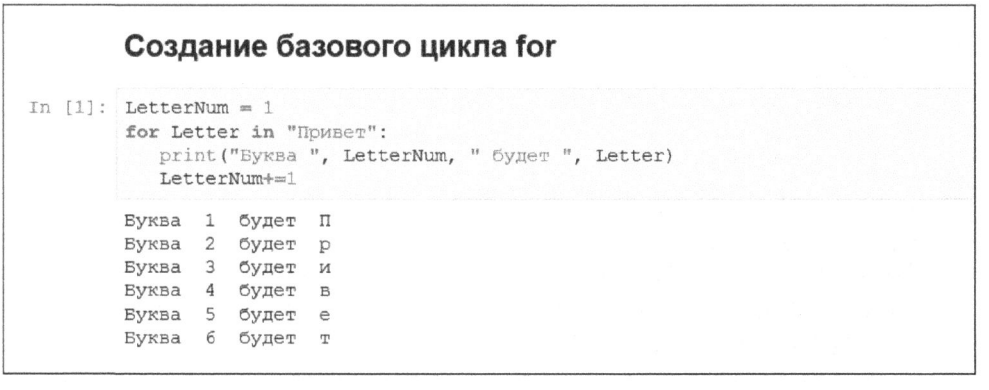


Рис. 9.1. Использование цикла `for` для последовательной обработки символов в строке

Управление выполнением цикла с помощью инструкции `break`

Из каждого правила бывают исключения. Возьмем, к примеру, конвейер для сборки часов. Допустим, в какой-то момент на сборочной линии заканчивается та или иная деталь. В отсутствие необходимого компонента конвейер останавливается в середине цикла сборки. Несмотря на то что сборка не завершена, линия все равно должна быть остановлена, пока не будет пополнен запас нужных деталей.

Подобные сбои случаются и в работе компьютеров. Например, во время загрузки данных из Интернета происходит сбой в сети, и соединение обрывается. Поток данных временно прекращается, в результате чего приложение завершает работу, хотя файл так и не был загружен.

С помощью инструкции `break` можно прекратить выполнение цикла. Она не просто включается в код, а вкладывается в инструкцию `if`, которая определяет условие прерывания цикла.

В следующем примере цикл досрочно завершается, когда при обработке строки значение счетчика цикла становится равным определенной величине. Это немного искусственный пример, цель которого — продемонстрировать, что может произойти, когда элемент данных обрабатывается слишком долго (возможно, вследствие ошибки).

1. **Введите следующий код в блокноте.**

```
Value = input("Введите менее 6 символов: ")
LetterNum = 1
for Letter in Value:
   print("Буква ", LetterNum, " будет ", Letter)
   LetterNum+=1
   if LetterNum > 6:
       print("Строка слишком длинная!")
       break
```

За основу взят пример из предыдущего раздела, только теперь пользователь может ввести строку переменной длины. Если строка содержит более шести символов, приложение прекращает ее обработку.

Инструкция `if` содержит условие прерывания цикла. Если значение переменной `LetterNum` больше 6, значит, строка слишком длинная. Обратите внимание на второй уровень отступа в инструкции `if`. При выполнении этого блока кода выводится сообщение, указывающее на слишком длинную строку. Далее выполняется инструкция `break`, которая завершает выполнение цикла.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа отображает приглашение на ввод данных.

3. **Введите слово Лампа и нажмите клавишу <Enter>.**

   Программа перечисляет каждый символ в строке (рис. 9.2).

```
In [1]: Value = input("Введите менее 6 символов: ")
        LetterNum = 1
        for Letter in Value:
           print("Буква ", LetterNum, " будет ", Letter)
           LetterNum+=1
           if LetterNum > 6:
              print("Строка слишком длинная!")
              break

        Введите менее 6 символов: Лампа
        Буква  1  будет  Л
        Буква  2  будет  а
        Буква  3  будет  м
        Буква  4  будет  п
        Буква  5  будет  а
```

Рис. 9.2. Короткая строка успешно обрабатывается приложением

4. **Выполните пп. 2 и 3 еще раз, но вместо слова Лампа введите фразу Слишком длинная строка.**

   Программа выдаст сообщение об ошибке и прекратит обрабатывать строку на символе 6 (рис. 9.3).

```
In [3]: Value = input("Введите менее 6 символов: ")
        LetterNum = 1
        for Letter in Value:
           print("Буква ", LetterNum, " будет ", Letter)
           LetterNum+=1
           if LetterNum > 6:
              print("Строка слишком длинная!")
              break

        Введите менее 6 символов: Слишком длинная строка
        Буква  1  будет  С
        Буква  2  будет  л
        Буква  3  будет  и
        Буква  4  будет  ш
        Буква  5  будет  к
        Буква  6  будет  о
        Строка слишком длинная!
```

Рис. 9.3. Длинные строки обрезаются, чтобы не превышать определенный размер

СОВЕТ

Это пример проверки предельной длины строки. В главе 8 рассматривались способы проверки диапазонов, позволявшие контролировать вводимые значения. Благодаря такого рода проверкам можно быть уверенным в том, что данные, особенно строки, не будут превышать размер выделенного для них пространства. Кроме того, ограничения на размер вводимых данных затрудняют определенные виды хакерских атак, что делает приложение более защищенным.

Управление выполнением цикла с помощью инструкции `continue`

Иногда нужно проверить каждый элемент последовательности, зная, что некоторые элементы не требуют обработки. Например, ваша задача — обработать информацию по каждому автомобилю в базе данных, кроме автомобилей коричневого цвета. Возможно, вам просто не нужна информация об этих автомобилях. В данной ситуации инструкцию `break` использовать нельзя, поскольку оно просто завершит цикл, и вы не увидите оставшиеся элементы последовательности.

ЗАПОМНИ!

Альтернативным вариантом для `break` является инструкция `continue`. Она тоже вкладывается в инструкцию `if`, которая содержит условие для перехода к очередной итерации цикла.

В следующем примере демонстрируются отличия инструкции `continue` от инструкции `break`. В данном случае программа отказывается обрабатывать букву и, но продолжит обрабатывать все остальные буквы.

1. **Введите следующий код в блокноте.**

```
LetterNum = 1
for Letter in "Привет":
   if Letter == "и":
      continue
      print("Найдено и, не обработано.")
   print("Буква ", LetterNum, " будет ", Letter)
   LetterNum+=1
```

За основу взят пример из раздела "Создание базового цикла `for`", к которому добавлен блок `if` с инструкцией `continue`. Обратите внимание на функцию `print()`, которая является частью блока `if`. Эта функция никогда не будет выполнена, потому что текущая итерация цикла заканчивается сразу же.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа отобразит последовательность букв вместе с их номерами (рис. 9.4). Несложно заметить влияния инструкции `continue` — буква и не обрабатывается.

```
In [2]: LetterNum = 1
        for Letter in "Привет":
           if Letter == "и":
              continue
              print("Найдено и, не обработано.")
           print("Буква ", LetterNum, " будет " , Letter)
           LetterNum+=1

        Буква  1  будет  П
        Буква  2  будет  р
        Буква  3  будет  в
        Буква  4  будет  е
        Буква  5  будет  т
```

Рис. 9.4. Используйте инструкцию `continue`, чтобы пропустить обработку определенных элементов

Управление выполнением цикла с помощью инструкции `pass`

В Python есть одна особенность, которая отличает его от других языков: вторая разновидность инструкции `continue` — инструкция `pass`. От инструкции `continue` ее отличает то, что она позволяет завершить выполнение блока `if`, в котором находится. Приводимый далее пример аналогичен примеру из предыдущего раздела, с той лишь разницей, что здесь используется инструкция `pass`.

1. **Введите следующий код в блокноте.**

```
LetterNum = 1
for Letter in "Привет":
   if Letter == "и":
      pass
      print("Найдено и, не обработано.")
   print("Буква ", LetterNum, " будет ", Letter)
   LetterNum+=1
```

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение отобразит последовательность букв с их номерами (рис. 9.5). Буква и здесь тоже не обрабатывается, зато выводится строка, которая не отображалась в примере с инструкцией `continue`.

ЗАПОМНИ!

Инструкция `continue` позволяет пропустить определенные элементы последовательности и не выполнять для них дополнительный код. Инструкция `pass` используется, когда необходимо выполнить дополнительную обработку при обнаружении того или иного

элемента. Это может быть, например, регистрация элемента в журнале ошибок, вывод сообщения об ошибке и т.п. Таким образом, инструкции `continue` и `pass` делают примерно одно и то же, но применяются в разных ситуациях.

```
In [3]: LetterNum = 1
        for Letter in "Привет":
           if Letter == "и":
              pass
              print("Найдено и, не обработано.")
           print("Буква ", LetterNum, " будет ", Letter)
           LetterNum+=1

        Буква  1  будет  П
        Буква  2  будет  р
        Найдено и, не обработано.
        Буква  3  будет  и
        Буква  4  будет  в
        Буква  5  будет  е
        Буква  6  будет  т
```

Рис. 9.5. Инструкция `pass` позволяет выполнить определенную обработку в случае появления нежелательных входных данных

Управление выполнением цикла с помощью предложения `else`

В Python есть еще один элемент цикла, которого нет в других языках программирования: это предложение `else`. Оно позволяет выполнять код, даже когда в заданной последовательности нет ни одного элемента для обработки.

1. **Введите следующий код в блокноте.**

```
Value = input("Введите менее 6 символов: ")
LetterNum = 1
for Letter in Value:
   print("Буква ", LetterNum, " будет ", Letter)
   LetterNum+=1
else:
   print("Строка пуста.")
```

   За основу взят пример из раздела "Создание базового цикла `for`". Если, запустив программу, пользователь просто нажмет клавишу <Enter>, будет выполнена ветвь предложения `else`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение выдаст запрос на ввод данных.

3. **Введите слово `Лампа` и нажмите клавишу <Enter>.**

Приложение перечислит каждый символ в строке (см. рис. 9.2). Как ни странно, сообщение о том, что строка пустая, тоже будет выведено. Дело в том, что при нормальном завершении цикла `for` (т.е. без вызова инструкции `break`) ветвь `else` всегда выполняется. С другой стороны, если переменная цикла не была инициализирована (цикл не выполнился ни разу), ветвь `else` тоже выполняется. Благодаря этой особенности она может использоваться в качестве блока финальных инструкций для любого цикла `for`. Подробнее об этом можно прочитать в статье, доступной по следующему адресу:

`http://python-notes.curiousefficiency.org/en/latest/python_concepts/break_else.html`

4. **Повторите пп. 2 и 3, но на этот раз просто нажмите клавишу <Enter>, не вводя никакого текста.**

   Теперь появится лишь сообщение о том, что строка пуста (рис. 9.6).

```
In [1]: Value = input("Введите менее 6 символов: ")
        LetterNum = 1
        for Letter in Value:
            print("Буква ", LetterNum, " будет ", Letter)
            LetterNum+=1
        else:
            print("Строка пуста.")

        Введите менее 6 символов:
        Строка пуста.
```

Рис. 9.6. Предложение `else` позволяет выполнять обработку в случае пустой последовательности

ВНИМАНИЕ!

Предложение `else` часто вызывает путаницу, поскольку пустая последовательность может означать не только отсутствие входных данных. Это может быть признаком ошибки или другой ситуации, которая требует иной обработки по сравнению с отсутствием входных данных. Постарайтесь тщательно разобраться с тем, как приложение работает с данными, чтобы не использовать предложение `else` там, где возникают потенциальные ошибки, которые нужно выявлять и устранять.

Обработка данных с помощью цикла `while`

Цикл `while` используется в ситуациях, когда вы не знаете, какой объем данных должно обработать приложение. Вместо того чтобы задавать фиксированное количество элементов, вы просите Python обрабатывать данные до тех пор, пока они не закончатся. Такого рода циклы применяются, например, при

загрузке файлов неизвестного размера или получении стриминговых данных от радиостанций в Интернете.

Знакомство с циклом `while`

В инструкции цикла `while` задается не последовательность, а условие. Цикл выполняется до тех пор, пока условие остается истинным. Представьте, например, покупателей, стоящих в очереди. Продавец обслуживает покупателей до последнего человека в очереди. Очередь может расти, если будут подходить другие покупатели, поэтому невозможно заранее сказать, сколько людей обслужит продавец. Он лишь знает, что торговля идет и нужно продолжать отпускать товар, пока есть спрос.

Вот как выглядит цикл `while`:

```
while Sum < 5:
```

Цикл начинается с ключевого слова `while`, за которым следует условие. В этом примере значение переменной `Sum` должно быть меньше `5`, чтобы цикл продолжался. О текущем значении переменной `Sum` здесь ничего не сказано, как и о том, каким образом оно будет меняться. Известно лишь, что для продолжения цикла значение переменной `Sum` должно быть меньше `5`. Условие заканчивается двоеточием, после которого следует список инструкций с отступом.

ВНИМАНИЕ!

Поскольку цикл `while` выполняется неопределенное число раз, возможно образование *бесконечного цикла,* который никогда не завершится. Допустим, перед началом цикла переменной `Sum` присвоено значение `0` и условие цикла требует, чтобы значение `Sum` оставалось меньшим `5`. Если значение переменной `Sum` не будет увеличиваться, то цикл будет выполняться бесконечно долго (по крайней мере, до момента выключения компьютера). Бесконечные циклы могут стать причиной множества проблем в системе, вызывая замедление или даже "зависание" компьютера, поэтому их лучше избегать. При использовании цикла `while` нужно продумать гарантированный способ его завершения (в отличие от цикла `for`, в котором конец последовательности означает конец цикла). Таким образом, нужно выполнить три действия:

1) подготовить проверяемую переменную (например, присвоить переменной `Sum` значение `0`);
2) задать условие цикла `while` (например, `Sum < 5`);
3) каким-то образом обновить проверяемую переменную в цикле, чтобы в конечном счете цикл гарантированно завершился (например, `Sum+=1`).

ЗАПОМНИ!

Как и в случае с циклом `for`, стандартное поведение цикла `while` можно менять с помощью тех же четырех команд:

- `break` — завершает текущий цикл;
- `continue` — немедленно завершает обработку текущего элемента;
- `pass` — завершает обработку текущего элемента после выполнения инструкций в блоке `if`;
- `else` — предлагает альтернативный способ обработки при несоблюдении условия цикла.

Использование цикла `while` в приложении

Рассмотрим самый простой пример, в котором программа выводит значение счетчика `Sum`. Чтобы создать и протестировать пример, выполните следующие действия.

1. **Введите следующий код в блокноте.**

```
Sum = 0
while Sum < 5:
   print(Sum)
   Sum+=1
```

Здесь демонстрируется выполнение трех типичных задач. Сначала переменной `Sum` присваивается значение `0`. Далее проверяется условие цикла. В последней инструкции цикла выполняется приращение счетчика. Прежде чем обновить значение переменной `Sum`, программа отображает ее текущее значение.

ЗАПОМНИ!

Цикл `while` обеспечивает гибкость, которой не может похвастаться инструкция `for`. В данном примере показан относительно простой способ обновления переменной `Sum`, хотя она может обновляться произвольным образом. Кроме того, условие цикла может быть сколь угодно сложным. При желании можно отслеживать текущие значения трех или четырех переменных. Понятно, что чем сложнее условия, тем выше вероятность создать бесконечный цикл. На практике пользователь сам решает, насколько сложным должно быть условие цикла `while`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа выполнит цикл `while` и отобразит числовую последовательность (рис. 9.7).

```
In [8]: Sum = 0
        while Sum < 5:
            print(Sum)
            Sum+=1

        0
        1
        2
        3
        4
```

Рис. 9.7. Простой цикл `while`*, в котором отображается последовательность чисел*

Вложенные циклы

Иногда циклы `for` и `while` позволяют достичь одного и того же результата. Несмотря на то что сами циклы работают по-разному, эффект получается одинаковый. В данном примере мы создадим генератор таблицы умножения путем вложения цикла `while` в цикл `for`. Чтобы конечный результат выглядел красиво, мы применим специальное форматирование (подробности будут приведены в главе 12).

1. **Введите следующий код в блокноте.**

```
X = 1
Y = 1
print('{:>4}'.format(' '), end= ' ')
for X in range(1, 11):
    print('{:>4}'.format(X), end=' ')
print()
for X in range(1, 11):
    print('{:>4}'.format(X), end=' ')
    while Y <= 10:
        print('{:>4}'.format(X * Y), end=' ')
        Y+=1
    print()
    Y=1
```

Программа начинается с создания двух переменных, `X` и `Y`, которые содержат значения строки и столбца таблицы. `X` — это номер строки, а `Y` — номер столбца.

Чтобы таблица получилась наглядной, необходимо создать одну строку сверху и один столбец слева. Значение, стоящее на пересечении строки и столбца, представляет собой произведение их номеров.

Первая инструкция `print()` добавляет пустое пространство в левом верхнем углу таблицы (рис. 9.8). Суть правила форматирования заключается в создании столбца шириной 4 символа и заполнении его пробелами. Выражение `{:>4}` задает размер столбца. Функция `format(' ')` определяет символ-заполнитель. Атрибут `end` функции `print()` заменяет конечный символ возврата каретки обычным пробелом.

Первый цикл `for` предназначен для отображения чисел от 1 до 10 в заголовке таблицы. Функция `range()` создает числовую последовательность. При этом указываются начальное (`1`) и конечное значения (`11`).

По завершении цикла курсор находится в конце строки заголовка. Чтобы переместить его на следующую строку, необходимо вызвать функцию `print()` без аргументов.

Следующую часть программы мы проанализируем строка за строкой. Поскольку в таблице умножения отображаются значения от `1*1` до `10*10`, нам нужны десять строк и десять столбцов. В цикле `for` создает десять строк.

Взгляните еще раз на рис. 9.8. У каждой строки есть свой номер, который отображается с помощью первой функции `print()` в цикле. Этот номер тоже должен быть отформатирован. Программа формирует столбец из четырех символов, который завершается пробелом, а не символом возврата каретки.

Далее идет цикл `while`, в котором формируется строка результатов умножения столбец за столбцом. Значения столбцов — это произведения `X * Y`. Чтобы столбец занимал четыре символа, результат тоже следует отформатировать. В конце значение переменной `Y` увеличивается на единицу с помощью инструкции `Y+=1`, после чего цикл `while` завершается.

Мы снова возвращаемся к циклу `for`. Функция `print()` завершает текущую строку. Переменная `Y` должна быть установлена равной `1`, чтобы начать следующую строку с умножения на `1`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Появится таблица умножения (рис. 9.8).

Вложенные циклы

```
In [9]: X = 1
        Y = 1
        print ('{:>4}'.format(' '), end= ' ')
        for X in range(1, 11):
           print('{:>4}'.format(X), end=' ')
        print()
        for X in range(1,11):
           print('{:>4}'.format(X), end=' ')
           while Y <= 10:
              print('{:>4}'.format(X * Y), end=' ')
              Y+=1
           print()
           Y=1
```

```
     1    2    3    4    5    6    7    8    9   10
 1   1    2    3    4    5    6    7    8    9   10
 2   2    4    6    8   10   12   14   16   18   20
 3   3    6    9   12   15   18   21   24   27   30
 4   4    8   12   16   20   24   28   32   36   40
 5   5   10   15   20   25   30   35   40   45   50
 6   6   12   18   24   30   36   42   48   54   60
 7   7   14   21   28   35   42   49   56   63   70
 8   8   16   24   32   40   48   56   64   72   80
 9   9   18   27   36   45   54   63   72   81   90
10  10   20   30   40   50   60   70   80   90  100
```

Рис. 9.8. Благодаря форматированию таблица умножения выглядит красиво

Глава 10

Обработка ошибок

В ЭТОЙ ГЛАВЕ...

- Причины возникновения ошибок
- Индикаторы ошибок
- Создание собственных индикаторов ошибок
- Выполнение обработки после возникновения ошибок

В любом коде независимо от уровня его сложности содержатся ошибки. Внезапное "зависание" приложения без видимой причины является следствием ошибки. Непонятные сообщения в диалоговых окнах тоже возникают из-за ошибок. К сожалению, программы не всегда уведомляют пользователей об ошибках. Если вы предоставите неверные данные, приложение выполнит неверные расчеты и выдаст некорректный результат, а вы даже не поймете этого, пока кто-то не подскажет вам или вы сами не проверите расчеты. Ошибки могут проявляться спорадически. Порой вы их замечаете, порой — нет. Например, ошибка может возникать в плохих погодных условиях или при перегрузке сети. Иными словами, ошибки возникают в произвольных ситуациях и в силу непредсказуемых причин. В этой главе рассматриваются различные типы ошибок и способы их устранения.

Появление ошибок не должно вас удивлять. Программы пишутся людьми, а людям, как известно, свойственно ошибаться. Многие разработчики называют ошибки приложения *исключениями*, подразумевая исключения из правил. Исключения в приложениях следует обнаруживать и при первой же возможности обрабатывать. Процесс обнаружения и обработки исключений называется *обработкой ошибок* или *обработкой исключений*. Для правильного выявления

ошибок нужно понимать их источники и причины их возникновения. Обнаруженную ошибку обрабатывают путем *перехвата* исключения. Это подразумевает анализ исключения и выполнение определенных действий. Вторая часть главы посвящена способам обработки исключений в приложениях.

Иногда программа сама обнаруживает ошибочную ситуацию. В таком случае следует *возбудить* или *сгенерировать* исключение. Оба термина используются для обозначения ситуации, когда программа сталкивается с ошибкой, не может с ней справиться и поэтому передает информацию о ней другому коду для *обработки* (интерпретации и, если повезет, исправления). В некоторых случаях для передачи информации используются настраиваемые объекты сообщений об ошибках. Python располагает множеством объектов обобщенных сообщений, подходящих для большинства ситуаций, но порой и их оказывается недостаточно. Например, вы хотите написать приложение для работы с базой данных, а в Python для этого нет объектов обобщенных сообщений. Необходимо знать, когда следует обрабатывать исключения локально, когда отправлять их вызывающему коду, а когда создавать специальные исключения, чтобы в каждом блоке приложения выполнялась корректная обработка исключений. Все эти вопросы тоже рассматриваются в данной главе.

Часто требуется обеспечить корректную обработку исключений, даже если это подразумевает прекращение работы программы. К счастью, в Python есть предложение `finally`, которое выполняется всегда, даже при возникновении исключений. В него можно добавить код, предназначенный для закрытия файлов или выполнения других финальных задач. Об этом мы поговорим в конце главы. Исходный код примеров данной главы находится в файле `BPPD_10_Dealing_with_Errors.ipynb` (см. введение).

Почему Python вас не понимает

Очень часто разработчики сталкиваются с ситуациями, когда программы и компьютеры становятся неуправляемыми. Проблема, разумеется, не в языках программирования и не в компьютерах, ведь они не умеют ни думать, ни хотеть чего-либо. Просто они в буквальном виде воспринимают все то, что им сообщает разработчик. В этом и кроется суть проблемы.

ЗАПОМНИ!

Ни Python, ни компьютер не понимают смысла инструкций. Они выполняют любые инструкции буквально, так, как их дает пользователь. Вряд ли вы вздумаете попросить Python, чтобы он удалил важный файл, если только не возникнет какая-то аномальная ситуация. Но Python нужны четкие инструкции относительно того, как именно

распознать аномалию, и если вы не предоставили таких инструкций, то он может удалить файл помимо вашего желания. В подобных ситуациях обычно говорят о *сбое*, или ошибкеприложения. Ошибки в коде устраняют с помощью отладчика.

Отладчик — это специальная утилита, которая позволяет остановить или приостановить выполнение приложения, просмотреть содержимое переменных и проанализировать код в поисках причины ошибки.

Ошибки чаще всего возникают, когда разработчик делает неверные предположения. Сюда можно отнести и предположения о пользователе, который, вероятно, даже не догадывается о том, сколько труда и заботы вкладывает разработчик в свое приложение. Иногда пользователь вводит неверные данные, а Python не понимает, что данные ошибочны, и продолжает их обрабатывать, даже если вы не планировали обработку некорректных данных. Python не видит различий между верными и неверными данными; он просто обрабатывает их согласно заданным правилам. А это значит, что нужно устанавливать такие правила, которые защищали бы пользователей от них же самих.

Python ничего не может продумывать наперед, поскольку не обладает аналитическим мышлением, присущим человеку. В случае сетевой ошибки или непредвиденных действий пользователя Python не сможет придумать решение для устранения ошибки. Он просто выполняет код. Если вы сами не предоставите код для решения проблемы, то приложение, скорее всего, аварийно завершится, потеряв все несохраненные данные. В силу того что разработчик не может предвидеть все возможные ситуации, даже в очень сложных приложениях возникают ошибки по недоработке самого разработчика.

ВНИМАНИЕ!

Некоторые разработчики наивно полагают, что могут создавать супернадежный код, несмотря на абсурдность самой этой мысли. Более грамотные разработчики допускают, что какие-то ошибки останутся незамеченными в процессе отладки, что человек и природа будут по-прежнему вызывать непредвиденные события и что даже самый умный программист не способен предусмотреть все возможные условия возникновения ошибок. Не забывайте о том, что ваше приложение всегда будет подвержено ошибкам, вызывающим исключения. Рассуждая подобным образом, вы станете на путь создания действительно надежных приложений. Согласно закону Мерфи "все, что может пойти не так, пойдет не так". (Дополнительные сведения о законах Мерфи можно найти на сайте `www.murphys-laws.com`.)

Возможные причины появления ошибок

Путем гадания на кофейной гуще можно было бы предсказать потенциальные источники появления ошибок в приложении, но вряд ли это поможет. Ошибки обычно делятся на две четкие категории, которые до определенной степени помогают предугадать, когда и где они произойдут. Держа в голове эти категории, можно с большей долей вероятности обнаружить потенциальные источники ошибок, прежде чем они навредят приложению. Основные категории ошибок таковы:

- ошибки, возникающие в определенное время;
- ошибки, имеющие специфический тип.

В следующих разделах эти две категории ошибок рассматриваются более детально. Прежде чем приступать к обнаружению и устранению потенциальных ошибок, необходимо разобраться в их классификации.

Этапы появления ошибок

Ошибки проявляются в определенное время, а именно:

- на этапе компиляции;
- на этапе выполнения.

Независимо от времени возникновения все ошибки приводят к сбою в работе приложения. В следующих разделах эти категории ошибок рассматриваются подробнее.

Ошибки компиляции

Ошибка компиляции возникает, когда вы просите Python запустить приложение. Прежде чем Python сможет выполнить приложение, он должен интерпретировать код и преобразовать его в форму, понятную компьютеру. Компьютер ориентируется на машинный код, специфичный для данного процессора и архитектуры. Если инструкции написаны неправильно или неполные, Python не сможет выполнить необходимое преобразование и сообщит об ошибке, которую следует устранить, прежде чем приложение сможет быть запущено.

К счастью, ошибки компиляции обнаруживаются и устраняются легче всего. Пользователь даже не заметит их, поскольку приложение просто не запустится. Ошибки этого типа устраняются в процессе написания кода.

СОВЕТ

Ошибка компиляции должна подсказать, что в данном фрагменте кода могут быть и другие ошибки или пропуски. Рекомендуется проверять весь окружающий код на предмет наличия других

потенциальных проблем, которые могут быть не обнаружены в процессе компиляции.

Ошибки выполнения

Ошибка выполнения возникает, когда Python скомпилировал написанный код и компьютер начинает его выполнять. Ошибки выполнения бывают нескольких типов, причем одни из них обнаружить сложнее, чем другие. Когда происходит ошибка выполнения, приложение внезапно завершает работу и выводит диалоговое окно с описанием исключения. Или же пользователь начинает жаловаться на получение неверных результатов или, крайней мере, нестабильную работу программы.

ЗАПОМНИ!

Не все ошибки выполнения приводят к возникновению исключений. В результате некоторых ошибок приложение начинает работать нестабильно (зависает), выдавать некорректные результаты или повреждать данные. Ошибки выполнения могут негативно влиять на другие приложения или вызывать сбои в системе, в которой запускается приложение. Другими словами, подобные ошибки могут доставить пользователю немало проблем.

Многие ошибки выполнения возникают из-за ошибок в коде. Например, вы неправильно указали имя переменной, и в итоге Python записывает данные не в ту переменную. Или был пропущен необязательный, но важный аргумент при вызове метода. Это примеры ошибок кодирования. Обнаружить их можно с помощью отладчика или просто путем построчной вычитки кода.

Ошибки выполнения могут быть также вызваны внешними причинами, не связанными с кодом. Например, если пользователь введет некорректную информацию, не ожидаемую приложением, может быть сгенерировано исключение. Из-за сетевой ошибки требуемый ресурс может стать недоступным. Иногда даже сбой аппаратного обеспечения компьютера вызывает неповторяющуюся ошибку приложения. От всех вышеперечисленных ошибок приложение можно избавить, если в нем есть код перехвата ошибок. Существуют два типа ошибок выполнения, которые допускаются при написании приложения: ошибки в результате нарушения правил и пропуски.

Типы ошибок

Ошибки можно различать в зависимости от того, как они возникают. Зная, какие бывают типы ошибок, вы будете представлять, где их искать в приложении. Здесь все как в жизни. Например, зная, что электронные устройства не работают без питания, вы всегда проверяете, подключен ли к розетке шнур питания телевизора, если при включении он не работает.

СОВЕТ

Понимание типов ошибок способствует их более быстрому обнаружению, благодаря чему сокращается время отладки. Опытные разработчики знают, что проще устранять ошибки в процессе написания кода, чем во время эксплуатации приложения, когда пользователи начинают проявлять нетерпение и хотят немедленного исправления программы. Кроме того, отладка на стадии разработки всегда проще, чем на завершающей стадии, поскольку придется проверять меньше кода.

Важно знать, где искать ошибки. В Python (как и в большинстве языков программирования) ошибки делятся на следующие типы:

- синтаксические;
- семантические;
- логические.

В следующих разделах более подробно описывается каждый из этих типов ошибок. Мы начнем с самых легких в обнаружении синтаксических ошибок, а закончим самыми трудными — логическими.

Синтаксические ошибки

Любая допущенная вами опечатка — это синтаксическая ошибка. Некоторые синтаксические ошибки довольно легко обнаруживаются в Python, поскольку приложение просто не запускается. Интерпретатор может даже указать на ошибку, выделив ошибочный код и отобразив сообщение об ошибке. Тем не менее некоторые синтаксические ошибки довольно сложно выявить. Python, например, чувствителен к регистру символов, поэтому, если в имени переменной где-то был использован неверный регистр, программа может работать не так, как ожидается. И найти то место, где произошел случайный сбой, окажется непросто.

ЗАПОМНИ!

Большинство синтаксических ошибок проявляется в процессе компиляции, и интерпретатор на них указывает явно. Устранение таких ошибок не представляет трудности благодаря тому, что интерпретатор достаточно точно сообщает, что именно нужно исправить. Даже если интерпретатор не выявил проблему, синтаксические ошибки приводят к некорректной работе приложения, и, как следствие, любые ошибки, не обнаруженные интерпретатором, всплывают в ходе тестирования. Если приложение было адекватно протестировано, то навряд ли в финальном релизе останутся синтаксические ошибки.

Семантические ошибки

Когда вы создаете цикл, который выполняется избыточное число раз, вы, скорее всего, не получите никаких сообщений об ошибках на этапе компиляции. Приложение успешно запустится, поскольку интерпретатор считает, что все работает корректно. Тем не менее одна лишняя итерация цикла способна привести к целому каскаду ошибок. Такого рода ошибки в коде называется *семантическими*.

Семантические ошибки возникают из-за неверного понимания последовательности выполняемых действий. Программа выдает некорректные результаты даже несмотря на то, что, казалось бы, работает в точности так, как ожидает разработчик. Такого рода ошибки трудно обнаруживать, и приходится прибегать к помощи отладчика. (В главе 20 рассматриваются инструменты, применяемые для отладки приложений Python.)

Логические ошибки

Некоторые разработчики не разделяют семантические и логические ошибки, хотя это разные вещи. Семантическая ошибка имеет место в том случае, когда синтаксически правильный код используется некорректно (например, цикл выполняется излишнее число раз). Логические ошибки являются результатом неверных рассуждений разработчика. Например, разработчик может путать и неверно применять логические операторы или, допустим, полагать, что данные всегда хранятся на локальном жестком диске, и, как следствие, при попытке загрузить данные с сетевого диска приложение ведет себя некорректно.

Логические ошибки очень трудно выявлять, ведь проблема, по сути, не в самом коде, а в его неверном понимании. В данном случае ошибочны сами рассуждения, которые привели к написанию именно такого кода, а значит, разработчику, допустившему ошибку, будет сложно самому ее обнаружить. Опытный разработчик для выявления логических ошибок обращается за помощью к коллеге. Поможет также наличие формальной спецификации приложения, поскольку при ее составлении формализуются все логические рассуждения.

Перехват исключений

Вообще говоря, пользователь не должен видеть никаких сообщений об исключении. Приложение всегда должно перехватывать исключения и

обрабатывать их до того, как о них узнает пользователь. Конечно, в реальности все немного не так, и пользователи время от времени сталкиваются с непредвиденными исключениями. И тем не менее при разработке приложения ставится цель обработать каждое потенциальное исключение. В следующих разделах описывается порядок перехвата и обработки исключений.

Базовая обработка исключений

Чтобы обрабатывать исключения, нужно сообщить Python о своем намерении и предоставить соответствующий код. Это можно сделать несколькими способами. Сначала мы рассмотрим простейший способ, а затем перейдем к более гибким решениям.

ВСТРОЕННЫЕ ИСКЛЮЧЕНИЯ

В Python поддерживается множество встроенных исключений — намного больше, чем можно себе представить. Список этих исключений доступен на сайте `https://docs.python.org/3.7/library/exceptions.html`. В документации список исключений разбит на категории. Ниже представлен краткий обзор категорий исключений Python, с которыми вы будете работать регулярно.

- **Базовые классы.** Это в основном строительные блоки для других исключений, как, например, исключение `Exception`. Тем не менее с некоторыми из этих исключений, в частности, `ArithmeticError`, можно столкнуться в приложении.
- **Конкретные исключения.** Приложения могут сталкиваться с серьезными ошибками, которые трудно исправить, поскольку не существует "хорошего" способа обработки таких ошибок либо они сигнализируют о событиях, которые должны обрабатываться приложением. Например, когда в системе имеет место нехватка памяти, Python генерирует исключение `MemoryError`. Устранить такую ошибку навряд ли получится, поскольку не всегда можно освободить память, занимаемую другими приложениями. Когда пользователь нажимает комбинацию клавиш <Ctrl+C> или <Delete>, сигнализирующую о прерывании процесса, Python генерирует исключение `KeyboardInterrupt`. Приложение должно обработать это исключение перед выполнением любых других задач.
- **Исключения операционной системы.** Операционная система может генерировать ошибки, которые Python передает приложению. Например, если приложение пытается открыть несуществующий файл, операционная система генерирует исключение `FileNotFoundError`.

- **Предупреждения.** Python пытается предупредить пользователей о непредвиденных событиях или действиях, которые могут позднее привести к ошибкам. Например, если попытаться использовать ресурс, такой как значок, ненадлежащим образом, Python сгенерирует исключение `ResourceWarning`. Следует помнить о том, что это лишь предупреждения, а не ошибки. Конечно, если на них не обращать внимания, то это может привести к проблемам в дальнейшем, но все же предупреждения можно игнорировать.

Обработка единственного исключения

Программы, которые рассматривались в главе 8, имели неприятную особенность: они генерировали исключения в случае ввода пользователем непредвиденных значений. Частично проблема решается путем проверки диапазона. Но это не поможет, если пользователь вместо числовых значений начнет вводить текст, например `Hello`. Более целостное решение обеспечивается путем обработки исключений, как описано ниже.

1. **Откройте новый блокнот.**

   Можете также воспользоваться файлом исходного кода `BPPD_10_Dealing_with_Errors.ipynb` (см. введение).

2. **Введите следующий код в блокноте.**

```
try:
   Value = int(input("Введите число от 1 до 10: "))
except ValueError:
   print("Нужно ввести число от 1 до 10!")
else:
   if (Value > 0) and (Value <= 10):
      print("Вы ввели: ", Value)
   else:
      print("Введенное значение некорректно!")
```

   Инструкция в блоке `try` может сгенерировать исключение, которое будет обработано. В данном случае исключение возникает при получении данных от пользователя с помощью вызова `int(input())`. Исключения, возникающие за пределами этого блока, не обрабатываются. У вас может возникнуть соблазн заключить весь исполняемый код в блок `try` и тем самым обрабатывать каждое исключение. Но на практике код обработки делают максимально компактным и конкретным, чтобы четко локализовать источник проблемы.

ЗАПОМНИ!

   В нашем примере блок `except` перехватывает одно конкретное исключение: `ValueError`. Оно будет сгенерировано и обработано, когда пользователь введет строку, например `Hello`, вместо числового значения.

Другие исключения, сгенерированные каким-то иным способом, не будут обрабатываться в данном блоке `except`.

Блок `else` содержит код, который выполняется, когда блок `try` завершается успешно (не генерируя исключение). Этот код не должен выполняться, если пользователь вводит некорректные данные. Когда пользователь вводит целое число, программа проверяет, попадает ли число в заданный диапазон.

3. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа попросит ввести число от 1 до 10.

4. **Введите `Hello` и нажмите клавишу <Enter>.**

   Программа отобразит сообщение об ошибке (рис. 10.1).

Перехват исключений

Базовая обработка исключений

Обработка единственного исключения

```
In [2]: try:
            Value = int(input("Введите число от 1 до 10: "))
        except ValueError:
            print("Нужно ввести число от 1 до 10!")
        else:
            if (Value > 0) and (Value <= 10):
                print("Вы ввели: ", Value)
            else:
                print("Введенное значение некорректно!")

        Введите число от 1 до 10: Hello
        Нужно ввести число от 1 до 10!
```

Рис. 10.1. Ввод некорректных данных теперь приводит к появлению сообщения об ошибке, а не исключению

5. **Повторите пп. 3 и 4, но вместо `Hello` введите `5.5`.**

   Приложение генерирует такое же сообщение об ошибке, как на рис. 10.1.

6. **Повторите пп. 3 и 4, но вместо `Hello` введите `22`.**

   Теперь программа выводит сообщение об ошибке диапазона (рис. 10.2). Обработка исключений не устраняет такого рода ошибки. Их все равно придется проверять отдельно.

7. **Повторите пп. 3 и 4, но на этот раз введите `7` вместо `Hello`.**

   Приложение сообщит о том, что было введено значение `7`. Несмотря на то что подобный уровень проверки потребовал большого объема работы, он все же необходим для обеспечения корректной работы приложения.

```
In [3]: try:
            Value = int(input("Введите число от 1 до 10: "))
        except ValueError:
            print("Нужно ввести число от 1 до 10!")
        else:
            if (Value > 0) and (Value <= 10):
                print("Вы ввели: ", Value)
            else:
                print("Введенное значение некорректно!")

        Введите число от 1 до 10: 22
        Введенное значение некорректно!
```

Рис. 10.2. Обработка исключений не гарантирует, что значение будет находиться в корректном диапазоне

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Возможно, придется проверять и другие ошибки, в зависимости от среды, в которой выполняется тестирование. Например, если вы работаете в IDLE, а не Notebook, то при нажатии комбинации клавиш <Ctrl+C>, <Cmd+C> или каком-то другом способе неожиданного прерывания будет сгенерировано исключение `KeyboardInterrupt`. Notebook обрабатывает это исключение автоматически, так что в этой среде вам ничего не придется делать. Но нельзя предполагать, что и все остальные пользователи приложения будут работать только в Notebook.

Использование предложения `except` без указания исключения

Можно создать обобщенный блок обработки исключений в Python, который будет предназначен для перехвата любых исключений. Чаще всего мы перехватываем конкретные исключения, руководствуясь следующими соображениями:

- чтобы не пропустить исключение, которое не было учетно при разработке приложения;
- чтобы другие пользователи точно знали, какие именно исключения обрабатывает ваше приложение;
- чтобы корректно обработать исключение, используя специально предназначенный для него код.

Но иногда требуется именно универсальный обработчик, например при использовании сторонних библиотек или взаимодействии с внешними службами. В следующей пошаговой инструкции демонстрируется использование предложения `except`, в котором не указано никакого конкретного исключения.

1. **Введите следующий код в блокноте.**

```
try:
   Value = int(input("Введите число от 1 до 10: "))
except:
   print("Обобщенная ошибка!")
except ValueError:
   print("Нужно ввести число от 1 до 10!")
else:
   if (Value > 0) and (Value <= 10):
      print("Вы ввели: ", Value)
   else:
      print("Введенное значение некорректно!")
```

Единственное отличие от предыдущего примера заключается в появлении предложения `except`, не связанного с исключением `ValueError`. В результате это предложение может перехватывать любое другое возникающее исключение.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Это приведет к появлению сообщения об ошибке (рис. 10.3). Python автоматически обнаруживает, что порядок расположения обработчиков исключений некорректен. (Эта тема будет подробно рассмотрена далее.) Поменяйте местами обработчики, чтобы код выглядел так, как показано ниже.

```
try:
   Value = int(input("Введите число от 1 до 10: "))
except ValueError
   print("Нужно ввести число от 1 до 10!")
except:
   print("Обобщенная ошибка!")
else:
   if (Value > 0) and (Value <= 10):
      print("Вы ввели: ", Value)
   else:
      print("Введенное значение некорректно!")
```

3. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение попросит ввести число, находящееся в диапазоне от 1 до 10.

4. **Введите слово `Hello` и нажмите клавишу <Enter>.**

Приложение выдаст сообщение об ошибке (см. рис. 10.1). Если обработчики стоят в правильном порядке, то программа в первую очередь обработает конкретную ошибку, а менее конкретные обработчики вызываются только тогда, когда это необходимо.

5. **Опять щелкните на кнопке Run Cell.**

Приложение попросит ввести число, находящееся в диапазоне от 1 до 10.

```
In [1]: try:
            Value = int(input("Введите число от 1 до 10: "))
        except:
            print("Обобщенная ошибка!")
        except ValueError:
            print("Нужно ввести число от 1 до 10!")
        else:
            if (Value > 0) and (Value <= 10):
                print("Вы ввели: ", Value)
            else:
                print("Введенное значение некорректно!")

          File "<ipython-input-1-e575efc38fb4>", line 2
            Value = int(input("Введите число от 1 до 10: "))
            ^
        SyntaxError: default 'except:' must be last
```

Рис. 10.3. Порядок расположения обработчиков исключений некорректен

6. **Выполните команду Kernel⇨Interrupt (Ядро⇨Прервать).**

   Это действие равнозначно нажатию комбинации клавиш <Ctrl+C> или <Cmd+C> в других IDE. Сначала кажется, будто ничего не происходит. Но посмотрите на окно сервера, и вы увидите там сообщение `Kernel Interrupted` (Выполнение ядра прервано).

7. **Введите `5.5` и нажмите клавишу <Enter>.**

   В результате появится обобщенное сообщение об ошибке (рис. 10.4), поскольку Notebook реагирует на прерывание, а не на ввод некорректных данных, ведь прерывание произошло раньше. Python формирует очередь ошибок в том порядке, в котором он их получает. Из-за этого иногда возникает впечатление, что приложение Python выводит некорректные сообщения об ошибках.

```
In [7]: try:
            Value = int(input("Введите число от 1 до 10: "))
        except ValueError:
            print("Нужно ввести число от 1 до 10!")
        except:
            print("Обобщенная ошибка!")
        else:
            if (Value > 0) and (Value <= 10):
                print("Вы ввели: ", Value)
            else:
                print("Введенное значение некорректно!")

        Введите число от 1 до 10: 5.5
        Обобщенная ошибка!
```

Рис. 10.4. Обобщенный обработчик перехватывает исключение `KeyboardInterrupt`

8. **Повторите пп. 3 и 4, но вместо `Hello` введите `5.5`.**

   Приложение выдаст то же сообщение об ошибке, что и в самом начале (см. рис. 10.1). В этом случае прерывания не происходит, поэтому отображается ожидаемое сообщение.

Обработка аргументов исключения

Большинство исключений не содержит аргументов (т.е. список значений, которые можно проверять для получения дополнительной информации). Исключение либо перехватывается, либо нет. Но некоторые исключения все же предоставляют аргументы, и в последующих главах вы увидите, как они используются на практике. Аргументы содержат информацию об исключении, требуемую для корректной обработки.

Для полноты изложения эта глава содержит простой пример, в котором генерируется исключение с аргументами. Можете без особого ущерба для понимания материала пропустить оставшуюся часть раздела, поскольку мы вернемся к этой теме в последующих главах.

1. **Введите следующий код в блокноте.**

```
import sys
try:
   File = open('myfile.txt')
except IOError as e:
   print("Ошибка при открытии файла!\r\n" +
      "Номер ошибки: {0}\r\n".format(e.errno) +
      "Текст ошибки: {0}".format(e.strerror))
else:
   print("Файл открыт, как ожидалось.")
   File.close();
```

   В этом примере используются несколько не рассматривавшихся ранее элементов. В частности, инструкция `import` подключает код из другого файла. Подробнее она будет описана в главе 11.

   Функция `open()` открывает файл и предоставляет доступ к нему через переменную `File`. Порядок доступа к файлам будет описан в главе 16. Учитывая то, что файла `myfile.txt` нет в каталоге приложения, операционная система не сможет его открыть и сообщит об этом Python.

   Попытка открытия несуществующего файла приводит к появлению исключения `IOError`, которое содержит два аргумента:

   - `errno` — целочисленный номер ошибки, полученный от операционной системы;
   - `strerror` — строка с информацией об ошибке.

Предложение `as` помещает информацию об исключении в переменную `e`, к которой можно получить доступ в блоке `except`. Далее вызывается функция `print()`, которая выводит на экран описание ошибки.

Если вы все же решите создать файл `myfile.txt`, то будет вызвано предложение `else`. В данном случае выводится сообщение о том, что файл открыт в нормальном режиме. Затем программа закрывает файл, не выполняя над ним никаких действий.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение отобразит информацию об ошибке открытия файла, как показано на рис. 10.5.

```
In [8]: import sys
        try:
           File = open('myfile.txt')
        except IOError as e:
           print("Ошибка при открытии файла!\r\n" +
              "Номер ошибки: {0}\r\n".format(e.errno) +
              "Текст ошибки: {0}".format(e.strerror))
        else:
           print("Файл открыт, как ожидалось.")
           File.close();

        Ошибка при открытии файла!
        Номер ошибки: 2
        Текст ошибки: No such file or directory
```

Рис. 10.5. Попытка открыть несуществующий файл завершилась неудачей

ПОЛУЧЕНИЕ СПИСКА АРГУМЕНТОВ ИСКЛЮЧЕНИЯ

Список аргументов, передаваемый вместе с исключением, зависит как от самого исключения, так и от того, что предоставляет отправитель. Не всегда можно понять, какая именно информация предоставляется в ваше распоряжение. Один из способов решения проблемы — просто вывести весь список с помощью следующего кода.

```
import sys
try:
   File = open('myfile.txt')
except IOError as e:
   for Arg in e.args:
      print(Arg)
else:
   print("Файл открыт, как ожидалось.")
   File.close();
```

Свойство `args` всегда содержит список аргументов исключения в строковом формате. Можно воспользоваться простым циклом `for` для вывода на экран каждого аргумента. Однако возникает одна проблема: отсутствуют имена аргументов, поэтому вы видите выводимую информацию, но не знаете, каким аргументам она соответствует.

Более сложный подход, позволяющий устранить описанную проблему, заключается в выводе на экран имен и значений аргументов.

```
import sys
try:
   File = open('myfile.txt')
except IOError as e:
   for Entry in dir(e):
      if (not Entry.startswith("_")):
         try:
            print(Entry, " = ", e.__getattribute__(Entry))
         except AttributeError:
            print("Атрибут ", Entry, " недоступен.")
else:
   print("Файл открыт, как ожидалось.")
   File.close();
```

В этом случае все начинается с получения списка атрибутов, связанных с объектом ошибки, с помощью функции `dir()`. Она возвращает список строк, содержащих имена атрибутов, которые можно вывести на экран. Но только те аргументы, которые не начинаются символом подчеркивания (_), содержат полезную информацию об исключении. Впрочем, не все из этих аргументов доступны, поэтому нужно заключить код вывода во второй блок `try...except` (подробности будут приведены в разделе "Обработка вложенных исключений").

Имя атрибута вывести несложно, поскольку оно содержится в переменной `Entry`. А вот чтобы получить значение, связанное с этим атрибутом, нужно воспользоваться функцией `__getattribute()`, передав ей имя атрибута. После запуска программы отобразится список имен и значений каждого атрибута, переданного вместе с объектом исключения. В данном случае мы получаем следующий результат.

```
args = (2, 'No such file or directory')
Attribute characters_written not accessible.
errno = 2
filename = myfile.txt
filename2 = None
strerror = No such file or directory
winerror = None
with_traceback = <built-in method with_traceback of
   FileNotFoundError object at 0x0000000003416DC8>
```

Обработка нескольких исключений с помощью одного предложения `except`

Большинство приложений может генерировать несколько исключений в одной строке кода, как было продемонстрировано в разделе "Использование предложения `except` без указания исключения". Порядок обработки нескольких исключений зависит от целей приложения, типов исключений и уровня пользователей. Иногда менее опытным пользователям проще заявить о том, что в приложении возникла неустранимая ошибка, и занести детали в файл журнала, находящийся в каталоге приложения или в центральном хранилище.

ЗАПОМНИ!

Использование одного предложения `except` для перехвата нескольких исключений допустимо, только если все эти исключения обрабатываются по одной и той же процедуре. В остальных случаях нужно обрабатывать каждое исключение отдельно. Ниже демонстрируется обработка нескольких исключений с помощью единственного предложения `except`.

1. **Введите следующий код в блокноте.**

```
try:
   Value = int(input("Введите число от 1 до 10: "))
except (ValueError, KeyboardInterrupt):
   print("Нужно ввести число от 1 до 10!")
else:
   if (Value > 0) and (Value <= 10):
      print("Вы ввели: ", Value)
   else:
      print("Введенное значение некорректно!")
```

ЗАПОМНИ!

Обратите внимание на то, что теперь предложение `except` перехватывает исключения `ValueError` и `KeyboardInterrupt`. Они заключены в скобки и разделены запятыми.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение попросит ввести число от 1 до 10.

3. **Введите `Hello` и нажмите клавишу <Enter>.**

   Приложение отобразит сообщение об ошибке (см. рис. 10.1).

4. **Снова щелкните на кнопке Run Cell.**

   Приложение выдаст тот же запрос.

5. **Выполните команду Kernel⇨Interrupt (Ядро⇨Прервать).**

   Это эквивалентно нажатию комбинации клавиш <Ctrl+C> или <Cmd+C> в других IDE.

6. **Введите `5.5` и нажмите клавишу <Enter>.**

   Приложение отобразит то же самое сообщение об ошибке.

7. **Повторите пп. 2 и 3, но вместо `Hello` введите `7`.**

   Приложение сообщит о том, что было введено корректное значение, равное 7.

Обработка нескольких исключений с помощью нескольких предложений `except`

При обработке нескольких исключений обычно рекомендуется перехватывать каждое исключение с помощью собственного предложения `except`. Это позволяет настраивать обработку для каждого исключения и дает пользователю понять, что именно пошло не так. Естественно, такой подход оказывается более трудоемким. Ниже показано, как выполнить обработку исключений с помощью нескольких предложений `except`.

1. **Введите следующий код в блокноте.**

```
try:
   Value = int(input("Введите число от 1 до 10: "))
except ValueError:
   print("Нужно ввести число от 1 до 10!")
except KeyboardInterrupt:
   print("Вы нажали клавиши Ctrl+C!")
else:
   if (Value > 0) and (Value <= 10):
      print("Вы ввели: ", Value)
   else:
      print("Введенное значение некорректно!")
```

ЗАПОМНИ!

Обратите внимание на то, что в этом случае используются несколько предложений `except`. Каждое из них обрабатывает другое исключение. Вполне допустим и комбинированный подход, когда некоторые предложения `except` перехватывают только одно исключение, а некоторые — несколько исключений. Вы вольны выбрать подход, который лучше всего подходит для конкретной ситуации.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение попросит ввести число от 1 до 10.

3. **Введите `Hello` и нажмите клавишу <Enter>.**

   Приложение выведет сообщение об ошибке (см. рис. 10.1).

4. **Повторите пп. 2 и 3, но вместо `Hello` введите `22`.**

   Приложение отобразит сообщение об ошибке диапазона (см. рис. 10.2).

5. **Повторите пп. 2 и 3, но теперь выполните команду Kernel⇨Interrupt (Ядро⇨Прервать), введите 5.5 и нажмите клавишу <Enter>.**

   Приложение выдаст специфическое сообщение об ошибке, как показано на рис. 10.6.

```
In [8]: try:
            Value = int(input("Введите число от 1 до 10: "))
        except ValueError:
            print("Нужно ввести число от 1 до 10!")
        except KeyboardInterrupt:
            print("Вы нажали клавиши Ctrl+C!")
        else:
            if (Value > 0) and (Value <= 10):
                print("Вы ввели: ", Value)
            else:
                print("Введенное значение некорректно!")

        Введите число от 1 до 10: 5.5
        Вы нажали клавиши Ctrl+C!
```

Рис. 10.6. Благодаря использованию нескольких предложений `except` *можно генерировать специфические сообщения об ошибках*

6. **Повторите пп. 2 и 3, но вместо `Hello` введите 7.**

   На этот раз приложение сообщит о том, что было введено корректное значение, равное 7.

Иерархическая обработка менее специфичных исключений

Одна из стратегий, применяемых для обработки исключений, заключается в использовании конкретизированных предложений `except` для обработки всех известных исключений и обобщенных предложений `except` для обработки неизвестных исключений. Иерархия исключений Python описана по следующему адресу:

`https://docs.python.org/3/library/exceptions.html#exception-hierarchy`

Просматривая иерархию, обратите внимание на то, что на самом верхнем уровне находится исключение `BaseException`. Ну а большинство исключений являются производными от объекта `Exception`. При обработке математических ошибок можно использовать как обобщенное исключение `ArithmeticError`, так и, к примеру, конкретизированное исключение `ZeroDivisionError`.

Python просматривает предложения `except` в том порядке, в котором они встречаются в исходном коде. Первое предложение проверяется первым, второе — вторым и т.д. Следующий пример демонстрирует, насколько важно соблюдать корректный порядок следования исключений. В данном случае будут выполнены операции, приводящие к математическим ошибкам.

1. **Введите следующий код в блокноте.**

```
try:
   Value1 = int(input("Введите первое число: "))
   Value2 = int(input("Введите второе число: "))
   Output = Value1 / Value2
except ValueError:
   print("Нужно ввести целое число!")
except KeyboardInterrupt:
   print("Вы нажали клавиши Ctrl+C!")
except ArithmeticError:
   print("Произошла непредвиденная математическая ошибка.")
except ZeroDivisionError:
   print("Попытка деления на нуль!")
else:
   print(Output)
```

Сначала вводятся два значения: `Value1` и `Value2`. Первые два предложения `except` предназначены для обработки результатов некорректного ввода. Следующие два предложения `except` предназначены для обработки математических исключений, таких как деление на 0. Если при выполнении приложения не возникает ошибок, вызывается предложение `else`, в котором выводится результат выполнения операции.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа попросит ввести первое число.

3. **Введите `Hello` и нажмите клавишу <Enter>.**

Как и ожидалось, Python отобразит сообщение, соответствующее исключению `ValueError`.

4. **Снова щелкните на кнопке Run Cell.**

Программа опять попросит ввести первое число.

5. **Введите `8` и нажмите клавишу <Enter>.**

Программа запросит ввод второго числа.

6. **Введите `0` и нажмите клавишу <Enter>.**

Появится сообщение, соответствующее исключению `ArithmeticError`, как показано на рис. 10.7. Но это не совсем то, чего мы ожидали, ведь исключение `ZeroDivisionError` более специфично, чем `ArithmeticError`.

7. **Поменяем порядок обработки исключений, как показано ниже.**

```
except ZeroDivisionError:
   print("Попытка деления на 0")
except ArithmeticError:
   print("Произошла непредвиденная математическая ошибка.")
```

```
In [9]: try:
            Value1 = int(input("Введите первое число: "))
            Value2 = int(input("Введите второе число: "))
            Output = Value1 / Value2
        except ValueError:
            print("Нужно ввести целое число!")
        except KeyboardInterrupt:
            print("Вы нажали клавиши Ctrl+C!")
        except ArithmeticError:
            print("Произошла непредвиденная математическая ошибка.")
        except ZeroDivisionError:
            print("Попытка деления на нуль!")
        else:
            print(Output)

        Введите первое число: 8
        Введите второе число: 0
        Произошла непредвиденная математическая ошибка.
```

Рис. 10.7. Порядок обработки исключений в Python имеет значение

8. **Снова выполните пп. 4–6.**

   Появится сообщение, соответствующее исключению `ZeroDivisionError`, поскольку теперь исключения перехватываются корректно.

9. **Повторите пп. 4–5, но на этот раз в качестве второго числа введите 2 вместо 0.**

   Приложение выведет результат `4.0`, как показано на рис. 10.8.

```
In [10]: try:
             Value1 = int(input("Введите первое число: "))
             Value2 = int(input("Введите второе число: "))
             Output = Value1 / Value2
         except ValueError:
             print("Нужно ввести целое число!")
         except KeyboardInterrupt:
             print("Вы нажали клавиши Ctrl+C!")
         except ArithmeticError:
             print("Произошла непредвиденная математическая ошибка.")
         except ZeroDivisionError:
             print("Попытка деления на нуль!")
         else:
             print(Output)

         Введите первое число: 8
         Введите второе число: 2
         4.0
```

Рис. 10.8. При правильных входных данных приложение возвращает корректный результат

Заметьте, что результат, показанный на рис. 10.8, является значением с плавающей точкой. Дело в том, что результатом деления всегда будет значение с плавающей точкой. Если же вам нужен целочисленный результат, воспользуйтесь оператором деления нацело (//).

Обработка вложенных исключений

Иногда обработчик одного исключения необходимо расположить внутри другого обработчика. Это называется *вложением исключений*. При вложении обработчиков Python сначала пытается найти обработчик на вложенном уровне, а затем — на внешнем. Глубина вложений может быть произвольной.

Двойная обработка исключений применяется в том случае, когда нужно получать входные данные от пользователя в цикле, чтобы гарантировать получение всей информации. Ниже показан пример такого кода.

1. **Введите следующий код в блокноте.**

```
TryAgain = True
while TryAgain:
   try:
      Value = int(input("Введите целое число."))
   except ValueError:
      print("Нужно ввести целое число!")
      try:
         DoOver = input("Попытаться снова (y/n)? ")
      except:
         print("Хорошо, увидимся в следующий раз!")
         TryAgain = False
      else:
         if (str.upper(DoOver) == "N"):
            TryAgain = False
   except KeyboardInterrupt:
      print("Вы нажали клавиши Ctrl+C!")
      print("Увидимся в следующий раз!")
      TryAgain = False
   else:
      print(Value)
      TryAgain = False
```

Программа начинается с создания цикла ввода. Использование циклов в подобных случаях оправдано, поскольку вы не хотите, чтобы приложение завершало работу всякий раз, когда допускается ошибка ввода данных. В этом примере цикл упрощен, так как обычно код выносится в отдельную функцию.

Когда начинается цикл, программа предлагает пользователю ввести целое число, которое может быть любым. Если пользователь вводит нецелочисленное значение либо нажимает комбинацию клавиш <Ctrl+C>, <Cmd+C> и т.п.,

возникает исключение. В противном случае приложение отображает значение, введенное пользователем, и присваивает переменной `TryAgain` значение `False`, которое вызывает завершение цикла.

Исключение `ValueError` возникает в тех случаях, когда пользователь допускает ошибку. Мы не знаем, почему пользователь вводит неправильное значение, а потому спрашиваем, хочет ли он повторить попытку. Ну а поскольку пользователь должен ввести ответ на этот вопрос, может возникнуть новое исключение. Оно будет обрабатываться внутренним блоком `try...except`.

СОВЕТ

Обратите внимание на использование функции `str.upper()` при получении символьных данных, вводимых пользователем. Это позволяет не делать различий между символами `n` и `N`, принимая оба варианта. Всякий раз, когда вы запрашиваете у пользователя ввод символов, имеет смысл переводить символы нижнего регистра в верхний регистр, чтобы вместо двух проверок делать одну (уменьшая тем самым вероятность ошибок).

ЗАПОМНИ!

Исключение `KeyboardInterrupt` возникает, когда пользователь нажимает определенную комбинацию клавиш, чтобы завершить выполнение приложения. Обработчик исключения `KeyboardInterrupt` отображает два сообщения и присваивает переменной `TryAgain` значение `False`, что приводит к завершению цикла.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение попросит ввести целое число.

3. **Введите `Hello` и нажмите клавишу <Enter>.**

   Приложение отобразит сообщение об ошибке и спросит, хотите ли вы попытаться еще раз.

4. **Введите `Y` и нажмите клавишу <Enter>.**

   Приложение снова запросит ввод целого числа, как показано на рис. 10.9.

5. **Введите значение `5.5` и нажмите клавишу <Enter>.**

   Приложение отобразит сообщение об ошибке и запросит повторный ввод данных.

6. **Теперь выполните команду Kernel⇨Interrupt (Ядро⇨Прервать), чтобы прервать работу ядра, введите `Y` и нажмите клавишу <Enter>.**

   Приложение завершит работу, как показано на рис. 10.10. Обратите внимание на то, что сообщение выводится внутренним обработчиком. Внешний обработчик никогда не будет вызван, потому что внутренний обработчик перехватывает все исключения.

7. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение попросит ввести целое число.

```
In [*]: TryAgain = True
        while TryAgain:
            try:
                Value = int(input("Введите целое число. "))
            except ValueError:
                print("Нужно ввести целое число!")
                try:
                    DoOver = input("Попытаться снова (y/n)? ")
                except:
                    print("Хорошо, увидимся в следующий раз!")
                    TryAgain = False
                else:
                    if (str.upper(DoOver) == "N"):
                        TryAgain = False
            except KeyboardInterrupt:
                print("Вы нажали клавиши Ctrl+C!")
                print("Увидимся в следующий раз!")
                TryAgain = False
            else:
                print(Value)
                TryAgain = False

        Введите целое число. Hello
        Нужно ввести целое число!
        Попытаться снова (y/n)? Y

        Введите целое число.
```

Рис. 10.9. Использование цикла позволяет приложению обрабатывать ошибки ввода

8. **Выполните команду Kernel⇨Interrupt (Ядро⇨Прервать), чтобы прервать работу ядра, введите значение 5 . 5 и нажмите клавишу <Enter>.**

   Приложение завершится, как показано на рис. 10.11. На этот раз сообщение выводится внешним обработчиком. В пп. 6 и 8 пользователь завершает работу приложения, нажимая соответствующую комбинацию клавиш, но в каждом из случаев вызываются разные обработчики.

Генерирование исключений

До сих пор в примерах главы мы занимались обработкой исключений. Возникала та или иная нештатная ситуация, и приложение определенным образом реагировало на нее. Но иногда вы не знаете заранее, на этапе разработки приложения, как обработать ошибку. Возможно, ошибка даже не подлежит обработке на текущем уровне, и ее нужно передать на какой-то другой уровень для обработки. Короче говоря, в определенных ситуациях приложение должно *генерировать* исключение. В следующих разделах описаны распространенные сценарии, когда это происходит.

```
In [13]: TryAgain = True
         while TryAgain:
            try:
               Value = int(input("Введите целое число. "))
            except ValueError:
               print("Нужно ввести целое число!")
               try:
                  DoOver = input("Попытаться снова (y/n)? ")
               except:
                  print("Хорошо, увидимся в следующий раз!")
                  TryAgain = False
               else:
                  if (str.upper(DoOver) == "N"):
                     TryAgain = False
            except KeyboardInterrupt:
               print("Вы нажали клавиши Ctrl+C!")
               print("Увидимся в следующий раз!")
               TryAgain = False
            else:
               print(Value)
               TryAgain = False

         Введите целое число. Hello
         Нужно ввести целое число!
         Попытаться снова (y/n)? Y
         Введите целое число. 5.5
         Нужно ввести целое число!
         Попытаться снова (y/n)? Y
         Хорошо, увидимся в следующий раз!
```

Рис. 10.10. Внутренний обработчик исключений позволяет запрашивать повторный ввод данных

Генерирование исключений в нештатных ситуациях

В этом примере демонстрируется генерирование простого исключения, не требующего ничего особенного. Мы генерируем и тут же обрабатываем исключение.

1. **Введите следующий код в блокноте.**

```
try:
   raise ValueError
except ValueError:
   print("Исключение ValueError!")
```

Вряд ли есть какая-то практическая польза от этого кода, но он дает общее представление о том, как генерируются исключения на простейшем уровне. В данном случае исключение генерируется в блоке `try...except`. В простейшем случае просто указывается имя исключения. Можно также задавать аргументы для передачи дополнительной информации.

```
In [1]: TryAgain = True
        while TryAgain:
           try:
              Value = int(input("Введите целое число. "))
           except ValueError:
              print("Нужно ввести целое число!")
              try:
                 DoOver = input("Попытаться снова (y/n)? ")
              except:
                 print("Хорошо, увидимся в следующий раз!")
                 TryAgain = False
              else:
                 if (str.upper(DoOver) == "N"):
                    TryAgain = False
           except KeyboardInterrupt:
              print("Вы нажали клавиши Ctrl+C!")
              print("Увидимся в следующий раз!")
              TryAgain = False
           else:
              print(Value)
              TryAgain = False

        Введите целое число. 5.5
        Вы нажали клавиши Ctrl+C!
        Увидимся в следующий раз!
```

Рис. 10.11. Внешний обработчик исключений позволяет контролировать правильность ввода данных

ЗАПОМНИ!

Обратите внимание на то, что в этом блоке `try...except` отсутствует предложение `else`, что редко встречается на практике. Как бы там ни было, предложение `else` является необязательным, в отличие от предложения `except`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа отобразит сообщение об исключении, как показано на рис. 10.12.

```
In [1]: try:
           raise ValueError
        except ValueError:
           print("Исключение ValueError!")

        Исключение ValueError!
```

Рис. 10.12. Чтобы сгенерировать исключение, достаточно выполнить инструкцию `raise`

Передача информации об ошибке вызывающему коду

В Python обеспечивается исключительно гибкая обработка ошибок, поскольку можно передавать информацию вызывающему коду независимо от используемого исключения. Конечно, вызывающий код не знает заранее о том, что ему передаются аргументы, и это приводит к определенным трудностям. Если вы работаете с чужим кодом и не знаете, доступна ли дополнительная информация, вы всегда можете использовать методику, описанную во врезке "Получение списка аргументов исключения".

Вы наверняка задались вопросом, можно ли предоставить лучшую информацию при работе с исключением `ValueError`, чем та, что передается встроенными средствами Python. Ниже показано, как это сделать.

1. **Введите следующий код в блокноте.**

```
try:
   Ex = ValueError()
   Ex.strerror = "Значение должно находиться в пределах от 1 до 10."
   raise Ex
except ValueError as e:
   print("Исключение ValueError!", e.strerror)
```

Исключение `ValueError` обычно не включает атрибут `strerror` (строка с описанием ошибки), но его можно добавить путем простого присваивания значения. Генерируемое исключение перехватывается предложением `except`, которое получает доступ к его атрибутам через объект `e`. Нужное нам значение содержится в переменной `e.strerror`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение отобразит расширенную информацию об исключении `ValueError`, как показано на рис. 10.13.

```
In [2]: try:
            Ex = ValueError()
            Ex.strerror = "Значение должно находиться в пределах от 1 до 10."
            raise Ex
        except ValueError as e:
            print("Исключение ValueError!", e.strerror)

        Исключение ValueError! Значение должно находиться в пределах от 1 до 10.
```

Рис. 10.13. К любому исключению можно добавить информацию об ошибке

Создание и использование настраиваемых исключений

В Python доступно множество стандартных исключений, которые следует использовать в большинстве ситуаций. Эти исключения невероятно гибкие и легко поддаются модификации (разумеется, в разумных пределах), если возникает такая необходимость. Например, в предыдущем разделе демонстрировалось, как изменить исключение `ValueError` таким образом, чтобы оно включало дополнительные данные. И все же бывают ситуации, когда вы вынуждены создавать настраиваемые исключения, поскольку ни одно из стандартных исключений не подходит. Это может понадобиться при работе с базами данных или веб-службами.

ВНИМАНИЕ!

Пример, рассматриваемый в этом разделе, может показаться немного сложным, поскольку мы еще не работали с классами. О них мы детально поговорим в главе 15. Можете смело пропустить этот раздел и вернуться к нему после того, как прочитаете главу 15.

В приводимом ниже примере демонстрируется быстрый способ создания настраиваемых исключений. Для этого нам нужно создать класс, основанный на существующем исключении. Чтобы немного облегчить задачу, мы создадим новое исключение на основе исключения `ValueError`. По сравнению с подходом, описанным в предыдущем разделе, это дает нам то преимущество, что любой программист, который впоследствии будет работать с данным кодом, сможет четко понять, каким именно образом модифицируется исключение `ValueError`. Кроме того, это упрощает работу с измененным исключением.

1. **Введите следующий код в блокноте.**

```
class CustomValueError(ValueError):
   def __init__(self, arg):
      self.strerror = arg
      self.args = {arg}
try:
   raise CustomValueError("Значение должно находиться в пределах
      от 1 до 10.")
except CustomValueError as e:
   print("Исключение CustomValueError!", e.strerror)
```

Функционально это тот же пример, что и в предыдущем разделе, только здесь одна и та же информация помещается в атрибуты `strerror` и `args`, благодаря чему разработчик может получить доступ к обоим атрибутам (что обычно и происходит на практике).

Вначале мы создаем класс `CustomValueError`, который использует класс исключения `ValueError` в качестве стартовой точки. Функция-конструктор `__init__()` предназначена для создания нового экземпляра класса. Класс можно рассматривать как чертеж здания, а экземпляр — как само здание, построенное на основе этого чертежа.

ЗАПОМНИ!

Заметьте, что атрибуту `strerror` непосредственно присваивается значение, тогда как атрибут `args` получает его в виде массива. Переменная-член `args` обычно содержит массив всех аргументов исключения, так что это стандартная практика, даже если атрибут `args` содержит лишь одно значение, как в данном случае.

Код, генерирующий исключение, здесь понятнее, чем в случае непосредственного изменения исключения `ValueError`. Все, что нам нужно сделать, — вызвать инструкцию `raise`, указав имя исключения и список передаваемых вместе с ним аргументов.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа выведет информацию о настраиваемом исключении, как показано на рис. 10.14.

```
In [3]: class CustomValueError(ValueError):
            def __init__(self, arg):
                self.strerror = arg
                self.args = (arg)
        try:
            raise CustomValueError("Значение должно находиться в пределах от 1 до 10.")
        except CustomValueError as e:
            print("Исключение CustomValueError!", e.strerror)

        Исключение CustomValueError! Значение должно находиться в пределах от 1 до 10.
```

Рис. 10.14. Благодаря настраиваемым исключениям код становится более понятным

Использование предложения `finally`

Обычно любые исключения должны обрабатываться таким образом, чтобы приложение не завершало работу аварийно. Но иногда ничего нельзя сделать, и аварийное завершение неизбежно. В таком случае наша цель — позволить приложению корректно завершиться, т.е. закрыть все файлы для предотвращения потери пользовательских данных и выполнить другие задачи подобного рода. То есть нужно сделать все возможное, чтобы минимизировать ущерб для системы и данных.

В подобной ситуации на помощь приходит предложение `finally`. В нем можно выполнять любые завершающие действия. Обычно его стараются сделать максимально кратким, включив только те вызовы, которые гаранти-

рованно завершатся успешно и не вызовут дальнейших проблем. Крайне важно закрыть все файлы, завершить пользовательский сеанс и выполнить другие необходимые действия, после чего позволить приложению аварийно завершиться, прежде чем произойдет что-нибудь совсем плохое (например, системный сбой). И все это можно сделать с помощью предложения `finally`, как продемонстрировано в следующей пошаговой инструкции.

1. **Введите следующий код в блокноте.**

```
import sys
try:
   raise ValueError
   print("Генерирование исключения.")
except ValueError:
   print("Исключение ValueError!")
   sys.exit()
finally:
   print("Уделим внимание последним деталям.")
print("Этот код не будет выполняться.")
```

В этом примере программа генерирует исключение `ValueError`. Предложение `except` выполняется как обычно. Вызов `sys.exit()` означает, что приложение завершает работу по окончании обработки исключения. Вот почему финальный вызов функции `print()` никогда не произойдет.

ЗАПОМНИ!

Код предложения `finally` выполняется всегда. И не важно, возникло исключение или нет. Код, находящийся в этом блоке, должен быть универсальным, таким, который всегда нужен. Например, при работе с файлом в блок `finally` включается код закрытия файла, гарантирующий, что данные не будут повреждены, оставаясь в памяти, а не на диске.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение выведет сообщения, находящиеся в предложениях `except` и `finally`, как показано на рис. 10.15. Вызов `sys.exit()` предотвращает вызов любого последующего кода.

ЗАПОМНИ!

Учтите, что это не является нормальным завершением приложения, и Notebook выводит соответствующую информацию по данному поводу. В других IDE, таких как IDLE, приложение просто завершает работу, не выводя никакой дополнительной информации.

3. **А теперь закомментируйте вызов `ValueError`, предварив его двумя символами решетки, как показано ниже:**

```
##raise ValueError
```

Это позволит лучше понять, как работает предложение `finally`.

4. **Снова щелкните на кнопке Run Cell.**

Приложение отобразит ряд сообщений, включая сообщение блока `finally`, как показано на рис. 10.16. Это показывает, что предложение `finally` выполняется всегда, поэтому используйте его аккуратно.

```
In [1]: import sys
        try:
           raise ValueError
           print("Генерирование исключения.")
        except ValueError:
           print("Исключение ValueError!")
           sys.exit()
        finally:
           print("Уделим внимание последним деталям.")
        print("Этот код не будет выполняться.")

        Исключение ValueError!
        Уделим внимание последним деталям.

        An exception has occurred, use %tb to see the full traceback.

        SystemExit


        F:\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py:2971: UserWarning: To exit: use 'exit', 'quit', or Ct
        rl-D.
          warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

Рис. 10.15. Использование предложения `finally` для выполнения необходимых действий перед завершением работы приложения

```
In [2]: import sys
        try:
           ## raise ValueError
           print("Генерирование исключения.")
        except ValueError:
           print("Исключение ValueError!")
           sys.exit()
        finally:
           print("Уделим внимание последним деталям.")
        print("Этот код не будет выполняться.")

        Генерирование исключения.
        Уделим внимание последним деталям.
        Этот код не будет выполняться.
```

Рис. 10.16. Необходимо учитывать, что предложение `finally` выполняется всегда

3 Профессиональные методики программирования на Python

В ЭТОЙ ЧАСТИ...

- Импорт и использование пакетов
- Работа со строками
- Создание списков объектов
- Использование коллекций для расширения функциональности списка
- Разработка классов

Глава 11

Пакеты

В ЭТОЙ ГЛАВЕ...

- Структурирование кода
- Добавление кода из внешних источников
- Поиск и просмотр библиотек кода
- Получение и чтение документации к библиотекам

Поскольку примеры, используемые в книге, довольно небольшие, функциональность результирующих приложений тоже будет сильно ограничена. Реальные приложения, даже самые маленькие, содержат тысячи строк кода. Нередко встречаются приложения, содержащие миллионы строк кода. Представьте себе, что вы пытаетесь что-либо найти в подобном огромном файле. Вряд ли у вас это получится. Короче говоря, нужен какой-то способ структурирования кода, позволяющий разбить его на фрагменты, которыми было бы легче управлять. Решение, применяемое в Python, заключается в размещении кода в отдельных блоках, называемых *пакетами* (или *модулями*, что одно и то же.) Часто применяемые пакеты, содержащие исходный код общего назначения, называются *библиотеками*.

ЗАПОМНИ!

Пакеты содержатся в отдельных файлах. Чтобы использовать пакет, необходимо попросить Python загрузить файл и подключить его к текущему приложению. Процесс получения кода, находящегося во внешних файлах, называется *импортом*. В нескольких примерах книги уже было продемонстрировано использование инструкции `import`; в этой главе мы рассмотрим его подробнее.

В процессе начальной установки Python создает ссылку на библиотеки общего назначения. Вот почему можно просто добавить в программу инструкцию `import` с именем библиотеки, и Python найдет ее. Тем не менее полезно знать, как найти файлы на диске, если нужно обновить их или добавить собственные пакеты и библиотеки в список файлов, используемых в Python.

Код библиотеки является автономным и хорошо документированным (по крайней мере, в большинстве случаев). Необходимость просматривать исходный код библиотеки обычно не возникает, однако вы вправе это сделать, если хотите разобраться, как он работает. Кроме того, просматривая код библиотеки, можно научиться новым приемам программирования, которые вы навряд ли встретите где-то еще. То есть это не обязательная, но полезная практика.

Главное, что нужно знать, — как получить документацию по библиотеке Python. В этой главе будет показано, как воспользоваться документацией в процессе создания приложения. Исходный код примеров данной главы находится в файле `BPPD_11_Interacting_with_Packages.ipynb` (см. введение). Примеры пакетов доступны в файле `BPPD_11_Packages.ipynb`.

Создание блоков кода

Код становится проще использовать, редактировать и понимать, когда он разбит на логические блоки. По мере разрастания приложения управлять кодом, хранящимся в одном файле, будет все сложнее и сложнее, и в какой-то момент вы поймете, что уже не в состоянии контролировать настолько большой листинг.

ЗАПОМНИ!

В данном случае термин *код* употребляется в широком смысле. Блоки кода могут включать:

- классы;
- функции;
- переменные;
- исполняемые файлы.

Коллекции, состоящие из классов, функций, переменных и исполняемого кода, называются *атрибутами*. Пакет содержит атрибуты, к которым пользователь обращается по именам. Далее мы рассмотрим, как именно осуществляется доступ к пакетам.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Исполняемый код может быть написан на других языках программирования. Например, нередко можно встретить пакеты, написанные на C/C++ (а не на Python). Причина, по которой некоторые разработчики используют готовый код, заключается в том, что это позволяет сделать приложение Python более быстрым, менее ресурсоемким и лучше использующим ресурсы конкретной платформы. Однако применение исполняемого кода приводит к тому, что приложение становится менее переносимым (способным работать на других платформах), если только в вашем распоряжении нет пакетов исполняемого кода для каждой платформы, которую нужно поддерживать. Кроме того, многоязычные приложения сложнее поддерживать, поскольку в вашей команде должны быть разработчики, знающие каждый из компьютерных языков, используемых в приложении.

Самый распространенный способ создания пакета — вынести группируемый код в отдельный файл. Например, можно написать процедуру печати, предназначенную для использования в разных частях приложения. Она не является самостоятельной программой, это просто структурная часть приложения, но она выделяется в отдельный файл, поскольку обращение к ней происходит из разных файлов и, возможно, даже из других программ. Повторное использование кода — одно из главных преимуществ пакетов.

Для простоты в примерах этой главы используется общий пакет. Он не особо полезен, его назначение — демонстрация принципов работы с пакетами. Откройте проект Python 3 Notebook, назовите его `BPPD_11_Packages` и введите код, показанный в листинге 11.1. После этого загрузите код в виде нового файла Python с именем `BPPD_11_Packages.py`, выбрав в блокноте команду File⇨Download As⇨Python (.py).

Листинг 11.1. Простой демонстрационный пакет

```
def SayHello(Name):
   print("Привет, ", Name)
   return
def SayGoodbye(Name):
   print("Пока, ", Name)
   return
```

Может понадобиться скопировать полученный файл в существующую папку `BPPD`, в зависимости от того, куда ваш браузер обычно загружает файлы. Если все сделано правильно, файл появится на панели Notebook, как показано на рис. 11.1. Все существенно упростится, если выполнить импорт блокнота, как описывалось в главе 4.

Рис. 11.1. Не забудьте поместить копию пакета в папку `BPPD`

В нашем пакете содержатся две простые функции: `SayHello()` и `SayGoodbye()`. В обоих случаях нужно указать аргумент `Name`, и функция выведет его на экран вместе с приветствием для пользователя, после чего вернет управление вызывающему коду. Конечно, можно создать и более сложные функции, но для целей главы вполне подойдут и эти.

Типы пакетов

Система поддержки Python огромна. Вы никогда не будете использовать ее в полном объеме, даже в самых требовательных приложениях. И дело даже не в громоздкости Python, который на самом деле довольно лаконичен по сравнению со многими другими языками программирования. Необходимость такой поддержки объясняется наличием большого количества пакетов Python, которые применяются везде: в интенсивных научных вычислениях, задачах искусственного интеллекта, космических исследованиях, биологическом моделировании и пр. Однако не все эти пакеты доступны на постоянной основе, поэтому нужно знать, какие пакеты поддерживает Python и где их можно найти (в порядке предпочтения).

- **Встроенные пакеты.** Они позволяют решать все распространенные задачи и находятся в папке `Adaconda3\Lib`. Все, что вам нужно сделать, чтобы их использовать, — импортировать их в свое приложение.

- **Пользовательские пакеты.** Как будет показано далее, можно создавать собственные пакеты и использовать их в случае необходимости. Они обычно находятся в том же каталоге, что и файлы проекта, и вы просто импортируете их в приложение.
- **Пакеты conda.** Существует множество пакетов, специально разработанных для Anaconda. Многие из них доступны по адресу `http://conda.anaconda.org/mutirri`. Прежде чем их использовать, нужно установить их с помощью утилиты `conda` в командной строке Anaconda, как будет описано в разделе "Установка пакетов `conda`". После завершения установки пакет можно использовать подобно любому встроенному пакету.
- **Пакеты, не являющиеся пакетами conda.** Тот факт, что пакет не предназначен специально для использования с Anaconda, не означает, что его нельзя использовать. Можно найти огромное количество полезных пакетов, разработанных сторонними разработчиками. Для их установки используется утилита командной строки Anaconda `pip`, как будет описано в разделе "Установка пакетов с помощью утилиты `pip`". После установки пакета может потребоваться сконфигурировать его в соответствии с требованиями разработчика. После того как пакет настроен, он используется как обычный встроенный пакет.

Кеш пакетов

В Anaconda имеется кеш пакетов, благодаря которому можно легко взаимодействовать с пакетами Anaconda с помощью утилиты командной строки `conda`. Чтобы увидеть, как используется кеш пакетов, откройте командную строку Anaconda или окно терминала, воспользовавшись ссылкой `Anaconda Prompt` в папке `Anaconda3`. Введите команду **`conda list`**, чтобы просмотреть список установленных пакетов. На рис. 11.2 показаны типичные результаты, которые получены в данном случае.

ЗАПОМНИ!

Обратите внимание на то, что имя пакета отображается в том виде, в котором оно выводится в окне Anaconda. Также отображаются версия пакета и соответствующая версия Python. Ниже перечислены основные команды `conda`, применяемые для управления пакетами.

- `conda clean`. Удаление неиспользуемых пакетов.
- `conda config`. Конфигурирование кеша пакетов.
- `conda create`. Создание новой среды `conda` с конкретным списком пакетов, что позволяет упростить управление пакетами и ускорить работу приложения.

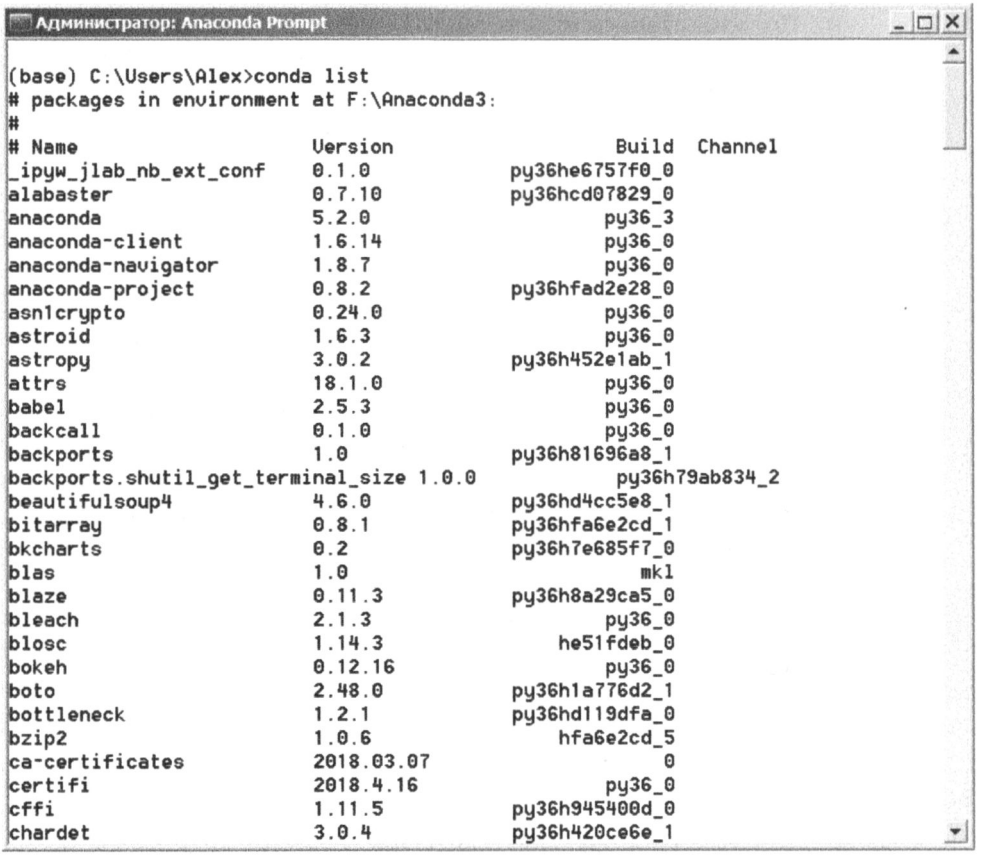


Рис. 11.2. Список пакетов Anaconda, полученный с помощью утилиты команд ной строки `conda`

- `conda help`. Вывод полного списка команд `conda`.
- `conda info`. Отображение сведений о конфигурации `conda`, в том числе о местонахождении пакетов `conda` и новых пакетов.
- `conda install`. Установка одного или нескольких пакетов в заданную по умолчанию или указанную среду `conda`.
- `conda list`. Вывод списка пакетов `conda` с настраиваемым уровнем детализации. Можно указать, какие именно пакеты должны выводиться и в каких средах их следует искать.
- `conda remove`. Удаление одного или нескольких пакетов из кеша пакетов.
- `conda search`. Поиск указанных пакетов с помощью заданных критериев поиска.
- `conda update`. Обновление некоторых или всех пакетов в кеше пакетов.

СОВЕТ

Поскольку всю эту информацию невозможно держать в голове, пользуйтесь переключателем командной строки `--help` для получения подробных сведений по конкретной команде. Например, для получения дополнительных сведений о команде `conda list` введите **`conda list --help`** и нажмите клавишу <Enter>.

Импорт пакетов

Чтобы воспользоваться пакетом, его нужно импортировать. Python помещает код пакета вместе с остальной частью приложения в память, как будто был создан один огромный файл. При этом сами файлы на диске не меняются — это по-прежнему разные файлы, просто Python трактует их код по-другому.

ЗАПОМНИ!

Импортировать пакеты можно двумя способами, в зависимости от ситуации.

- **Инструкция `import`.** Применяется в случае импорта целого пакета. Этот способ чаще всего используется разработчиками кода, поскольку экономит время и требует всего лишь одной строки кода. С другой стороны, такой подход требует больше памяти, чем подход, предусматривающий выборочный импорт атрибутов.
- **Инструкция `from...import`.** Применяется в том случае, когда нужно выборочно импортировать отдельные атрибуты пакета. Это позволяет экономить ресурсы, но сам по себе такой способ сложнее. Кроме того, если попытаться использовать атрибут, который не был импортирован, Python выдаст ошибку. Да, пакет содержит атрибут, но Python не видит, поскольку он не был импортирован.

В следующих разделах эти инструкции будут рассмотрены подробнее.

ВЗАИМОДЕЙСТВИЕ С ТЕКУЩИМ КАТАЛОГОМ PYTHON

Каталог, который Python использует для доступа к коду, влияет на то, какие пакеты можно загрузить. Каталог библиотечных файлов всегда включен в список мест, к которым Python может получить доступ, но Python ничего не знает о каталоге, который вы используете для хранения исходного кода, если только вы не попросите просматривать этот каталог. Это требует знакомства с каталожными функциями, позволяющими сообщить Python, где искать нужные фрагменты кода. Соответствующий пример можно найти в файле `BPPD_11_Directory.ipynb` (см. введение).

1. **Откройте новый блокнот.**
2. **Введите `import os`.**

   Это действие приводит к импорту библиотеки `os`. Ее нужно импортировать, чтобы заменить текущий каталог (место, которое Python ищет на диске) рабочим каталогом данной книги.
3. **Введите `print(os.getcwd())` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   В результате вы увидите имя текущего рабочего каталога (`cwd`), который используется в Python для получения локального кода.
4. **В новой ячейке введите `for entry in os.listdir() : print(entry)` и щелкните на кнопке Run Cell.**

   На экране отобразится список записей каталога. Это позволяет определить, находится ли нужный вам файл в каталоге `cwd`. Если нет, нужно заменить текущий каталог каталогом, содержащим нужный файл.

Чтобы изменить местоположение текущего каталога, воспользуйтесь методом `os.chdir()`, указав имя каталога в виде строки, например `os.chdir('C:\MyDir')`. Но обычно вы увидите, что в каталоге `cwd` находятся файлы текущего проекта.

Использование инструкции `import`

Для импорта пакетов в Python чаще всего применяется инструкция `import`. Этот подход достаточно быстрый и гарантирует, что весь пакет будет готов к использованию. Выполните следующие действия.

1. **Откройте новый блокнот.**

   Можете также воспользоваться файлом исходного кода `BPPD_11_Interacting_with_Packages.ipynb` (см. введение).
2. **При необходимости перейдите в каталог загружаемого исходного кода.**

   Обычно Notebook выбирает корректный каталог для использования файлов исходного кода, поэтому выполнять данный шаг не понадобится. Прочитайте также врезку "Взаимодействие с текущим каталогом Python".
3. **Введите `import BPPD_11_Packages`.**

   Эта инструкция указывает Python на необходимость импорта содержимого файла `BPPD_11_Packages.py`, который был создан в разделе "Создание блоков кода". Теперь вся библиотека готова к использованию.

ВНИМАНИЕ!

Важно понимать, что Python также создает кеш пакета в подкаталоге `__pycache__` после первого импорта файла `BPPD_11_Packages`. Если нужно внести изменения в пакет, удалите этот подкаталог, иначе Python продолжит использовать старый файл кеша вместо обновленного файла исходного кода.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Кешированное имя файла включает версию Python, для которой предназначен пакет. В данном случае это `BPPD_11_Packages.cpython-37.pyc`. Число `37` в имени файла означает, что файл относится к Python 3.7. Файл с расширением `.pyc` представляет собой скомпилированный файл Python, который используется для увеличения быстродействия приложения.

4. **Введите `dir(BPPD_11_Packages)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появится список содержимого пакета, который включает функции `SayHello()` и `SayGoodbye()`, как показано на рис. 11.3. (Дополнительная информация будет приведена в разделе "Просмотр содержимого пакета".)

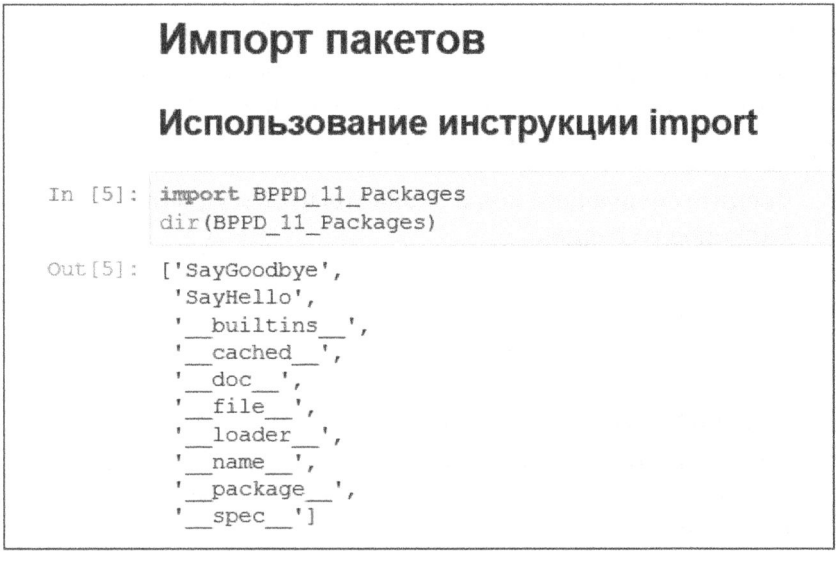


Рис. 11.3. Python импортирует обе функции из пакета

5. **В новой ячейке введите `BPPD_11_Packages.SayHello("Иван")`.**

   Учтите, что имя атрибута — в данном случае функцию `SayHello()` — нужно предварять именем пакета, т.е. `BPPD_11_Packages`. Два элемента разделены точкой. Подобным образов осуществляются любые обращения к импортируемым пакетам.

6. **Введите `BPPD_11_Packages.SayGoodbye("Мария")` и щелкните на кнопке Run Cell (Выполнить ячейку).**

Функции `SayHello()` и `SayGoodbye()` выводят ожидаемый результат, как показано на рис. 11.4.

```
In [2]: BPPD_11_Packages.SayHello("Иван")
        BPPD_11_Packages.SayGoodbye("Мария")

        Привет,  Иван
        Пока,  Мария
```

Рис. 11.4. Функции `SayHello()` и `SayGoodbye()` выводят ожидаемый результат

Использование инструкции `from...import`

Преимущество инструкции `from...import` заключается в импорте из пакета только необходимых атрибутов. Это означает, что пакет потребляет меньше памяти и других системных ресурсов, чем при обращении к инструкции `import`. Кроме того, инструкция `from...import` упрощает использование пакета, поскольку некоторые команды, такие как `dir()`, отображают меньше информации или только ту информацию, которая вам действительно нужна. В результате вы получаете только то, что хотите, и ничего больше. Ниже пошагово продемонстрировано использование инструкции `from...import`.

1. **Введите следующий код в блокноте, чтобы удалить пакет `BPPD_11_Packages` из среды.**

```
import sys
del sys.modules["BPPD_11_Packages"]
del BPPD_11_Packages
dir(BPPD_11_Packages)
```

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Появится сообщение об ошибке, показанное на рис. 11.5. Вывод содержимого пакета `BPPD_11_Packages` невозможен, поскольку этот пакет больше не загружается.

3. **В новой ячейке введите `from BPPD_11_Packages import SayHello`.**

   Python импортирует функцию `SayHello()`, которая была создана в разделе "Создание блоков кода". Только эту конкретную функцию можно теперь использовать в приложении.

ЗАПОМНИ!

При желании можно импортировать весь пакет. Это реализуется двумя способами. Во-первых, можно указать список атрибутов через запятую, например `from BPPD_11_Packages import SayHello, SayGoodbye`, или поставить звездочку (`*`) вместо имени атрибута. Звездочка — это символ подстановки, означающий импорт всего содержимого пакета.

```
In [13]: import sys
         del sys.modules["BPPD_11_Packages"]
         del BPPD_11_Packages
         dir(BPPD_11_Packages)

         ---------------------------------------------------------------------------
         NameError                                 Traceback (most recent call last)
         <ipython-input-13-a56565a3615c> in <module>()
               2 del sys.modules["BPPD_11_Packages"]
               3 del BPPD_11_Packages
         ----> 4 dir(BPPD_11_Packages)

         NameError: name 'BPPD_11_Packages' is not defined
```

Рис. 11.5. Процесс удаления пакета из среды

4. **Введите `dir(BPPD_11_Packages)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Python отобразит сообщение об ошибке, показанное на рис. 11.5. Причина в том, что Python импортирует только те атрибуты, которые были явным образом запрошены. Это означает, что пакет `BPPD_11_Packages` не загружен в память; в памяти находятся лишь конкретные его атрибуты.

5. **В новой ячейке введите `dir(SayHello)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Отобразится список атрибутов, которые связаны с функцией `SayHello()`, как показано на рис. 11.6 (и это далеко не полный список). Сейчас вам не нужно знать, что они означают; с некоторыми из них вы еще столкнетесь в книге.

6. **В новой ячейке введите `SayHello("Андрей")` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Функция `SayHello()` выводит ожидаемый текст, как показано на рис. 11.7.

ЗАПОМНИ!

Если атрибут был импортирован с помощью инструкции `from...import`, имя атрибута больше не нужно предварять именем пакета, что упрощает доступ к атрибуту.

ВНИМАНИЕ!

Инструкция `from...import` может вызывать проблемы. Если у двух атрибутов одно и то же имя, допускается импортировать лишь один из них. В то же время инструкция `import` предотвращает конфликт имен, который может иметь место в случае импорта большого количества атрибутов. Поэтому следует соблюдать осторожность при использовании инструкции `from...import`.

7. **В новой ячейке введите `SayGoodbye("Григорий")` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Поскольку была импортирована только функция `SayHello()`, Python ничего не знает о функции `SayGoodbye()`, а потому отображает сообщение

об ошибке. Избирательный характер инструкции `from...import` чреват проблемами, когда вы предполагаете, что атрибут имеется, хотя на самом деле его нет.

```
In [16]: dir(SayHello)

Out[16]: ['__annotations__',
          '__call__',
          '__class__',
          '__closure__',
          '__code__',
          '__defaults__',
          '__delattr__',
          '__dict__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__get__',
          '__getattribute__',
          '__globals__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__kwdefaults__',
          '__le__',
          '__lt__',
          '__module__',
          '__name__',
          '__ne__',
          '__new__',
          '__qualname__',
          '__reduce__',
          '__reduce_ex__',
```

Рис. 11.6. Использование функции `dir()` для получения информации об импортируемых атрибутах

```
In [8]: SayHello("Андрей")

        Привет,  Андрей
```

Рис. 11.7. Для функции `SayHello()` больше не нужно указывать имя пакета

Поиск пакетов на диске

Чтобы можно было использовать код из пакета, Python должен суметь найти пакет и загрузить его в память. Информация о местонахождении пакета хранится в виде путей в Python. Всякий раз, когда вы запрашиваете импорт пакета, Python просматривает все файлы в своем списке путей, чтобы его найти. Информация о пути поступает из следующих трех источников.

- **Переменные среды.** В главе 3 уже рассматривались переменные среды Python, такие как `PYTHONPATH`, которые сообщают Python о местонахождении пакетов на диске.
- **Текущий каталог.** Ранее уже говорилось о том, как изменить текущий каталог Python, чтобы находить любые пакеты, используемые приложением.
- **Каталоги, заданные по умолчанию.** Даже если вы не определяете никаких переменных среды, а в текущем каталоге отсутствуют какие-либо пригодные для использования пакеты, Python все равно может найти свои библиотеки в наборе каталогов по умолчанию, которые подключены как часть его собственной информации о пути.

Знать информацию о текущем пути полезно, потому что отсутствие нужного пути может привести к сбою приложения. Чтобы получить информацию о пути, в новой ячейке введите **`for p in sys.path: print(p)`** и щелкните на кнопке Run Cell (Выполнить ячейку). На экране появится информация о пути (рис. 11.8). Ваш список может отличаться от показанного на рис. 11.8, в зависимости о платформы, версии Python и установленных модулей Python.

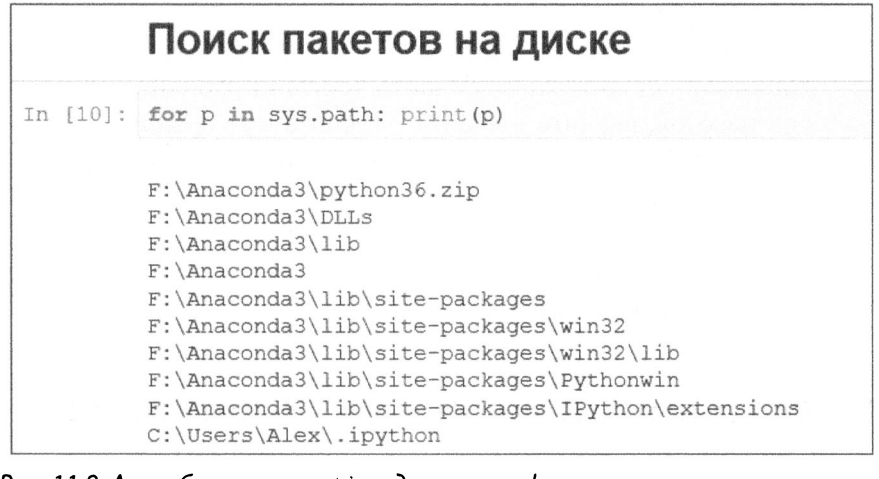


Рис. 11.8. Атрибут `sys.path` содержит информацию о путях в системе

На атрибут `sys.path` вполне можно положиться, но он далеко не всегда содержит все пути, которые может видеть Python. Если вы не находите нужного пути, проверьте другое место, где Python ищет информацию. В следующей пошаговой инструкции показано, как это сделать.

1. **В новой ячейке введите `import os`.**
2. **Введите `os.environ['PYTHONPATH'].split(os.pathsep)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Если переменная среды `PYTHONPATH` определена, то появится список путей, как показано на рис. 11.9. Если же эта переменная не определена, будет выдано сообщение об ошибке.

```
In [9]: import os
        os.environ['PYTHONPATH'].split(os.pathsep)

Out[9]: ['C:\\BP4D']
```

Рис. 11.9. Запрос информации о переменных среды

Атрибут `sys.path` не включает функцию `split()`, поэтому в данном примере используется цикл `for`. А вот атрибут `os.environ['PYTHONPATH']` поддерживает эту функцию, поэтому ее можно использовать для создания списка отдельных путей.

Функции `split()` нужно предоставить значение, которое следует искать при разбиении списка элементов. *Константа* `os.pathsep` (переменная, имеющая одно фиксированное значение) задает разделитель пути для текущей платформы, чтобы можно было использовать один и тот же код на любой платформе, поддерживающей Python.

СОВЕТ

Атрибут `sys.path` поддерживает добавление и удаление элементов. Например, если нужно добавить текущий рабочий каталог в список пакетов, введите в ячейку блокнота **`sys.path.append(os.getcwd())`** и щелкните на кнопке Run Cell (Выполнить ячейку). При повторном выводе содержимого атрибута `sys.path` новая запись добавляется в конец списка. Аналогично, чтобы удалить запись, нужно ввести **`sys.path.remove(os.getcwd())`** в ячейку блокнота и щелкнуть на кнопке Run Cell.

Загрузка пакетов из других источников

Ваша копия Python и связанный с ней компонент Jupyter Notebook поставляются с широким ассортиментом пакетов, которые удовлетворят большинство

потребностей. Вам довольно редко придется выходить за рамки этих пакетов, разве что в целях эксперимента, поскольку в системе уже установлено достаточное количество пакетов. Конечно, при желании всегда можно придумать новый способ решения задачи, который потребует нового кода и пакетов для его хранения. Кроме того, некоторые задачи настолько специфичны, что включение пакетов для их поддержки в стандартную установку приведет к пустой трате ресурсов, поскольку большинство людей никогда не будут их использовать. Следовательно, вам придется время от времени устанавливать пакеты из Интернета или других источников.

Чаще всего для получения новых пакетов применяются утилиты `conda` и `pip`. Но есть и пакеты, при установке которых используются другие методики. Утилиты `conda` и `pip` служат разным целям. Бытует множество неверных представлений относительно этих двух менеджеров пакетов, но на самом деле все сводится к тому, что `conda` обеспечивает универсальное управление пакетами для широкого спектра языков в своей собственной среде, тогда как `pip` предоставляет услуги специально для Python, причем в любой среде. Если вам нужен пакет, специфичный для Python, в первую очередь обратите внимание на утилиту `pip`. Например, она предоставляет доступ к индексу пакетов Python (Python Package Index — PyPI), доступному на сайте `https://pypi.python.org/pypi`. Оба менеджера пакетов подробно рассматриваются в следующих разделах.

Открытие командной строки Anaconda

Прежде чем вплотную заняться вопросами управления пакетами, нужно вызвать утилиту Anaconda Prompt. Она аналогична любой другой командной строке или окну терминала, но предоставляет специальные функции настройки, облегчающие работу с различными утилитами командной строки, входящими в комплект поставки Anaconda. Чтобы открыть командную строку, найдите папку `Anaconda3`. Например, в Windows можно открыть выбрать команду Пуск⇨Все программы⇨Anaconda3⇨Anaconda Prompt.

Работа с пакетами `conda`

С помощью `conda` можно решать самый широкий спектр задач. В следующих разделах описывается, как решать основные задачи с помощью `conda`. Чтобы получить справку по этой утилите, введите **`conda --help`**.

Просмотр пакетов `conda`

Просматривать пакеты `conda` можно двумя способами. Первый способ заключается в создании списка доступных пакетов, второй — в поиске

определенного пакета. Благодаря списку пакетов можно узнать, установлен ли требуемый пакет. Ну а поиск поможет получить сведения об установленном пакете.

Можно выполнить общий поиск, чтобы найти все, что установлено в конкретной системе. В этом случае используются следующие команды.

```
conda list
conda search
```

Вывод этих команд очень длинный и может заполнить весь экранный буфер, из-за чего выполнить прокрутку назад и просмотреть все результаты не получится. Например, на рис. 11.10 показано, что произойдет, если воспользоваться командой `conda list`.

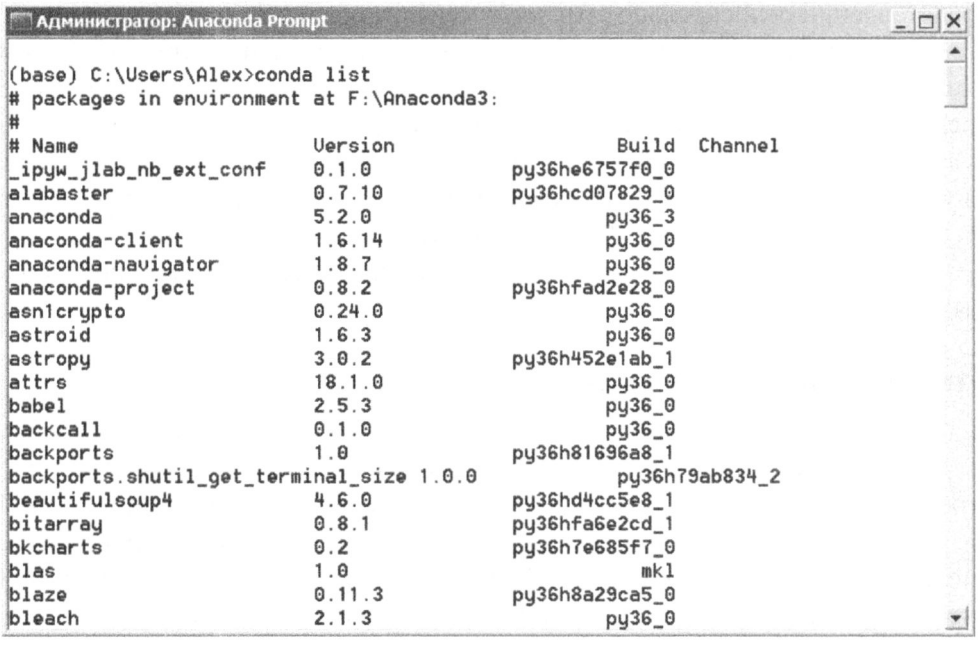


Рис. 11.10. Вывод команды `conda list` довольно длинный и может вызвать переполнение буфера экрана

Обратите внимание на то, что результат выполнения команды включает имя пакета, его версию и соответствующую версию Python. Просмотрите этот список, чтобы узнать, установлен ли нужный пакет в вашей системе. Иногда требуется получить больше, чем выдает обычный поиск. Например, предположим, что вы хотите узнать, какие компоненты были установлены из пакета `scikit-learn` для 64-разрядной платформы Windows. Для этого нужно ввести команду **`conda search --platform win-64 scikit-learn`**. Это приведет к выводу подробных сведений (рис. 11.11).

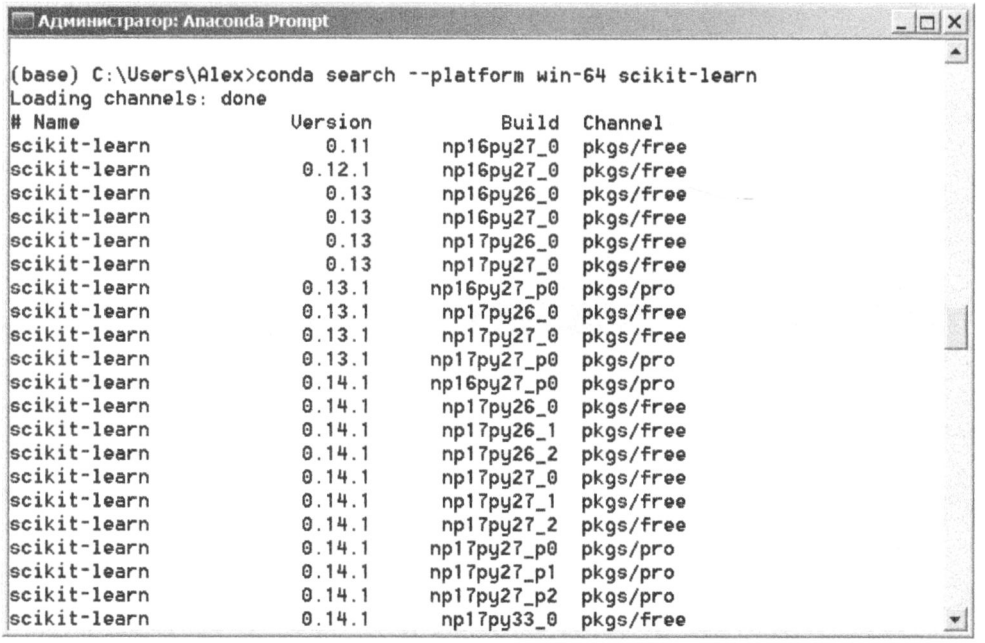


Рис. 11.11. Команда `conda search` позволяет получить дополнительную информацию о пакетах

СОВЕТ

Поддерживаются дополнительные флаги, позволяющие увеличить объем получаемой информации. Например, при использовании флага `--json` вы получаете такие сведения, как полный список зависимостей для пакета, информация о том, полностью ли установлен пакет, и URL-адрес, содержащий расположение пакетов в сети.

Установка пакетов `conda`

Пакеты `conda` доступны на сайте `https://anaconda.org/`. Чтобы определить, доступен ли конкретный пакет, такой как SciPy, введите его имя в поле поиска в верхней части окна. Скорее всего, вы получите целый список кандидатов, как показано на рис. 11.12.

Чтобы разобраться в длинном списке кандидатов, следует щелкать на отдельных ссылках, которые приведут вас к странице с подробностями, подобной показанной на рис. 11.13. Учтите, что вы получите ссылки на все виды информации о конкретной копии выбранного пакета. Но наиболее важным для нас является раздел Installers (Установщики). Можно загрузить установщик или использовать для этого утилиту `conda`. Соответствующая команда может выглядеть так:

```
conda install -c anaconda scipy
```

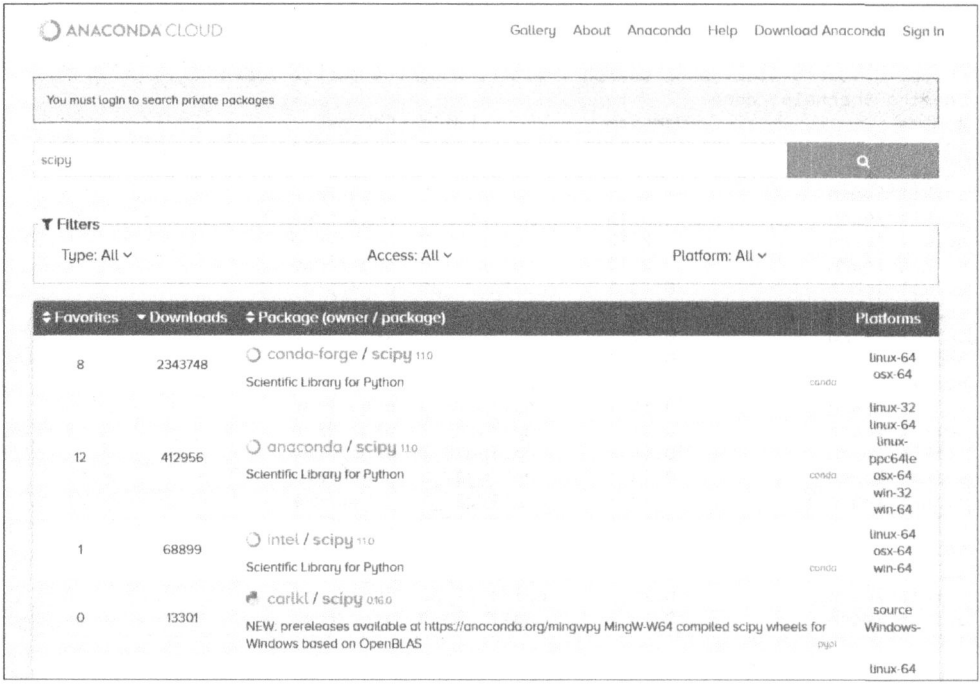


Рис. 11.12. Выберите версию используемого пакета

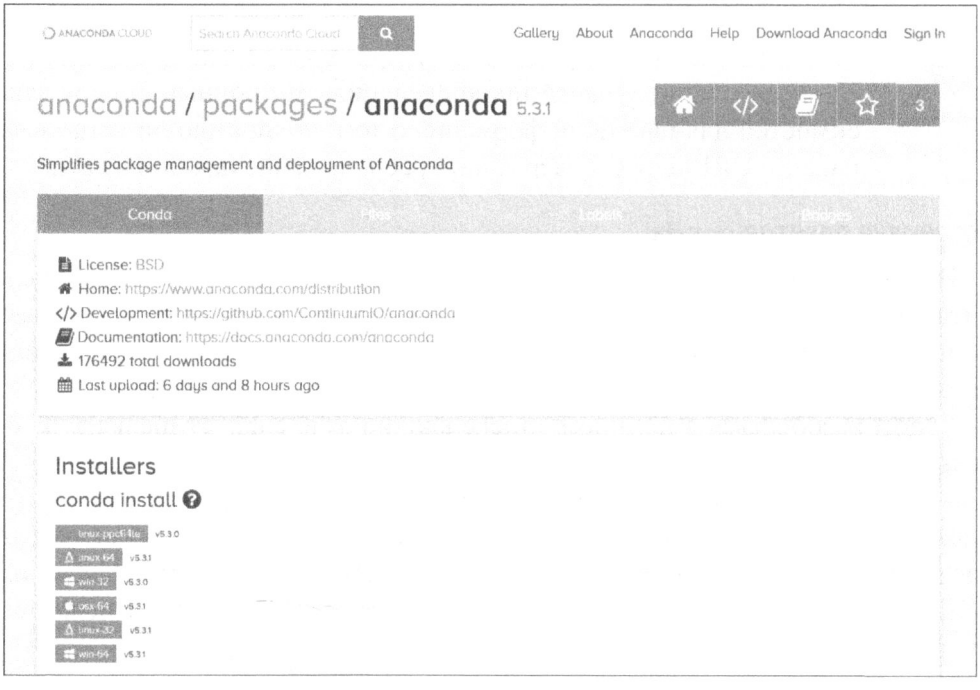


Рис. 11.13. Выбор устанавливаемого дистрибутива

ИСПОЛЬЗОВАНИЕ КОМАНДЫ CONDA INFO

Несмотря на то что команда `conda info` обычно используется для получения информации об окружении, ее также можно применять для работы с пакетами. Чтобы узнать особенности конкретного пакета, нужно просто добавить имя пакета, например `conda info numpy`. К сожалению, эта команда часто выдает огромный список сведений, который желательно немного сократить. Один из способов сделать это — добавить номер версии после имени пакета, отделив его знаком равенства (=), например `conda info numpy = 1.13.1` для версии 1.13.1 пакета NumPy.

В большинстве случаев при использовании ключа `--verbose` применительно к пакетам вы не получите какой-либо дополнительной информации. А вот ключ `--json` позволяет узнать немного больше, причем информация подается так, что ею можно легко манипулировать из программы, например из сценария. В общем, с помощью команды `conda info` можно получить весьма содержательную информацию о пакетах.

Обновление пакетов conda

Пакеты, применяемые для разработки приложений, могут со временем устаревать. Разработчики, которые их поддерживают, периодически добавляют новые функции или исправляют ошибки. Проблема с обновлениями заключается в том, что они могут привести к некорректной работе приложения, а иногда и вовсе вывести его из строя. Тем не менее рекомендуется обновлять пакеты, даже если обновление предназначено лишь для исправления ошибок, связанных с безопасностью. Конечно, для начала нужно как-то узнать, что пакет требует обновления. Чтобы найти устаревшие пакеты, воспользуйтесь командой `conda search --outdated`, указав имя проверяемого пакета.

СОВЕТ

Если нужно проверить все пакеты, то при выполнении поиска просто не указывайте имя пакета. К сожалению, на этом этапе вывод становится настолько длинным, что действительно сложно что-либо увидеть. В таком случае поможет команда `conda search --outdated --namesonly`, которая сообщает только имена пакетов, требующих обновления.

Когда становится ясно, что именно требует обновления, можете воспользоваться командой `conda update`. Например, можно обновить пакет NumPy, выполнив команду `conda update numpy`. Мало какие пакеты являются автономными, поэтому утилита `conda` сообщит список элементов, которые необходимо обновить вместе с пакетом NumPy. Введите **y** и нажмите клавишу <Enter> для

продолжения. На рис. 11.14 показана типичная последовательность событий в процессе обновления.

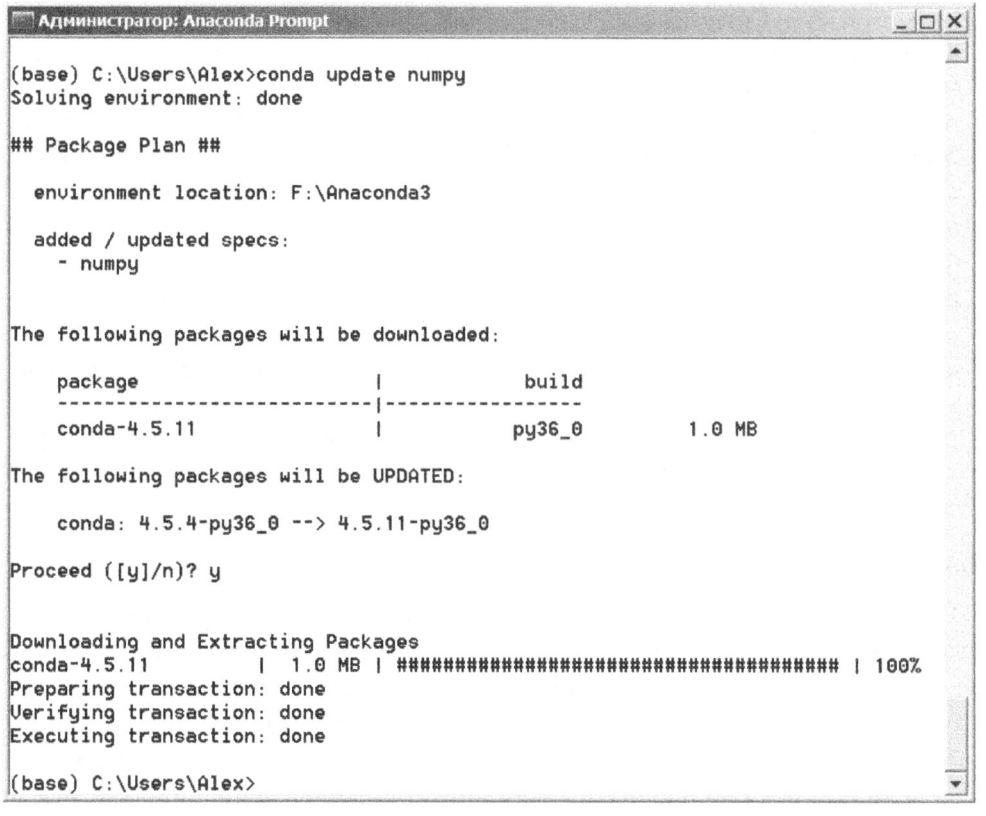


Рис. 11.14. В процессе обновления отображается много информации

Можно обновить все пакеты одновременно, воспользовавшись командой `conda update --all`. Но в этом случае взаимодействие между пакетами сделает обновление менее успешным, чем если бы пакеты обновлялись по отдельности. Кроме того, обновление может занять много времени, поэтому придется запастись терпением.

Удаление пакетов `conda`

В какой-то момент вы приходите к выводу о том, что вам больше не нужен пакет `conda`. Единственная проблема заключается в том, что вы не знаете, зависят ли от него другие пакеты. Поскольку зависимости пакетов могут быть довольно сложными и вы хотите быть уверены, что ваши приложения продолжат работать, нужно проверить, какие пакеты зависят от этого конкретного

пакета. К сожалению, команда `conda info` сообщает вам только о требованиях к пакету, т.е. от чего они зависят. Поэтому рекомендуется не трогать пакеты после их установки.

ЗАПОМНИ!

Если же вы все-таки решили удалить пакет, то воспользуйтесь командой `conda remove`. Она удаляет указанный вами пакет вместе со всеми пакетами, которые от него зависят. В таком случае рекомендуется сначала использовать параметр командной строки `--dry-run`, чтобы понять, действительно ли вы хотите удалить пакет. Предположим, вы решили удалить пакет NumPy. Тогда введите команду **`conda remove --dry-run numpy`**. Сам пакет не будет удален; утилита `conda` просто показывает, что произойдет, если вы запустите процесс удаления (рис. 11.15).

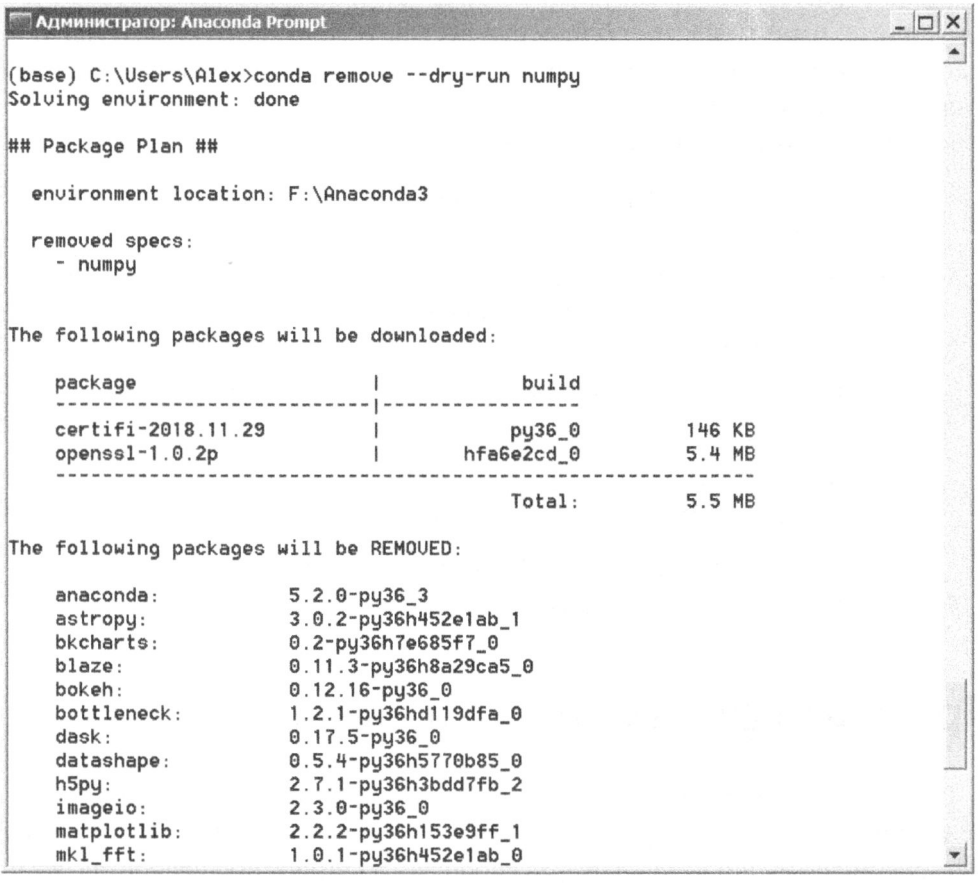


Рис. 11.15. Удаление единственного пакета может вызвать цепную реакцию

Как видите, один пакет может поддерживать множество других пакетов, и некоторые из них вам могут понадобиться. Если же вы действительно хотите удалить пакет, введите ту же команду, что и раньше, только без ключа `--dry-run`.

Никогда не используйте ключ `--force`. В этом случае команда удаляет пакет, не трогая зависимые пакеты, что в итоге приводит к повреждению установки Python. Если нужно удалить пакет, удаляйте и все зависимые пакеты, чтобы сохранить свою установку в целости и сохранности.

Установка пакетов с помощью утилиты `pip`

Как ни странно, использование утилиты `pip` очень напоминает работу с утилитой `conda`. Обе они, по сути, решают одни и те же задачи, поэтому, если вы знаете, как использовать одну утилиту, то будете знать, как использовать другую. Обратившись на сайт `https://pip.pypa.io/en/stable/reference/`, вы увидите, что утилита `pip` поддерживает практически те же команды, что и `conda` (с незначительными отличиями). Например, если вы хотите найти список устаревших пакетов, то введите команду **`pip list --outdated`**. Вот список общих команд, поддерживаемых утилитой `pip`.

- `check`. Проверка совместимости зависимостей для установленных пакетов.
- `download`. Загрузка указанных пакетов для последующей установки.
- `freeze`. Вывод информации об установленных пакетах в формате требований.
- `help`. Отображение экрана справки со списком команд.
- `install`. Установка указанных пакетов.
- `list`. Вывод списка установленных пакетов.
- `search`. Поиск пакетов на сайте PyPI.
- `show`. Вывод информации об установленных пакетах.
- `uninstall`. Удаление указанных пакетов.

Просмотр содержимого пакета

Python поддерживает несколько способов просмотра содержимого пакета. Основной способ, используемый большинством разработчиков, заключается в применении функции `dir()`, которая сообщает об атрибутах, предоставляемых пакетом.

Обратите внимание на рис. 11.3. Помимо рассмотренных ранее функций `SayGoodbye()` и `SayHello()` в списке содержатся и другие записи. Эти атрибуты автоматически генерируются в Python. Рассмотрим их назначение.

- `__builtins__`. Содержит список всех встроенных атрибутов, которые доступны в пакете. Python добавляет эти атрибуты автоматически.
- `__cached__`. Сообщает имя и местоположение кешированного файла, который связан с пакетом. Информация о местоположении является относительной и зависит от текущего каталога Python.
- `__doc__`. Выводит справочную информацию о пакете, если таковая существует. Например, если ввести **`os.__doc__`**, Python выдаст справку, связанную с библиотекой `os`.
- `__file__`. Сообщает имя и местонахождение пакета. Информация о местоположении является относительной и зависит от текущего каталога Python.
- `__initializing__`. Определяет, находится ли пакет в процессе инициализации. Обычно этот атрибут возвращает значение `False`. Данный атрибут полезен, когда нужно дождаться завершения загрузки одного пакета, прежде чем импортировать другой пакет, который зависит от предыдущего пакета.
- `__loader__`. Выводит информацию о загрузчике для данного пакета. Загрузчик — это утилита, которая получает пакет и помещает его в память, чтобы Python мог его использовать. На практике данный атрибут используется довольно редко.
- `__name__`. Сообщает название пакета.
- `__package__`. Этот внутренний атрибут используется системой импорта для упрощения загрузки и управления пакетами.

Можно получить еще более подробную информацию об атрибутах. Например, введите **`dir(BPPD_11_Packages.SayHello)`**. Вы увидите список атрибутов, показанный на рис. 11.16.

Одни из записей, такие как `__name__`, понятны без дополнительных объяснений, другие требуют комментариев. Например, предположим, что нужно получить дополнительную информацию о записи `__sizeof__`. Один из способов сделать это — ввести **`help("__sizeof__")`**. На экране появится довольно скудная (но полезная) справочная информация (рис. 11.17).

С Python не случится ничего плохого, если попробовать воспользоваться каким-либо атрибутом. Даже если у программы Notebook возникнут проблемы, всегда можно перезапустить ядро (или среду в целом). Например, если ввести

BPPD_11_Packages.SayHello.__sizeof__(), отобразится размер функции `SayHello()` в байтах, как показано на рис. 11.18.

Просмотр содержимого пакета

```
In [1]: import BPPD_11_Packages
        dir(BPPD_11_Packages.SayHello)

Out[1]: ['__annotations__',
         '__call__',
         '__class__',
         '__closure__',
         '__code__',
         '__defaults__',
         '__delattr__',
         '__dict__',
         '__dir__',
         '__doc__',
         '__eq__',
         '__format__',
         '__ge__',
         '__get__',
         '__getattribute__',
         '__globals__',
         '__gt__',
         '__hash__',
         '__init__',
         '__init_subclass__',
         '__kwdefaults__',
         '__le__',
         '__lt__',
         '__module__',
         '__name__',
         '__ne__',
         '__new__',
         '__qualname__',
         '__reduce__',
```

Рис. 11.16. Просмотр списка атрибутов, используемых в Python

```
In [2]: help("__sizeof__")

        Help on built-in function __sizeof__:

        __sizeof__(...)
            __sizeof__() -> int
            size of object in memory, in bytes
```

Рис. 11.17. Получение полезной информации о нужном атрибуте

```
In [3]: BPPD_11_Packages.SayHello.__sizeof__()

Out[3]: 112
```

Рис. 11.18. Пример использования атрибута

В отличие от многих других языков программирования, Python делает доступным исходный код своих внутренних библиотек. Например, открыв каталог `\Python37\Lib`, вы увидите список файлов `.py`, которые можно без проблем открыть в Notebook. Попробуйте загрузить библиотеку `os.py`, которую вы используете в примерах этой главы, с помощью кнопки Upload (Загрузить) панели инструментов Notebook. Щелкните на кнопке Upload рядом с именем файла после его открытия, после чего щелкните на полученной ссылке, и вы увидите результат, показанный на рис. 11.19. Заметьте, что файлы `.py` открыва-

jupyter os.py✔ несколько секунд назад

File Edit View Language

```
r"""OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix or nt, e.g. unlink, stat, etc.
  - os.path is either posixpath or ntpath
  - os.name is either 'posix' or 'nt'
  - os.curdir is a string representing the current directory (always '.')
  - os.pardir is a string representing the parent directory (always '..')
  - os.sep is the (or a most common) pathname separator ('/' or '\\')
  - os.extsep is the extension separator (always '.')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
"""

#'
import abc
import sys
import stat as st

_names = sys.builtin_module_names

# Note:  more names are added to __all__ later.
__all__ = ["altsep", "curdir", "pardir", "sep", "pathsep", "linesep",
           "defpath", "name", "path", "devnull", "SEEK_SET", "SEEK_CUR",
           "SEEK_END", "fsencode", "fsdecode", "get_exec_path", "fdopen",
           "popen", "extsep"]
```

Рис. 11.19. Просмотрите код пакета, чтобы лучше понять, как он работает

ются в более простом редакторе и не отображают ячейки, в отличие от файлов блокнота, которые вы использовали на протяжении всей книги.

Непосредственный просмотр содержимого пакета поможет вам открыть для себя новые приемы программирования и лучше понять, как работает библиотека. Чем больше времени вы потратите на изучение Python, тем лучше будете использовать его возможности для создания интересных приложений.

ВНИМАНИЕ!

Убедитесь в том, что вы просто просматриваете код библиотеки, не изменяя его. Если вы случайно измените код, приложения могут перестать работать. Хуже того, вы внесете незаметные ошибки, которые будут проявляться только в вашей системе и больше нигде. Всегда проявляйте осторожность, работая с библиотечным кодом.

Просмотр документации по пакетам

Используйте функцию `doc()` всякий раз, когда нужна быстрая помощь. Но для получения сведений по пакетам и библиотекам Python лучше воспользоваться компонентом Python Package Documentation (документация по пакетам Python). Он часто представляется как Package Docs в папке Python и также называется Pydoc. Независимо от названия этот компонент значительно облегчает жизнь разработчикам. В следующих разделах описано, как с ним работать.

Открытие приложения Pydoc

Pydoc — это еще одно полезное приложение Python. Оно находится в каталоге `\Python37\Lib` вашей системы в виде файла `pydoc.py`. Как и в случае с другими файлами `.py`, можно открыть этот файл с помощью Notebook и посмотреть, как он работает. Pydoc также можно запустить с помощью ярлыка Python 3.7 Module Docs, который находится в системной папке `Python 3.7`, либо команды Anaconda Prompt (см. раздел "Открытие командной строки Anaconda").

СОВЕТ

Pydoc можно использовать как в графическом, так и в текстовом режиме. При открытии Anaconda Prompt можно указать ключевое слово, например `JSON`, после чего Pydoc отобразит текстовую справку. Используя опцию командной строки `-k`, за которой следует ключевое слово, например `if`, можно отобразить список мест, где встречается это ключевое слова. Чтобы запустить сам сервер, нужно ввести **`Pydoc -b`**. Если нужно использовать определенный порт для браузера, добавьте ключ командной строки `-p`, указав номер порта.

В графическом режиме создается локализованный сервер Pydoc, который взаимодействует с браузером для отображения информации о пакетах и библиотеках Python. Поэтому, запустив данное приложение, вы увидите окно командной строки (терминала).

ЗАПОМНИ!

Как и в случае с любым сервером, система может запросить у вас разрешение. Например, может появиться предупреждение брандмауэра о том, что Pydoc пытается получить доступ к локальной системе. Следует дать Pydoc разрешение на работу с системой, чтобы можно было видеть предоставляемую информацию. В случае срабатывания антивирусной программы понадобится разрешить продолжение работы Pydoc. Некоторые системы, как, например, Windows, могут потребовать дополнительных полномочий для запуска Pydoc.

Обычно сервер автоматически открывает новое окно браузера, как показано на рис. 11.20. В нем содержатся ссылки на различные пакеты, которые есть в вашей системе, включая любые пользовательские пакеты, которые создаются и включаются в путь поиска Python. Чтобы просмотреть информацию о любом пакете, просто щелкните на соответствующей ссылке.

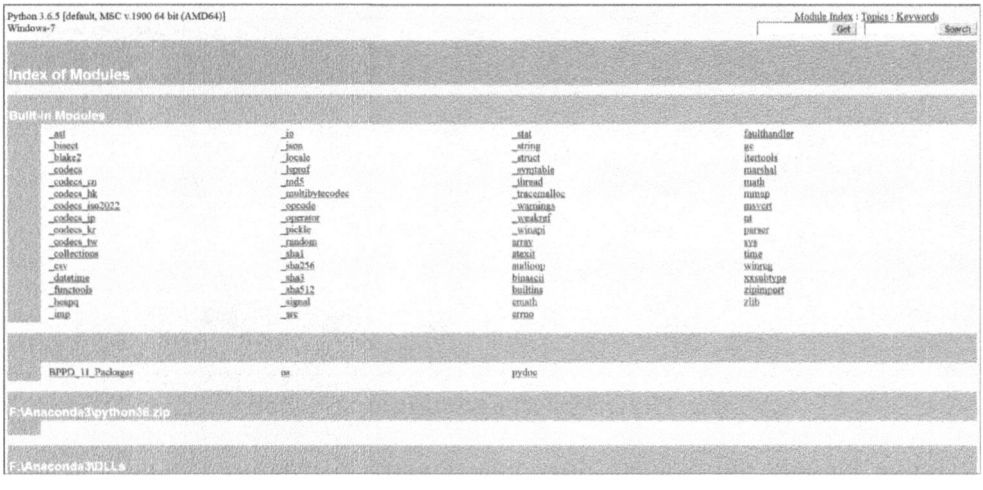


Рис. 11.20. Индексная страница модулей в окне браузера

Утилита Anaconda Prompt предлагает две команды, обеспечивающие управление сервером. Достаточно ввести букву, связанную с командой, и нажать клавишу <Enter> для ее активизации. Вот эти две команды:

- `b` — запуск новой копии браузера, заданного по умолчанию, с загруженной индексной страницей;
- `q` — останов сервера.

ЗАПОМНИ!

Завершив просмотр справочной информации, остановите выполнение сервера, введя `q` в командной строке и нажав клавишу <Enter>. После остановки сервера освобождаются все используемые ресурсы и закрываются все порты брандмауэра, открытые при установке Pydoc.

Использование ссылок быстрого доступа

Снова вернемся к рис. 11.20. В верхней части веб-страницы находятся три ссылки. Благодаря им обеспечивается быстрый доступ к средствам сайта. Браузер сначала открывает страницу `Module Index`. Если в какой-то момент потребуется вернуться к этой странице, щелкните на ссылке Module Index (Индекс модулей).

После щелчка на ссылке Topics (Темы) появится страница, показанная на рис. 11.21. Здесь находятся ссылки на важные темы Python. Например, чтобы получить сведения о булевых значениях, щелкните на ссылке BOOLEAN. Будет открыта страница с описанием того, как работают булевы значения в Python. В нижней части страницы находятся ссылки на страницы, содержащие дополнительную полезную информацию.

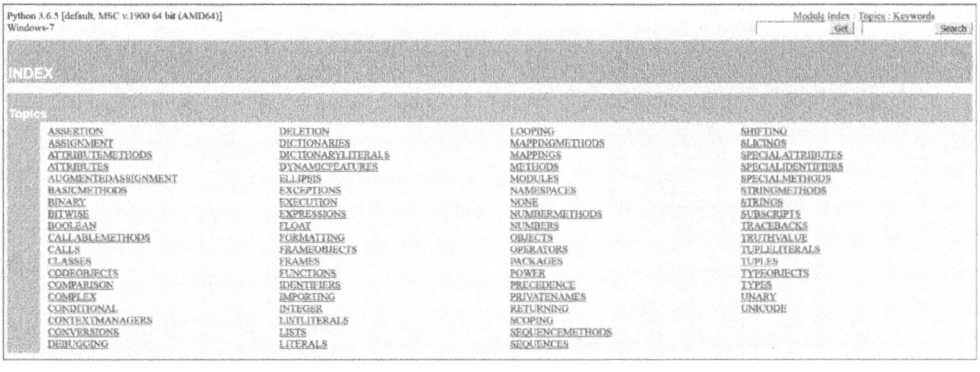


Рис. 11.21. Перечень важных тем Python

После щелчка на ссылке Keywords появится страница, показанная на рис. 11.22. Здесь находится список ключевых слов, поддерживаемых в Python. Например, чтобы получить дополнительные сведения о создании циклов, щелкните на ссылке for.

Ввод поискового термина

В верхней части страницы также находятся два текстовых поля. Рядом с левым полем расположена кнопка Get (Получить), а рядом с правым — кнопка Search (Поиск). После ввода поискового термина в первое текстовое поле

щелкните на кнопке Get, и вы получите доступ к документации по соответствующему пакету или атрибуту. На рис. 11.23 показан результат, получаемый после ввода слова `print`.

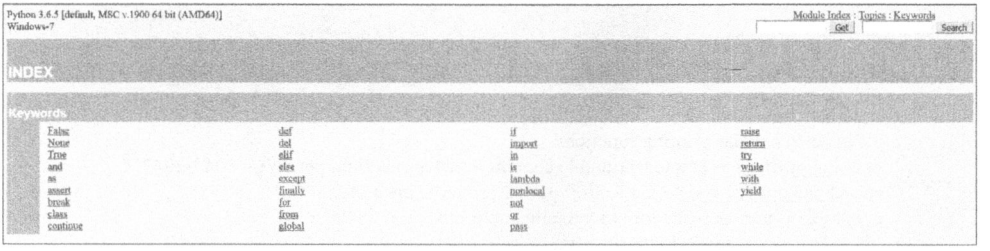


Рис. 11.22. Перечень ключевых слов, поддерживаемых в Python

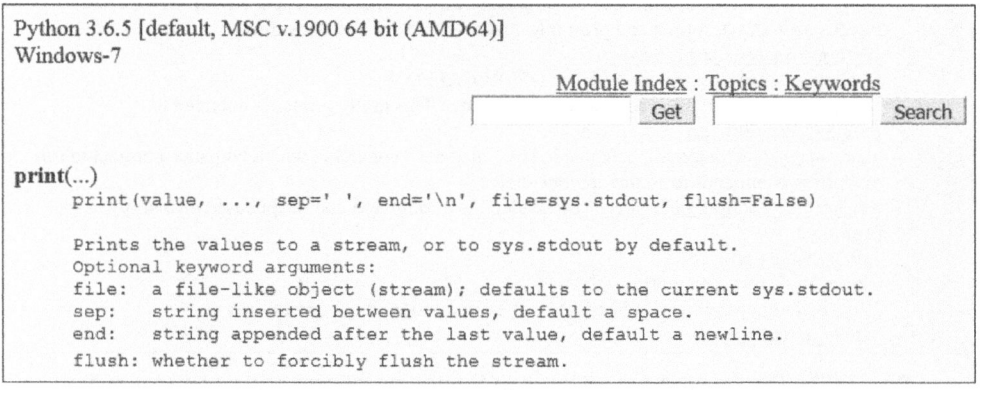


Рис. 11.23. Информация о поисковом термине, полученная с помощью поля Get

Если ввести поисковый термин во второе текстовое поле и щелкнуть на кнопке Search (Поиск), отобразится список тем, связанных с данным поисковым термином. На рис. 11.24 показаны типичные результаты ввода ключевого слова `print`. В данном случае нужно щелкнуть на ссылке, чтобы получить дополнительные сведения.

Просмотр результатов

Результаты, полученные при просмотре страницы, зависят от темы. Одни темы краткие, как, например, тема для функции `print()` на рис. 11.23. Другие же темы могут быть довольно обширными. Например, после щелчка на ссылке Calendar, показанной на рис. 11.24, выводится большой массив данных (рис. 11.25).

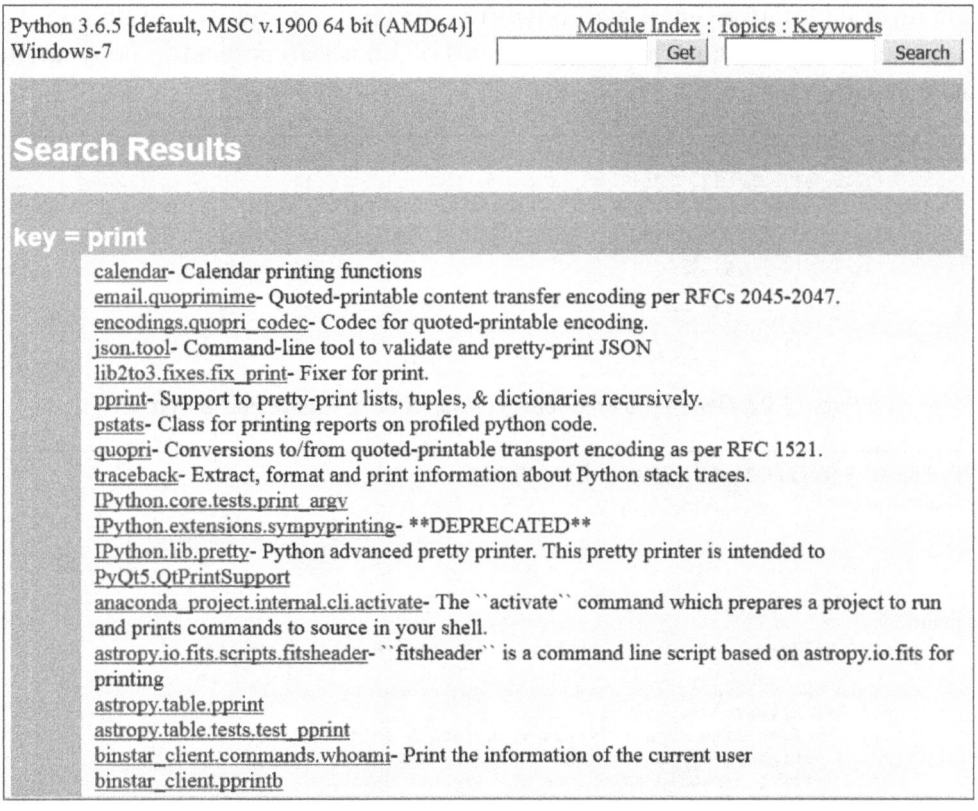


Рис. 11.24. После щелчка на кнопке Search отображается список тем, относящихся к поисковому термину

В этом конкретном случае вы видите связанную информацию о пакете, сведения об ошибках, функциях, параметрах и всем остальном, что касается функций вывода календаря. Количество информации, которую вы получаете, частично зависит от сложности темы и частично — от объема информации, предоставленной разработчиком вместе с пакетом. Например, выбрав `BPPD_11_Packages` на странице Module Index, вы получите только список функций (и никакой документации).

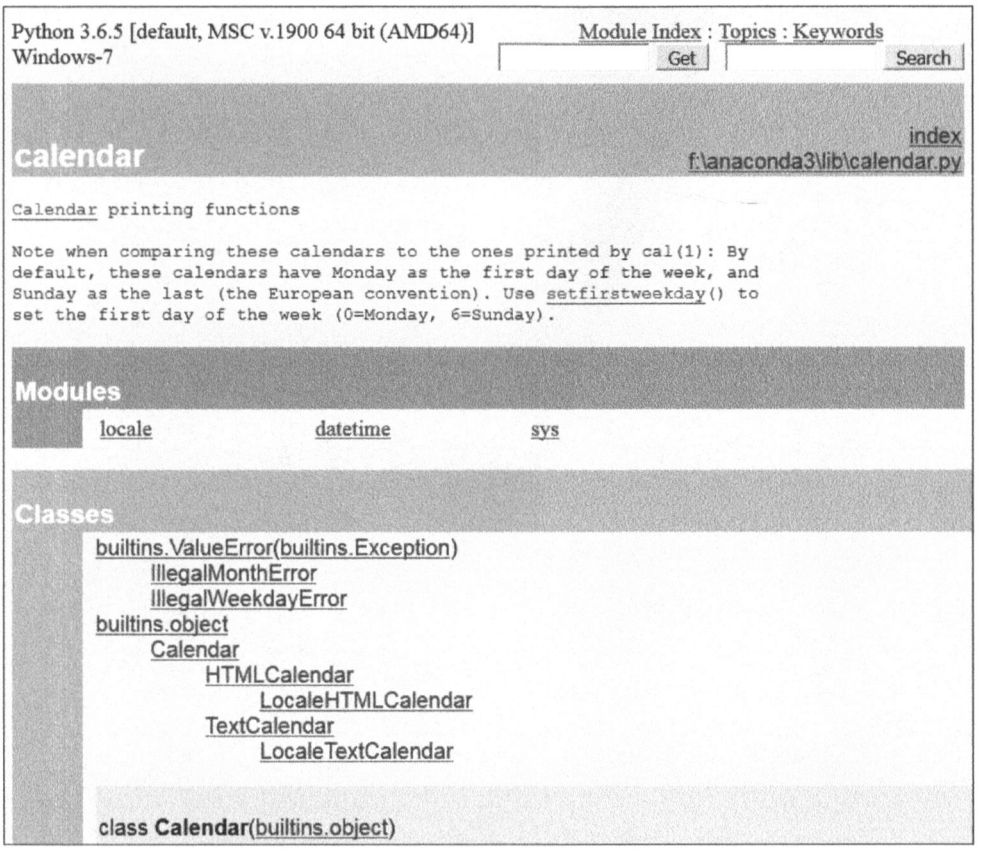


Рис. 11.25. Некоторые страницы содержат большой объем данных

Глава 12

Работа со строками

В ЭТОЙ ГЛАВЕ...

- Знакомство со строками
- Работа со специальными и одиночными символами
- Обработка строк и поиск в строках
- Форматирование строк

Компьютеры не понимают строковых данных. Они говорят на языке цифр, а не букв. Строка, отображаемая на экране, воспринимается компьютером как последовательность чисел. Но поскольку люди понимают строки, приложения должны иметь возможность работать с ними. К счастью, в Python работать со строками относительно не сложно. Понятные пользователям строки переводятся в числа, воспринимаемые компьютером, и наоборот.

Чтобы от строк был какой-либо толк, нужно уметь их обрабатывать. На практике это означает, что нужно уметь анализировать строки, извлекать из них фрагменты и искать в строке требуемую информацию. В этой главе рассматривается, как создавать строки в Python, разбивать их на части и использовать только те части, которые нужны.

Строки, создаваемые в программах, нужно представлять пользователям в удобном для чтения виде. Компьютер, в общем-то, не заботится о том, как выглядит строка на экране, поэтому отображаемый текст порой довольно трудно читать. Важно понимать принципы форматирования строк, чтобы пользователи могли получать информацию в наглядном виде. Прочитав главу, вы будете знать, как создавать и форматировать строки должным образом. Исходный код примеров данной главы находится в файле `BPPD_12_Working_with_Strings.ipynb` (см. введение).

Различия между строками

Многие начинающие разработчики (и даже те, кто поопытнее) не понимают, что на самом деле компьютеры работают лишь с нулями и единицами. Даже большие числа фактически состоят из 0 и 1. При сравнении учитываются лишь нули и единицы. Данные перемещаются с использованием нулей и единиц. Такие объекты, как строки, для компьютера выглядят как последовательности, состоящие из 0 и 1. И хотя сгруппировать нули и единицы для создания чисел относительно просто, со строками все намного сложнее, поскольку в этом случае речь идет об информации, которой компьютер должен манипулировать как числами, а представлять — как символы.

Для компьютера понятия "строка" не существует. Строки состоят из символов, а отдельные символы в действительности являются числовыми значениями. Работая со строками, Python создает последовательность символов, которую компьютер воспринимает как числовой набор.

Определение символа с помощью чисел

Чтобы создать символ, нужно сначала определить связь между ним и соответствующим ему числом. Важно понимать, что приложение, работающее со строками, выполняет преобразование числовыми кодами и символами, выводимыми на экран. Для подобного преобразования обычно применяется стандарт ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией). Например, код ASCII используется в Python для преобразования числа `65` в букву `A`. На сайте `www.asciitable.com` можно просмотреть всю таблицу ASCII, включающую различные числовые значения и соответствующие им символы.

Каждому символу должно соответствовать собственное числовое значение. Заглавной букве `A` соответствует значение `65`. Чтобы создать строчную букву `a`, следует воспользоваться другим числовым значением, в данном случае `97`. Компьютер рассматривает `A` и `a` как два совершенно разных символа, а вот для пользователя это две версии одного и того же символа в нижнем и верхнем регистрах.

В этой главе используются десятичные числа, но компьютер рассматривает их как совокупности нулей и единиц. Например, буква `A` в действительности представляется как `01000001`, а буква `a` — как `01100001`. Когда пользователь видит на экране букву `A`, компьютер видит соответствующее двоичное значение.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Было бы удобно иметь один набор символов на все случаи жизни. Но на практике довольно сложно подобрать единый набор числовых значений, который годился бы для представления всех символов. Частично суть проблемы заключается в том, что стандарт ASCII не поддерживает языки с нелатинской письменностью; кроме того, в нем нет средств преобразования специальных символов в экранное представление. На самом деле помимо ASCII существует много других наборов символов. Некоторые из них доступны на сайте `http://www.i18nguy.com/unicode/codepages.html`. Щелкните на названии интересующего вас символьного набора, чтобы увидеть, как в нем присваиваются конкретные числовые значения каждому символу. Большинство наборов символов в качестве отправной точки использует таблицу ASCII.

Создание строк на основе символов

Для создания строк в Python от вас не потребуется ничего экстраординарного. Для начала разберемся с термином *строка*. Представьте, что это бусы. При создании ожерелья на нитку поочередно нанизываются отдельные бусинки. После завершения этой работы вы получите готовое украшение, состоящее из отдельных бусин.

Аналогичная концепция применяется для описания компьютерных строк. Видя строку, вы понимаете, что она состоит из отдельных символов, объединяемых с помощью языка программирования. Средства языка создают структуру, которая включает всю цепочку символов. Поэтому именно программа, а не компьютер, знает о том, что весь этот числовой ряд (в котором каждое число представляет символ) определяет строку.

ЗАПОМНИ!

Почему так важно знать, каким образом Python работает с символами? Дело в том, что многие функции и специальные средства Python работают на уровне отдельных символов. Даже когда вы видите предложение, Python видит цепочку символов.

В отличие от большинства языков программирования, строки в Python могут заключаться как в одинарные, так и в двойные кавычки. Например, `"Привет всем"` и `'Привет всем'` — это одна и та же строка. В Python также поддерживаются тройные двойные и одинарные кавычки, которые позволяют создавать строки, разделенные на несколько строк. В следующей пошаговой инструкции демонстрируется работа со строками в Python.

1. **Откройте новый блокнот.**

   Можете воспользоваться файлом исходного кода `BPPD_12_Working_with_Strings.ipynb` (см. введение).

2. **Введите следующий код в блокноте.**

```
print('Привет всем (одинарные кавычки)!')
print("Привет всем (двойные кавычки)!")
print("""Многострочная текстовая строка с
тремя двойными кавычками.
Можно также использовать три одинарные кавычки.""")
```

   Каждый из трех вызовов функции `print()` демонстрирует различные методики работы со строками. Строки можно заключать как в одинарные, так и в двойные кавычки. При использовании трех кавычек (одинарных или двойных) текст можно разбивать на несколько строк.

3. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа выведет ожидаемый текст. Обратите внимание на то, что многострочный текст отображается в трех строках (рис. 12.1), как и в файле исходного кода, так что это своего рода форматирование. Многострочное форматирование применяется для разбиения отображаемого текста в тех местах, где это нужно.

Создание строк на основе символов

```
In [1]: print('Привет всем (одинарные кавычки)!')
        print("Привет всем (двойные кавычки)!")
        print("""Многострочная текстовая строка с
        тремя двойными кавычками.
        Можно также использовать три одинарные кавычки.""")

        Привет всем (одинарные кавычки)!
        Привет всем (двойные кавычки)!
        Многострочная текстовая строка с
        тремя двойными кавычками.
        Можно также использовать три одинарные кавычки.
```

Рис. 12.1. Строки состоят из цепочек символов

Создание строк, включающих специальные символы

В некоторых строках встречаются специальные символы. Они отличаются от привычных нам алфавитно-цифровых символов и знаков пунктуации и делятся на следующие категории.

- **Управляющие символы.** Это символы, которые предназначены не для отображения, а для управления выводом. Многие из них влияют на положение *указателя вставки* — маленькой линии, которую вы видите, когда вводите текст. Типичный пример — символ табуляции. Он создает пространство между двумя элементами, а величина этого пространства определяется позицией табуляции. Если вы хотите перейти на следующую строку, вы вставляете символ возврата каретки (возвращает указатель вставки в начало строки) и символ перевода строки (перемещает указатель вставки на следующую строку).
- **Акцентированные символы**. Это символы с различными диакритическими знаками, такими как акут ('), гравис (`), циркумфлекс (^), диерезис (¨), тильда (~) или кольцо (°). Подобные символы имеются во многих алфавитах.
- **Символы рисования.** С помощью некоторых символов можно создавать элементарную графику. Примеры можно увидеть на сайте `http://jrgraphix.net/r/Unicode/2500-257F`.
- **Типографские символы.** Некоторые символы, например символ абзаца (¶), применяются при отображении на экране определенных видов текста, особенно когда приложение используется в качестве редактора.
- **Другие специальные символы.** В зависимости от используемого набора символов вам могут встретиться и другие специальные символы, число которых практически бесконечно.

Чаще всего при работе со строками, даже в простых консольных приложениях, используются управляющие символы. С учетом этого в Python применяются управляющие последовательности, с помощью которых можно непосредственно вводить такие символы.

ЗАПОМНИ!

Управляющая последовательность изменяет привычное назначение символа, придавая ему новый смысл. Сочетание обратной косой черты (\) и буквы (например, `a`) обычно рассматривается разработчиками как один элемент — *управляющий символ* или *управляющий код*. В табл. 12.1 представлены управляющие последовательности, применяемые в Python.

Наилучший способ понять, как работают управляющие последовательности, — испытать их в действии. Для этого выполните следующую пошаговую инструкцию.

Таблица 12.1. Управляющие последовательности в Python

Управляющая последовательность	Значение
\newline	Игнорируется
\\	Символ обратной косой черты (\)
\'	Одинарная кавычка (')
\"	Двойная кавычка (")
\a	ASCII-символ звонка (BEL)
\b	ASCII-символ возврата на одну позицию (BS)
\f	ASCII-символ прокрутки страницы (FF)
\n	ASCII-символ перевода строки (LF)
\r	ASCII-символ возврата каретки (CR)
\t	ASCII-символ горизонтальной табуляции (TAB)
\u*hhhh*	Символ Unicode (набор символов, широко применяемый во всем мире), шестнадцатеричное значение которого задается символами *hhhh*
\v	ASCII-символ вертикальной табуляции (VT)
ooo	ASCII-символ, восьмеричное значение которого задается символами *ooo*
\x*hh*	ASCII-символ, шестнадцатеричное значение которого задается символами *hh*

1. **Введите следующий код в блокноте.**

```
print("Часть этого текста находится на\r\nследующей строке.")
print("Буква А с грависом: \xC0.")
print("Символ рисования: \u2562.")
print("Знак абзаца: \266.")
print("Знак деления: \xF7.")
```

В этом примере используются различные способы достижения одной и той же цели — создание специального символа. В первой строке продемонстрировано непосредственное использование управляющих символов. В то же время многие специальные символы формируются с помощью шестнадцатеричного числа, которое состоит из двух цифр, как показано во второй и пятой строках.

Иногда приходится использовать символы Unicode, состоящие из четырех цифр, как показано в третьей строке. Восьмеричные значения состоят их трех цифр и записываются без буквенного префикса, как показано в четвертой строке.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа отобразит ожидаемый текст и специальные символы (рис. 12.2).

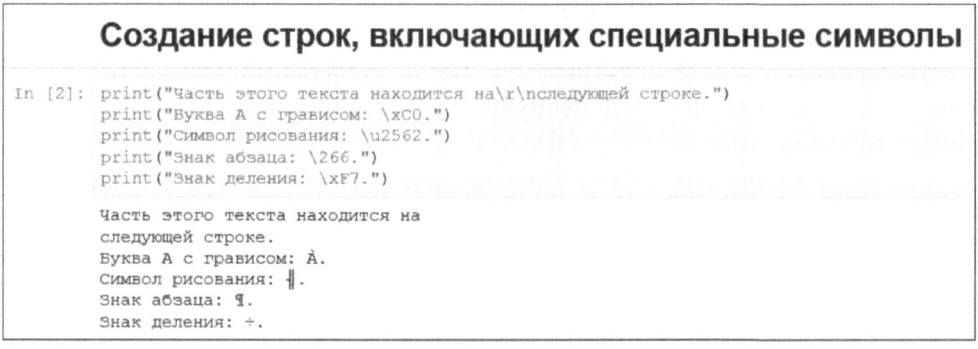


Рис. 12.2. Специальные символы используются для форматирования вывода

ВНИМАНИЕ!

В Notebook используется стандартный набор символов для разных платформ, поэтому вы будете видеть одни и те же специальные символы независимо от того, на какой платформе работаете. Однако, создавая приложение, обязательно протестируйте его на разных платформах, чтобы увидеть, не изменится ли его поведение. Набор символов для каждой конкретной платформы может использовать другие числа для представления специальных символов. Кроме того, выбор пользователем наборов символов может повлиять на отображение специальных символов вашим приложением. Всегда тщательно тестируйте используемые специальные символы.

Выбор отдельных символов

Выше уже отмечалось, что строки состоят из отдельных символов. Они действительно похожи на ожерелье — каждая бусина является отдельным элементом всей цепочки.

В Python можно получить доступ к отдельным символам строки. Это важная особенность, позволяющая создавать новые строки, которые содержат только часть исходной строки. Можно также комбинировать строки для создания новых строк. В таких операциях применяются квадратные скобки: после

имени переменной ставятся квадратные скобки, в которых указывается число, как показано ниже.

```
MyString = "Здравствуй, мир"
print(MyString[0])
```

ЗАПОМНИ!

В результате отобразится буква З. Строки в Python индексируются, начиная с позиции 0. Например, если использовать вызов `print(MyString[1])`, то будет выведена буква д.

Можно также извлечь подстроку из строки. Для этого нужно в квадратных скобках указать начальную и конечную позиции, разделенные двоеточием. Например, результатом выполнения функции `print(MyString[12:14])` будет слово мир. Вывод начинается с буквы, находящейся на 13-й позиции, и завершается буквой, находящейся на 15-й позиции (не забывайте о том, что индексация ведется с нуля). В следующей пошаговой инструкции демонстрируется, как работать с подстроками в Python.

1. **Введите следующий код в блокноте.**

```
String1 = "Здравствуй, мир"
String2 = "Python – это супер!"
print(String1[0])
print(String1[0:10])
print(String1[:10])
print(String1[10:])
String3 = String1[:11] + String2[:6]
print(String3)
print(String2[:7]*9)
```

Пример начинается с создания двух строк. Затем демонстрируются различные способы индексации первой строки. Заметьте, что можно не указывать начальную или конечную позицию в диапазоне, если нужно работать с остальной частью строки.

Далее мы комбинируем две подстроки, соединяя начало подстроки `String1` с началом подстроки `String2`, чтобы создать строку `String3`.

ЗАПОМНИ!

Операция, связанная с использованием символа + для объединения двух строк, называется *конкатенацией*. Это один из самых удобных операторов, о котором нужно помнить, работая со строками в приложении.

Последний шаг заключается в использовании функциональной возможности Python, которая называется *повторением*. С ее помощью создается несколько копий строки или подстроки.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа отобразит различные варианты подстрок и комбинаций строк (рис. 12.3).

```
In [5]: String1 = "Здравствуй, мир"
        String2 = "Python - это супер!"
        print(String1[0])
        print(String1[0:10])
        print(String1[:10])
        print(String1[11:])
        String3 = String1[:11] + String2[:6]
        print(String3)
        print(String2[:7]*9)

        З
        Здравствуй
        Здравствуй
         мир
        Здравствуй,Python
        Python Python Python Python Python Python Python Python Python
```

Рис. 12.3. Можно выбирать отдельные фрагменты строк

Обработка строк

Работа с диапазонами символов обеспечивает определенную степень гибкости, но не дает возможности манипулировать строковым содержимым или анализировать его. Например, может потребоваться перевести символы в верхний регистр или определить, содержит ли строка одни лишь буквы. К счастью, в Python есть функции, которые помогают решать задачи такого рода. Вот наиболее часто используемые функции.

- `capitalize()`. Перевод первого символа строки в верхний регистр.
- `center(`*`ширина`*`, `*`заполнитель`*`=" ")`. Центрирует строку таким образом, чтобы ее ширина стала равна значению *`ширина`*. Если вы предоставляете аргумент *`заполнитель`*, функция использует этот символ. В противном случае используются пробелы для выравнивания строки до нужной ширины.
- `expandtabs(`*`размер_табуляции`*`=8)`. Раскрывает символы табуляции в строке, заменяя их количеством пробелов, заданным аргументом *`размер_табуляции`*. Если размер табуляции не задан, то по умолчанию он равен 8 пробелам.
- `isalnum()`. Возвращает `True`, если строка включает хотя бы один символ и все символы являются алфавитно-цифровыми.
- `isalpha()`. Возвращает `True`, если строка включает хотя бы один символ и все символы являются буквами.
- `isdecimal()`. Возвращает `True`, если строка Unicode содержит только десятичные символы.

- `isdigit()`. Возвращает `True`, если строка содержит только цифры.
- `islower()`. Возвращает `True`, если строка содержит хотя бы один буквенный символ и все буквенные символы находятся в нижнем регистре.
- `isnumeric()`. Возвращает `True`, если строка Unicode содержит только числовые символы.
- `isspace()`. Возвращает `True`, если строка содержит только символы пропуска (пробелы, табуляция, возврат каретки, перевод строки, прокрутка страницы, вертикальная табуляция, но не символ возврата на один символ).
- `istitle()`. Возвращает `True`, если все слова в строке начинаются с прописной буквы, например `Здравствуй, Мир`. Это касается даже однобуквенных слов. Например, для строки `Ты и я` возвращается значение `False`, а для строки `Ты И Я` — значение `True`.
- `isupper()`. Возвращает `True`, если строка содержит хотя бы один буквенный символ и все буквенные символы находятся в верхнем регистре.
- `join(`*последовательность*`)`. Создает строку, в которой базовая строка по очереди разделяется каждым символом *последовательности*, образуя повторяющиеся фрагменты. Например, если начать с команды `MyString = "Привет"` и ввести `print(MyString.join("!*!"))`, то результат будет `!Привет*Привет!`.
- `len(`*строка*`)`. Возвращает длину строки.
- `ljust(`*ширина*`, `*заполнитель*`=" ")`. Выравнивает строку влево таким образом, чтобы ее ширина стала равна значению *ширина*. Если вы предоставляете аргумент *заполнитель*, функция использует этот символ. В противном случае используются пробелы для выравнивания строки до нужной ширины.
- `lower()`. Преобразует все символы верхнего регистра в строке в символы нижнего регистра.
- `lstrip()`. Удаляет все ведущие пробелы в строке.
- `max(`*строка*`)`. Возвращает символ, который имеет максимальное числовое значение в *строке*. Например, числовое значение символа `a` больше числового значения символа `A`.
- `min(`*строка*`)`. Возвращает символ, который имеет минимальное числовое значение в *строке*. Например, числовое значение символа `A` меньше числового значения символа `a`.
- `rjust(`*ширина*`, `*заполнитель*`=" ")`. Выравнивает строку вправо таким образом, чтобы ее ширина стала равна значению *ширина*. Если вы предоставляете аргумент *заполнитель*, функция

использует этот символ. В противном случае используются пробелы для выравнивания строки до нужной ширины.

- `rstrip()`. Удаляет все хвостовые пробелы в строке.
- `split(`*`строка`*`=" ", `*`число`*`=string.count(`*`строка`*`))`. Разбивает строку на подстроки с помощью разделителя, указанного в первом аргументе. По умолчанию в качестве разделителя используется пробел. Следовательно, если строка содержит фразу `A Fine Day`, будут получены три подстроки: `A`, `Fine` и `Day`. Параметр *`число`* используется для определения количества возвращаемых подстрок. По умолчанию возвращается каждая подстрока, создаваемая функцией.
- `splitlines(`*`число`*`=string.count('\n'))`. Разбивает строку, содержащую символы перевода строки (`\n`), на отдельные строки. Каждое разбиение имеет место там, где стоит символ новой строки. В результирующих строках символы новой строки отсутствуют. Количество возвращаемых строк указывается с помощью параметра *`число`*.
- `strip()`. Удаляет все ведущие и хвостовые символы в строке.
- `swapcase()`. Инвертирует регистр каждого буквенного символа в строке.
- `title()`. Возвращает строку, в которой начальная буква каждого слова переведена в верхний регистр, а все остальные символы слова — в нижний.
- `upper()`. Преобразует все символы нижнего регистра в символы верхнего регистра.
- `zfill(`*`ширина`*`)`. Заполняет строку нулями слева, чтобы ее ширина стала равна значению *`ширина`*. Эта функция предназначена для использования со строками, содержащими числовые значения. Исходный знак числа сохраняется.

Вы сможете лучше понять эти функции, если поэкспериментируете с ними. Ниже демонстрируется использование некоторых из функций.

1. **Введите следующий код в блокноте.**

```
MyString = "  Здравствуй, мир  "
print(MyString.upper())
print(MyString.strip())
print(MyString.center(21, "*"))
print(MyString.strip().center(21, "*"))
print(MyString.isdigit())
print(MyString.istitle())
print(max(MyString))
```

```
print(MyString.split())
print(MyString.split()[0])
```

Программа начинается с создания строки `MyString`, которая включает пробелы до и после текста, чтобы вы могли видеть, как работают связанные с пробелами функции. Первая из вызываемых функций, `upper()`, преобразует все символы строки в верхний регистр.

Удаление лишних пробелов является обычной задачей при разработке приложений. И с ней хорошо справляется функция `strip()`. Функция `center()` позволяет добавлять заполнители как в левой, так и в правой части строки, чтобы строка занимала требуемое пространство. Результат, полученный в результате комбинирования функций `strip()` и `center()`, будет отличаться от результата, полученного при использовании одной функции `center()`.

ЗАПОМНИ!

Функции разрешается комбинировать. Python по очереди вызывает каждую функцию в направлении слева направо. Порядок вызова функций будет влиять на результат, и разработчики часто допускают ошибки, указывая функции в неправильном порядке. Если результаты отличаются от ожидаемых, попробуйте изменить порядок расположения функций.

Некоторые функции работают со строками как с аргументами, в частности, функция `max()`. Если ввести `MyString.max()`, Python выдаст ошибку.

Если функция возвращает список строк, то можно получить доступ к отдельному элементу списка, указав его индекс. В примере показано, как использовать функцию `split()` для разбиения строки на подстроки. В конце демонстрируется порядок получения доступа только к первой подстроке в списке. Подробнее работа со списками будет рассмотрена в главе 13.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа выведет модифицированные строки, как показано на рис. 12.4.

Поиск значения в строке

Иногда требуется найти конкретную информацию в строке, например узнать, содержит ли строка слово `Привет`. В Python имеется целый ряд функций для анализа строк. Вот наиболее часто используемые функции.

- `count(`*`стр`*`, `*`начало`*`=0, `*`конец`*`=len(`*`строка`*`))`. Подсчитывает, сколько раз подстрока *`стр`* встречается в строке. Можно ограничить поиск путем указания начального индекса с помощью параметра *`начало`* и конечного индекса с помощью параметра *`конец`*.

```
In [6]: MyString = "  Здравствуй, мир  "
        print(MyString.upper())
        print(MyString.strip())
        print(MyString.center(21, "*"))
        print(MyString.strip().center(21, "*"))
        print(MyString.isdigit())
        print(MyString.istitle())
        print(max(MyString))
        print(MyString.split())
        print(MyString.split()[0])

  ЗДРАВСТВУЙ, МИР
Здравствуй, мир
*  Здравствуй, мир  *
***Здравствуй, мир***
False
False
у
['Здравствуй,', 'мир']
Здравствуй,
```

Рис. 12.4. Благодаря функциям обработка строк становится более гибкой

- `endswith(`*суффикс*`, `*начало*`=0, `*конец*`=len(`*строка*`))`. Возвращает `True`, если строка завершается символами, заданными параметром *суффикс*. Можно ограничить поиск путем указания начального индекса с помощью параметра *начало* и конечного индекса с помощью параметра *конец*.
- `find(`*стр*`, `*начало*`=0, `*конец*`=len(`*строка*`))`. Определяет, встречается ли подстрока *стр* в строке, и выводит индекс местоположения. Можно ограничить поиск путем указания начального индекса с помощью параметра *начало* и конечного индекса с помощью параметра *конец*.
- `index(`*стр*`, `*начало*`=0, `*конец*`=len(`*строка*`))`. Реализует ту же функциональность, что и функция `find()`, но генерирует исключение, если подстрока *стр* не найдена.
- `replace(`*старый*`, `*новый*` [, `*макс*`])`. Заменяет все вхождения подстроки *старый* подстрокой *новый*. С помощью параметра *макс* можно ограничить количество замен.
- `rfind(`*стр*`, `*начало*`=0, `*конец*`=len(`*строка*`))`. Реализует ту же функциональность, что и функция `find()`, но поиск выполняется с конца строки, а не с начала.
- `rindex(`*стр*`, `*начало*`=0, `*конец*`=len(`*строка*`))`. Реализует ту же функциональность, что и функция `index()`, но поиск выполняется с конца строки, а не с начала.

- `startswith(`*`префикс`*`, `*`начало`*`=0, `*`конец`*`=len(`*`начало`*`))`. Возвращает `True`, если строка начинается символами, заданными параметром *`префикс`*. Можно ограничить поиск путем указания начального индекса с помощью параметра *`начало`* и конечного индекса с помощью параметра *`конец`*.

Поиск требуемых данных — важная задача программирования, с которой вы будете сталкиваться независимо от того, какое приложение создаете. Ниже показано, как выполнять поиск в строках.

1. **Введите следующий код в блокноте.**

```
SearchMe = "Малина окрашена в красный цвет, а голубика окрашена
           в синий цвет!"
print(SearchMe.find("в"))
print(SearchMe.rfind("в"))
print(SearchMe.count("а"))
print(SearchMe.startswith("красный"))
print(SearchMe.endswith("синий"))
print(SearchMe.replace("Малина", "Машина")
      .replace("голубика", "ракета"))
```

Пример начинается с создания строки `SearchMe`, содержащей два экземпляра слова `в`. Это позволит продемонстрировать, как различаются результаты поиска в зависимости от того, с какой позиции вы начинаете. При использовании функции `find()` поиск выполняется с начала строки, а при использовании функции `rfind()` — с конца строки.

Иногда нужно узнать, сколько раз в строке встречается определенный набор символов. В таком случае на помощь придет функция `count()`.

Далее с помощью функции `startswith()` мы проверяем, начинается ли строка с определенного шаблона, и точно также с помощью функции `endswith()` проверяется, заканчивается ли строка указанным шаблоном.

Последняя инструкция в примере заменяет слово `"Малина"` словом `"Машина"`, а слово `"голубика"` — словом `"ракета"`. Обратите внимание на то, что инструкция занимает две строки, а функция `replace()` вызывается дважды. Подобный способ записи позволяет улучшить наглядность кода.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Результаты работы программы показаны на рис. 12.5. Принципиальный момент — поисковые функции возвращают разные индексы в зависимости от того, с какого места строки был начат поиск. Важно выбирать подходящую функцию для получения правильных результатов.

```
In [9]: SearchMe = "Малина окрашена в красный цвет, а голубика окрашена в синий цвет!"
        print(SearchMe.find("в"))
        print(SearchMe.rfind("в"))
        print(SearchMe.count("а"))
        print(SearchMe.startswith("красный"))
        print(SearchMe.endswith("синий"))
        print(SearchMe.replace("Малина", "Машина")
              .replace("голубика", "ракета"))

        16
        61
        9
        False
        False
        Машина окрашена в красный цвет, а ракета окрашена в синий цвет!
```

Рис. 12.5. Применение функций поиска и замены

Форматирование строк

Задача форматирования — представить строки в виде, удобном для пользователя и простом для восприятия. Форматирование не означает добавления специальных шрифтов или эффектов, а связано лишь с представлением данных. Например, пользователю может понадобиться вывести число с фиксированной запятой, а не десятичное число.

Существует несколько способов форматирования строк, которые будут продемонстрированы в книге. Большинство из них основано на использовании функции `format()`. Сначала вы создаете строку со спецификацией формата, а затем вызываете функцию `format()`, добавляя данные в эту строку. Спецификация формата может быть очень простой и содержать, например, две фигурные скобки `{}`, которые задают заполнитель для данных. Заполнители можно нумеровать, создавая более сложные правила форматирования. Например, выражение `{0}` ссылается на первый элемент данных в строке. Если элементы данных пронумерованы, можно повторять их, чтобы одни и те же данные появлялись в строке более одного раза.

Спецификация формата стоит после двоеточия. Если нужно создать только спецификацию, то в фигурных скобках указывается лишь двоеточие и любое форматирование, которое нужно использовать. Например, спецификация `{: f}` задает вывод числа с фиксированной точкой. Если вы нумеруете элементы данных, то число, которое предшествует двоеточию, скажем, `{0: f}`, задает вывод числа с фиксированной запятой только для этого элемента. Синтаксис спецификации формата таков:

`[[`*`заполнитель`*`]`*`выравнивание`*`][`*`знак`*`][#][0][`*`ширина`*`][,][.`*`точность`*`][`*`тип`*`]`

Этот синтаксис подробно описан по адресу `https://docs.python.org/3/library/string.html`.

Рассмотрим назначение основных элементов спецификации.

- *заполнитель*. Задает символ, используемый для выравнивания данных, которые слишком малы, чтобы поместиться в отведенном для них пространстве.
- *выравнивание*. Задает способ выравнивания данных в области отображения. Можно использовать следующие типы выравнивания:
  - **<** — выравнивание влево;
  - **>** — выравнивание вправо;
  - **^** — выравнивание по центру;
  - **=** — выравнивание по ширине.
- *знак*. Определяет использование знаков в выводимых результатах:
  - **+** — положительные числа имеют знак "плюс", а отрицательные — знак "минус";
  - **-** — отрицательные числа снабжаются знаком "минус";
  - **<пробел>** — положительным числам предшествует пробел, а отрицательным числам — знак "минус".
- **#**. Указывает на то, что следует использовать альтернативный формат отображения чисел. Например, к шестнадцатеричным числам будет добавлен префикс `0x`.
- **0**. Указывает на то, что выводимые числа должны иметь знаки и дополняться нулями по мере необходимости для обеспечения согласованного вывода.
- *ширина*. Определяет общую ширину поля данных (даже если данные не помещаются в отведенное пространство).
- **,**. Указывает на то, что числовые данные должны содержать запятые в качестве разделителя разрядов.
- **.** *точность*. Определяет количество символов после десятичной точки.
- *тип*. Определяет тип вывода, даже если тип входных данных не совпадает. Типы вывода делятся на следующие три группы.
  - *Строка*. Можно использовать тип `s` или просто ничего не указывать.
  - *Целое число*. Вот целочисленные типы: `b` (двоичный), `c` (символьный), `d` (десятичный), `o` (восьмеричный), `x` (шестнадцатеричный с буквами в нижнем регистре), `X` (шестнадцатеричный с буквами в верхнем регистре) и `n` (зависящее от локальных настроек

десятичное число, использующее соответствующие символы в качестве разделителя разрядов).

- *Число с плавающей точкой*. Вот типы с плавающей точкой: `e` (экспоненциальная запись, использующая символ нижнего регистра `e` в качестве разделителя), `E` (экспоненциальная запись, использующая символ верхнего регистра `E` в качестве разделителя), `f` (число с фиксированной точкой с символами нижнего регистра), `F` (число с фиксированной точкой с символами верхнего регистра), `g` (общий формат в нижнем регистре), `G` (общий формат в верхнем регистре), `n` (зависящий от локальных настроек общий формат, в котором используются соответствующие символы для десятичного разделителя и разделителя разрядов) и `%` (процентное значение).

Элементы спецификации формата должны стоять в корректном порядке, иначе Python не будет знать, что с ними делать. Если вы укажете выравнивание перед заполнителем, Python выведет сообщение об ошибке и не станет применять форматирование. Следующая пошаговая инструкция поможет вам увидеть, как применяются спецификации форматирования и в каком порядке задаются элементы спецификации.

1. **Введите следующий код в блокноте.**

```
Formatted = "{:d}"
print(Formatted.format(7000))
Formatted = "{:,d}"
print(Formatted.format(7000))
Formatted = "{:^15,d}"
print(Formatted.format(7000))
Formatted = "{:*^15,d}"
print(Formatted.format(7000))
Formatted = "{:*^15.2f}"
print(Formatted.format(7000))
Formatted = "{:*>15X}"
print(Formatted.format(7000))
Formatted = "{:*<#15x}"
print(Formatted.format(7000))
Formatted = "{0} {1} и {0} {2}."
print(Formatted.format("синяя", "машина", "ракета"))
```

Пример начинается с форматирования поля в виде десятичного значения. Затем к выводу добавляется разделитель разрядов. Следующий шаг — сделать поле шире, чем необходимо для хранения данных, и центрировать данные в поле. Наконец, в поле добавляется заполнитель-звездочка.

После этого мы переходим к другим типам данных. Сначала мы выводим те же данные в формате с фиксированной запятой. В примере также показан вывод

данных в шестнадцатеричном формате в верхнем и нижнем регистре. Число с символами верхнего регистра выравнивается по правому краю, а с символами нижнего регистра — по левому краю.

И наконец, в примере демонстрируется, как использовать пронумерованные поля. В данном случае создается строка вывода, в которой повторяется одно из входных значений.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа выведет данные в разных форматах, как показано на рис. 12.6.

```
In [19]: Formatted = "{:d}"
         print(Formatted.format(7000))
         Formatted = "{:,d}"
         print(Formatted.format(7000))
         Formatted = "{:^15,d}"
         print(Formatted.format(7000))
         Formatted = "{:*^15,d}"
         print(Formatted.format(7000))
         Formatted = "{:*^15.2f}"
         print(Formatted.format(7000))
         Formatted = "{:*>15X}"
         print(Formatted.format(7000))
         Formatted = "{:*<#15x}"
         print(Formatted.format(7000))
         Formatted = "{0} {1} и {0} {2}."
         print(Formatted.format("синяя", "машина", "ракета"))

         7000
         7,000
              7,000
         *****7,000*****
         ****7000.00****
         ***********1B58
         0x1b58*********
         синяя машина и синяя ракета.
```

Рис. 12.6. Использование форматирования для представления данных в нужном виде

Глава 13

Управление списками

В ЭТОЙ ГЛАВЕ...

- Знакомство со списками
- Создание списков
- Управление списками
- Использование объекта `Counter`

Многие упускают из виду тот факт, что большинство приемов программирования взято из реальной жизни. Отчасти это связано с употреблением незнакомой терминологии. Например, большинство людей называют место для хранения чего-либо коробкой или ящиком, тогда как для программистов привычнее термин *переменная*. Но со списками иная ситуация: они знакомы всем. Каждый человек составляет списки и использует их в повседневной жизни. Так что многое в этой главе должно быть вам понятно. Единственное — нужно научиться думать о списках так, как это принято в Python.

Многие считают, что со списками трудно работать, хотя это не так. Создавая список, мы просто записываем элементы в том порядке, который считаем правильным. Иногда мы переупорядочиваем список, переставляя его элементы. Порой мы используем палец для поиска нужной записи в списке. Все то, что мы привыкли делать со списками, выполнимо и в Python. Разница лишь в том, что теперь нужно продумывать каждый шаг, чтобы Python понимал наши намерения.

Эта глава посвящена рассмотрению способов создания, управления, поиска и вывода списков. Изучив главу, вы сможете применять списки, чтобы

делать свои приложения более надежными, быстрыми и гибкими. Исходный код примеров этой главы находится в файле `BPPD_13_Managing_Lists.ipynb` (см. введение).

Упорядочение информации в приложении

Люди создают списки, чтобы упорядочить информацию и упростить доступ к ней. В Python списки служат тем же целям, и часто без них не обойтись. Например, может понадобиться создать объект, предназначенный для хранения названий дней недели или месяцев в году. Эти названия сформируют список, как если бы вы записали их на бумаге. В следующих разделах мы поговорим о принципах работы со списками.

Знакомство со списками

В Python *список* определяется как разновидность последовательности. *Последовательность* — это способ хранения данных в одном месте, но в виде отдельных объектов. Представьте себе почтовые ящики, которые устанавливаются в жилых домах. Один почтовый шкаф содержит несколько небольших ящиков, в каждый из которых кладется корреспонденция для конкретной квартиры. В Python поддерживаются и другие виды последовательностей, которые будут рассмотрены в главе 14:

- кортежи;
- словари;
- стеки;
- очереди;
- двухсторонние очереди.

ЗАПОМНИ!

Из всех последовательностей списки наиболее просты для понимания. Освоив списки, вы научитесь лучше работать с другими видами последовательностей, которые обеспечивают бо́льшую функциональность и гибкость. Суть в том, что данные хранятся в списке почти так же, как если бы вы записывали их на листе бумаги: один элемент следует за другим, как показано на рис. 13.1. Любой список имеет начало, середину и конец. Как показано на рисунке, элементы списка пронумерованы. (Даже если вы обычно не нумеруете их в реальности, Python всегда нумерует элементы за вас.)

Рис. 13.1. Список является последовательностью элементов, подобно обычному списку, записанному в блокноте

Как компьютер просматривает списки

Компьютер не просматривает списки так, как мы. У него нет блокнота и ручки. Зато у компьютера есть память, поэтому он сохраняет каждый элемент списка в отдельной ячейке памяти, как показано на рис. 13.2. Память имеет непрерывный характер, и каждый новый элемент добавляется в следующую ячейку памяти.

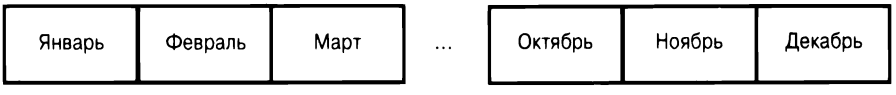


Рис. 13.2. Каждый элемент, добавляемый в список, занимает отдельную ячейку памяти

Компьютеры используют нечто наподобие почтовых ящиков для хранения списков. Список в целом представляет собой своего рода почтовый шкаф. Когда поступает новый элемент данных, компьютер помещает его в следующий почтовый ящик, доступный в шкафу.

ЗАПОМНИ!

Ячейки памяти, в которых хранятся элементы списка, нумеруются подобно почтовым ящикам, на которых указаны номера квартир. Только в данном случае нумерация начинается с 0, а не с 1. Каждый очередной ящик получает следующий номер по порядку. Список с названиями месяцев года будет содержать 12 элементов, которые будут пронумерованы от 0 до 11 (а не от 1 до 12, как можно было

предположить). Важно как можно быстрее привыкнуть к компьютерной системе нумерации, потому что даже опытные программисты допускают ошибки, используя 1, а не 0 в качестве отправной точки для нумерации.

В зависимости от того, какая информация хранится в каждом почтовом ящике, они не обязательно должны быть одинакового размера. Python позволяет хранить строку в одном элементе, целое число — в другом и число с плавающей точкой — в третьем. Компьютер не знает, какая информация хранится в каждом почтовом ящике, и ему это не нужно. Все, что он видит, — это один длинный список чисел, которые могут обозначать все что угодно. Python берет на себя всю заботу по обработке элементов списка в соответствии с корректным типом данных. Он гарантирует, что если вы запрашиваете пятый элемент списка, то получите именно его.

СОВЕТ

Как правило, рекомендуется создавать списки однотипных элементов, чтобы упростить управление данными. Например, при создании списка, состоящего только из целых чисел, вам не придется тратить время на постоянные проверки типа. Но бывают ситуации, когда возникает потребность в использовании смешанных данных. Многие языки программирования требуют, чтобы списки имели только один тип данных, но Python в этом плане более гибкий и допускает наличие смешанных данных. Просто помните о том, что использование смешанных данных в списке означает необходимость постоянно проверять их тип при извлечении информации. Если вы попытаетесь обработать строку как целое число, приложение может аварийно завершиться.

Создание списков

Как уже упоминалось, списки в Python могут быть смешанного типа. Однако ограничение списка одним типом данных всегда является наилучшей практикой. В следующей пошаговой инструкции демонстрируется, как создавать списки в Python.

1. **Откройте новый блокнот.**

   Можете также воспользоваться файлом исходного кода `BPPD_13_Managing_Lists.ipynb` (см. введение).

2. **Введите** `List1 = ["Один", 1, "Два", True]`.

Python создаст список `List1`. Список содержит два строковых значения (`"Один"` и `"Два"`), целое число (`1`) и булево значение (`True`). Естественно, вы не увидите результат, поскольку Python создает список в памяти.

ЗАПОМНИ!

Вы заметите, что каждый тип данных выделяется своим цветом. Если используется цветовая схема, заданная по умолчанию, Python отображает строки оранжевым цветом, цифры — черным, булевы значения — зеленым. Цвет записи — это сигнал, позволяющий понять, правильно ли вы ввели данные. Благодаря этому уменьшается количество ошибок при создании списков.

3. **Введите `print(List1)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране отобразится содержимое всего списка, как показано на рис. 13.3. Обратите внимание: строковые значения отображаются в одинарных кавычках, даже если они изначально были взяты в двойные кавычки. Оба варианта одинаково допустимы в Python.

Создание списков

```
In [1]: List1 = ["Один", 1, "Два", True]
        print(List1)

        ['Один', 1, 'Два', True]
```

Рис. 13.3. Python отображает содержимое списка `List1`

4. **Введите `dir(List1)` и щелкните на кнопке Run Cell.**

   Python отобразит список действий, которые можно выполнять с помощью списков, как показано (частично) на рис. 13.4. Выводимый результат сам по себе является списком. Таким образом, вы используете один список, чтобы определить, какие действия можно выполнить над другим списком.

ЗАПОМНИ!

Когда вы начнете работать со сложными объектами, вы увидите, что функция `dir()` сообщает, какие действия можно выполнить, используя этот объект. Действия, отображаемые без знаков подчеркивания, являются основными действиями, которые можно выполнять над списком. Вот они:

- `append`;
- `clear`;
- `copy`;
- `count`;

```
In [2]: dir(List1)

Out[2]: ['__add__',
         '__class__',
         '__contains__',
         '__delattr__',
         '__delitem__',
         '__dir__',
         '__doc__',
         '__eq__',
         '__format__',
         '__ge__',
         '__getattribute__',
         '__getitem__',
         '__gt__',
         '__hash__',
         '__iadd__',
         '__imul__',
         '__init__',
         '__init_subclass__',
         '__iter__',
```

Рис. 13.4. Python предоставляет список действий, которые можно выполнять с помощью списка

- extend;
- index;
- insert;
- pop;
- remove;
- reverse;
- sort.

Доступ к спискам

После создания списка понадобится получать доступ к информации, которую он содержит. В предыдущем разделе было показано, как использовать функции `print()` и `dir()` для взаимодействия со списком, но существуют и другие способы обращения к спискам, как описано в следующей пошаговой инструкции.

1. **Введите `List1 = ["Один", 1, "Два", True]` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Python создаст список `List1`.

2. **Введите `List1[1]` и щелкните на кнопке Run Cell.**

Будет выдан результат 1, как показано на рис. 13.5. Число, заключенное в квадратные скобки, называется *индексом*. Python всегда начинает индексацию с 0, поэтому запрос элемента с индексом 1 означает получение второго элемента в списке.

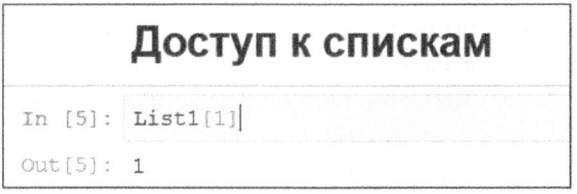


Рис. 13.5. Убедитесь в использовании корректного индекса

3. **Введите `List1[1:3]` и щелкните на кнопке Run Cell.**

Будет выдан диапазон значений, который включает два элемента, как показано на рис. 13.6. Верхняя граница диапазона всегда на единицу больше, чем количество возвращаемых элементов. В данном случае это означает, что вы получаете элементы 1 и 2, а не элементы с 1 по 3, как можно было бы ожидать.

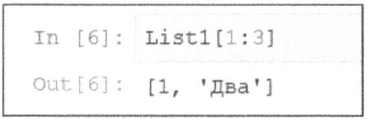


Рис. 13.6. Диапазоны содержат несколько значений

4. **Введите `List1[1:]` и щелкните на кнопке Run Cell.**

На экране появятся все элементы, начиная с элемента 1 и до конца списка, как показано на рис. 13.7. Если верхняя граница диапазона не указана, значит, подразумевается вся оставшаяся часть списка.

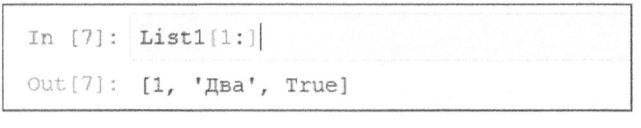


Рис. 13.7. Если не указать верхнюю границу диапазона, будет выведена оставшаяся часть списка

5. **Введите `List1[:3]` и щелкните на кнопке Run Cell.**

Отобразятся элементы от 0 до 2. Если нижняя граница диапазона не указана, вывод начинается с элемента 0, как показано на рис. 13.8.

```
In [8]: List1[:3]

Out[8]: ['Один', 1, 'Два']
```

Рис. 13.8. Если не указать нижнюю границу диапазона, вывод начнется с элемента 0

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В Python можно использовать отрицательные индексы, хотя это и кажется странным. При наличии отрицательного индекса Python будет просматривать список в направлении справа налево. Например, если вы создадите список `List1 = ["Один", 1, "Два", True]` и введете `List1[-2]`, то получите результат `"Два"`. Аналогично при вводе `List[-3]` вы получите значение `1`. В этом случае крайний справа элемент имеет индекс –1.

Циклический обход списков

Чтобы автоматизировать обработку элементов списка, требуется какой-то способ циклического обхода списка. Простейший способ выполнить это — воспользоваться инструкцией `for`, как описано ниже.

1. **Введите следующий код в блокноте.**

```
List1 = [0, 1, 2, 3, 4, 5]
for Item in List1:
   print(Item)
```

Пример начинается с создания списка, содержащего числовые значения. Затем используется цикл `for` для получения каждого элемента списка и отображения его на экране.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Python отобразит содержимое списка по одному элементу в строке, как показано на рис. 13.9.

Изменение списков

Содержимое списка можно изменить, если возникнет такая необходимость. Под изменением списка понимается добавление новых записей и удаление существующих. При работе со списками применяют акроним CRUD, означающий "Create, Read, Update, Delete" (создать, прочесть, обновить, удалить), т.е. четыре основные операции работы с объектами данных. Вот функции, связанные с CRUD:

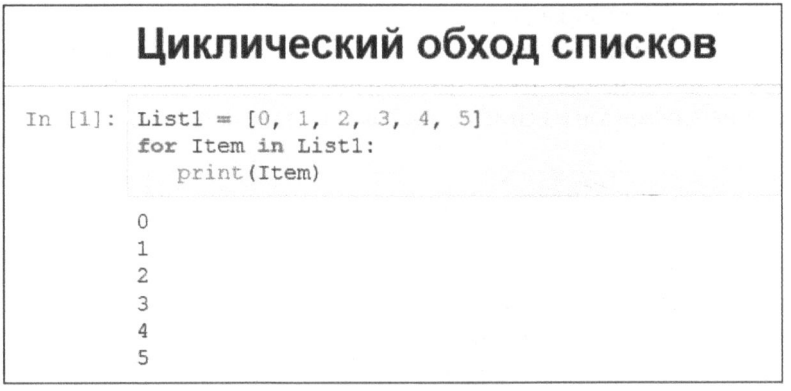


Рис. 13.9. Благодаря циклу облегчается получение копии каждого элемента списка

- `append()` — добавление новой записи в конец списка;
- `clear()` — удаление всех записей из списка;
- `copy()` — создание копии текущего списка и вставка ее в новый список;
- `extend()` — добавление элементов из существующего списка в текущий список;
- `insert()` — добавление новой записи в указанную позицию списка;
- `pop()` — удаление записи из конца списка;
- `remove()` — удаление записи из указанной позиции списка.

В следующей пошаговой инструкции показано, как модифицировать списки. Это всего лишь практическое упражнение, цель которого — помочь вам понять, как работают списки. По мере чтения книги вы увидите, как те же самые функции применяются в приложениях.

1. **Введите `List2 = [ ]`.**

   Python создаст список `List2`.

ЗАПОМНИ!

   Обратите внимание на то, что в квадратных скобках ничего нет. Список `List2` не содержит никаких записей. Можно создавать пустые списки, которые будут заполняться информацией позже. Фактически именно так создается большинство списков, поскольку вы обычно не знаете, какую информацию они будут содержать, пока пользователь не начнет взаимодействовать со списком.

2. **Введите `len(List2)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

Функция `len()` возвращает значение `0`, как показано на рис. 13.10. При создании приложения можно проверять, пуст ли список, используя функцию `len()`. Если список пустой, вы не сможете выполнять такие действия, как, например, удаление элементов из списка, поскольку удалять просто нечего.

```
In [2]: List2 = []
        len(List2)

Out[2]: 0
```

Рис. 13.10. Проверка списка на предмет наличия содержимого

3. **Введите `List2.append(1)` и нажмите клавишу <Enter>.**
4. **Введите `len(List2)` и щелкните на кнопке Run Cell.**

   Теперь функция `len()` сообщает о том, что длина списка равна `1`.
5. **Введите `List2[0]` и щелкните на кнопке Run Cell.**

   На экране появится значение, сохраненное в элементе 0 списка `List2`, как показано на рис. 13.11.

```
In [4]: List2[0]

Out[4]: 1
```

Рис. 13.11. Добавление элемента изменяет длину списка

6. **Введите `List2.insert(0, 2)`.**

   Функция `insert()` требует два аргумента. Первый аргумент — это индекс вставки, в данном случае `0`. Второй элемент — это объект, который нужно вставить в указанной позиции; в данном случае это число `2`.
7. **Введите `List2` и щелкните на кнопке Run Cell.**

   Python добавляет еще один элемент в список `List2`. С помощью функции `insert()` новый элемент можно вставить перед первым элементом списка, как показано на рис. 13.12.

```
In [5]: List2.insert(0, 2)
        List2

Out[5]: [2, 1]
```

Рис. 13.12. Благодаря функции `insert()` можно выбирать, где необходимо вставить новый элемент

8. **Введите `List3 = List2.copy()`.**

   Новый список `List3` становится точной копией списка `List2`. Копирование часто применяется для создания временной версии существующего списка, в результате чего пользователь сможет вносить изменения во временную копию, а не в исходный список. После завершения ввода данных пользователем приложение либо удаляет временный список, либо копирует его в исходный список.

9. **Введите `List2.extend(List3)`.**

   Python скопирует все элементы из списка `List3` в конец списка `List2`. Подобная операция часто применяется для объединения двух списков.

10. **Введите `List2` и щелкните на кнопке Run Cell.**

    После выполнения операций копирования и расширения список `List2` будет содержать значения `2`, `1`, `2` и `1`, как показано на рис. 13.13.

```
In [6]: List3 = List2.copy()
        List2.extend(List3)
        List2

Out[6]: [2, 1, 2, 1]
```

Рис. 13.13. Благодаря функциям `copy()` и `extend()` можно быстро перемещать большие объемы данных

11. **Введите `List2.pop()` и щелкните на кнопке Run Cell.**

    Python отобразит значение `1`, как показано на рис. 13.14. Это значение хранилось в конце списка, а функция `pop()` всегда удаляет значения из конца списка.

```
In [7]: List2.pop()

Out[7]: 1
```

Рис. 13.14. Функция `pop()` позволяет удалять элементы из конца списка

12. **Введите `List2.remove(1)` и щелкните на кнопке Run Cell.**

    На этот раз Python удалит элемент 1. В отличие от функции `pop()`, функция `remove()` не отображает значение удаляемого элемента.

13. **Введите `List2.clear()`.**

    После применения функции `clear()` список станет пустым.

14. **Введите `len(List2)` и щелкните на кнопке Run Cell.**

В данном случае результат будет 0, так как список List2 пуст. Итак, мы опробовали все функции модификации, которые Python предоставляет для работы со списками.

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРОВ ПРИ РАБОТЕ СО СПИСКАМИ

Некоторые действия над списками можно выполнять с помощью операторов. Например, если нужно создать список, содержащий четыре копии слова Здравствуй, то можно использовать инструкцию MyList = ["Здравствуй"] * 4 для заполнения списка. Список допускает повторение элементов. Оператор умножения (*) сообщает Python, сколько раз нужно повторить данный элемент. Python создаст соответствующее количество копий элемента. В результате список MyList будет выглядеть так: ['Здравствуй', 'Здравствуй', 'Здравствуй', 'Здравствуй'].

Для заполнения списка можно также использовать конкатенацию. Например, в результате выполнения инструкции MyList = ["Здравствуй"] + ["мир"] + ["!"] * 4 будут созданы 6 элементов в списке MyList. Первым элементом будет "Здравствуй", вторым — "мир", а за ними последуют четыре элемента, содержащих один восклицательный знак ("!").

Оператор членства (in) также работает со списками. В этой главе используется простой и понятный способ поиска в списках, но с помощью оператора членства можно упростить некоторые инструкции, сделав их более короткими. Например, выражение "Здравствуй" in MyList вернет True, если список был заполнен значениями ['Здравствуй', 'мир', '!', '!', '!', '!'].

Поиск в списках

Изменить содержимое списка будет непросто, если не знать, что в нем содержится. И здесь приходит на помощь возможность поиска в списке. Ниже показано, как искать в списке нужные значения.

1. **Введите следующий код в блокноте.**

```
Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
ColorSelect = ""
while str.upper(ColorSelect) != "ВЫХОД":
   ColorSelect = input("Укажите название цвета: ")
   if (Colors.count(ColorSelect) >= 1):
      print("Этот цвет указан в списке!")
   elif (str.upper(ColorSelect) != "ВЫХОД"):
      print("Этот цвет отсутствует в списке!")
```

Пример начинается с создания списка `Colors`, в котором содержатся названия цветов. Также создается переменная `ColorSelect`, в которой хранится название цвета, введенного пользователем. Затем приложение входит в цикл, в котором у пользователя запрашивается название цвета, сохраняемое в переменной `ColorSelect`. До тех пор пока этой переменной не будет присвоено значение `"ВЫХОД"`, приложение будет продолжать выполнять цикл, запрашивая ввод данных.

Всякий раз, когда пользователь вводит название цвета, приложение подсчитывает количество вхождений этого цвета в списке. Если число больше или равно единице, значит, цвет содержится в списке, и на экране появляется соответствующее сообщение. С другой стороны, если запрошенного цвета нет в списке, на экране появляется альтернативное сообщение.

СОВЕТ

Обратите внимание на то, каким образом в этом примере используется предложение `elif`. Оно позволяет поверить, не содержит ли переменная `ColorSelect` слово `"ВЫХОД"`. Соответствующее сообщение не должно быть выведено, если пользователь хочет завершить работу с приложением. Такого рода проверки нужно всегда применять во избежание путаницы или даже потери данных (когда приложение выполняет задачу, которую пользователь фактически не запрашивал).

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа попросит ввести название цвета.

3. **Введите Синий.**

   На экране появится сообщение о том, что этот цвет включен в список (рис. 13.15).

Поиск в списках

```
In [*]: Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
        ColorSelect = ""
        while str.upper(ColorSelect) != "ВЫХОД":
            ColorSelect = input("Укажите название цвета: ")
            if (Colors.count(ColorSelect) >= 1):
                print("Этот цвет указан в списке!")
            elif (str.upper(ColorSelect) != "ВЫХОД"):
                print("Этот цвет отсутствует в списке!")

        Укажите название цвета: Синий
        Этот цвет указан в списке!
```

Рис. 13.15. Сообщение о том, что цвет имеется в списке

4. **Введите Пурпурный.**

   Появится сообщение о том, что цвет отсутствует в списке, как показано на рис. 13.16.

Поиск в списках

```
In [*]: Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
        ColorSelect = ""
        while str.upper(ColorSelect) != "ВЫХОД":
           ColorSelect = input("Укажите название цвета: ")
           if (Colors.count(ColorSelect) >= 1):
              print("Этот цвет указан в списке!")
           elif (str.upper(ColorSelect) != "ВЫХОД"):
              print("Этот цвет отсутствует в списке!")

        Укажите название цвета: Синий
        Этот цвет указан в списке!
        Укажите название цвета: Пурпурный
        Этот цвет отсутствует в списке!
```

Рис. 13.16. Сообщение об отсутствии цвета в списке

5. **Теперь введите ВЫХОД.**

   Приложение завершит работу, не выдав никаких сообщений.

Сортировка списков

Компьютер найдет информацию в списке независимо от того, в каком порядке она хранится. Но, несомненно, поиск в длинных списках выполняется эффективнее, когда они отсортированы. Тем не менее основная причина сортировки списка состоит в том, чтобы облегчить пользователю просмотр информации, содержащейся в списке. Люди лучше воспринимают отсортированную информацию. Ниже показано, как выполнить сортировку простого списка.

1. **Введите следующий код в блокноте.**

```
Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
for Item in Colors:
   print(Item, end=" ")
print()
Colors.sort()
for Item in Colors:
   print(Item, end=" ")
print()
```

   Пример начинается с создания массива `Colors`. В данный момент цвета не отсортированы. Далее названия цветов выводятся в порядке их появления. Обратите внимание на использование аргумента `end=""` функции `print()`, благодаря чему все названия цветов выводятся в одну строку (это делает сравнение более наглядным).

ЗАПОМНИ!

Сортировка списка выполняется простым вызовом функции `sort()`. После этого содержимое списка снова выводится на экран, но уже в отсортированном порядке.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Программа выведет неотсортированный и отсортированный списки, как показано на рис. 13.17.

Сортировка списков

```
In [1]: Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
        for Item in Colors:
           print(Item, end=" ")
        print()
        Colors.sort()
        for Item in Colors:
           print(Item, end=" ")
        print()

        Красный Оранжевый Желтый Зеленый Синий
        Желтый Зеленый Красный Оранжевый Синий
```

Рис. 13.17. Сортировка списка заключается в простом вызове функции `sort()`

СОВЕТ

Иногда нужно отсортировать элементы списка в обратном порядке. Для этого используется функция `reverse()`. Она должна вызываться в отдельной строке. Предыдущий пример, выполняющий сортировку цветов в обратном порядке, будет выглядеть следующим образом.

```
Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
for Item in Colors:
   print(Item, end=" ")
print()
Colors.sort()
Colors.reverse()
for Item in Colors:
   print(Item, end=" ")
print()
```

Вывод списков

В Python доступно множество способов вывода информации. В этой главе было продемонстрировано лишь несколько простейших способов вывода списков. В реальных приложениях процедуры вывода будут намного сложнее, поэтому вам нужно освоить ряд дополнительных методик.

1. **Введите следующий код в блокноте.**

```
Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
print(*Colors, sep='\n')
```

Мы начинаем с создания того же списка цветов, что и в предыдущем разделе. Там для вывода содержимого списка применялся цикл `for`. Здесь же мы воспользуемся специальным оператором-звездочкой (`*`), который называется *оператором позиционного расширения*. Он выполняет распаковку списка и поочередно передает каждый элемент функции `print()`. Аргумент `sep` определяет способ разделения выводимых элементов. В данном случае каждый элемент записывается с новой строки.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа выводит список строка за строкой, как показано на рис. 13.18.

```
In [2]: Colors = ["Красный", "Оранжевый", "Желтый", "Зеленый", "Синий"]
        print(*Colors, sep='\n')

        Красный
        Оранжевый
        Желтый
        Зеленый
        Синий
```

*Рис. 13.18. Оператор * позволяет существенно сократить и упростить код*

3. **Введите следующий код в блокноте и щелкните на кнопке Run Cell.**

```
for Item in Colors: print(Item.rjust(8), sep='/n')
```

Программа не обязана располагаться на нескольких строках. В этом примере две строки кода соединены в одну строку. Здесь также демонстрируется использование метода `rjust()`, который задает выравнивание строки вправо, как показано на рис. 13.19. Многочисленные методы работы со строками описаны по адресу `https://docs.python.org/2/library/string.html`.

```
In [3]: for Item in Colors: print(Item.rjust(8), sep='/n')

          Красный
        Оранжевый
          Желтый
         Зеленый
           Синий
```

Рис. 13.19. С помощью строковых функций можно форматировать результат нужным образом

4. **Введите следующий код в блокноте и щелкните на кнопке Run Cell.**

```
print('\n'.join(Colors))
```

В Python предусмотрено несколько способов решения любой задачи. В данном случае используется функция `join()` для объединения символа новой строки с каждым элементом списка `Colors`. Результаты будут такими же, как на рис. 13.18, хотя достигается это по-другому. Вы всегда можете выбрать тот подход, который для вас удобнее.

5. **Введите следующий код в блокноте и щелкните на кнопке Run Cell.**

```
print('First: {0}\nSecond: {1}'.format(*Colors))
```

Здесь создается спецификация формата, содержащая поясняющий текст, причем в выводимые результаты включаются не все элементы списка `Colors`. Записи `{0}` и `{1}` соответствуют первому и второму элементам входных данных (они извлекаются при раскрытии списка с помощью оператора `*`). На рис. 13.20 показан полученный результат. Дополнительные сведения по этой теме можно найти по адресу `https://docs.python.org/3/tutorial/inputoutput.html`.

```
In [5]: print('Первый: {0}\nВторой: {1}'.format(*Colors))

        Первый: Красный
        Второй: Оранжевый
```

Рис. 13.20. Используйте функцию `format()`, когда выводимые данные нужно определенным образом форматировать

ЗАПОМНИ!

В этом разделе были рассмотрены лишь некоторые из применяемых в Python способов форматирования вывода. На самом деле их гораздо больше. Многие из них будут продемонстрированы в следующих главах. Выбирайте для себя тот подход, который вам понятнее и удобнее.

Работа с объектом `Counter`

Предположим, имеется источник данных и нужно узнать, как часто происходят определенные события (например, сколько раз тот или иной элемент появляется в списке). Если список короткий, то подсчет можно провести самостоятельно. Но если у вас по-настоящему длинный список, сами вы точно не справитесь. Представьте, что вас ждет, если, к примеру, нужно подсчитать частоту слов в романе "Война и мир", текст которого хранится в источнике данных. Решить такую задачу невозможно без помощи компьютера.

ЗАПОМНИ!

Объект `Counter` позволяет быстро сосчитать элементы списка. Кроме того, он невероятно прост в использовании. В этой главе будет показано, как применять его для работы со списками. Ниже создается список с повторяющимися элементами, а затем подсчитывается, сколько раз эти элементы встречаются в списке.

1. **Введите следующий код в блокноте.**

```
from collections import Counter
MyList = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1, 5]
ListCount = Counter(MyList)
print(ListCount)
for ThisItem in ListCount.items():
   print("Элемент: ", ThisItem[0],
         " Появляется: ", ThisItem[1])
print("Значение 1 появляется {0} раз."
      .format(ListCount.get(1)))
```

Для использования объекта `Counter` его нужно импортировать из пакета `collections`. Если в приложении используются и другие типы коллекций, необходимо импортировать весь пакет с помощью команды `import collections`.

Пример начинается с создания списка `MyList`, содержащего повторяющиеся числа. Несложно заметить, что некоторые элементы появляются более одного раза. Далее на основе списка `MyList` создается новый объект `ListCount` типа `Counter`. Объекты `Counter` можно создавать по-разному, но этот способ наиболее удобный при работе со списком.

ЗАПОМНИ!

Объект `Counter` и список сами по себе никак не связаны. При изменении содержимого списка придется заново создать объект `Counter`, поскольку он не увидит изменений. Альтернативный вариант — вызвать сначала метод `clear()`, чтобы очистить объект `Counter`, а затем — метод `update()` для заполнения объекта `Counter` новыми данными.

Приложение выводит содержимое переменной `ListCount` несколькими способами. Сначала выводится объект `Counter` в чистом виде, без какого-либо форматирования. Далее перечисляются уникальные элементы списка `MyList` с указанием количества появлений каждого элемента. Чтобы получить эти данные, следует использовать метод `items()` объекта `Counter`. В конце демонстрируется, как получить индивидуальный результат подсчета с помощью метода `get()`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа выведет результаты, полученные с помощью объекта `Counter`, как показано на рис. 13.21.

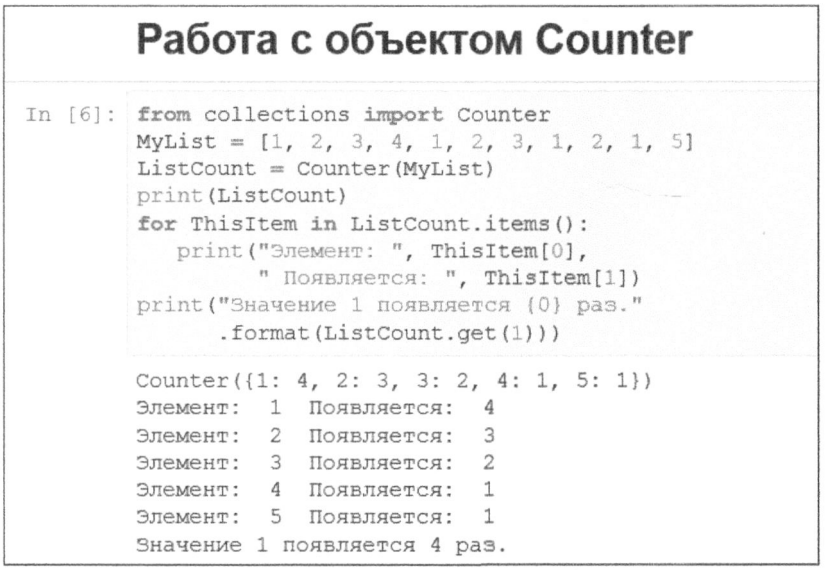


Рис. 13.21. С помощью объекта `Counter` можно получить статистику для длинных списков

Вы, должно быть, заметили, что информация фактически хранится в объекте `Counter` в виде пары "ключ — значение". Эта тема будет подробно рассмотрена в главе 14. Пока что вам достаточно знать, что элемент, найденный в списке `MyList`, становится в списке `ListCount` ключом, который идентифицирует уникальное имя элемента. Значение показывает, сколько раз этот элемент появляется в списке `MyList`.

Глава 14

Коллекции данных

В ЭТОЙ ГЛАВЕ...

- Понятие о коллекции
- Использование кортежей и словарей
- Создание стеков с помощью списков
- Использование очередей и деков

Люди коллекционируют все что угодно. Компакт-диски, декоративные тарелки, бейсбольные карточки — все это примеры коллекций. Коллекции, с которыми сталкивается разработчик при написании приложения, представляют собой то же самое. *Коллекция* — это набор схожих элементов, собранных в одном месте и организованных в простом и понятном порядке.

ЗАПОМНИ!

В этой главе рассматриваются самые разные коллекции. Основная цель сбора любой коллекции — создание такой среды, в которой благодаря правильной организации можно было бы быстро найти необходимый элемент в любое время. Книжные полки наиболее оптимальны для хранения книг, DVD и других плоских предметов. Коллекции ручек лучше хранить в пеналах или футлярах. Места хранения не влияют на саму суть домашних коллекций. То же самое можно сказать и о компьютерных коллекциях. Безусловно, между стеком и очередью имеются различия, но основная идея сводится к правильной организации данных, чтобы их можно было быстро находить. Исходный код примеров этой главы находится в файле `BPPD_14_Collecting_All_Sorts_of_Data.ipynb` (см. введение).

Понятие коллекции

В главе 13 вы познакомились с последовательностями. *Последовательность* — это ряд значений, хранящихся в одном в контейнере. Самой простой последовательностью является строка, представляющая собой цепочку символов. Списки, рассмотренные в главе 13, — это наборы объектов. И то и то — последовательности, но между строками и списками имеются существенные различия. Например, при работе со строками все символы можно перевести в нижний регистр, чего нельзя сделать при использовании списков. С другой стороны, списки позволяют добавлять новые элементы, а в строках эта возможность не поддерживается (в результате конкатенации фактически создается новая строка). Коллекция — это еще один вид последовательности, хотя и более сложный, чем строка или список.

ЗАПОМНИ!

Любая последовательность поддерживает две функции: `index()` и `count()`. Функция `index()` всегда возвращает позицию указанного элемента в последовательности. Например, можно определить положение символа в строке или элемента в списке. Функция `count()` сообщает, сколько раз указанный элемент появляется в списке. Опять же, тип элемента зависит от вида последовательности.

В Python с помощью коллекций можно создавать структуры наподобие баз данных. Каждый тип коллекции имеет определенное назначение, причем его можно использовать разными способами. Важно запомнить, что коллекция — один из видов последовательности. Как и любая другая разновидность последовательности, коллекции всегда поддерживают функции `index()` и `count()` как часть своей базовой функциональности.

В Python имеется базовый набор коллекций, с помощью которых можно создавать любые приложения. В этой главе рассматриваются наиболее популярные коллекции.

- **Кортеж.** Это коллекция, используемая для создания сложных последовательностей наподобие списков. Преимуществом кортежей является возможность вложения содержимого кортежа. Эта особенность позволяет создавать структуры, содержащие, к примеру, записи о сотрудниках или координатные пары *x*–*y*.
- **Словарь.** Как и в настоящих словарях, содержит пары "ключ/значение" (их можно рассматривать как слово и его определение). Словарь позволяет значительно сократить время поиска данных и упростить их организацию.

- **Стек.** Во многих языках программирования предлагается непосредственная поддержка стеков. Python не поддерживает стеки, хотя и предлагает частичное решение. Стек — это последовательность типа LIFO (Last In/First Out — "последним пришел — первым ушел"). В качестве иллюстрации представьте себе стопку блинчиков. Каждый новый блинчик, который кладут сверху, тут же съедают. Стек — это важный тип коллекции, который в Python можно смоделировать с помощью списка (именно эта методика демонстрируется в данной главе).
- **Очередь.** Это коллекция типа FIFO (First In/First Out — "первым пришел — первым ушел"), с помощью которой можно отслеживать элементы, требующие той или иной обработки. Представьте себе очередь в супермаркете. Вы подходите к кассе, становитесь в очередь и, наконец, оформляете покупку.
- **Двухсторонняя очередь (дек).** Это структура наподобие очереди, позволяющая добавлять или удалять элементы с обоих концов очереди, но не из середины. Дек можно использовать как очередь, стек или любой другой вид коллекции, допускающий добавление или удаление элементов в определенном порядке (в отличие от списков, кортежей и словарей, реализующих произвольный доступ).

Работа с кортежами

Как уже говорилось, кортеж — это коллекция, используемая для создания сложных списков, в которых один кортеж может вкладываться в другой. Такое вложение позволяет формировать иерархии наподобие листинга каталогов жесткого диска или организационной структуры компании.

СОВЕТ

Кортежи не поддаются изменению. Можно создать новый кортеж с таким же именем и измененной структурой, но нельзя изменить уже имеющийся кортеж. Списки же поддаются изменению. На первый взгляд может показаться, что именно в этом проявляется недостаток кортежа. Тем не менее в неизменности имеются и свои преимущества, а именно: более высокая степень безопасности и быстродействия. Более того, неизменяемые объекты проще в использовании в случае параллельной обработки, осуществляемой несколькими процессорами.

Основные различия между кортежем и списком заключаются в том, что кортеж неизменяем и позволяет вкладывать одни кортежи в другие. В следующей пошаговой инструкции демонстрируется работа с кортежами в Python.

1. **Откройте новый блокнот.**

   Можете также использовать файл исходного кода `BPPD_14_Collecting_All_Sorts_of_Data.ipynb` (см. введение).

2. **Введите `MyTuple = ('Красный', 'Синий', 'Зеленый')`.**

   Python создаст кортеж, состоящий из трех строк.

3. **Введите `MyTuple` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Отобразится содержимое кортежа `MyTuple`, состоящего из трех строк (рис. 14.1). Учтите, что строки будут заключены в одинарные кавычки, даже если при создании кортежа использовались двойные кавычки. Кроме того, в кортеже, в отличие от списка, применяются круглые скобки вместо квадратных.

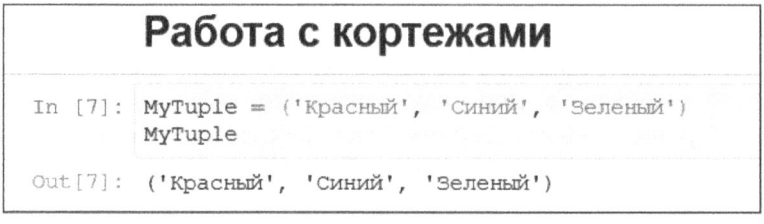


Рис. 14.1. В кортежах используются круглые скобки вместо квадратных

4. **Введите `print(dir(MyTuple))` и щелкните на кнопке Run Cell.**

   Python выведет список функций, поддерживаемых кортежами (рис. 14.2). Это напоминает перечень функций списков, который был рассмотрен в главе 13. Здесь также имеются функции `count()` и `index()`.

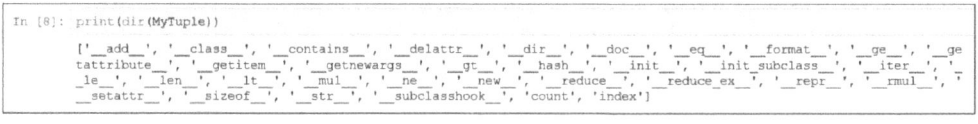


Рис. 14.2. В кортежах используется меньше функций, чем в списках

СОВЕТ

Результат, выводимый при использовании функции `print()` вместе с функцией `dir()`, отличается от результата, выдаваемого самой функцией `dir()` (см. рис. 13.4). Это больше напоминает результаты, получаемые в других IDE, например IDLE. Многие пользователи предпочитают применяемый в Notebook способ, когда названия выводятся построчно, поскольку их легче читать. В то же время комбинированный вывод определенно получается намного более компактным.

5. **Введите `MyTuple = MyTuple.__add__(('Пурпурный',))`.**

   Этот код добавляет новый кортеж в `MyTuple` и помещает результат в новый экземпляр `MyTuple`. После выполнения инструкции старый экземпляр `MyTuple` удаляется.

ЗАПОМНИ!

Функция `__add__()` в качестве входных данных поддерживает только кортежи. Это означает, что добавляемый элемент следует заключить в круглые скобки. Более того, при создании кортежа с одной записью после нее следует ставить запятую, как показано в примере. Не забывайте об этом, иначе получите следующее сообщение об ошибке:

```
TypeError: can only concatenate tuple (not "str") to tuple
```

6. **Введите `MyTuple` и щелкните на кнопке Run Cell.**

   В конце кортежа появится новый элемент, расположенный на том же уровне, что и другие элементы (рис. 14.3).

```
In [9]: MyTuple = MyTuple.__add__(('Пурпурный',))
        MyTuple

Out[9]: ('Красный', 'Синий', 'Зеленый', 'Пурпурный')
```

Рис. 14.3. В новом экземпляре `MyTuple` содержится дополнительный элемент

7. **Введите `MyTuple = MyTuple.__add__(('Желтый', ('Оранжевый', 'Черный')))`.**

   На этом шаге добавляются три элемента: `Желтый`, `Оранжевый` и `Черный`. Однако элементы `Оранжевый` и `Черный` становятся вложенным кортежем, образуя иерархию. Они рассматриваются как единый элемент внутри главного кортежа.

СОВЕТ

Функцию `__add__()` можно заменить оператором конкатенации. Например, если нужно добавить элемент `Фуксия` в начало кортежа, введите **`MyTuple = ("Фуксия",) + MyTuple`**.

8. **Введите `MyTuple[4]` и щелкните на кнопке Run Cell.**

   Python отобразит соответствующий элемент кортежа `MyTuple`: `Желтый`. Кортежи, как и списки, используют индексы для получения доступа к отдельным элементам. При необходимости можно указать диапазон. Все операции, выполняемые с индексом списка, можно выполнять и с индексом кортежа.

9. **Введите `MyTuple[5]`.**

   Отобразится кортеж, содержащий элементы `Оранжевый` и `Черный`.

СОВЕТ

Кортежи часто имеют иерархическую структуру. Чтобы выяснить, находится ли по указанному индексу другой кортеж, нужно выполнить проверку типа. В данном примере для проверки шестого элемента (с индексом 5) нужно ввести **`type(MyTuple[5]) == tuple`**. В результате получим значение `True`.

10. **Введите `MyTuple[5][0]`.**

На этом этапе отобразится результат `Оранжевый`. Индексы всегда указываются в порядке своего уровня в иерархии.

```
In [13]: MyTuple[5][0]
Out[13]: 'Оранжевый'
```

Рис. 14.4. Использование индексов для получения доступа к отдельным элементам кортежа

СОВЕТ

С помощью индексов и функции `__add__()` (или оператора конкатенации +) можно создавать гибкие приложения, использующие кортежи. Например, можно удалить элемент из кортежа, приравняв последний к диапазону значений. Чтобы удалить кортеж, содержащий элементы `Оранжевый` и `Черный`, введите **`MyTuple = MyTuple[0:5]`**.

Работа со словарями

Словарь в Python организован так же, как и реальные словари, и содержит пары "ключ — значение" (слово и его определение в словаре). Как и списки, словари можно изменять. Основная идея словаря заключается в ускорении процесса поиска данных. Благодаря коротким и уникальным ключам компьютер быстро находит необходимую информацию.

В следующих разделах рассматриваются вопросы создания и использования словарей. Зная, как работать со словарями, можно применять эти знания для исправления недоработок в языке Python. Большинство языков программирования включает инструкцию `switch`, позволяющую создавать меню из различных вариантов, среди которых выбирается один. В Python такой инструкции нет, поэтому для решения соответствующих задач обычно приходится использовать инструкцию `if...elif`, которую нельзя считать самым эффективным решением.

Создание и использование словаря

Работа со словарем во многом напоминает работу со списком, с той лишь разницей, что для словаря задается пара, состоящая из ключа и значения. При создании ключей необходимо соблюдать определенные правила.

- **Ключ должен быть уникальным.** При вводе дублирующегося ключа новая запись получает преимущество над старой, которая просто затирается.
- **Ключ должен быть неизменяемым.** Это означает, что в качестве ключа можно использовать строки, числа или кортежи, но не списки.

Ограничения на используемые значения отсутствуют. Значением может быть любой объект Python, благодаря чему с помощью словаря можно получить доступ к учетной карточке сотрудника или другим сложным данным. В следующей пошаговой инструкции демонстрируется использование словаря.

1. **Введите `Colors = {"Сэм": "Синий", "Эмма": "Красный", "Сара": "Желтый"}`.**

   Python создаст словарь, содержащий три записи, соответствующие любимым цветам трех людей. Обратите внимание на то, как создается пара "ключ — значение". Сначала указывается ключ, потом двоеточие и за ним — значение. Записи разделяются запятыми.

2. **Введите `Colors` и щелкните на кнопке Run Cell.**

   Отобразится содержимое словаря (рис. 14.5). Во многих случаях записи будут отсортированы по ключам. Словарь по возможности автоматически сортирует ключи для ускорения доступа, благодаря чему сокращается время поиска даже при работе с большими объемами данных. Недостаток словаря заключается в том, что его создание отнимает больше времени, чем использование того же списка, поскольку компьютер тратит время на сортировку записей.

Создание и использование словаря

```
In [14]: Colors = {"Сэм": "Синий", "Эмма": "Красный", "Сара": "Желтый"}
         Colors

Out[14]: {'Сэм': 'Синий', 'Эмма': 'Красный', 'Сара': 'Желтый'}
```

Рис. 14.5. Вывод содержимого словаря

3. **Введите `Colors["Сара"]` и щелкните на кнопке Run Cell.**

   Отобразится цвет, который нравится Саре. В данном случае это `Желтый` (рис. 14.6). Использование строки в качестве ключа вместо числового индекса упрощает чтение кода, который в определенной степени становится самодокументируемым. Это экономит разработчикам немало времени, что объясняет высокую популярность словарей. Тем не менее удобство достигается за счет дополнительных трат времени и ресурсов, так что нужно взвешивать все плюсы и минусы.

```
In [9]: Colors["Сара"]

Out[9]: 'Желтый'
```

Рис. 14.6. Словари упрощают доступ к значениям и делают их самодокументируемыми

4. **Введите `Colors.keys()` и щелкните на кнопке Run Cell.**

   Отобразится перечень ключей словаря (рис. 14.7). Эти ключи можно использовать для автоматизации доступа к словарю.

```
In [16]: Colors.keys()

Out[16]: dict_keys(['Сэм', 'Эмма', 'Сара'])
```

Рис. 14.7. Можно запросить список ключей словаря

5. **Введите следующий код в блокноте, после чего щелкните на кнопке Run Cell.**

```
for Item in Colors.keys():
   print("{0} любит цвет {1}."
      .format(Item, Colors[Item]))
```

ЗАПОМНИ!

Программа выведет список имен пользователей вместе с их любимыми цветами (рис. 14.8). Благодаря словарям получение такого рода результатов существенно упрощается. Использование понятного ключа означает, что сам ключ легко может стать частью результата.

```
In [15]: for Item in Colors.keys():
            print("{0} любит цвет {1}."
               .format(Item, Colors[Item]))

         Сэм любит цвет Синий.
         Эмма любит цвет Красный.
         Сара любит цвет Желтый.
```

Рис. 14.8. Благодаря ключам нужная информация выводится намного быстрее

6. **Введите `Colors["Сара"] = "Пурпурный"`.**

   Содержимое словаря обновится, и теперь Саре нравится не желтый, а пурпурный цвет.

7. **Введите `Colors.update({"Гарри": "Оранжевый"})`.**

   В результате в словарь добавится новая запись.

8. **Введите `del Colors["Сэм"]`.**

   Программа удалит из словаря запись Сэма.

9. **Введите следующий код в блокноте.**

```
for name, color in Colors.items():
   print("{0} любит цвет {1}."
      .format(name, color))
```

   Сравните этот код с кодом, рассмотренным в п. 5. В этой версии каждый элемент обрабатывается последовательно: ключ присваивается переменной `name`, а значение — переменной `color`. Результат будет тем же. Во многих случаях такой вариант проще читать. С точки зрения быстродействия оба варианта одинаковы.

10. **Щелкните на кнопке Run Cell.**

    Обновленный результат представлен на рис. 14.9. Запись для Гарри добавлена в конец словаря. Кроме того, для Сары любимый цвет стал пурпурным.

```
In [13]: del Colors["Сэм"]
         for name, color in Colors.items():
             print("{0} любит цвет {1}."
                   .format(name, color))

         Эмма любит цвет Красный.
         Сара любит цвет Пурпурный.
         Гарри любит цвет Оранжевый.
```

Рис. 14.9. Словари нетрудно изменить

11. **Введите `len(Colors)` и щелкните на кнопке Run Cell.**

    Выводимое значение `3` подтвердит, что теперь словарь содержит три записи вместо четырех.

12. **Введите `Colors.clear()`.**

13. **Введите `len(Colors)` и щелкните на кнопке Run Cell.**

    Программа сообщит, что `Colors` содержит 0 записей, поэтому словарь пуст.

Замена инструкции `switch` словарем

В большинстве языков программирования имеется инструкция `switch`, которая позволяет создавать меню выбора. Из нескольких предоставляемых вариантов пользователю разрешается выбрать лишь один. На основе выбора пользователя программа выполняет определенные действия. Ниже приведен пример такой инструкции (не пытайтесь его выполнить).

```
switch(n)
{
```

```
   case 0:
      print("Вы выбрали синий.");
      break;
   case 1:
      print("Вы выбрали желтый.");
      break;
   case 2:
      print("Вы выбрали зеленый.");
      break;
}
```

Приложение предлагает пользователю сделать выбор, получает от него номер варианта, после чего активирует соответствующую ветвь инструкции `switch`. Это проще и понятнее, чем использовать набор инструкций `if` для тех же целей.

К сожалению, в Python нет инструкции `switch`. Ее можно заменить инструкцией `if...elif`, но есть и другой вариант: смоделировать применение инструкции `switch` с помощью словаря. Ниже показано, как это сделать.

1. **Введите следующий код в блокноте.**

```
def PrintBlue():
   print("Вы выбрали синий!\r\n")
def PrintRed():
   print("Вы выбрали красный!\r\n")
def PrintOrange():
   print("Вы выбрали оранжевый!\r\n")
def PrintYellow():
   print("Вы выбрали желтый!\r\n")
```

Каждая из этих функций реализует соответствующую ветвь выбора цвета. Вызвать можно будет только одну функцию.

2. **Введите следующий код в блокноте.**

```
ColorSelect = {
   0: PrintBlue,
   1: PrintRed,
   2: PrintOrange,
   3: PrintYellow
}
```

Здесь мы создаем словарь. Каждый ключ представляет собой ветвь `case` инструкции `switch`. Значение определяет необходимое действие. Иными словами, это структура типа `switch`. Созданные ранее функции — это блоки кода в соответствующих ветвях инструкции `switch`, т.е. код, находящийся между инструкциями `case` и `break`.

3. **Введите следующий код в блокноте.**

```
Selection = 0
while (Selection != 4):
   print("0. Синий")
   print("1. Красный")
   print("2. Оранжевый")
   print("3. Желтый")
   print("4. Выход")
   Selection = int(input("Выберите цвет: "))
   if (Selection >= 0) and (Selection < 4):
      ColorSelect[Selection]()
```

По сути, это та часть программы, которая реализует пользовательский интерфейс. Вначале мы создаем переменную `Selection`, хранящую результат выбора. Затем цикл выполняется до тех пор, пока пользователь не введет значение `4`.

На каждой итерации цикла программа отображает список вариантов и ожидает ввода пользовательских данных. Когда данные получены, приложение выполняет проверку диапазона. Выбор любого значения между `0` и `3` приводит к выполнению одной из функций словаря.

4. **Щелкните на кнопке Run Cell.**

Программа отобразит меню (рис. 14.10).

5. **Введите 0 и нажмите клавишу <Enter>.**

Программа сообщит о выборе синего цвета и еще раз отобразит меню (рис. 14.11).

6. **Введите 4 и нажмите клавишу <Enter>.**

Программа завершит работу.

Создание стеков с помощью списков

Стек — это удобная программная структура, которую можно использовать для хранения среды выполнения приложения (состояния переменных и других атрибутов программного окружения в любой момент времени). К сожалению, в Python стек не может быть коллекцией. Тем не менее стек можно смоделировать с помощью списка. Ниже показан пример использования списка в качестве стека.

Замена инструкции switch словарем

```
In [*]: def PrintBlue():
            print("Вы выбрали синий!\r\n")
        def PrintRed():
            print("Вы выбрали красный!\r\n")
        def PrintOrange():
            print("Вы выбрали оранжевый!\r\n")
        def PrintYellow():
            print("Вы выбрали желтый!\r\n")

        ColorSelect = {
            0: PrintBlue,
            1: PrintRed,
            2: PrintOrange,
            3: PrintYellow
        }

        Selection = 0
        while (Selection != 4):
            print("0. Синий")
            print("1. Красный")
            print("2. Оранжевый")
            print("3. Желтый")
            print("4. Выход")
            Selection = int(input("Выберите цвет: "))
            if (Selection >= 0) and (Selection < 4):
                ColorSelect[Selection]()
```

```
0. Синий
1. Красный
2. Оранжевый
3. Желтый
4. Выход

Выберите цвет:
```

Рис. 14.10. Приложение начинает работу с отображения меню

```
0. Синий
1. Красный
2. Оранжевый
3. Желтый
4. Выход
Выберите цвет: 0
Вы выбрали синий!

0. Синий
1. Красный
2. Оранжевый
3. Желтый
4. Выход

Выберите цвет:
```

Рис. 14.11. Если пользователь выбрал цвет, приложение снова отображает меню

1. **Введите следующий код в блокноте.**

```
MyStack = []
StackSize = 3
def DisplayStack():
   print("В настоящее время стек содержит:")
   for Item in MyStack:
      print(Item)
def Push(Value):
   if len(MyStack) < StackSize:
      MyStack.append(Value)
   else:
      print("Стек полон!")
def Pop():
   if len(MyStack) > 0:
       MyStack.pop()
   else:
       print("Стек пуст.")
Push(1)
Push(2)
Push(3)
DisplayStack()
input("Нажмите любую клавишу...")
Push(4)
DisplayStack()
input("Нажмите любую клавишу...")
Pop()
DisplayStack()
input("Нажмите любую клавишу...")
Pop()
Pop()
Pop()
DisplayStack()
```

В этом примере приложение создает список и переменную для определения максимального размера стека. Как правило, у стека есть ограничение по размеру. В данном случае мы имеем дело с маленьким стеком, но для демонстрации этого вполне достаточно.

ЗАПОМНИ!

Принцип работы стека заключается в добавлении новых значений на вершину стека и удалении запрашиваемых значений с вершины. Эти две задачи решают функции `Push()` и `Pop()`. В код также включена функция `DisplayStack()`, чтобы можно было быстро просмотреть содержимое стека.

Остальной код тестирует стек (проверяет его функциональность), помещая в него значения с последующим их удалением.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение заполняет стек данными и затем отображает его содержимое (рис. 14.12). В данном случае на вершине стека находится значение 3, поскольку это последнее добавленное значение.

```
DisplayStack()
input("Нажмите любую клавишу...")
Push(4)
DisplayStack()
input("Нажмите любую клавишу...")
Pop()
DisplayStack()
input("Нажмите любую клавишу...")
Pop()
Pop()
Pop()
DisplayStack()

В настоящее время стек содержит:
1
2
3

Нажмите любую клавишу...
```

Рис. 14.12. Стек заносит значения одно за другим

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В зависимости от используемой IDE сообщение `Нажмите любую клавишу...` может появиться как в верхней, так и в нижней части области вывода. В Notebook сообщение и соответствующее поле ввода первоначально появятся вверху, а затем переместятся вниз.

3. **Нажмите клавишу <Enter>.**

   Приложение попытается занести еще одно значение в стек. Но поскольку стек полон, эту задачу выполнить не удастся (рис. 14.13).

```
В настоящее время стек содержит:
1
2
3
Нажмите любую клавишу...
Стек полон!
В настоящее время стек содержит:
1
2
3

Нажмите любую клавишу...
```

Рис. 14.13. Заполненный стек больше не может принимать значения

4. **Нажмите клавишу <Enter>.**

   Приложение удалит значение с вершины стека. Поскольку на вершине стека находилось число 3, именно оно отсутствует на рис. 14.14.

```
В настоящее время стек содержит:
1
2

Нажмите любую клавишу...
```

Рис. 14.14. Удаление значения с вершины стека

5. **Нажмите клавишу <Enter>.**

   Приложение попытается удалить из стека больше значений, чем в нем содержится, в результате чего последняя операция не будет выполнена (рис. 14.15). В любой модели стека должна предусматриваться возможность обнаружения как переполнения стека (слишком много записей), так и его исчерпания (слишком мало записей).

```
Стек пуст.
В настоящее время стек содержит:
```

Рис. 14.15. Убедитесь в том, что реализация стека способна выявить его переполнение и исчерпание

Работа с очередями

Очередь работает не так, как стек. Представим обычную очередь в магазине: вы становитесь в конец, постепенно продвигаетесь вперед и, когда доходите до кассы, наконец совершаете покупку. Очереди часто используются для планирования задач и управления выполнением программ. В следующей инструкции показано, как создать приложение, работающее с очередью.

1. **Введите следующий код в блокноте.**

```
import queue
MyQueue = queue.Queue(3)
print(MyQueue.empty())
input("Нажмите любую клавишу...")
MyQueue.put(1)
MyQueue.put(2)
print(MyQueue.full())
input("Нажмите любую клавишу...")
MyQueue.put(3)
print(MyQueue.full())
input("Нажмите любую клавишу...")
print(MyQueue.get())
print(MyQueue.empty())
```

```
print(MyQueue.full())
input("Нажмите любую клавишу...")
print(MyQueue.get())
print(MyQueue.get())
```

Для создания очереди необходимо импортировать пакет `queue`. В нем содержится несколько типов очередей, но в данном примере используется лишь стандартная очередь FIFO.

Когда очередь пуста, функция `empty()` возвращает значение `True`. Аналогичным образом, когда очередь заполнена, функция `full()` возвращает значение `True`. Указанные две функции позволяют управлять очередью. С очередью нельзя работать в цикле `for`, как с другими типами коллекций. Вместо этого следует отслеживать ее состояние с помощью функций `empty()` и `full()`.

В Python имеются две функции, работающие с данными в очереди. Это функция `put()`, которая добавляет новые данные, и `get()`, которая их удаляет.

ЗАПОМНИ!

Очереди присущ один серьезный недостаток. При попытке добавить в нее большее количество элементов, чем она может вместить, очередь просто будет ожидать освобождения места. Как следствие, приложение может "зависнуть", если только оно не является *многопоточным* (т.е. содержит параллельные потоки выполнения).

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение проверит состояние очереди. В данном случае отображается результат `True`, а это значит, что очередь пуста.

3. **Нажмите клавишу <Enter>.**

   Приложение добавит в очередь два новых значения, поэтому перестанет быть пустой (рис. 14.16).

4. **Нажмите клавишу <Enter>.**

   Приложение добавляет в очередь еще одну, третью запись. Поскольку достигнут предельный размер очереди, она становится полной, и результатом вызова функции `full()` будет `True`.

5. **Нажмите клавишу <Enter>.**

   Чтобы освободить место в очереди, приложение удаляет одну из записей, вызывая функцию `get()`. Поскольку первым в очередь было занесено значение `1`, именно его и вернет функция `get()`, как показано на рис. 14.17. Теперь обе функции, `empty()` и `full()`, должны вернуть значение `False`.

6. **Нажмите клавишу <Enter>.**

   Приложение удалит оставшиеся две записи. В результате отобразятся значения `2` и `3` (в том порядке, в котором они заносились в очередь).

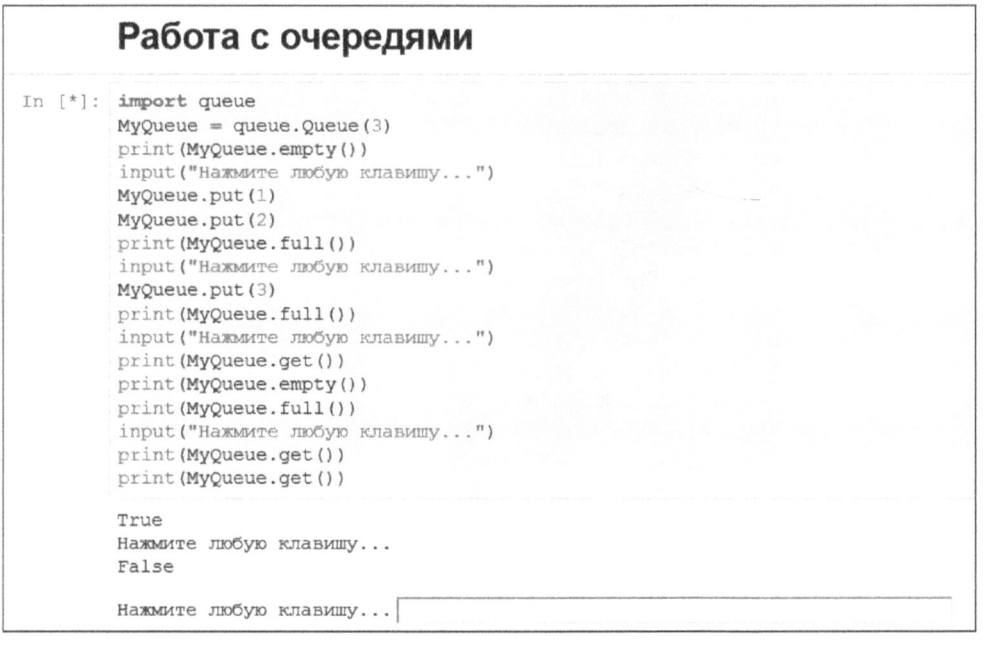


Рис. 14.16. После добавления новых записей очередь не будет пустой

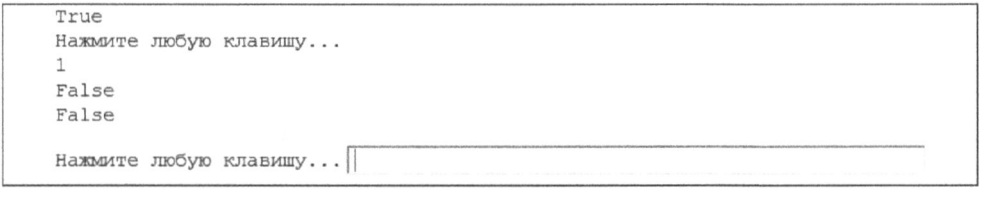


Рис. 14.17. При работе с очередью важно отслеживать ее состояние

Работа с двухсторонней очередью

Двухсторонняя очередь, или *дек*, — это очередь, позволяющая добавлять и удалять элементы с двух сторон. Во многих языках программирования очередь и стек поддерживают двухстороннюю функциональность, которая ограничивается в приложении в соответствии с его задачами.

При работе с двухсторонней очередью представляйте ее как горизонтальный ряд. Имеются функции, предназначенные для работы с левым и правым концами дека, что позволяет добавлять и удалять элементы с обеих сторон. В следующем примере демонстрируется работа с двухсторонней очередью.

1. **Введите следующий код в блокноте.**

```
import collections
MyDeque = collections.deque("abcdef", 10)
print("Начальное состояние:")
for Item in MyDeque:
   print(Item, end=" ")
print("\r\n\r\nДобавление и расширение вправо")
MyDeque.append("g")
MyDeque.extend("hi")
for Item in MyDeque:
   print(Item, end=" ")
print("\r\nMyDeque содержит {0} элементов."
      .format(len(MyDeque)))
print("\r\nВыталкивание вправо")
print("Выталкивание {0}".format(MyDeque.pop()))
for Item in MyDeque:
   print(Item, end=" ")
print("\r\n\r\nДобавление и расширение влево")
MyDeque.appendleft("a")
MyDeque.extendleft("bc")
for Item in MyDeque:
   print(Item, end=" ")
print("\r\nMyDeque содержит {0} элементов."
      .format(len(MyDeque)))
print("\r\nВыталкивание влево")
print("Выталкивание {0}".format(MyDeque.popleft()))
for Item in MyDeque:
   print(Item, end=" ")
print("\r\n\r\nУдаление")
MyDeque.remove("a")
for Item in MyDeque:
   print(Item, end=" ")
```

Поскольку реализация двухсторонней очереди содержится в пакете `collections`, его нужно импортировать. При создании дека можно дополнительно задать начальный список *итерируемых элементов* (элементы, подвергающиеся циклической обработке) и максимальный размер очереди.

ЗАПОМНИ!

Двухсторонняя очередь различает добавление одного элемента и добавление группы элементов. В первом случае используются функции `append()` и `appendleft()`. Функции `extend()` и `extendleft()` позволяют добавлять несколько элементов. Для удаления одного элемента используются функции `pop()` и `popleft()`, которые возвращают удаляемое значение. Особенность функции `remove()` состоит в том, что она всегда работает слева и удаляет первый экземпляр запрашиваемых данных.

В отличие от других коллекций, двухсторонняя очередь является полностью итерируемой. Это означает, что в любой момент можно получить содержимое очереди в цикле `for`.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение выведет информацию, представленную на рис. 14.18.

ВНИМАНИЕ!

Проанализируйте полученные результаты, наблюдая за тем, как меняется размер очереди. После удаления `i` в очереди содержится восемь элементов. При расширении слева приложение добавляет в очередь еще три элемента. Тем не менее она содержит только десять элементов. При превышении максимального размера очереди лишние данные просто выталкиваются с другой стороны.

```
Начальное состояние:
a b c d e f

Добавление и расширение вправо
a b c d e f g h i
MyDeque содержит 9 элементов.

Выталкивание вправо
Выталкивание i
a b c d e f g h

Добавление и расширение влево
c b a a b c d e f g
MyDeque содержит 10 элементов.

Выталкивание влево
Выталкивание c
b a a b c d e f g

Удаление
b a b c d e f g
```

Рис. 14.18. Пример использования двухсторонней очереди

Глава 15

Создание и использование классов

В ЭТОЙ ГЛАВЕ...

- Определение характеристик класса
- Задание компонентов класса
- Применение класса
- Работа с подклассами

В предыдущих главах вам уже приходилось работать с классами. Многие из примеров книги просты в создании и использовании благодаря тому, что основаны на классах Python. Впрочем, в предыдущих главах классы лишь вскользь упоминались, поскольку их обсуждению следует посвятить целую главу.

Классы упрощают работу с кодом Python, позволяя сделать приложения более компактными, наглядными и практичными. Классы служат для создания программных контейнеров, группирующих код и относящиеся к нему данные. Для внешнего кода класс является типичным черным ящиком, который получает данные и выдает данные.

ЗАПОМНИ!

Рано или поздно вы начнете создавать собственные классы, чтобы избежать типичного спагетти-кода, характерного для старых приложений. *Спагетти-код* представляет собой хаотичное нагромождение процедурного кода, в котором трудно понять, где заканчивается

одна задача и начинается следующая. Сопровождать спагетти-код практически невозможно, и некоторые организации вынуждены были отказаться от такого рода приложений только потому, что никто не мог их понять.

В этой главе вы изучите концепцию классов как программных контейнеров, позволяющих избежать спагетти-кода, и создадите свой первый класс. Вы получите представление о том, как работают классы Python и как сделать ваши приложения более удобными для работы. Исходный код примеров этой главы находится в файле `BPPD_15_Creating_and_Using_Classes.ipynb` (см. введение).

Класс как средство упаковки кода

По сути, класс является контейнером кода. Идея состоит в том, чтобы упростить повторное использование кода, сделать приложения более надежными и снизить вероятность ошибок. Хорошо спроектированные классы — это черные ящики, которые получают определенные входные данные и выдают соответствующие выходные данные. Другими словами, класс должен обладать предсказуемым и согласованным поведением. Не имеет значения, как именно класс выполняет свою работу. Сокрытие внутренних деталей является хорошей практикой программирования.

Прежде чем двигаться дальше, следует изучить ряд терминов, характерных для классов. Их необходимо знать, чтобы понять материал главы. Эти термины специфичны для Python. (В других языках программирования могут использоваться иные названия для тех же концепций; иногда те же самые термины употребляются в другом контексте.)

ЗАПОМНИ!

» **Класс.** Определяет схему, по которой создаются объекты. Представьте себе строителя, который хочет построить здание. Строитель использует архитектурный проект, позволяющий гарантировать, что здание будет соответствовать требуемым спецификациям. Аналогичным образом Python использует классы в качестве чертежей для создания новых объектов.

ВНИМАНИЕ!

» **Переменная класса.** Место хранения, используемое всеми методами экземпляра класса. Переменная класса определяется внутри самого класса, но за пределами любого из его методов. Такие переменные используются относительно редко, поскольку представляют потенциальную угрозу безопасности вследствие того, что каждый экземпляр класса имеет доступ к одной и той же информации.

Кроме того, они видны как часть определения класса и доступны в обход его методов, что несет дополнительные риски.

Использование глобальных переменных всегда считалось плохой практикой программирования, особенно это касается Python, поскольку каждый экземпляр может видеть одну и ту же информацию. В отличие от других языков программирования, в Python не поддерживается сокрытие данных. Каждая такая переменная всегда доступна, что препятствует реализации по-настоящему объектно-ориентированных решений.

- **Переменная-член.** Переменная класса или переменная экземпляра, используемая для хранения данных, связанных с классом и его объектами.
- **Перегрузка функции.** Создание нескольких версий функции, каждой из которых присуще свое поведение. Основная задача, решаемая функцией, может оставаться неизменной, но входные данные различаются, а следовательно, потенциально различаются и полученные результаты.

  Благодаря перегрузке функций обеспечивается гибкость программных решений, когда одна и та же функция работает в приложении по-разному, решая одну и ту же задачу для разных типов входных данных.
- **Наследование.** Использование родительского класса для создания дочерних классов с одинаковыми характеристиками. Дочерние классы обычно имеют расширенные функциональные возможности либо обеспечивают более специфическое поведение, чем родительский класс.

ЗАПОМНИ!

- **Экземпляр.** Объект класса, созданный на основе содержащейся в нем спецификации. Python может создать столько экземпляров класса, сколько запрашивает приложение. Каждый экземпляр уникален.
- **Переменная экземпляра.** Место хранения, используемое одним методом экземпляра класса. Переменная экземпляра определяется внутри метода. Такие переменные считаются более безопасными, чем переменные класса, потому что только один метод может получить доступ к ним. Метод получает данные через аргументы, что позволяет выполнять проверку входных данных и осуществлять контроль типов.

ЗАПОМНИ!

- **Создание экземпляра.** Выполнение специальной функции-конструктора, создающей экземпляр класса. Полученный объект представляет собой уникальный экземпляр класса, отличающийся от других экземпляров.

» **Метод.** Функция, являющаяся частью класса. По сути, метод и функция — это одно и то же, но термин "метод" считается более конкретным, поскольку методы есть только у классов.

» **Объект.** Уникальный экземпляр класса. Объект содержит все методы и переменные исходного класса, но данные для каждого объекта различаются. Каждый объект хранится отдельно.

» **Перегрузка оператора.** Создание нескольких версий функции, связанной с оператором, таким как +, –, / или *, что приводит к различным вариантам поведения. Назначение оператора может оставаться тем же, но способ обработки операндов отличается. Благодаря перегрузке операторов обеспечивается гибкость программных решений, когда один и тот же оператор можно применять по отношению к данным разного типа.

Компоненты класса

Класс имеет четкую структуру. Каждая часть класса служит конкретной цели, определяя его полезные характеристики. Прежде всего, класс заключается в контейнер, включающий весь его код. Об этом мы поговорим в следующем разделе, а затем перейдем к рассмотрению других компонентов класса.

Создание определения класса

Класс не обязан быть чересчур сложным. Фактически можно создать саму оболочку контейнера, включить в нее одну переменную и назвать все это классом. Конечно, полученный таким образом класс будет не слишком полезным, но все равно вы сможете создать его экземпляр и работать с ним как с любым другим классом. В следующей пошаговой инструкции демонстрируется создание простейшего класса.

1. **Откройте новый блокнот.**

   Можете также воспользоваться файлом исходного кода `BPPD_15_Creating_and_Using_Classes.ipynb` (см. введение).

2. **Введите следующий код в блокноте.**

```
class MyClass:
   MyVar = 0
```

   В первой строке определяется контейнер класса, который состоит из ключевого слова `class` и имени класса `MyClass`. Каждый создаваемый класс должен начинаться именно так.

Вторая строка представляет собой содержимое класса. Все элементы, составляющие класс, называются его *членами*. В данном случае вы видите переменную класса `MyVar`, значение которой равно `0`. Каждый экземпляр класса будет включать одну и ту же переменную, имеющую одно и то же начальное значение.

3. **Введите `MyInstance = MyClass()`.**

   Вы только что создали экземпляр класса `MyClass` с именем `MyInstance`! Давайте теперь убедимся в этом.

4. **Введите `MyInstance.MyVar` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Полученный результат `0` (рис. 15.1) означает, что экземпляр `MyInstance` действительно содержит переменную класса с именем `MyVar`.

Рис. 15.1. Созданный экземпляр содержит требуемую переменную

5. **Введите `MyInstance.__class__` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Python сообщит, какой класс применялся для создания указанного экземпляра (рис. 15.2). Префикс `__main__` означает, что данный класс включен непосредственно в код приложения и не является частью другого пакета.

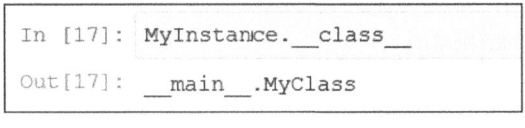


Рис. 15.2. Данный экземпляр был создан с помощью класса `MyClass`

Встроенные атрибуты класса

В процессе создания класса вам кажется, будто класс содержит только то, что вы ему назначаете. Но на самом деле это не так, поскольку Python добавляет в класс встроенную функциональность. Например, в предыдущем разделе вы применили встроенный атрибут `__class__`, который сами не создавали. Полезно знать о наличии таких атрибутов, чтобы не пытаться добавлять их самостоятельно. Подобные возможности будут востребованы довольно часто, поэтому Python включает их в каждый класс. В следующей пошаговой инструкции показано, как работать со встроенными атрибутами класса. При этом предполагается, что вы выполнили действия, описанные в предыдущем разделе.

1. **Введите `print(dir(MyInstance))` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появится список атрибутов, показанный на рис. 15.3. Эти атрибуты реализуют универсальную функциональность класса. Они являются общими для всех создаваемых классов, поэтому на них всегда можно рассчитывать.

```
In [4]: print(dir(MyInstance))

        ['MyVar', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattri
        bute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__'
        , '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__
        ']
```

Рис. 15.3. Использование функции `dir()` для идентификации встроенных атрибутов

2. **Введите `help('__class__')`.**

   Python отобразит информацию об атрибуте `__class__`, как показано на рис. 15.4. Подобным образом можно получить справку о любом атрибуте класса.

Работа с методами

Метод — это просто функция, определенная в классе. Метод создается и применяется точно так же, как и функция, за исключением того, что он всегда связан с классом (не существует автономных методов, вызываемых вне класса). Существуют два вида методов: те, которые связаны с самим классом, и те, которые связаны с экземпляром класса. Важно не путать их.

Создание методов класса

Метод класса — это метод, который можно вызвать непосредственно из класса, не создавая экземпляра. Иногда требуется создавать такие методы, которые применяются к любым объектам класса. Например, в главе 10 мы

использовали функцию `str.upper()`, которая определена в классе `str`. В следующей пошаговой инструкции демонстрируется создание и использование метода класса.

```
In [5]: help('__class__')

        Help on class module in module builtins:

        __class__ = class module(object)
         |  module(name[, doc])
         |
         |  Create a module object.
         |  The name must be a string; the optional doc argument can have any type.
         |
         |  Methods defined here:
         |
         |  __delattr__(self, name, /)
         |      Implement delattr(self, name).
         |
         |  __dir__(...)
         |      __dir__() -> list
         |      specialized dir() implementation
         |
         |  __getattribute__(self, name, /)
         |      Return getattr(self, name).
         |
         |  __init__(self, /, *args, **kwargs)
         |      Initialize self.  See help(type(self)) for accurate signature.
         |
         |  __new__(*args, **kwargs) from builtins.type
         |      Create and return a new object.  See help(type) for accurate signature.
         |
         |  __repr__(self, /)
         |      Return repr(self).
         |
         |  __setattr__(self, name, value, /)
         |      Implement setattr(self, name, value).
         |
         |  ----------------------------------------------------------------------
```

Рис. 15.4. Python предоставляет справку для каждого атрибута класса

1. **Введите следующий код в блокноте.**

```
class MyClass:
   def SayHello():
      print("Привет всем!")
```

Созданный класс содержит единственный метод `SayHello()`, который не имеет никаких аргументов и не возвращает никаких значений. Он просто выводит сообщение. Для демонстрационных целей этого вполне достаточно.

2. **Введите `MyClass.SayHello()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

Программа выведет ожидаемую строку, как показано на рис. 15.5. Обратите внимание на то, что вам не пришлось создавать экземпляр класса, поскольку метод вызывается прямо из класса.

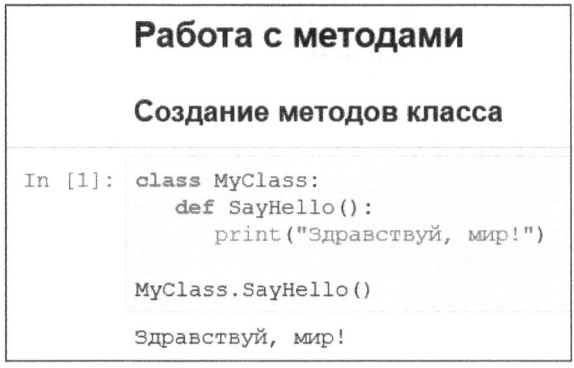


Рис. 15.5. Метод класса выводит простое сообщение

Метод класса может работать только с данными класса. Он не имеет доступа ни к каким данным, связанным с экземплярами класса. Ему лишь можно передавать данные в виде аргументов. Таким образом, нужно проявлять осторожность при создании методов класса и следить за тем, чтобы они были самодостаточными.

Создание методов экземпляра

Метод экземпляра является членом отдельного экземпляра класса. Такие методы применяются для работы с данными, хранящимися в экземпляре. Как следствие, их нельзя использовать до тех пор, пока не будет создан объект класса.

Все методы экземпляра имеют как минимум один аргумент: `self`. Он ссылается на текущий экземпляр, из которого был вызван метод. Без аргумента `self` метод не знал бы о том, какие данные использовать. Значение для `self` предоставляет сам Python, и его невозможно изменить при вызове метода. Ниже показано, как создавать и использовать методы экземпляра в Python.

1. **Введите следующий код в блокноте.**

```
class MyClass:
   def SayHello(self):
      print("Привет всем!")
```

Созданный класс содержит единственный метод `SayHello()`, который не имеет никаких аргументов и не возвращает никаких значений. Он просто выводит сообщение. Тем не менее для демонстрационных целей этого вполне достаточно.

2. **Введите `MyInstance = MyClass()`.**

   Python создаст экземпляр класса `MyClass` с именем `MyInstance`.

3. **Введите `MyInstance.SayHello()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появится сообщение, показанное на рис. 15.6.

Создание методов экземпляра

```
In [2]: class MyClass:
            def SayHello(self):
                print("Здравствуй, мир!")

        MyInstance = MyClass()
        MyInstance.SayHello()

        Здравствуй, мир!
```

Рис. 15.6. Метод экземпляра вызывается по отношению к конкретному объекту и выводит простое сообщение

Работа с конструкторами

Конструктор — это особый метод, вызываемый в Python при создании объекта. Python использует конструктор для выполнения таких действий, как *инициализация* (присвоение начальных значений) любых переменных экземпляра, которые понадобятся объекту при запуске. Конструктор также может проверить, достаточно ли ресурсов для объекта, и выполнить любые другие начальные действия.

ЗАПОМНИ!

Имя конструктора всегда одно: `__init__()`. Конструктор может иметь аргументы, если это необходимо для создания объекта. Когда создается класс без конструктора, Python автоматически создает конструктор по умолчанию, который не выполняет никаких действий. Каждый класс должен иметь конструктор, даже если он просто вызывает конструктор, заданный по умолчанию. Ниже показано, как создать конструктор.

1. **Введите следующий код в блокноте.**

```
class MyClass:
   Greeting = ""
   def __init__(self, Name="мир"):
      self.Greeting = Name + "!"
   def SayHello(self):
      print("Здравствуй, {0}".format(self.Greeting))
```

Это наш первый пример перегрузки функций. В данном случае существуют две версии конструктора `__init__()`. Первый конструктор не требует предоставления входного аргумента, поскольку использует значение `Name`, заданное по умолчанию: `"мир"`. Второй конструктор требует указать имя. Это имя вместе с восклицательным знаком записывается в переменную `Greeting`. Метод `SayHello()`, по сути, такой же, как и в предыдущих примерах.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В Python не поддерживается истинная перегрузка функций. Ярые приверженцы строгих принципов объектно-ориентированного программирования (ООП) считают, что значения, заданные по умолчанию, — это совсем не то же самое, что перегрузка функций. Однако результат достигается тот же самый, и это единственная альтернатива, доступная в Python. При истинной перегрузке функций будет создано несколько версий одной и той же функции, каждая из которых будет по-разному обрабатывать входные данные.

2. **Введите `MyInstance = MyClass()`.**

   Python создаст экземпляр класса `MyClass` с именем `MyInstance`.

3. **Введите `MyInstance.SayHello()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появится сообщение, показанное на рис. 15.7. В данном случае выводится приветствие, заданное по умолчанию.

Работа с конструкторами

```
In [3]: class MyClass:
            Greeting = ""
            def __init__(self, Name="мир"):
                self.Greeting = Name + "!"
            def SayHello(self):
                print("Здравствуй, {0}".format(self.Greeting))

        MyInstance = MyClass()
        MyInstance.SayHello()

        Здравствуй, мир!
```

Рис. 15.7. Первая версия конструктора использует имя, заданное по умолчанию

4. **Введите `MyInstance2 = MyClass("Эмма")`.**

   Python создаст экземпляр класса `MyClass` с именем `MyInstance2`. Он полностью отличается от экземпляра `MyInstance`.

5. **Введите `MyInstance2.SayHello()`.**

Python сформирует сообщение для Эммы, отличающееся от того, которое было получено для экземпляра класса `MyInstance`.

6. **Введите `MyInstance.Greeting ="Гарри!".`**

   Приветствие для экземпляра `MyInstance` изменится, не затрагивая приветствие для экземпляра `MyInstance2`.

7. **Введите `MyInstance.SayHello()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появятся сообщения, показанные на рис. 15.8. Для каждого экземпляра выводится свое приветствие. Каждый экземпляр уникален, и можно изменить сообщение для первого экземпляра, не влияя на второй экземпляр.

```
In [4]: MyInstance2 = MyClass("Эмма")
        MyInstance2.SayHello()
        MyInstance.Greeting = "Гарри!"
        MyInstance.SayHello()

        Здравствуй, Эмма!
        Здравствуй, Гарри!
```

Рис. 15.8. Передавая конструктору имя, можно настраивать сообщение

Работа с переменными

Как упоминалось ранее, переменные являются контейнерами для хранения данных. Работая с классами, необходимо учитывать, как хранятся данные и каким образом ими можно управлять. Класс может включать как переменные класса, так и переменные экземпляра. Первые создаются как часть самого класса, а вторые — как часть методов. В следующих разделах будет показано, как использовать оба типа переменных.

Создание переменных класса

Переменные класса обеспечивают глобальный доступ к данным, которыми ваш класс каким-либо образом работает. В большинстве случаев глобальные переменные инициализируются в конструкторе, так как это позволяет гарантировать, что они содержат известное начальное значение. Ниже показано, как работают переменные класса.

1. **Введите следующий код в блокноте.**

```
class MyClass:
   Greeting = ""
   def SayHello(self):
      print("Здравствуй, {0}".format(self.Greeting))
```

Это версия программы из предыдущего раздела, только в данном случае она не включает конструктор. Обычно конструктор добавляют, чтобы обеспечить корректную инициализацию переменных класса. Здесь же показано, что произойдет, если этого не сделать.

2. **Введите `MyClass.Greeting = "Зельда!"`.**

   Переменной `Greeting` будет присвоено значение, отличающееся от значения, которое использовалось при создании класса. Конечно, кто угодно может сделать подобное изменение. Вопрос в том, произойдет ли оно.

3. **Введите `MyClass.Greeting` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Вы увидите, что значение переменной `Greeting` было изменено (рис. 15.9).

Создание переменных класса

```
In [10]: class MyClass:
             Greeting = ""
             def SayHello(self):
                 print("Здравствуй, {0}".format(self.Greeting))

         MyClass.Greeting = "Зельда!"
         MyClass.Greeting

Out[10]: 'Зельда!'
```

Рис. 15.9. Пользователь может изменить значение переменной `Greeting`

4. **Введите `MyInstance = MyClass()`.**

   Python создаст экземпляр класса `MyClass` с именем `MyInstance`.

5. **Введите `MyInstance.SayHello()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появится сообщение, показанное на рис. 15.10. Изменение, внесенное в переменную `Greeting`, отразилось на экземпляре класса. В данном случае использование переменной класса не вызвало проблем, но в реальном приложении подобная практика может привести к серьезным неприятностям.

```
In [11]: MyInstance = MyClass()
         MyInstance.SayHello()

         Здравствуй, Зельда!
```

Рис. 15.10. Изменение значения переменной `Greeting` отражается на экземпляре класса

Это всего лишь простой пример ситуации, в которой переменные класса могут повлиять на корректную работу программы. Отсюда можно вынести два урока:

- по возможности не используйте переменные класса, поскольку они могут быть небезопасными;
- всегда инициализируйте переменные класса известным значением в коде конструктора.

Создание переменных экземпляра

Переменные экземпляра всегда определяются внутри метода. Аргументы метода тоже считаются переменными экземпляра, поскольку они существуют только тогда, когда существует метод. Использовать переменные экземпляра обычно безопаснее, чем переменные класса, поскольку их легче контролировать. Вы всегда сможете проверить, что вызывающий код предоставил корректные данные. Ниже приведен пример работы с переменными экземпляра.

1. **Введите следующий код в блокноте.**

```
class MyClass:
   def DoAdd(self, Value1=0, Value2=0):
       Sum = Value1 + Value2
       print("Сумма {0} и {1} равна {2}."
             .format(Value1, Value2, Sum))
```

   В данном случае у нас есть три переменные экземпляра. Входные аргументы `Value1` и `Value2` имеют значения, заданные по умолчанию (`0`), поэтому метод `DoAdd()` не сможет завершиться сбоем только из-за того, что пользователь забыл указать значения. Конечно, пользователь еще может передать нечисловые данные, поэтому в код нужно включить соответствующие проверки. Третья переменная экземпляра — это `Sum`. Ее значение вычисляется по формуле `Value1 + Value2`. Программа просто суммирует два числа и выводит результат.

2. **Введите `MyInstance = MyClass()`.**

   Python создаст экземпляр класса `MyClass` с именем `MyInstance`.

3. **Введите `MyInstance.DoAdd(1, 4)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   На экране появится сообщение, показанное на рис. 15.11. В данном случае отображается сумма чисел 1 и 4.

Использование методов с переменным числом аргументов

Время от времени вам будут требоваться методы, которые могут принимать переменное число аргументов. Обработка подобных ситуаций очень хорошо

Создание переменных экземпляра

```
In [3]: class MyClass:
            def DoAdd(self, Value1=0, Value2=0):
                Sum = Value1 + Value2
                print("Сумма {0} и {1} равна {2}."
                      .format(Value1, Value2, Sum))

        MyInstance = MyClass()
        MyInstance.DoAdd(1, 4)

        Сумма 1 и 4 равна 5.
```

Рис. 15.11. Результат представляет собой сумму двух чисел

реализована в Python. Вот два типа переменных списков аргументов, которые можно создать:

- `*args` — список неименованных аргументов;
- `**kwargs` — список именованных аргументов.

ЗАПОМНИ!

Фактические имена аргументов не важны, но в соответствии с общепринятым соглашением разработчики Python используют имена `*args` и `**kwargs`, давая тем самым понять, что это переменный список аргументов. В первом случае имя содержит только одну звездочку (*), означающую использование неименованных аргументов. Во втором случае имя сожержит две звездочки, что означает использование именованных аргументов. Ниже показано применение обоих типов списков.

1. **Введите следующий код в блокноте.**

```
class MyClass:
   def PrintList1(*args):
      for Count, Item in enumerate(args):
         print("{0}. {1}".format(Count, Item))
   def PrintList2(**kwargs):
      for Name, Value in kwargs.items():
         print("{0} любит {1}".format(Name, Value))
MyClass.PrintList1("Красный", "Синий", "Зеленый")
MyClass.PrintList2(Джордж="Красный", Сью="Синий",
                   Зара="Зеленый")
```

В этом примере аргументы реализованы как часть метода класса. Но их так же легко можно использовать в методе экземпляра.

СОВЕТ

В методе `PrintList1()` продемонстрирован новый способ использования цикла `for` для перебора списка. В данном случае функция `enumerate()` возвращает как номер аргумента, так и строку, переданную вызывающим кодом.

Функция `PrintList2()` получает данные в виде словаря. Как и в случае с функцией `PrintList1()`, список аргументов может быть любой длины. При этом для получения отдельных значений нужно вызвать функцию `items()`, извлекающую данные из словаря.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Результат выполнения программы показан на рис. 15.12. Попробуйте поэкспериментировать с кодом, чтобы лучше понять концепцию переменных списков. Например, попробуйте смешать числа и строки в первом списке и посмотрите, что произойдет. Попробуйте также добавить булевы значения. Суть в том, что эта методика придает методам невероятную гибкость, если все, что вам нужно, — это список входных значений.

Использование методов с переменными списками аргументов

```
In [5]: class MyClass:
            def PrintList1(*args):
                for Count, Item in enumerate(args):
                    print("{0}. {1}".format(Count, Item))
            def PrintList2(**kwargs):
                for Name, Value in kwargs.items():
                    print("{0} любит {1}".format(Name, Value))
        MyClass.PrintList1("Красный", "Синий", "Зеленый")
        MyClass.PrintList2(Джордж="Красный", Сью="Синий",
                           Зара="Зеленый")

        0. Красный
        1. Синий
        2. Зеленый
        Джордж любит Красный
        Сью любит Синий
        Зара любит Зеленый
```

Рис. 15.12. Метод может обрабатывать любое количество аргументов

Перегрузка операторов

Иногда при использовании стандартных операторов, таких как + (сложение), требуется выполнить что-то нестандартное. Благодаря перегрузке можно назначить новую функциональность существующим операторам, чтобы они выполняли то, что нужно вам, а не то, что по умолчанию предполагается в Python. Ниже показано, как выполнить перегрузку оператора и применить измененный оператор в приложении.

1. **Введите следующий код в блокноте.**

```
class MyClass:
   def __init__(self, *args):
      self.Input = args
   def __add__(self, Other):
      Output = MyClass()
      Output.Input = self.Input + Other.Input
      return Output
   def __str__(self):
      Output = ""
      for Item in self.Input:
         Output += Item
         Output += " "
      return Output
Value1 = MyClass("Красный", "Зеленый", "Синий")
Value2 = MyClass("Желтый", "Пурпурный", "Голубой")
Value3 = Value1 + Value2
print("{0}+ {1}= {2}"
      .format(Value1, Value2, Value3))
```

В примере демонстрируется несколько новых методик. В частности, в конструкторе `__init__()` показано, как создать переменную экземпляра, связанную с объектом `self`. Аналогичным образом можно создать произвольное количество переменных.

ЗАПОМНИ!

Когда создаются новые классы, оператор + для них не определяется по умолчанию. Его нужно создать самостоятельно. Единственное исключение имеет место в случае наследования существующего класса, в котором данный оператор уже определен (дополнительные сведения по этой теме будут приведены далее). Чтобы иметь возможность суммировать два объекта `MyClass`, нужно определить метод `__add__()`, который соответствует оператору +.

Код метода `__add__()` может показаться немного странным, поэтому рассмотрим его построчно. Метод начинается с создания нового объекта `Output` класса `MyClass`. На данный момент в объекте `Output` ничего нет, он пуст. Два объекта, содержимое которых нужно суммировать, `self.Input` и `Other.Input`, в действительности являются кортежами. (Дополнительные сведения о кортежах были приведены в главе 14.) Метод помещает сумму двух кортежей в переменную `Output.Input`, после чего возвращает итоговый объект вызывающему коду.

Вы спросите, почему нельзя просто сложить два входных кортежа? Ответ заключается в том, что в результате вы тоже получите кортеж, а не объект `MyClass`. Тип результирующего значения изменится, и вы не сможете использовать его так, как исходные объекты.

Для корректного вывода содержимого объекта `MyClass` необходимо определить метод `__str__()`. Он извлекает все аргументы из переменной `self.Input`, формируя единую строку, разделенную пробелами. В собственных классах вы вправе выводить любые строки.

В основной процедуре создаются два тестовых объекта, `Value1` и `Value2`. Они суммируются, и результат помещается в `Value3`, содержимое которой затем выводится на экран.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

На рис. 15.13 показан результат сложения двух объектов. Нам пришлось написать немало кода как для такого простого результата, но зато вы узнали, как создавать классы, которые являются автономными и полностью функциональными.

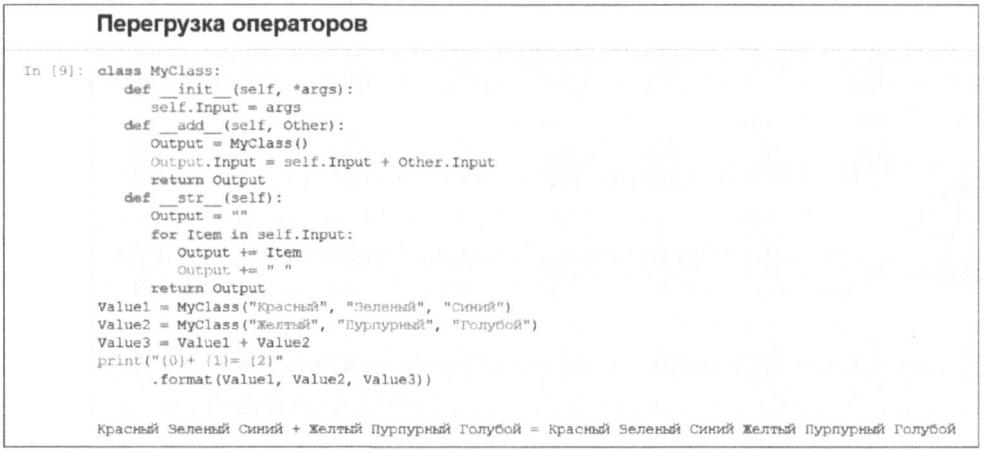


Рис. 15.13. В результате сложения двух объектов `MyClass` получается третий объект такого же типа

Создание класса

Весь предыдущий материал главы готовил вас к созданию собственного класса. Сейчас мы создадим класс, который помещается во внешний пакет, чтобы к нему можно было получать доступ из приложений.

Определение класса `MyClass`

В листинге 15.1 приведен код класса. Он содержится в файле `BPPD_15_MyClass.ipynb` (см. введение).

Листинг 15.1. Создание внешнего класса

```
class MyClass:
   def __init__(self, Name="Сэм", Age=32):
       self.Name = Name
       self.Age = Age
   def GetName(self):
      return self.Name
   def SetName(self, Name):
      self.Name = Name
   def GetAge(self):
      return self.Age
   def SetAge(self, Age):
      self.Age = Age
   def __str__(self):
      return "{0} в возрасте {1}.".format(self.Name, self.Age)
```

Код класса начинается с создания объекта, включающего две переменные экземпляра: `Name` и `Age`. Если пользователь не присвоит им значения, они по умолчанию получат значения `Сэм` и `32`.

ЗАПОМНИ!

В этом примере представлена еще одна функциональная особенность классов: *методы доступа*. Они обеспечивают доступ к внутренним переменным объектов. Существуют два типа методов доступа: *геттеры* (от англ. "getter" — получатель) и *сеттеры* (от англ. "setter" — установщик). Методы `GetName()` и `GetAge()` являются геттерами, позволяющими прочитать значение переменной. Методы `SetName()` и `SetAge()` являются сеттерами, позволяющими изменить значение переменной. Подобное разделение методов дает возможность проверять корректность входных данных и контролировать, имеет ли вызывающий код разрешение на просмотр информации.

Как и для любого другого создаваемого вами класса, нужно определить метод `__str__()`, если вы хотите, чтобы пользователь мог выводить содержимое объекта на экран. В данном случае класс формирует строку вывода, в которой содержатся обе переменные экземпляра.

Сохранение класса на диске

Класс может находиться в том же файле, что и код, предназначенный для его тестирования, но на практике такое встречается нечасто. Чтобы класс можно было тестировать в последующих примерах, выполните следующие действия.

1. **Создайте новый блокнот под названием `BPPD_15_MyClass.ipynb`.**
2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Если код был введен корректно, Python выполнит его без ошибок.
3. **Выполните команду File⇨Save and Checkpoint (Файл⇨Сохранить и создать точку восстановления).**

   Notebook сохранит файл.
4. **Выберите команду File⇨Download As⇨Python (.py) (Файл⇨Загрузить как⇨Python (.py)).**

   Notebook выведет код в виде файла Python.
5. **Импортируйте полученный файл в свой блокнот.**

   О том, как это сделать, рассказывалось в разделе "Импорт блокнота" главы 4.

Использование класса в приложении

Чаще всего вы будете использовать внешние классы при работе с Python. Класс редко включают в файл приложения, поскольку такой файл быстро становится слишком большим и неуправляемым. Кроме того, нельзя будет повторно использовать код класса в другом приложении. В следующей пошаговой инструкции демонстрируется применение класса `MyClass`, который был создан в предыдущем разделе.

1. **Введите следующий код в блокноте.**

```
import BPPD_15_MyClass
SamsRecord = BPPD_15_MyClass.MyClass()
AmysRecord = BPPD_15_MyClass.MyClass("Эмма", 44)
print(SamsRecord.GetAge())
SamsRecord.SetAge(33)
print(AmysRecord.GetName())
AmysRecord.SetName("Эйми")
print(SamsRecord)
print(AmysRecord)
```

   Программа начинается с импорта пакета `BPPD_15_MyClass`. Имя пакета — это не имя класса, а имя файла, содержащего код класса. Один пакет может содержать несколько классов, поэтому всегда думайте о пакете как о фактическом файле, который используется для хранения одного или нескольких классов.

   Далее приложение создает два объекта `MyClass`. Обратите внимание на то, что сначала указывается имя пакета, а затем — имя класса. Первый объект, `SamsRecord`, создается с настройками по умолчанию. Второй объект, `AmysRecord`, содержит данные, полученные от пользователя.

С момента создания приложения Сэм стал старше на год. Сначала приложение сообщает его текущий возраст, а затем, получив новые данные, обновляет возраст Сэма.

Каким-то образом сотрудник отдела кадров неправильно записал имя Эйми. Оказывается, Эмма — это неправильное написание. Опять-таки, сначала приложение сообщает текущее имя, после чего вносит исправления в объект `AmysRecord`. Последний шаг заключается в выводе на экран обеих записей.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение выводит сообщения, генерируемые классом `MyClass`, как показано на рис. 15.14. Теперь вы знаете, как создавать собственные классы.

Использование класса в приложении

```
In [1]: import BPPD_15_MyClass
        SamsRecord = BPPD_15_MyClass.MyClass()
        AmysRecord = BPPD_15_MyClass.MyClass("Эмма", 44)
        print(SamsRecord.GetAge())
        SamsRecord.SetAge(33)
        print(AmysRecord.GetName())
        AmysRecord.SetName("Эйми")
        print(SamsRecord)
        print(AmysRecord)

        32
        Эмма
        Сэм в возрасте 33.
        Эйми в возрасте 44.
```

Рис. 15.14. Полученные результаты говорят о том, что класс полностью функционален

Расширение классов для создания новых классов

В профессиональных приложениях, применяемых в производственной среде, создание классов отнимает много времени, так как необходимо запрограммировать множество задач. К счастью, в Python поддерживается механизм *наследования*, с помощью которого можно получить нужную функциональность из родительского класса при создании дочернего класса. Переопределяя методы, которые вам не нужны, и добавляя новые методы, вы сможете создавать новые классы относительно быстро и с гораздо меньшими усилиями. Кроме того, поскольку родительский код уже протестирован, вам не придется прилагать много усилий, чтобы гарантировать корректную работу нового класса. Далее будет показано, как создавать и использовать дочерние классы.

Создание дочернего класса

Родительские классы обычно являются супермножествами чего-либо. Например, можно создать родительский класс `Car`, а затем определить дочерние классы, описывающие различные типы автомобилей. В следующем примере создается родительский класс `Animal` (Животное), на основе которого определяется дочерний класс `Chicken` (Курица). Можно легко добавить и другие дочерние классы, например `Gorilla` (Горилла), но мы ограничимся только одним родительским и одним дочерним классом (листинг 15.2). Чтобы класс можно было использовать далее, его нужно сохранить на диске, как описывалось ранее. Назовите файл `BPPD_15_Animals.ipynb`.

Листинг 15.2. Создание родительского и дочернего классов

```
class Animal:
   def __init__(self, Name="", Age=0, Type=""):
      self.Name = Name
      self.Age = Age
      self.Type = Type
   def GetName(self):
      return self.Name
   def SetName(self, Name):
      self.Name = Name
   def GetAge(self):
      return self.Age
   def SetAge(self, Age):
      self.Age = Age
   def GetType(self):
      return self.Type
   def SetType(self, Type):
      self.Type = Type
   def __str__(self):
      return "{0} будет {1} в возрасте {2}".format(self.Name,
                                                   self.Type,
                                                   self.Age)

class Chicken(Animal):
   def __init__(self, Name="", Age=0):
      self.Name = Name
      self.Age = Age
      self.Type = "курица"
   def SetType(self, Type):
      print("Извините, {0} всегда будет {1}"
            .format(self.Name, self.Type))
   def MakeSound(self):
      print("{0} говорит ко, ко, ко!".format(self.Name))
```

Класс Animal отслеживает три характеристики: Name, Age и Type. Производственное приложение, вероятно, будет отслеживать больше характеристик, но для данного примера этого вполне достаточно. Класс также включает необходимые методы доступа для каждой из характеристик. Метод __str__() выводит на экран простое сообщение с указанием характеристик животного.

Класс Chicken наследуется от класса Animal. Имя родительского класса указывается в скобках после имени дочернего класса. Это подсказывает Python, что класс Chicken производный от класса Animal, т.е. наследующий его характеристики.

Заметьте, что конструктор класса Chicken получает только значения Name и Age. Пользователю не нужно указывать значение Type, поскольку вы и так знаете, что это курица. Новый конструктор переопределяет конструктор класса Animal. Все три атрибута остаются на месте, но атрибут Type инициализируется непосредственно в конструкторе Chicken.

Кто-то может забавы ради поменять атрибут Type, выставив, например, свою курицу гориллой. Учитывая это, класс Chicken также переопределяет метод-сеттер SetType(). Если кто-то попытается изменить тип класса Chicken, то получит соответствующее сообщение. Обычно проблема такого рода решается с помощью исключения, но сообщение делает код более компактным и понятным.

Наконец, класс Chicken добавляет новый метод MakeSound(). Если кто-то захочет услышать звук, издаваемый курицей, то сможет вызвать этот метод, чтобы реализовать свою мечту.

Тестирование класса в приложении

Тестирование класса Chicken также в определенной степени позволяет проверить работу класса Animal, который является родительским для различных классов животных. Это позволит нам увидеть, как работает наследование.

1. **Введите следующий код в блокноте.**

```
import BPPD_15_Animals
MyChicken = BPPD_15_Animals.Chicken("Салли", 2)
print(MyChicken)
MyChicken.SetAge(MyChicken.GetAge() + 1)
print(MyChicken)
MyChicken.SetType("Горилла")
print(MyChicken)
MyChicken.MakeSound()
```

Первый шаг заключается в импорте пакета Animals. Помните, что вы всегда указываете имя файла, а не класса. Файл пакета в действительности содержит два класса: Animal и Chicken.

В этом примере создается курица `MyChicken` по имени Салли, которой 2 года. Затем мы проверяем различные способы работы с классом `MyChicken`. Например, сегодня у Салли день рождения, поэтому программа увеличивает возраст Салли на один год. Обратите внимание на совместное использование методов `setAge()` и `GetAge()`. После каждого изменения программа выводит содержимое объекта. Последний шаг заключается в том, чтобы позволить Салли сказать несколько слов.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Результаты работы программы показаны на рис. 15.15. Как видите, использование наследования позволяет существенно упростить задачу создания новых классов в ситуации, когда несколько классов имеют общую функциональность. Вы создаете родительский класс, который содержит общий код, используемый в дочерних классах.

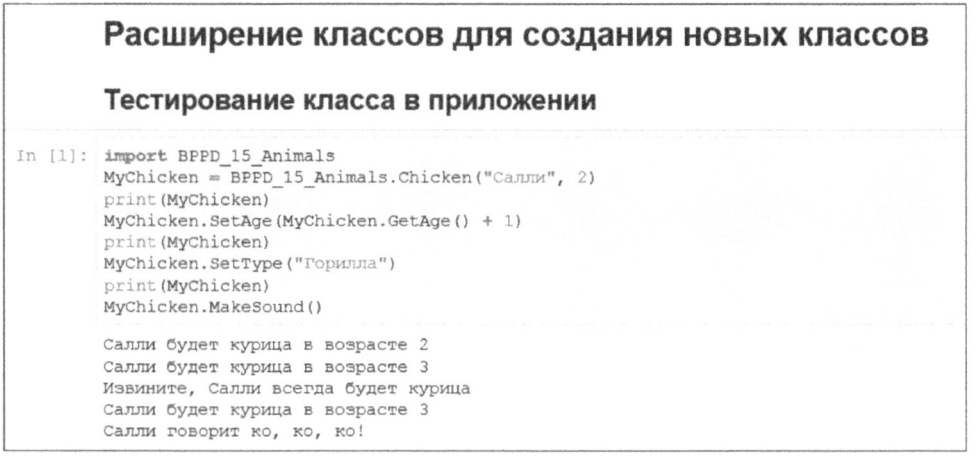


Рис. 15.15. Сегодня у Салли день рождения, и она хочет кое-что сказать нам

4 Решение сложных задач

В ЭТОЙ ЧАСТИ...

- Долговременное хранение данных на диске
- Создание, чтение, изменение и удаление данных в файлах
- Написание, отправка и просмотр сообщений электронной почты

Глава 16

Хранение данных в файлах

В ЭТОЙ ГЛАВЕ...

- Принципы работы долговременного хранилища
- Использование содержимого долговременного хранилища
- Создание, чтение, изменение и удаление данных в файлах

До сих пор мы разрабатывали приложения, которые в основном выводили информацию на экран. Но в действительности приложения больше ориентированы на работу с данными. Данные — это сердцевина любого приложения, поскольку именно они больше всего интересуют пользователя.

Будучи основой приложений, данные требуют самого тщательного подхода к организации долговременного хранения. Данные могут храниться на таких устройствах, как жесткие диски, SSD-накопители, USB-флешки и пр. (Популярное решение — облачные хранилища данных, но они в книге не рассматриваются, поскольку требуют иных подходов к программированию.) Данные, находящиеся в оперативной памяти, являются временными: они хранятся лишь до тех пор, пока работает компьютер. Долговременное хранилище позволяет запоминать данные даже после выключения компьютера и использовать их во время следующей сессии.

В этой главе рассматриваются вопросы, связанные не только с долговременным хранением, но и с основными операциями с файлами: создание, чтение, изменение и удаление (CRUD — Create, Read, Update, and Delete). Акроним CRUD известен многим раз-

работчикам. Независимо от способа хранения данных приложение должно поддерживать вышеперечисленные четыре операции, чтобы быть полезным для пользователя. Безусловно, методы CRUD должны быть надежными, безопасными и управляемыми. В этой главе будет показано, как осуществлять безопасную работу с файлами, чтобы гарантировать целостность данных. Исходный код примеров этой главы находится в файле `BPPD_16_Storing_Data_in_Files.ipynb` (см. введение).

Принципы работы долговременного хранилища

Для работы с долговременными хранилищами вовсе не обязательно изучать их детально. Например, вам не важно, как крутится жесткий диск (и крутится ли он вообще). Тем не менее необходимо понимать ряд общих правил, на которых основаны такие хранилища. Эти правила были сформулированы еще на заре компьютерной эры, в эпоху мейнфреймов.

Данные преимущественно хранятся в *файлах*. Думаю, каждый пользователь знает, что такое файлы, ведь практически любое приложение работает с ними. Например, открывая документ в текстовом редакторе, мы, по сути, открываем файл данных, содержащий введенные кем-то слова.

У файла обычно есть *расширение*, определяющее его тип. Расширение отделяется от имени файла точкой, например `MyData.txt`. Здесь `.txt` — это расширение имени файла, и в вашем компьютере наверняка есть приложение (и не одно) для открытия файлов такого типа.

Файлы структурируют хранимые данные определенным образом, чтобы упростить их чтение и запись. Любое создаваемое приложение должно понимать используемую структуру файлов, чтобы правильно взаимодействовать с данными, которые хранятся в файлах. Файловые структуры бывают довольно сложными, но в примерах этой главы используется простая файловая структура, позволяющая максимально упростить код доступа к данным.

Если бы все файлы хранились на жестком диске в полном беспорядке, то найти нужный файл было бы невозможно, поэтому файлы организованы в виде каталогов. (В современных компьютерных системах используется также термин *папка*.) Устройства долговременного хранения ориентируются на каталоги как на способ упорядочения файлов, упрощающий поиск отдельных файлов. Чтобы найти файл, необходимо знать, в каком каталоге он находится.

Каталоги имеют определенную иерархическую структуру, которая начинается на верхнем уровне жесткого диска. Например, файлы исходных кодов для этой книги находятся в каталоге `BPPD`. В моей системе Windows структура каталогов выглядит следующим образом: `C:\Users\John\BPPD`. В системах Mac и Linux тот же каталог `BPPD` будет включен в другую иерархию. В вашей системе, скорее всего, местоположение файлов кода тоже будет другим.

ЗАПОМНИ!

Заметьте, что я использовал обратную косую черту (\) для разделения каталогов. В некоторых системах аналогичной цели служит косая черта (/). В данной книге используется обратная косая черта, но вы должны проверить, какой вариант соответствует вашей системе.

В Python структура каталогов называется *путем*. Этот термин уже встречался вам ранее, поскольку Python должен находить любые ресурсы на основании предоставленного пути. Например, `C:\Users\John\BPPD` — это полный путь к файлам исходного кода данной главы в моей Windows-системе. Путь, охватывающий весь маршрут следования Python, называется *абсолютным*. Неполный путь к ресурсу, использующий в качестве отправной точки текущий каталог, называется *относительным*.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Для поиска файла с помощью относительного пути в качестве отправной точки обычно используется текущий каталог. Например, `BPPD\__pycache__` будет относительным путем, ведущим к кеш-памяти Python. Обратите внимание на отсутствие в начале пути буквы диска или обратной косой черты. Тем не менее иногда относительный путь начинается со специальных обозначений, которые перечислены ниже.

- `\`. Корневой каталог текущего диска. Диск является относительным, и путь начинается с корня этого диска.
- `.\`. Текущий каталог. Это сокращение используется для обозначения текущего каталога, если его имя неизвестно. Например, определить местонахождение кеш-памяти Python можно следующим образом: `.\__pycache__`.
- `..\`. Родительский каталог. Это сокращение используется, если имя родительского каталога неизвестно.
- `..\..\`. Родитель родительского каталога. При необходимости можно подниматься вверх по структуре каталогов для нахождения нужной стартовой точки, от которой потом спускаются по структуре каталогов вниз к новому местоположению.

Создание содержимого для долговременного хранения

Файл может содержать структурированные либо неструктурированные данные. Примером *структурированных данных* может служить база данных, в которой каждая запись несет в себе определенную информацию. База данных сотрудников компании содержит поля для имени, адреса, идентификатора исполнителя и т.п. Каждая запись относится к отдельному сотруднику, и каждая запись сотрудника содержит поля для имени, адреса и идентификатора. Примером *неструктурированных данных* служит файл текстового редактора, который может содержать любой текст в произвольном порядке. Для содержимого абзаца нет обязательного порядка, а предложения могут включать любое количество слов. Тем не менее в обоих случаях приложение должно знать, как выполнять операции CRUD с файлом. Это означает, что содержимое файла должно быть подготовлено таким образом, чтобы приложение могло осуществлять чтение из файла и запись в файл.

Даже в файлах текстового редактора хранение текста организовано в соответствии с определенными правилами. Предположим, что файлы — это простой текст. Каждый абзац должен иметь что-то вроде символа-разграничителя, определяющего начало нового абзаца. Приложение читает абзац до тех пор, пока не встретит разграничитель, после чего начинает чтение нового абзаца. Чем больше форматирования предлагает текстовый редактор, тем более структурированным получается результат.

ЗАПОМНИ!

Разметка, подготавливающая содержимое для записи в долговременное хранилище, зачастую невидима. Все, что мы видим в файле, — это сами данные. Разметка скрывается по ряду причин, перечисленных ниже.

- Маркер разметки — это управляющий символ, например символ возврата каретки или перевода строки, который обычно скрыт по умолчанию на платформенном уровне.
- Для разделения записей приложение использует специальные комбинации символов, например запятые и двойные кавычки. Эти специальные комбинации символов скрываются самим приложением при выводе текста.
- В процессе вывода некоторые символы преобразуются в другую форму, как, например, при форматировании содержимого файла в текстовом редакторе. На экране отображается результат

форматирования, а специальные символы, реализующие форматирование, скрываются.

- Файл фактически имеет альтернативный формат, например XML (eXtensible Markup Language — расширяемый язык разметки). Дополнительные сведения об XML доступны по следующему адресу:

  `http://www.w3schools.com/xml/default.ASP`

  Альтернативный формат интерпретируется и отображается на экране в понятном для пользователя виде.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Могут существовать и другие правила форматирования данных. Например, компания Microsoft в действительности использует ZIP-файл для хранения файлов последней версии текстового редактора (таких, как `.docx`). Используя сжатую папку, можно сохранить больший объем информации в хранилище того же размера. Всегда интересно изучать, как другие разработчики управляют своими хранилищами, так как это позволяет находить более эффективные и безопасные способы хранения данных в своих собственных приложениях.

Теперь, когда вы представляете себе общие принципы подготовки данных к хранению на диске, настало время рассмотреть пример. В данном случае стратегия форматирования довольно проста. Программа получает входные данные, форматирует их для последующего сохранения и выводит отформатированную версию на экран (не сохраняя их пока что).

1. **Откройте новый блокнот.**

   Можете использовать файлы исходного кода `BPPD_16_Storing_Data_in_Files.ipynb` (содержит код приложения) и `BPPD_16_FormattedData.ipynb` (содержит код класса `FormatData`).

2. **Введите следующий код в блокноте.**

```
class FormatData:
   def __init__(self, Name="", Age=0, Married=False):
      self.Name = Name
      self.Age = Age
      self.Married = Married

   def __str__(self):
      OutString = "'{0}', {1}, {2}".format(
         self.Name,
         self.Age,
         self.Married)
      return OutString
```

Это компактный вариант класса. Обычно в него еще добавляются методы доступа (геттеры и сеттеры) и код перехвата ошибок. (Вспомните, что *метод-геттер* обеспечивает чтение данных класса, а *метод-сеттер* — их запись.) Но для целей демонстрации этого вполне достаточно.

Главный элемент, на который следует обратить внимание, — это метод `__str__()`, определенным образом форматирующий выходные данные. Строка `self.Name` заключена в одинарные кавычки, а все значения разделены запятыми. По сути, это стандартный формат CSV (Comma-Separated Value — значения, разделенные запятыми), который широко применяется на всех платформах, поскольку легко преобразуется в простой текст и не требует сложной обработки.

3. **Сохраните файл под именем `BPPD_16_FormattedData.ipynb`.**

Чтобы класс можно было использовать в последующих примерах, необходимо сохранить его на диске, как описывалось в главе 15. Необходимо также создать файл `BPPD_16_FormattedData.py`, что позволит импортировать класс в приложение.

4. **Откройте еще один блокнот.**

5. **Введите следующий код в блокноте.**

```
from BPPD_16_FormattedData import FormatData
NewData = [FormatData("Джордж", 65, True),
           FormatData("Салли", 47, False),
           FormatData("Дуг", 52, True)]
for Entry in NewData:
   print(Entry)
```

Программа начинается с импорта класса `FormatData` из пакета `BPPD_16_FormattedData`. Этого можно было и не делать, поскольку на данный момент пакет `BPPD_16_FormattedData` содержит лишь один класс, но в дальнейшем мы планируем расширить его, так что о подобном приеме следует помнить.

Чаще всего при сохранении данных на диск мы имеем дело с множеством записей. В данном случае создается список записей, которые помещаются в объект `NewData`. Этот объект, по сути, и есть наш документ. В реальном приложении данные будут представляться по-другому, но идея в целом та же самая.

Любое приложение, сохраняющее данные, проходит через так называемый цикл вывода. В нашем цикле данные просто выводятся на экран. В последующих примерах данные будут записываться в файл.

6. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Результаты показаны на рис. 16.1. В данном случае все записи разделены комбинацией символов возврата каретки и перевода строки, т.е. записи `Джордж`, `Салли` и `Дуг` — это отдельные строки в файле. Все *поля* (элементы данных) разделены запятыми. Чтобы отличать текстовые поля от других типов данных, они взяты в кавычки.

```
In [1]: from BPPD_16_FormattedData import FormatData
        NewData = [FormatData("Джордж", 65, True),
                   FormatData("Салли", 47, False),
                   FormatData("Дуг", 52, True)]
        for Entry in NewData:
           print(Entry)

        'Джордж', 65, True
        'Салли', 47, False
        'Дуг', 52, True
```

Рис. 16.1. Данные в формате CSV

Создание файла

Любые данные, с которыми пользователь хочет работать в более чем одном сеансе, должны сохраняться на каком-то устройстве, предназначенном для постоянного хранения. Создание файла и сохранение в нем данных — это ключевые операции в Python. В следующей пошаговой инструкции показано, как создать код, который будет записывать данные на жесткий диск.

1. **Откройте файл исходного кода `BPPD_16_FormattedData.ipynb`.**

   На экране отобразится код, созданный в предыдущем разделе. Мы добавим в пакет новый класс, позволяющий сохранить файл на диске.

2. **Введите следующий код в блокноте.**

```
import csv

class FormatData2:
   def __init__(self, Name="", Age=0, Married=False):
      self.Name = Name
      self.Age = Age
      self.Married = Married

   def __str__(self):
      OutString = "'{0}', {1}, {2}".format(
         self.Name,
         self.Age,
         self.Married)
      return OutString

   def SaveData(Filename = "", DataList = []):
      with open(Filename, "w", newline='\n') as csvfile:
         DataWriter = csv.writer(
            csvfile,
            delimiter='\n',
            quotechar=" ",
            quoting=csv.QUOTE_NONNUMERIC)
```

```
DataWriter.writerow(DataList)
csvfile.close()
print("Данные сохранены!")
```

Модуль `csv` содержит все необходимое для работы с файлами CSV.

ЗАПОМНИ!

Python поддерживает огромное количество типов файлов, и существует множество сторонних библиотек, предлагающих дополнительную поддержку. Если вам нужно работать с каким-то типом файла, который не поддерживается стандартными средствами Python, вы всегда найдете для этого стороннюю библиотеку. К сожалению, не существует универсального списка поддерживаемых в Python типов файлов, поэтому информацию о том, как в Python реализована поддержка необходимого вам типа файла, придется искать в Интернете. Например, описание всех архивных форматов можно найти по следующей ссылке:

`https://docs.python.org/3/library/archiving.html`

Описание прочих форматов файлов доступно по следующему адресу:

`https://docs.python.org/3/library/fileformats.html`

СОВЕТ

В данном примере используется такой же код форматирования текста, как и в классе `FormatData`, только теперь добавлен метод `SaveData()` для сохранения отформатированных данных на диске. Использование нового класса сигнализирует о новых возможностях. По сути, класс `FormatData2` является расширенной версией класса `FormatData`.

Метод `SaveData()` получает два аргумента: имя файла, используемого для хранения данных, и список элементов для записи в файл. Это метод класса, а не метод экземпляра. Далее вы увидите, в чем преимущество такого подхода. Аргумент `DataList` по умолчанию содержит пустой список, чтобы при отсутствии входного значения программа не генерировала исключение. Вместо этого будет просто создан пустой выходной файл. При желании можно добавить код для обнаружения пустого списка, чтобы сообщать пользователю об ошибке.

ЗАПОМНИ!

Инструкция `with` дает Python указание выполнить ряд действий с конкретным ресурсом — открытым файлом `csvfile` (его именем будет `TestFile.csv`). Аргументы функции `open()` зависят от способа работы с файлом. В этом примере файл открывается в режиме записи (обозначается аргументом `"w"`). Аргумент `newline` означает, что Python должен воспринимать управляющий символ `\n` (перевод строки) как символ новой строки.

Для записи выводимых данных нужно создать объект записывающего устройства. Объект `DataWriter` конфигурируется таким образом, чтобы `csvfile` стал файлом вывода, в качестве разделителя записей использовался символ `\n`, поля записей разделялись пробелом, а в кавычки брались только нечисловые значения. Результаты применения этих настроек

будут приведены позже, а пока предположим, что это именно те результаты, которые нам нужны.

Собственно запись данных осуществляется проще, чем вы думаете. Все, что нам нужно, — это вызвать метод `DataWriter.writerow()`, передав ему список `DataList` в качестве аргумента. По окончании работы с файлом рекомендуется его закрывать. При закрытии файла данные сбрасываются на диск (выполняется подтверждение записи данных). В конце выводится сообщение о том, что данные сохранены.

3. **Сохраните файл под именем `BPPD_16_FormattedData.ipynb`.**

Чтобы созданный класс можно было использовать в последующих примерах, сохраните его на диске, как описывалось в главе 15. Необходимо также заново создать файл `BPPD_16_FormattedData.py`, с помощью которого класс будет импортироваться в приложение. Если не сделать этого, клиентский код не сможет импортировать класс `FormatData2`. Только убедитесь в том, что прежняя версия файла `BPPD_16_FormattedData.py` удалена из репозитория кода, прежде чем импортировать новую версию (можно также попросить Notebook перезаписать старый экземпляр).

4. **Введите следующий код в блокноте.**

```
from BPPD_16_FormattedData import FormatData2
NewData = [FormatData2("Джордж", 65, True),
           FormatData2("Салли", 47, False),
           FormatData2("Дуг", 52, True)]
FormatData2.SaveData("TestFile.csv", NewData)
```

Это напоминает пример, созданный в предыдущем разделе. Мы снова создаем объект `NewData` в виде списка. Только теперь информация не отображается на экране, а записывается в файл с помощью вызова `FormatData2.SaveData()`. Это одна из тех ситуаций, когда использование метода экземпляра будет неуместно. В случае метода экземпляра нам пришлось бы сначала создать экземпляр `FormatData2`, который не представляет никакого интереса для пользователя.

5. **Перезапустите ядро, выполнив команду Kernel⇨Restart (Ядро⇨Перезапуск) или щелкнув на кнопке Restart the Kernel (Перезапуск ядра).**

Этот шаг необходим для выгрузки предыдущей версии пакета `BPPD_16_FormattedData`. В противном случае пример не будет выполняться, даже несмотря на появление новой версии файла `BPPD_16_FormattedData.py` в каталоге кода.

6. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

После завершения работы приложения появится сообщение об успешном сохранении данных. Впрочем, это ничего не говорит нам о самих данных. В папке исходного кода появится новый файл с именем `TestFile.csv`. На каждой платформе есть приложения, которые по умолчанию открывают такие файлы.

В Windows можно воспользоваться программами Excel и WordPad. На рис. 16.2 показан результат открытия этого файла в Excel, а на рис. 16.3 — в WordPad. В обоих случаях результат совпадает с тем, который был показан на рис. 16.1.

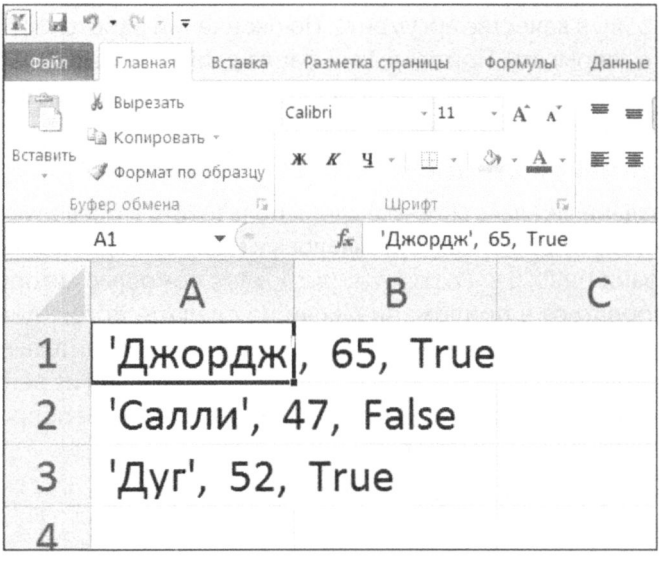


Рис. 16.2. Отображение данных в Excel

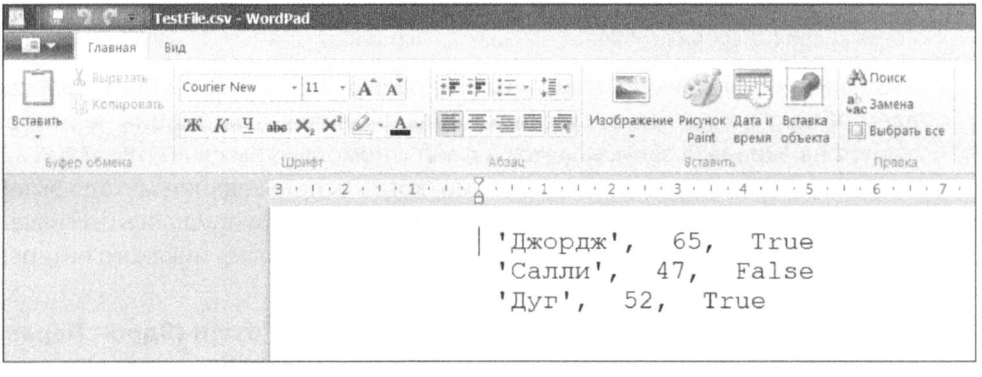


Рис. 16.3. Отображение данных в WordPad

Чтение содержимого файла

На данном этапе информация содержится на жестком диске. Там она в безопасности, но пока бесполезна для пользователя, поскольку скрыта. Чтобы отобразить данные, нужно сначала загрузить их в память. Ниже демонстрируется, как прочитать данные с жесткого диска и отобразить их на экране.

1. **Откройте ранее сохраненный файл `BPPD_16_FormattedData.ipynb`.**

   На экране отобразится код, созданный в разделе "Создание содержимого для долговременного хранения". В этом примере мы добавим в пакет новый класс, позволяющий прочитать файл с диска.

2. **Введите следующий код в блокноте.**

```
import csv

class FormatData3:
   def __init__(self, Name="", Age=0, Married=False):
      self.Name = Name
      self.Age = Age
      self.Married = Married

   def __str__(self):
      OutString = "'{0}', {1}, {2}".format(
         self.Name,
         self.Age,
         self.Married)
      return OutString

   def SaveData(Filename = "", DataList = []):
      with open(Filename, "w", newline='\n') as csvfile:
         DataWriter = csv.writer(
            csvfile,
            delimiter='\n',
            quotechar=" ",
            quoting=csv.QUOTE_NONNUMERIC)
         DataWriter.writerow(DataList)
         csvfile.close()
         print("Данные сохранены!")

   def ReadData(Filename = ""):
      with open(Filename, "r", newline='\n') as csvfile:
         DataReader = csv.reader(
            csvfile,
            delimiter="\n",
            quotechar=" ",
            quoting=csv.QUOTE_NONNUMERIC)
         Output = []
         for Item in DataReader:
            Output.append(Item[0])
         csvfile.close()
         print("Данные прочитаны!")
         return Output
```

Файл открывается для чтения той же функцией `open()`, что и для записи. Разница лишь в том, что вместо аргумента `"w"` ("write" — запись) указывается

аргумент "r" ("read" — чтение). Остальные аргументы одинаковы и действуют так же.

ЗАПОМНИ!

Важно помнить, что CSV-файл содержит простой текст с разделителями. При загрузке данных в память необходимо подготовить соответствующую структуру Python. В этом примере мы начинаем с пустого списка Output.

В файле содержатся три записи, разделенные управляющим символом \n. Все записи считываются в цикле for. Обратите внимание на необычное использование выражения Item[0]. В процессе чтения записей Python трактует нефинальные записи (те, которые не являются последними в файле) как два элемента списка. Первый элемент содержит данные, а второй пуст. Нам нужен только первый элемент. Все данные добавляются в переменную Output, в результате чего формируется полный список записей, содержащихся в файле.

Как уже упоминалось, по окончании работы с файлом следует закрыть его. Метод выводит сообщение об успешном получении данных и возвращает список Output вызывающему коду.

3. **Сохраните файл под именем BPPD_16_FormattedData.ipynb.**

Чтобы созданный класс можно было использовать в последующих примерах, сохраните его на диске, как описывалось в главе 15. Необходимо также заново создать файл BPPD_16_FormattedData.py, с помощью которого класс будет импортироваться в приложение. Если не сделать этого, клиентский код не сможет импортировать класс FormatData3. Только убедитесь в том, что прежняя версия файла BPPD_16_FormattedData.py удалена из репозитория кода, прежде чем импортировать новую версию.

4. **Введите следующий код в блокноте.**

```
from BPPD_16_FormattedData import FormatData3
NewData = FormatData3.ReadData("TestFile.csv")
for Entry in NewData:
   print(Entry)
```

Программа начинает работу с импорта класса FormatData3. Затем создается список NewData, возвращаемый методом FormatData3.ReadData(). Как и в предыдущем примере, это метод класса, что позволяет сократить и упростить код. Далее выполняется цикл for для отображения содержимого списка NewData.

5. **Перезапустите ядро и щелкните на кнопке Run Cell (Выполнить ячейку).**

Результат выполнения программы можно увидеть на рис. 16.4. Он аналогичен результату, показанному на рис. 16.1, при том что данные сначала были записаны на диск, а затем прочитаны обратно. Именно так и должны работать приложения, осуществляющие чтение и запись данных.

После считывания данные должны быть получены в таком же виде, в каком они были на момент их записи на диск. В противном случае приложение работает некорректно, внося неконтролируемые изменения в данные.

Чтение содержимого файла

```
In [9]: from BPPD_16_FormattedData import FormatData3
        NewData = FormatData3.ReadData("TestFile.csv")
        for Entry in NewData:
            print(Entry)

        Данные прочитаны!
        'Джордж', 65, True
        'Салли', 47, False
        'Дуг', 52, True
```

Рис. 16.4. Данные приложения после обработки

Изменение содержимого файла

Некоторые разработчики считают, что изменение файла — это сложная задача. Она может стать таковой, если рассматривать ее как единую операцию. Но в действительности это три последовательные операции:

1) считывание содержимого файла в память;
2) изменение представления данных в памяти;
3) запись полученного содержимого в долговременное хранилище.

Чаще всего второй этап дробится на несколько шагов. В процессе модификации данных приложение может выполнять некоторые или все из нижеперечисленных действий:

- отображение данных на экране;
- добавление данных в список;
- удаление данных из списка;
- внесение изменений в существующие данные, что может быть реализовано путем добавления новой записи с измененными данными и удаления старой записи.

До сего момента мы сталкивались почти со всеми операциями, перечисленными в этих списках, кроме одной. Мы осуществляли чтение и запись файла, добавляли данные в список и отображали их на экране. Единственное, что мы еще не делали, — не удаляли данные из списка. Изменение данных обычно

реализуется в два этапа: сначала на основе старой записи создается новая запись, а после ее добавления в список старая запись удаляется.

Не стоит полагать, будто все эти действия выполняются в каждом приложении. К примеру, программа-регистратор никогда не удаляет и не изменяет имеющиеся записи, а только создает новые записи. Приложение электронной почты позволяет создавать новые письма и удалять старые, но не изменять полученные или отправленные. С другой стороны, в текстовом редакторе реализуются все вышеупомянутые операции. Все это зависит исключительно от типа создаваемого приложения.

Важно отделить пользовательский интерфейс от тех действий, которые к нему не относятся. В следующем примере наглядно показано, что именно нужно отделить от пользовательского интерфейса, чтобы внести изменения в файл, созданный в разделе "Создание файла". Вы увидите, каким образом выполняются операции чтения, модификации и записи файла в процессе его обновления. Операция модификации предусматривает добавление, удаление и изменение данных. Чтобы пользователь мог выполнять приложение неоднократно, обновления записываются в другой файл.

1. **Введите следующий код в блокноте.**

```
from BPPD_16_FormattedData import FormatData3
import os.path

if not os.path.isfile("TestFile.csv"):
   print("Выполните пример, в котором создается файл TestFile.csv!")
   quit()

NewData = FormatData3.ReadData("TestFile.csv")
for Entry in NewData:
   print(Entry)

print("\r\nДобавление записи для Гарри.")
NewRecord = "'Гарри', 23, False"
NewData.append(NewRecord)
for Entry in NewData:
   print(Entry)

print("\r\nУдаление записи Дуга.")
Location = NewData.index("'Дуг', 52, True")
Record = NewData[Location]
NewData.remove(Record)
for Entry in NewData:
   print(Entry)
```

```
print("\r\nИзменение записи Салли.")
Location = NewData.index("'Салли', 47, False")
Record = NewData[Location]
Split = Record.split(",")
NewRecord = FormatData3(Split[0].replace("'", ""),
                        int(Split[1]),
                        bool(Split[2]))
NewRecord.Married = True
NewRecord.Age = 48
NewData.append(NewRecord.__str__())
NewData.remove(Record)
for Entry in NewData:
   print(Entry)

FormatData3.SaveData("ChangedFile.csv", NewData)
```

Пример содержит достаточно много кода. Сначала проверяется наличие файла `TestFile.csv`. Такого рода проверки всегда нужно выполнять при работе с существующими файлами. В данном случае мы не создаем новый файл, а лишь обновляем существующий. При отсутствии файла приложение завершает работу.

Далее мы загружаем данные в список `NewData`. Этот фрагмент аналогичен тому, что мы уже рассматривали ранее.

ЗАПОМНИ!

В главе 13 мы познакомились с функциями для работы со списками. В этом примере мы вновь прибегнем к ним. Функция `append()` добавляет новую запись в список `NewData`. Только учтите, что данные добавляются в виде строки, а не объекта `FormatData3`. На диске данные хранятся в виде строк и в таком же виде считываются обратно. Можно либо добавить новые данные в виде строки, либо создать объект `FormatData3` и затем использовать метод `__str__()`.

Следующий шаг заключается в удалении записи из списка `NewData`. Для этого необходимо сначала найти нужную запись. Подобное действие можно выполнить и вручную, ведь у нас всего четыре записи (напомним, что список `NewData` теперь включает запись для Гарри). Но когда записей много, поиск записей лучше выполнять с помощью функции `index()`. Она возвращает число, соответствующее позиции записи в списке. Это число мы используем для выборки самой записи. Полученную запись можно удалить с помощью функции `remove()`.

Код, связанный с изменением записи Салли, кажется пугающим, но в основном он посвящен обработке строк, которые хранятся на диске. Извлекаемая из списка `NewData` запись представляет собой единую строку, содержащую все три значения. Функция `split()` возвращает список, содержащий три строковых элемента, но этого недостаточно. К тому же имя Салли заключено как в двойные, так и в одинарные кавычки.

Самый простой способ управления записью — создание объекта `FormatData3` и занесение в него содержимого строкового списка. Это означает удаление из имени лишних кавычек, преобразование второго значения в тип `int` и преобразование третьего значения в тип `bool`. Класс `FormatData3` не содержит методов доступа, поэтому мы изменяем поля `Married` и `Age` напрямую. Правильнее было бы воспользоваться методами-сеттерами.

Далее приложение присоединяет новую запись к списку `NewData` и удаляет из него существующую запись. Обратите внимание на использование метода `NewRecord.__str__()` для преобразования объекта `FormatData3` в строку.

Последнее действие — сохранение измененного файла. Обычно данные сохраняются в том же файле, но в данном случае мы записываем полученный результат в другой файл, что позволяет просмотреть как старые, так и новые данные.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

На рис. 16.5 показаны результаты работы программы. После каждого этапа программа выводит список записей, что позволяет следить за изменением списка `NewData`. Это полезная методика отладки. Разумеется, код, выводящий информацию на экран, следует удалить перед созданием финального релиза.

```
Данные прочитаны!
'Джордж', 65, True
'Салли', 47, False
'Дуг', 52, True

Добавление записи для Гарри.
'Джордж', 65, True
'Салли', 47, False
'Дуг', 52, True
'Гарри', 23, False

Удаление записи Дуга.
'Джордж', 65, True
'Салли', 47, False
'Гарри', 23, False

Изменение записи Салли.
'Джордж', 65, True
'Гарри', 23, False
'Салли', 48, True
Данные сохранены!
```

Рис. 16.5. Программа пошагово отображает все изменения

3. **Откройте файл `ChangedFile.csv` в соответствующем редакторе.**

Содержимое файла, открытое в программе WordPad, показано на рис. 16.6.

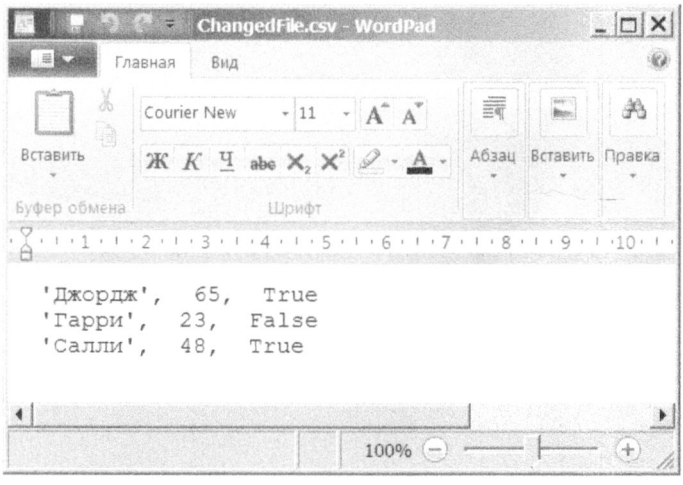


Рис. 16.6. Содержимое файла `ChangedFile.csv`

Удаление файла

В предыдущем разделе демонстрировалось, как добавлять, удалять и изменять записи в файле. Ниже будет показано, как удалить ненужный файл.

1. **Введите следующий код в блокноте.**

```
import os
os.remove("ChangedFile.csv")
print("Файл удален!")
```

ВНИМАНИЕ!

Все довольно просто. Для удаления файла вызовите функцию `os.remove()`, указав имя файла и путь к нему (если файл находится в текущем каталоге, то путь можно не указывать). Есть еще несколько функций, связанных с удалением файлов и каталогов.

- `os.rmdir()`. Удаляет заданный каталог. Каталог должен быть пустым, иначе Python выдаст сообщение об исключении.
- `shutil.rmtree()`. Удаляет заданный каталог, включая все подкаталоги и все файлы. Эта функция особенно опасна, поскольку удаляет все подряд без проверки (Python предполагает, что пользователь отдает отчет в своих действиях). Проявляйте особую осторожность, чтобы не потерять важные данные.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Появится сообщение `Файл удален!`. Заглянув в каталог, в котором изначально находился файл `ChangedFile.csv`, вы увидите, что этого файла там уже нет.

Глава 17

Отправка сообщений электронной почты

В ЭТОЙ ГЛАВЕ...

- Последовательность событий при отправке электронной почты
- Создание приложения электронной почты
- Тестирование приложения электронной почты

В этой главе рассматривается процесс рассылки электронных писем с использованием Python. Вы сможете понять, что происходит в процессе дистанционного общения. Несмотря на то что глава посвящена электронной почте, в ней излагается универсальный материал, который можно применять для решения других задач. Например, при работе с внешними службами часто требуется создавать пакеты того же типа, что и для электронной почты. Таким образом, информация, изложенная в этой главе, поможет вам создавать самые разные коммуникационные приложения.

Электронная почта была фактически смоделирована на основе реальной почты. Первоначально термин "электронная почта" использовался для любого вида передачи электронных документов, и в некоторых приложениях отправитель и получатель должны были находиться в сети одновременно. Это привело к тому, что в Интернете появилась масса невразумительной информации о происхождении и развитии электронной почты. В данной главе электронная почта рассматривается в том виде, в каком она существует сегодня, — как механизм обмена документами различных типов.

Примеры этой главы рассчитаны на использование сервера SMTP (Simple Mail Transfer Protocol — простой протокол передачи почты). Если этот термин вам незнаком, обратитесь к приведенной ниже врезке. Исходный код примеров этой главы находится в файле `BPPD_17_Sending_an_Email.ipynb` (см. введение).

SMTP (ПРОСТОЙ ПРОТОКОЛ ПЕРЕДАЧИ ПОЧТЫ)

Имея дело с электронной почтой, вы будете постоянно встречать упоминание SMTP. Конечно, этот термин звучит довольно загадочно, но разбираться в технических деталях вовсе не обязательно, достаточно лишь знать, для чего применяется данный протокол. С другой стороны, полезно немного разобраться в принципах работы SMTP-сервера, чтобы не воспринимать его на уровне "черного ящика", который принимает письмо от отправителя и пересылает получателю. Рассмотрим отдельные компоненты, образующие аббревиатуру SMTP, начав с конца.

- **Protocol (Протокол).** Это стандартный набор коммуникационных правил. Работа с электронной почтой требует соблюдения общепринятых правил, в противном случае почтовая система станет ненадежной.
- **Mail Transfer (Передача почты).** Электронные документы пересылаются из одного места в другое почти так же, как письма между почтовыми отделениями. В случае электронной почты процесс передачи основан на коротких командах, которые ваше почтовое приложение выдает SMTP-серверу. Например, команда `MAIL FROM` сообщает серверу имя отправителя электронной почты, а команда `RCPT TO` указывает, куда ее отправлять.
- **Simple (Простой).** Означает, что доставки почтовых отправлений выполняется с наименьшим количеством возможных усилий. Это позволяет повысить надежность процесса в целом.

Если вы изучите правила передачи информации, то обнаружите, что они вовсе не такие простые. Например, RFC1123 — это стандарт, который определяет, как должны работать интернет-хосты (`http://www.faqs.org/rfcs/rfc1123.html`). Подобные правила лежат в основе многих интернет-технологий, что объясняет, почему большинство из них работает примерно одинаково (несмотря на внешние различия).

Другой стандарт, RFC2821, описывает, каким образом в SMTP реализуются правила, описанные в RFC1123 (`http://www.faqs.org/rfcs/rfc2821.html`). Иными словами, существует множество всевозможных правил, написанных на языке, который понятен лишь узкому кругу специалистов (и то не всегда). Если хотите получить более внятное объяснение принципов работы электронной почты, обратитесь по адресу `https://ru.bmstu.wiki/SMTP_(Simple_Mail_Transfer_Protocol)`.

Что происходит при отправке электронной почты

Электронная почта стала настолько надежной и обыденной, что большинство людей даже не представляют, какое это чудо. В сущности, то же самое можно сказать и о реальной почтовой службе. Если задуматься, то вероятность того, что одно конкретное письмо будет доставлено именно туда, куда нужно, кажется мизерной. Но на самом деле это не так. И электронная почта, и ее физический эквивалент имеют несколько общих особенностей, которые резко повышают их надежность. В следующих разделах объясняется, что происходит, когда вы пишете электронное письмо, щелкаете на кнопке Отправить и письмо доставляется получателю. Вы удивитесь тому, сколько там всего интересного.

Структура электронного письма

Лучше всего рассматривать электронную почту как ее физический аналог. Когда вы пишете письмо, вам нужны как минимум две бумажки: первая — само письмо, вторая — конверт. Если почтовая служба работает честно, то содержимое письма никогда не просматривается никем, кроме получателя. То же самое можно сказать и об электронной почте. Она содержит следующие компоненты.

- **Сообщение**. Содержимое электронного письма, которое фактически состоит из двух частей.
  - *Заголовок*. Включает тему, список получателей и другие атрибуты, такие как срочность сообщения.
  - *Тело*. Фактическое сообщение. Может быть представлено в формате простого текста или HTML и содержать вложенные документы.
- **Конверт**. Контейнер для сообщения. Содержит информацию об отправителе и получателе так же, как и конверт, пересылаемый обычной почтой. Только на него не клеится почтовая марка.

При работе с электронной почтой вы создаете сообщение с помощью почтового приложения. В процессе конфигурирования приложения вы также задаете информацию о своей учетной записи. После щелчка на кнопке Отправить выполняются следующие действия.

1. Приложение электронной почты упаковывает ваше сообщение, содержащее заголовок, в конверт, который включает в себя информацию как об отправителе, так и о получателе.

2. Приложение электронной почты использует информацию учетной записи, чтобы связаться с SMTP-сервером и отправить сообщение получателю.
3. SMTP-сервер считывает только ту информацию, которая найдена в конверте сообщения, и перенаправляет вашу электронную почту получателю.
4. Приложение электронной почты получателя подключается к локальному серверу, загружает электронную почту и отображает сообщение пользователю.

В действительности все происходит немного сложнее, чем здесь описано, но вам главное понять суть. Это примерно то же самое, что происходит с физическими письмами. В случае обычной почты приложение электронной почты заменяется отправителем письма на одном конце и получателем письма — на другом. SMTP-сервер заменяется почтовым отделением и работающими в нем сотрудниками (включая почтальонов). В обоих случаях кто-то пишет письмо, кто-то его доставляет, а кто-то — получает.

Компоненты конверта электронного сообщения

Есть разница в том, как сконфигурирован конверт для электронной почты и как он фактически обрабатывается. Когда вы просматриваете конверт электронного письма, он выглядит подобно реальному письму в том смысле, что он содержит адрес отправителя и адрес получателя, т.е. те же самые компоненты, что и физический конверт. В реальном письме указываются имя отправителя, его почтовый адрес, город, область и почтовый индекс, и то же самое — для получателя. Эти элементы определяют адрес, по которому почтальон должен доставить письмо или вернуть письмо, если оно не может быть доставлено.

SMTP-сервер, обрабатывая конверт электронной почты, должен проанализировать адреса, и здесь аналогия с физическим конвертом начинает немного нарушаться. Адрес электронной почты содержит совсем не ту информацию, которую мы привыкли видеть в физическом адресе. Ниже описано содержимое электронного адреса.

- **Хост.** Подобен городу и области, указанным на почтовом конверте. Адрес хоста — это адрес, используемый сетевой картой, которая физически подключена к Интернету и обрабатывает весь сетевой трафик для этого конкретного компьютера. Пользователь компьютера может взаимодействовать с различными интернет-ресурсами, но адрес хоста во всех случаях будет одинаков.
- **Порт.** Напоминает адрес квартиры, указанный на почтовом конверте. Он определяет, какая часть почтовой системы должна получить сообщение. Например, SMTP-сервер, обрабатывающий исходящую почту, обычно использует порт 25. В то же время сервер POP3, обрабатывающий входящие сообщения, использует порт 110. Браузер,

как правило, взаимодействует с веб-сайтами через порт 80. Однако для защищенных веб-сайтов (в адресе которых указан протокол `https`, а не `http`) используется порт 443. Список типичных портов можно просмотреть по следующему адресу:

`https://ru.wikipedia.org/wiki/Список_портов_TCP_и_UDP`

» **Имя локального хоста.** Это комбинация хоста и порта. Например, имя веб-сайта `http://www.myplace.com` разрешается в адрес `55.225.163.40:80` (где первые четыре числа — это адрес хоста, а номер после двоеточия — порт). Python заботится об этих деталях в фоновом режиме, поэтому вы о них не будете знать.

Теперь, когда вы получили представление о том, как формируется адрес, рассмотрим его подробнее.

Хост

Адрес хоста — это идентификатор подключения к серверу. Точно так же, как адрес на конверте не является фактическим местоположением, так и адрес хоста не является фактическим сервером. Он просто указывает на местоположение сервера.

ЗАПОМНИ!

Объект подключения, соответствующий комбинации адреса хоста и порта, называется *сокетом*. С помощью сокета можно получить всю информацию, полезную для понимания работы электронной почты. В следующей пошаговой инструкции показано, как узнать используемые имена и адреса хостов. Благодаря этому вы начнете понимать всю идею конверта электронной почты и адресов, которые он содержит.

1. **Откройте новый блокнот.**

   Можете также воспользоваться файлом исходного кода `BPPD_17_Sending_an_Email.ipynb`, содержащим код приложения (см. введение).

2. **Введите `import socket`.**

   Прежде чем начать работать с сокетами, следует импортировать библиотеку сокетов. Она содержит множество интересных функций, которые помогут нам увидеть, как работают интернет-адреса.

3. **Введите `print(socket.gethostbyname("localhost"))`.**

   На экране отобразится адрес хоста. В данном случае вы должны увидеть `127.0.0.1`, поскольку `localhost` — это стандартное имя локального хоста, которому всегда соответствует адрес `127.0.0.1`.

4. **Введите `print(socket.gethostbyaddr("127.0.0.1"))` и щелкните на кнопке Run Cell (Выполнить ячейку).**

Приготовьтесь к сюрпризу. Результатом работы функции будет кортеж (рис. 17.1). Вместо `localhost` в качестве имени хоста вы увидите имя своего компьютера. Имя `localhost` — это универсальное обозначение локального компьютера, но если вы указываете адрес, то получаете имя компьютера. Учтите, что имя вашего компьютера будет отличаться от имени, показанного на рис. 17.1.

Хост

```
In [1]: import socket
        print(socket.gethostbyname("localhost"))
        print(socket.gethostbyaddr("127.0.0.1"))

        127.0.0.1
        ('Main', [], ['127.0.0.1'])

In [2]: print(socket.gethostbyname("www.johnmuellerbooks.com"))

        166.62.109.105
```

Рис. 17.1. Адрес локального хоста соответствует вашему компьютеру

5. **Введите `print(socket.gethostbyname("www.johnmuellerbooks.com"))` и щелкните на кнопке Run Cell.**

   На экране появится результат, показанный на рис. 17.2. Это адрес моего сайта. Он остается одинаковым, где бы вы ни находились, точно так же, как и адреса почтовых отправлений. Почтовая служба использует адреса, которые являются уникальными во всем мире. То же самое касается и Интернета.

```
In [2]: print(socket.gethostbyname("www.johnmuellerbooks.com"))

        166.62.109.105
```

Рис. 17.2. Адреса, используемые для отправки электронных сообщений, уникальны в Интернете

Порт

Порт — это точка входа на сервер. Адрес хоста указывает местоположение, а порт определяет, как именно осуществляется доступ к серверу. Даже если вы не указываете порт при обращении к адресу хоста, подставляется стандартный порт. Доступ всегда предоставляется с использованием комбинации адреса хоста и порта. В следующей пошаговой инструкции показано, как работать с портами для получения доступа к серверу.

1. **Введите `import socket`.**

   Помните о том, что сокет предоставляет адрес хоста и информацию о порте. Именно с помощью сокета создается объект подключения, содержащий оба элемента.

2. **Введите `socket.getaddrinfo("localhost", 110)` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Первое значение — это имя хоста, о котором вы хотите получить информацию. Второе значение — порт на указанном хосте. В данном случае вы запрашиваете информацию о порте 110 локального хоста.

   Результат показан на рис. 17.3. Выходные данные состоят из двух кортежей: один — для протокола IP версии 6 (IPv6), второй — для протокола IP версии 4 (IPv4). Каждый из этих кортежей содержит пять записей, о четырех из которых вам действительно не нужно беспокоиться, потому что они вам, вероятно, никогда не понадобятся. А вот самая последняя запись, `('127.0.0.1', 110)`, содержит адрес и порт локального хоста.

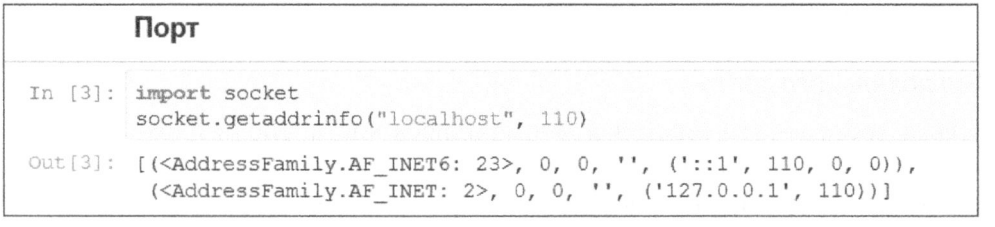


Рис. 17.3. Хост `localhost` поддерживает адреса IPv6 и IPv4

3. **Введите `socket.getaddrinfo("johnmuellerbooks.com", 80)`.**

   Результат показан на рис. 17.4. В данном случае предоставляется только адрес IPv4, которому соответствует порт 80. Метод `socket.getaddrinfo()` позволяет определить, можно ли получить доступ к определенному интернет-ресурсу. Протокол IPv6 обеспечивает значительные преимущества по сравнению с IPv4, но большинство веб-сайтов в настоящее время реализуют только поддержку IPv4.

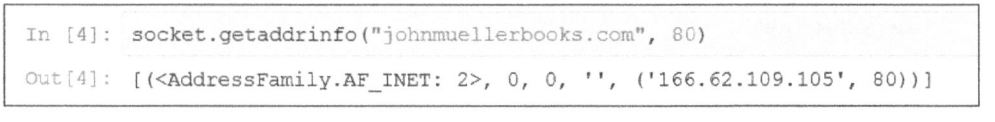


Рис. 17.4. Большинство веб-сайтов поддерживает только адреса IPv4

4. **Введите `socket.getservbyport(25)`.**

   Результат показан на рис. 17.5. Метод `socket.getservbyport()` позволяет определить, как используется заданный порт. Порт 25 всегда зарезервирован для поддержки SMTP. Поэтому, обратившись по адресу `127.0.0.1:25`, вы

запрашиваете доступ к SMTP-серверу на локальном хосте. Другими словами, порт обеспечивает доступ к конкретной службе.

```
In [5]: socket.getservbyport(25)

Out[5]: 'smtp'
```

Рис. 17.5. Стандартные порты служат для доступа к конкретным службам на каждом сервере

ЗАПОМНИ!

Некоторые пользователи думают, будто информация о порте предоставляется всегда. Однако это не совсем так. Python предоставит порт, заданный по умолчанию, если вы явно не укажете номер порта, но полагаться на такой порт не стоит, ведь вы не будете точно знать, с какой службой он связан. Кроме того, некоторые системы используют нестандартные номера портов исходя из соображений безопасности. Выработайте привычку всегда указывать номер порта и проверять, что работаете с корректным портом.

Имя локального хоста

Имя хоста — это текстовая форма адреса хоста. Люди плохо понимают смысл адреса 127.0.0.1 (адреса IPv6 еще менее понятны). А вот имя хоста понятно всем. Есть специальная служба, отвечающая за перевод понятных человеку имен хостов в адреса, но в этой книге мы не будем затрагивать данную тему.

В разделе "Хост" для определения имени хоста применялся метод `socket.gethostbyaddr()`, в качестве аргумента которого задавался сетевой адрес. Мы также выполняли и обратное преобразование с помощью метода `socket.gethostbyname()`. В следующей пошаговой инструкции показано, как работать с именем хоста.

1. **Введите `import socket`.**
2. **Введите `socket.gethostname()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

   Отобразится имя локального компьютера (рис. 17.6). В вашем случае имя наверняка будет другим.

```
In [7]: import socket
        socket.gethostname()

Out[7]: 'Alex'
```

Рис. 17.6. Иногда нужно знать имя локального компьютера

3. **Введите `socket.gethostbyname(socket.gethostname())` и щелкните на кнопке Run Cell.**

   Вы увидите IP-адрес локальной системы, как показано на рис. 17.7. Опять же, ваши настройки наверняка отличаются от настроек автора книги, поэтому будет отличаться и результат. Данный прием можно использовать в приложениях для определения адреса отправителя, когда это необходимо. Поскольку здесь не используются жестко заданные значения, прием будет работать в любой системе.

```
In [7]: socket.gethostbyname(socket.gethostname())

Out[7]: '192.168.0.101'
```

Рис. 17.7. По возможности избегайте жестко заданных значений для локальной системы

Компоненты сообщения

Конверт электронной почты используется SMTP-сервером для маршрутизации сообщения. Он не включает собственно содержимое, находящееся в письме. Многие разработчики путают эти два элемента, потому что письмо также содержит информацию об отправителе и получателе. Эти сведения появляются в письме подобно адресной информации, включаемой в деловое письмо, и служат исключительно для удобства пользователя. Когда вы отправляете деловое письмо, почтальон не будет вскрывать конверт, чтобы увидеть информацию об адресе внутри. Для него имеют значение только сведения, указанные на конверте.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Из-за того что информация в электронном письме отделена от информации, указанной в конверте, злоумышленники могут подделать адреса электронной почты. В результате конверт будет содержать корректную информацию об отправителе и получателе, а письмо — нет. (Когда вы получаете электронное сообщение, вы видите само письмо, но не конверт — он удаляется приложением электронной почты.)

Сообщение электронной почты состоит из нескольких компонентов, как и конверт. Ниже приведен краткий обзор трех компонентов сообщения.

- **Отправитель.** Информация об отправителе сообщения. Включает только адрес электронной почты отправителя.
- **Получатель.** Информация о получателе сообщения. В действительности это список адресов получателей. Даже если вы хотите

отправить сообщение только одному человеку, нужно добавить один адрес в список.

- **Сообщение.** Информация, которую должен увидеть получатель. Может включать следующие элементы.
  - **От.** Информация об отправителе в текстовом виде.
  - **Кому.** Информация о получателе в текстовом виде.
  - **Копия.** Отображаемые получатели, которым тоже адресуется сообщение, даже если они не указаны как основные получатели.
  - **Тема.** Назначение сообщения.
  - **Документы.** Один или несколько документов, включая текстовое сообщение, которые прилагаются к письму.

Электронные письма могут быть довольно сложными и длинными и включать множество дополнительных элементов. Но чаще всего они содержат лишь вышеперечисленные простые компоненты, требуемые для отправки письма из почтового приложения. В следующих разделах процесс создания письма и его компонентов будет описан более подробно.

Структура сообщения

Можно, конечно, успешно отправить кому-либо пустой конверт, но вряд ли в этом есть смысл. Чтобы от письма была польза, оно должно содержать сообщение. В Python поддерживается несколько способов создания сообщений. Самый простой и надежный способ — воспользоваться средствами MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения интернет-почты).

Как и многие другие протоколы электронной почты, MIME стандартизирован, поэтому работает одинаково вне зависимости от платформы. Существуют многочисленные форматы MIME, включенные в модуль `email.mime`, который описан по адресу `https://docs.python.org/3/library/email.mime.html`. Чаще всего вам понадобятся следующие форматы MIME.

- **`MIMEApplication`**. Позволяет отправлять и получать входные и выходные данные приложения.
- **`MIMEAudio`**. Содержит звуковой файл.
- **`MIMEImage`**. Содержит файл изображения.
- **`MIMEMultipart`**. Позволяет создавать сообщения, состоящие из нескольких частей, например текстовой и графической.
- **`MIMEText`**. Содержит текстовые данные в формате ASCII, HTML или другом стандартизированном формате.

Несмотря на то что с помощью Python можно создавать сообщения любого типа, проще всего начать с сообщения, содержащего обычный текст. Отсутствие форматирования позволит нам сосредоточиться на создании самого сообщения, а не его содержимого. В следующей пошаговой инструкции показано, как создается сообщение электронной почты, которое пока что никуда не отправляется.

1. **Введите следующий код в блокноте.**

```
from email.mime.text import MIMEText
msg = MIMEText("Hello there")
msg['Тема'] = "A Test Message"
msg['От'] = 'John Mueller <John@JohnMuellerBooks.com>'
msg['Кому'] = 'John Mueller <John@JohnMuellerBooks.com>'
```

ЗАПОМНИ!

Это простое текстовое сообщение, поэтому для него нужно импортировать класс `MIMEText`. Если бы создавался какой-то иной контент, нужно было бы импортировать другие классы или же весь модуль `email.mime`.

Конструктору `MIMEText()` нужно передать текст сообщения. Это тело сообщения, которое может быть довольно длинным. В данном случае сообщение относительно короткое и представляет собой простое приветствие.

Далее мы присваиваем значения стандартным атрибутам. В примере показаны три универсальных атрибута, которые определяется всегда: `Тема`, `От` и `Кому`. Два поля адреса, `От` и `Кому`, содержат как текстовое имя, так и адрес электронной почты. Обязательным является только адрес электронной почты.

2. **Введите `msg.as_string()` и щелкните на кнопке Run Cell (Выполнить ячейку).**

Результат показан на рис. 17.8. Вот как на самом деле выглядит сообщение. Если вы когда-либо просматривали исходное содержимое сообщений, создаваемых вашим почтовым приложением, то подобная структура будет вам знакома.

Параметр `Content-Type` определяет тип сообщения, которое в данном случае является текстовым. Параметр `charset` сообщает о том, какой набор символов используется в сообщении, чтобы получатель знал, как их обрабатывать. Параметр `MIME-Version` задает версию MIME, используемую для создания сообщения. Наконец, параметр `Context-Transfer-Encoding` определяет, каким образом сообщение преобразуется в битовый поток перед отправкой получателю.

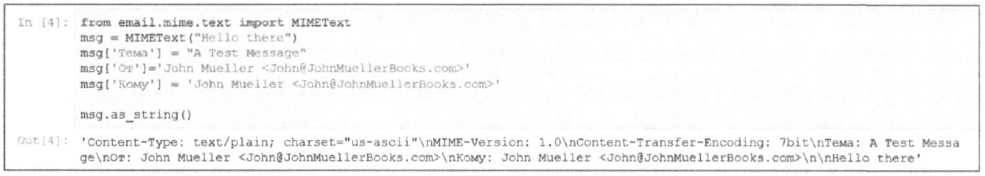


Рис. 17.8. Python добавляет служебную информацию, необходимую для создания сообщения

Процесс передачи сообщения

Выше уже говорилось о том, каким образом конверт используется для передачи сообщения из одного места в другое. При отправке сообщения необходимо определить способ передачи. Python создает сам конверт и выполняет передачу сообщения, но именно пользователь определяет особенности передачи. В следующей пошаговой инструкции демонстрируется самый простой подход к отправке сообщения в Python. У вас не получится отправить письмо, если вы не внесете коррективы в соответствии с конфигурацией вашего почтового сервера. Прочитайте приведенную ниже врезку "Знакомство с SMTP-сервером" для получения дополнительной информации.

1. **Введите следующий код в блокноте.**

   ```
   import smtplib
   s = smtplib.SMTP('localhost')
   ```

   Модуль `smtplib` содержит все, что нужно для создания и отправки конверта сообщения. Первым шагом в этом процессе является создание подключения к SMTP-серверу, который указывается в строковом виде в конструкторе. Если заданного вами SMTP-сервера не существует, приложение аварийно завершится, сообщив, что хост отказал в установке соединения.

2. **Введите `s.sendmail('`*`адрес_отправителя`*`', ['`*`адрес_получателя`*`'], msg.as_string())` и щелкните на кнопке Run Cell (Выполнить ячейку).**

ВНИМАНИЕ!

Чтобы этот код был работоспособным, нужно заменить *`адрес_отправителя`* и *`адрес_получателя`* реальными адресами. На этот раз не используйте описательные имена, поскольку серверу нужен только адрес. Если не указать реальный адрес, будет выдано сообщение об ошибке. Другие возможные причины появления сообщения об ошибке: почтовый сервер временно отключен от сети, произошел сетевой сбой или случилась какая-то другая нештатная ситуация. Если вы уверены, что ввели корректную информацию, попробуйте отправить сообщение еще раз, прежде чем начинать паниковать. Для получения дополнительной

информации обратитесь к приведенной ниже врезке "Знакомство с SMTP-сервером".

На этом шаге Python создает конверт, упаковывает сообщение электронной почты и отправляет его получателю. Обратите внимание на то, что информация об отправителе и получателе указывается отдельно от сообщения, которое SMTP-сервер не читает.

Подтипы сообщений

Выше рассматривались основные типы сообщений электронной почты. Если бы электронная почта полагалась только на эти типы, то передавать согласованные сообщения было будет затруднительно. Проблема в том, что одного типа недостаточно. Когда вы отправляете текстовое сообщение, вы должны знать, что это за текст, прежде чем сможете его корректно обработать. Текстовое сообщение может быть отформатировано как обычный текст или как HTML-страница. Вы не узнаете этого, просто прочитав метку типа, поэтому для сообщений требуется указывать подтип. Тип сообщения — текст, а подтип — HTML, если вы отправляете кому-либо HTML-страницу. Тип и подтип разделяются косой чертой, поэтому вы увидите `text/html`, если просмотрите исходное содержимое сообщения.

ЗАПОМНИ!

Теоретически количество подтипов не ограничено, если для платформы определены их обработчики. Но на практике все пользователи должны прийти к соглашению о подтипах, иначе обработчики не появятся (если только вы не имеете дело с пользовательским приложением, для которого обе стороны заранее договорились насчет собственного подтипа). Вы сможете найти список стандартных типов и подтипов на сайте `http://www.freeformatter.com/mime-types-list.html`. Приятной особенностью опубликованной там таблицы является то, что в ней приведены стандартные расширения файлов, связанные с каждым подтипом, и даны ссылки для получения дополнительной информации.

Создание электронного сообщения

До сих пор конверт и сообщение рассматривались по отдельности. Теперь пришло время соединить их и посмотреть, как работает электронная почта. В следующих разделах будет показано, как создать два сообщения. Первое из них — текстовое, а второе использует форматирование HTML. Оба сообщения должны поддерживаться большинством клиентов электронной почты.

Работа с текстовым сообщением

Текстовые сообщения — это наиболее эффективный и наименее ресурсоемкий способ передачи информации. С другой стороны, в них содержится меньше всего информации. Да, можно использовать смайлики для выражения эмоций, но отсутствие форматирования в определенных ситуациях становится проблемой. Ниже описано, как создать простое текстовое сообщение, используя Python.

1. **Введите следующий код в блокноте.**

```
from email.mime.text import MIMEText
import smtplib
msg = MIMEText("Hello There!")
msg['Тема'] = 'A Test Message'
msg['От'] = 'SenderAddress'
msg['Кому'] = 'RecipientAddress'
s = smtplib.SMTP('localhost')
s.sendmail('SenderAddress',
           ['RecipientAddress'],
           msg.as_string())
print("Сообщение отправлено!")
```

Этот пример представляет собой квинтэссенцию всего, что было рассмотрено в главе. Мы впервые свели все вместе. Заметьте, что сначала мы создаем сообщение, а затем — конверт (как и в реальной жизни).

ВНИМАНИЕ!

Программа выдаст ошибку, если не заменить `SenderAddress` и `RecipientAddress` реальными адресами. Эти записи используются только в качестве заполнителей. Как и в примере из предыдущего раздела, ошибки могут возникать по разным причинам, поэтому всегда пытайтесь отправить сообщение как минимум дважды, если видите ошибку впервые. Для получения дополнительной информации по этой теме обратитесь к врезке "Знакомство с SMTP-сервером".

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

Приложение уведомит о том, что сообщение было отправлено получателю.

Работа с HTML-сообщением

HTML-сообщение — это текстовое сообщение со специальным форматированием. В следующей пошаговой инструкции показано, как создать электронное письмо в формате HTML и отправить его.

ЗНАКОМСТВО С SMTP-СЕРВЕРОМ

Если вы попробовали выполнить пример этой главы, не внеся никаких изменений, то, скорее всего, у вас ничего не получилось. Маловероятно, чтобы имеющийся в вашей системе SMTP-сервер был подключен к сайту `localhost`. Причина, по которой в примерах используется адрес `localhost`, заключается в предоставлении заполнителя, который позже должен быть заменен реальным адресом в соответствии с конфигурацией конкретной системы.

Для того чтобы пример был работоспособным, нужны SMTP-сервер и реальная учетная запись электронной почты. Можете установить все программное обеспечение, необходимое для создания такой среды, в своей системе, и некоторые разработчики, которые интенсивно работают с почтовыми приложениями, именно так и поступают. В большинстве платформ имеется пакет электронной почты, доступный для установки, или же можно воспользоваться бесплатной альтернативой — открытым пакетом Sendmail, который можно скачать по следующей ссылке:

`https://downloads.tomsguide.com/SendMail,0301-2037.html`

Самый простой способ заставить пример заработать — использовать тот же самый SMTP-сервер, с которым взаимодействует ваше почтовое приложение. Когда вы конфигурировали приложение электронной почты, вы либо проинструктировали его обнаруживать SMTP-сервер, либо задали SMTP-сервер самостоятельно. Параметры конфигурации вашего почтового приложения должны содержать необходимую информацию. Где именно находится эта информация сильно зависит от конкретного приложения, так что сверьтесь с документацией.

Независимо от того, какой тип SMTP-сервера в конечном итоге будет использован, в большинстве случаев необходимо иметь учетную запись на этом сервере, чтобы работать с ним. Используйте в примерах адрес вашего SMTP-сервера, например `smtp.myisp.com`, и укажите ваш адрес электронной почты для отправителя и получателя. В противном случае пример работать не будет.

1. **Введите следующий код в блокноте.**

```
from email.mime.text import MIMEText
import smtplib
msg = MIMEText(
   "<h1>A Heading</h1><p>Hello There!</p>", "html")
msg['Тема'] = 'A Test HTML Message'
msg['От'] = 'SenderAddress'
msg['Кому'] = 'RecipientAddress'
s = smtplib.SMTP('localhost')
s.sendmail('SenderAddress',
```

```
            ['RecipientAddress'],
            msg.as_string())
print("Сообщение отправлено!")
```

Это фактически тот же пример, что и пример текстового сообщения в предыдущем разделе. Но обратите внимание на то, что сообщение теперь содержит HTML-теги. Мы создаем только тело HTML-сообщения, а не всю страницу. Сообщение содержит заголовок `<h1>` и абзац `<p>`.

ЗАПОМНИ!

Наиболее важной частью примера является текст, который следует за сообщением. Аргумент `"html"` изменяет подтип с `text/plain` на `text/html`, давая получателю понять, что сообщение нужно обрабатывать как HTML-содержимое. Если не внести это изменение, получатель не увидит HTML-контент.

2. **Щелкните на кнопке Run Cell (Выполнить ячейку).**

   Приложение уведомит о том, что сообщение было отправлено получателю.

Просмотр электронного сообщения

На данный момент в вашей папке Входящие должно находиться от одного до трех сообщений, сгенерированных приложением. Чтобы вы могли просмотреть сообщения, созданные в предыдущих разделах, ваше почтовое приложение должно получить их от сервера — так же, как и любую другую электронную почту. На рис. 17.9 показан пример HTML-версии сообщения.

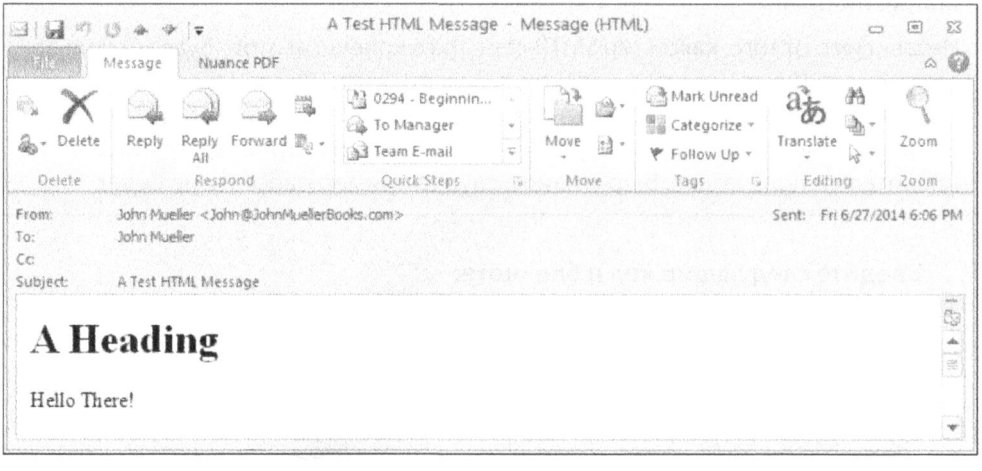


Рис. 17.9. HTML-сообщение содержит заголовок и абзац

Если почтовое приложение позволяет просмотреть источник сообщения, вы обнаружите, что сообщение содержит ту информацию, которую вы уже видели ранее. Ничего не поменялось, поскольку сообщение не подвергается никаким изменениям в процессе передачи получателю.

ЗАПОМНИ!

Создавать собственное приложение для отправки и получения электронной почты неудобно — гораздо лучше использовать готовое приложение. Целью примеров, приведенных в этой главе, было продемонстрировать гибкость Python. При создании собственного приложения вы контролируете каждый аспект сообщения. Python скрывает большую часть деталей, давая вам возможность сконцентрироваться на том, как создать и передать сообщение, используя корректные аргументы.

Великолепные десятки

В ЭТОЙ ЧАСТИ...

- Продолжаем изучать Python
- Упрощаем работу с Python с помощью утилит
- Расширяем возможности Python с помощью библиотек

Глава 18

Десять ценных ресурсов для программистов

В ЭТОЙ ГЛАВЕ...

- Источники дополнительной информации о Python
- Создание веб-приложений с помощью Python
- Расширение программной среды Python
- Улучшение производительности приложений и самих разработчиков

Эта книга способна дать хороший старт вашей карьере программиста на Python, но в какой-то момент вам потребуются дополнительные ресурсы. В данной главе вашему вниманию будут представлены десять полезных ресурсов, которые можно использовать для улучшения своих навыков программирования. Они помогут вам экономить время и усилия при создании великолепных приложений Python.

ЗАПОМНИ!

Эта глава содержит лишь краткий обзор ресурсов Python. В Интернете имеется огромное количество документации по Python и буквально горы готового кода. Можно написать целую книгу (и не одну), посвященную исключительно библиотекам Python. Так что не останавливайтесь в своих поисках и относитесь к главе как к началу большого пути.

Онлайн-документация Python

Работая с Python, важно знать, что доступно в базовом языке и как его можно расширить для решения других задач. В документации Python, доступной на сайте `https://docs.python.org/3/` (создана для версии 3.7*x* на момент выхода книги; со временем может быть обновлена), содержится гораздо больше, чем просто описание средств языка, которые вы получаете в составе загружаемого дистрибутива. Фактически в документацию включено следующее:

- новые функции текущей версии языка;
- полноценный учебник по Python;
- полное справочное руководство по библиотекам;
- полное справочное руководство по языку;
- пошаговая инструкция по установке и конфигурированию Python;
- описание способов решения отдельных задач на Python;
- справка по установке внешних модулей, выступающих в качестве средств расширения Python;
- справка по распространению пользовательских модулей Python;
- инструкция по расширению Python с помощью C/C++ и внедрению новых функций;
- полное руководство для разработчиков на C/C++, которые хотят расширить свои приложения с помощью Python;
- страницы с ответами на часто задаваемые вопросы (Frequently Asked Questions — FAQ).

ЗАПОМНИ!

Вся эта информация приведена в простой и доступной форме. В дополнение к привычному оглавлению можно воспользоваться различными поисковыми индексами. Например, если нужно найти определенный модуль, класс или метод, можно воспользоваться Global Module Index (глобальный индекс модулей).

На сайте можно также сообщить о проблемах, возникших при работе с Python (`https://docs.python.org/3/bugs.html`). Как и в любом другом языке программирования, в Python есть ошибки. Обнаружив и устранив их, вы поможете сделать Python лучше.

СОВЕТ

С документацией можно работать по-разному. В левом верхнем углу страницы доступны два раскрывающихся списка. В первом из них можно выбрать предпочитаемый язык (на данный момент можно

выбрать английский, французский, японский, корейский и китайский). С помощью второго списка обеспечивается доступ к документации по более ранним версиям Python, включая версию 2.7.

Руководство на сайте `LearnPython.org`

Существует много прекрасных учебных пособий по Python, но в них нет главного, что предлагается на сайте `LearnPython.org`: интерактивности. Здесь вы можете не только прочитать о различных средствах Python, но и опробовать их самостоятельно в интерактивном режиме.

Уровень простых примеров вы уже прошли в данной книге. Если вам интересно, попробуйте теперь освоить более сложные примеры на сайте `LearnPython.org`. Они охватывают следующие темы.

- **Generators (Генераторы).** Специализированные функции, возвращающие итераторы.
- **List comprehensions (Списковые включения).** Технология создания новых списков на основе существующих.
- **Multiple function arguments (Функции с переменным числом аргументов).** Та же самая идея применительно к методам описывалась в главе 15.
- **Regular expressions (Регулярные выражения).** Используются для создания поисковых шаблонов, используемых, например, при работе с телефонными номерами.
- **Exception handling (Обработка исключений).** Данная тема рассматривалась в главе 10.
- **Sets (Наборы).** Особый вид списка, не содержащий повторяющихся записей.
- **Serialization (Сериализация).** Методология хранения данных на основе JSON (JavaScript Object Notation — нотация объектов JavaScript).
- **Partial functions (Частичные функции).** Методика создания специализированных версий функций, которые являются производными от более сложных функций. Например, если есть функция `multiply()`, которая требует двух аргументов, то для частичной функции `double()` может требоваться только один аргумент, который всегда умножается на 2.

- **Code introspection (Интроспекция).** Возможность исследовать классы, функции и ключевые слова, чтобы определить их назначение и возможности.
- **Closures (Замыкания).** Функции, которые ссылаются на внешние переменные, объявленные в их области видимости.
- **Decorators (Декораторы).** Шаблоны, применяемые для внесения простых модификаций в вызываемые объекты.

Веб-программирование на Python

Эта книга посвящена основам программирования, поэтому мы ориентировались на настольные приложения в силу их простоты. Тем не менее многие разработчики специализируются на создании веб-приложений с использованием Python. Сайт *Web Programming in Python* (`https://wiki.python.org/moin/WebProgramming`) поможет вам перейти от настольных к веб-приложениям. На сайте рассмотрены практически все виды веб-приложений. Учебные пособия разделены на следующие категории.

- Серверное программирование.
  - Разработка серверных фреймворков для приложений.
  - Создание скриптов CGI (Common Gateway Interface — общий интерфейс шлюза).
  - Предоставление серверных приложений.
  - Разработка систем управления контентом (Content Management Systems — CMS).
  - Разработка методов доступа к данным с помощью веб-служб.
- Клиентское программирование.
  - Взаимодействие с браузерами и браузерными приложениями.
  - Создание клиентов, основанных на использовании браузеров.
  - Разработка методов доступа к данным с помощью веб-служб.
- Дополнительные темы.
  - Создание общих решений для интернет-вычислений на основе Python.
  - Взаимодействие с системами управления базами данных (СУБД).
  - Разработка шаблонов приложений.
  - Построение интранет-решений.

Дополнительные библиотеки

На сайте Pythonware (`http://www.pythonware.com/`) размещены ссылки для доступа к ряду сторонних библиотек Python.

В первую очередь просмотрите ссылку Downloads (downloads.effbot.org). Вы получите доступ к следующим библиотекам.

- `aggdraw`. Библиотека, упрощающая создание сглаженных рисунков.
- `celementtree`. Дополнение к библиотеке `elementtree`, которое обеспечивает более эффективную и быструю обработку данных XML.
- `console`. Интерфейс для Windows, позволяющий создавать консольные приложения.
- `effbot`. Коллекция полезных дополнений и утилит, включая программу чтения новостей EffNews.
- `elementsoap`. Библиотека, позволяющая создавать подключения SOAP (Simple Object Access Protocol — простой протокол доступа к объекту) к провайдерам веб-служб.
- `elementtidy`. Дополнение к библиотеке `elementtree`, которое позволяет создавать более привлекательные и функциональные отображения дерева XML, чем стандартные средства Python.
- `elementtree`. Библиотека, которая позволяет взаимодействовать с данными XML более эффективно, чем стандартные средства Python.
- `exemaker`. Утилита, которая создает исполняемую программу на основе сценария Python, чтобы сценарий можно было выполнять так же, как и любое другое приложение.
- `ftpparse`. Библиотека для работы с FTP-сайтами.
- `grabscreen`. Библиотека для создания экранных снимков.
- `imaging`. Предоставляет исходный код для библиотеки изображений Python (Python Imaging Library — PIL), которая позволяет добавлять возможности обработки изображений в интерпретатор Python. Благодаря исходному коду можно настроить библиотеку PIL под свои потребности.
- `pil`. Двоичные установщики для PIL, которые упрощают получение нужной инсталляции для системы. (Существуют и другие библиотеки на основе PIL, такие как `pilfont`, предназначенная для добавления расширенных шрифтовых средств в PIL-приложения.)
- `pythondoc`. Утилита, предназначенная для создания документации на основе комментариев в коде Python (работает так же, как JavaDoc).

- squeeze. Утилита, предназначенная для преобразования Python-приложения, содержащегося в нескольких файлах, в одно- или двухфайловый дистрибутив, который будет выполняться интерпретатором Python.
- tkinter3000. Библиотека создания виджетов для Python, которая включает несколько модулей. *Виджеты* — это фрагменты кода, создающие элементы управления, например кнопки, для графических приложений. Существует ряд надстроек, таких как wckgraph, которые помогают добавить в приложение поддержку графики.

Быстрое создание приложений с помощью IDE

Интерактивная среда разработки (IDE) помогает создавать приложения на определенном языке программирования. Редактор IDLE (Integrated DeveLopment Environment), входящий в состав Python, удобен для выполнения экспериментов, но через некоторое время его возможностей может оказаться недостаточно. Например, IDLE не предоставляет расширенных функций отладки, к которым привыкли многие разработчики. Кроме того, с помощью IDLE создание графических приложений может оказаться непростой задачей.

Ограничения в IDLE являются причиной, по которой в книге вместо IDLE используется Jupyter Notebook. Тем не менее возможностей Jupyter Notebook тоже порой оказывается недостаточно. Если вы пообщаетесь с 50 разработчиками и попробуете выяснить, какую IDE они предпочитают, то получите множество разных ответов. У каждого разработчика есть своя любимая среда разработки, и его нелегко заставить попробовать что-то другое. Разработчики тратят много времени на изучение конкретной IDE и ее расширение под свои задачи.

ЗАПОМНИ!

Зачастую заменить ранее выбранную IDE невозможно, поэтому важно попробовать несколько различных IDE, прежде чем остановиться на одной из них. (Самая распространенная причина, по которой невозможно перейти на другую IDE, — это несовместимость типов проектов. Вам придется повторно создавать проекты каждый раз, когда выбирается другой редактор. Существуют и другие причины, список которых можно найти в Интернете.) В статье PythonEditors, доступной по адресу https://wiki.python.org/moin/PythonEditors, приводится обширный перечень IDE, которые можно попробовать в работе. В соответствующей таблице приведены подробные сведения о каждом редакторе, которые помогут вам сразу же исключить некоторые неподходящие варианты.

Удобная проверка синтаксиса

Редактор IDLE реализует базовую подсветку синтаксиса, что полезно при поиске ошибок. Например, если неправильно ввести ключевое слово, то его цвет не изменится на цвет, используемый для выделения ключевых слов в вашей системе. Увидев, что он не изменился, вы сможете сразу же исправить ошибку, вместо того чтобы запускать приложение и потом выяснять, что же пошло не так.

Среда Jupyter Notebook также обеспечивает подсветку синтаксиса и предоставляет расширенные средства проверки ошибок, которые отсутствуют в стандартной IDE. Тем не менее для некоторых разработчиков этого тоже может оказаться недостаточно, поскольку для просмотра информации об ошибках придется выполнять ячейки. Многие разработчики предпочитают интерактивную проверку синтаксиса, при которой IDE немедленно выделяет ошибку, еще до того, как разработчик покинет строку с ошибочным кодом.

Утилита `python.vim` (`http://www.vim.org/scripts/script.php?script_id=790`) обеспечивает улучшенную подсветку синтаксиса, которая упрощает поиск ошибок в сценарии Python. Эта утилита работает как сценарий, что делает ее быстрой и эффективной для использования на любой платформе. Кроме того, можно настроить исходный код утилиты под свои потребности.

Использование XML

В настоящее время расширяемый язык разметки (XML) применяется для хранения данных любых типов в большинстве приложений. Возможно, в вашей системе есть XML-файлы, а вы даже не знаете об этом, потому что они имеют другие расширения. Например, многие файлы `.config`, используемые для хранения настроек приложения, основаны на XML. Так что при написании приложений на Python вы рано или поздно столкнетесь с XML.

XML имеет ряд преимуществ перед другими форматами хранения данных. Например он не зависит от платформы. Вы сможете использовать XML в любой системе, и созданный на компьютере файл можно будет прочитать на любом другом компьютере. Благодаря тому что XML не зависит от платформы, он применяется во многих технологиях, например в веб-службах. Кроме того, XML относительно прост в изучении, а поскольку это текстовый формат, возникающие проблемы можно достаточно легко устранять.

Чтобы освоить XML, можно воспользоваться учебным пособием, доступным на сайте W3Schools по адресу `http://www.w3schools.com/xml/default.ASP`. Некоторые разработчики пренебрегают изучением XML, в потом обнаруживают, что не в состоянии разобраться в специфичном коде Python, предполагающем наличие навыков по созданию базовых XML-файлов. Сайт W3Schools удобен тем, что процесс обучения разбивается на главы, так что можно осваивать XML постепенно:

- базовый курс XML;
- валидация файлов XML;
- использование XML совместно с JavaScript;
- обзор технологий, связанных с XML;
- использование расширенных методик XML;
- работа с примерами XML, чтобы увидеть XML в действии.

После того как вы ознакомитесь с основами, вам понадобится ресурс, где демонстрируется совместное использование XML и Python. Одним из лучших мест, где можно найти соответствующую информацию, является сайт *Tutorials on XML Processing with Python*, который доступен по следующему адресу:

```
https://wiki.python.org/moin/Tutorials%20on%20XML%20processing
%20with%20Python
```

Благодаря этим двум ресурсам вы сможете быстро получить необходимые знания по XML, что позволит вам начать создавать приложения Python, использующие XML.

ВОСПОЛЬЗУЙТЕСЬ РЕСУРСАМИ W3SCHOOLS

Одним из самых популярных веб-ресурсов, используемых для изучения компьютерных технологий в Интернете, является W3Schools (`www.w3schools.com`). Этот сайт поможет вам освоить все веб-технологии, необходимые для создания любого современного приложения, которое только можно себе представить. На этом ресурсе доступны следующие темы:

- HTML;
- CSS;
- JavaScript;
- SQL;
- JQuery;

- PHP;
- XML;
- ASP.NET.

Следует понимать, что это лишь отправная точка для разработчиков Python. Используйте материалы сайта W3Schools, чтобы получить представление о базовой технологии, а затем обратитесь к ресурсам, специфичным для Python.

Устранение типичных ошибок, присущих новичкам

Абсолютно все допускают ошибки при программировании, даже крутые профессионалы с более чем 30-летним опытом. Конечно, никто не любит ошибаться и тем более признавать это, но от ошибок не застрахован никто. Поэтому не стоит слишком переживать, допустив ошибку. Просто исправьте ее и двигайтесь дальше.

ЗАПОМНИ!

Есть разница между непредсказуемой ошибкой и ошибкой типичной, которую можно и нужно было избежать. Безусловно, даже профессионалы иногда допускают ошибки, которых можно избежать, но это гораздо менее вероятно, поскольку они уже сталкивались с такого рода ошибками и приучили себя не допускать их. Вы сможете получить преимущество перед конкурентами, научившись избегать ошибок новичков, о которых каждый должен знать. Для этого просмотрите следующий ресурс из двух частей.

» *Python: Common Newbie Mistakes, Part 1.*

```
http://blog.amir.rachum.com/blog/2013/07/06/
python-common-newbie-mistakes-part-1/
```

» *Python: Common Newbie Mistakes, Part 2.*

```
http://blog.amir.rachum.com/blog/2013/07/09/
python-common-newbie-mistakes-part-2/
```

Для новичков в Python доступно множество других сайтов, но указанные ресурсы достаточно краткие и простые для понимания. Вы сможете ознакомиться с ними за относительно короткое время, сделать для себя выводы и начать наконец избегать тех неприятных ошибок, которых не следует допускать.

Знакомство с Unicode

В этой книге мы постарались обойти стороной непростую тему Unicode, но вы неизбежно с ней столкнетесь, когда начнете создавать серьезные приложения. К сожалению, до сих пор не существует единого определения Unicode, поскольку работа над стандартом продолжается непрерывно.

Вы столкнетесь с множеством вариантов Unicode, когда начнете работать с многоязычными приложениями Python, поскольку для каждого языка стандартизируется своя таблица символов. Если хотите изучить Unicode, обратитесь к следующим ресурсам.

- *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!).*

  `http://www.joelonsoftware.com/articles/Unicode.html`

- *The Updated Guide to Unicode on Python.*

  `http://lucumr.pocoo.org/2013/7/2/the-updated-guide-to-unicode/`

- *Python Encodings and Unicode.*

  `http://eric.themoritzfamily.com/python-encodings-and-unicode.html`

- *Explain it like I'm five: Python and Unicode?*

  `http://www.reddit.com/r/Python/comments/1g62eh/explain_it_like_im_five_python_and_unicode/`

- *Unicode Pain.*

  `http://nedbatch26elder.com/text/unipain.html`

Учимся создавать быстродействующие приложения

Ничто так не раздражает пользователя, как медленное приложение. В подобном случае велика вероятность того, что с приложением вообще никто не будет работать. Низкая производительность является основной причиной провала приложений в корпоративных средах. Организация может потратить кучу денег на создание красивого приложения, которое делает все что угодно, но оно в итоге окажется никому не нужным, поскольку работает слишком медленно или имеет другие серьезные проблемы, связанные с производительностью.

Производительность приложения — это сочетание его надежности, безопасности и скорости работы. Этот "треугольник производительности" описывается в блоге автора по следующему адресу:

```
http://blog.johnmuellerbooks.com/2012/04/16/considering-the-
performance-triangle/
```

Многие разработчики фокусируются только на быстродействии и в итоге не достигают поставленных целей. Важно рассмотреть все аспекты использования ресурсов вашим приложением и убедиться в том, что применяются оптимальные приемы программирования.

Доступны многочисленные ресурсы, которые помогут вам изучить вопросы производительности применительно к приложениям Python. Рекомендую прочитать статью "PythonSpeed Performance Tips", доступную по следующему адресу:

```
https://wiki.python.org/moin/PythonSpeed/PerformanceTips
```

Глава 19

Десять утилит, улучшающих работу с Python

В ЭТОЙ ГЛАВЕ...

- Отладка, тестирование и развертывание приложений
- Ведение документации и управление версиями приложений
- Написание кода приложения
- Работа в интерактивной среде

Python, как и большинство языков программирования, располагает серьезной внешней поддержкой, которая доступна в виде различных утилит. *Утилита* — это любая программа, которая расширяет возможности Python в плане создания приложений. Таким образом, отладчик является утилитой, поскольку это отдельная программа, а вот библиотеки не относятся к утилитам. Тем не менее библиотеки помогают создавать более качественные приложения. (О некоторых из них будет рассказываться в главе 21.)

Для Python имеется множество универсальных и специализированных утилит. На сайте `https://wiki.python.org/moin/DevelopmentTools`, утилиты разбиты на следующие 13 категорий:

- `AutomatedRefactoringTools` (автоматизированные утилиты рефакторинга);
- `BugTracking` (отслеживание ошибок);

- ConfigurationAndBuildTools (утилиты конфигурирования и сборки);
- DistributionUtilities (дистрибутивные утилиты);
- DocumentationTools (утилиты документирования);
- IntegratedDevelopmentEnvironments (интегрированные среды разработки);
- PythonDebuggers (отладчики Python);
- PythonEditors (редакторы Python);
- PythonShells (командные оболочки Python);
- SkeletonBuilderTools (утилиты — построители каркасов приложений);
- TestSoftware (тестовое ПО);
- UsefulModules (полезные модули);
- VersionControl (контроль версий).

Отметим, что это далеко не полные списки. В Интернете можно найти много других списков.

Все это многообразие невозможно охватить в одной главе, поэтому мы рассмотрим лишь наиболее интересные утилиты, заслуживающие вашего внимания. После ознакомления с данной главой желательно поискать в Интернете и другие утилиты. Может оказаться, что утилита, которую вы планируете создать, уже существует, причем в нескольких вариантах.

Отслеживание ошибок с помощью Roundup Issue Tracker

Существует целый ряд сайтов, помогающих отслеживать ошибки в приложениях Python, среди которых — Github (https://github.com/), Google Code (https://code.google.com/), BitBucket (https://bitbucket.org/) и Launchpad (https://launchpad.net/). Это общедоступные сайты, но, как правило, они не столь удобны в использовании, как специализированное программное обеспечение. Одним из наилучших инструментов является Roundup Issue Tracker (http://roundup.sourceforge.net/). Он работает на любой платформе, где поддерживается Python, и предлагает следующие основные возможности:

- отслеживание ошибок;
- управление списком запланированных действий.

Приложив чуть больше усилий в процессе установки программы, вы сможете получить доступ к расширенным возможностям, которые делают программу особенно полезной. Для этого вам придется установить несколько дополнительных программ, в частности, СУБД (система управления базами данных). В инструкции указано, что именно следует установить и какие сторонние приложения являются совместимыми. Вот перечень расширенных возможностей, которые вы получите.

- Служба поддержки клиентов, предлагающая следующие услуги:
  - мастер ответов по телефону;
  - сетевые ссылки;
  - трекеры системных и других ошибок.
- Управление проблемами для рабочих групп IETF (Internet Engineering Task Force).
- Отслеживание потенциальных клиентов.
- Представление материалов для конференций.
- Двойной "слепой" метод управления лицами, принимающими решения.
- Ведение блога (простейшие возможности, которые со временем будут усложняться).

Создание виртуальной среды с помощью VirtualEnv

Существует много причин для создания виртуальных сред, но главная из них при работе с Python — обеспечение более безопасной и надежной среды тестирования. Применяя одинаковую среду тестирования, можно обеспечивать стабильность работы приложения до тех пор, пока оно не будет достаточно готово для тестирования в производственной среде. Программа VirtualEnv (`https://pypi.python.org/pypi/virtualenv`) предоставляет инструменты для создания виртуальной среды Python, которую можно использовать на начальных этапах процесса тестирования или при диагностике проблем, связанных с программной средой. Следует помнить, что существуют как минимум три стандартных уровня тестирования, которые нужно выполнять.

- **Отслеживание ошибок**. Проверка приложения на наличие ошибок.
- **Производительность**. Проверка соответствия приложения требованиям быстродействия, надежности и безопасности.

» **Юзабилити (удобство использования)**. Проверка того, что приложение соответствует требованиям пользователя и реагирует на ввод данных предсказуемым образом.

ЗАПОМНИ!

Большинство приложений Python не следует запускать в виртуальной среде после того, как они были развернуты в производственной системе. Это связано с тем, что большинству приложений Python требуется доступ к внешнему миру, а изоляция в виртуальной среде препятствует этому.

НЕ ПРОВОДИТЕ ТЕСТИРОВАНИЕ НА ПРОИЗВОДСТВЕННОМ СЕРВЕРЕ

Некоторые разработчики допускают серьезную ошибку, проводя тестирование еще незаконченного приложения на производственном сервере, к которому получают доступ пользователи. Основная причина, по которой не следует тестировать приложение на рабочем сервере, — возможная потеря данных. Если предоставить пользователям доступ к тестовой версии вашего приложения, то ошибки, связанные с некорректным обращением к базам данных или другим источникам данных, могут привести к безвозвратной потере данных.

Следует понимать, что у вас есть лишь один шанс для того, чтобы произвести впечатление на пользователей. Многие программные проекты терпят неудачу из-за того, что пользователям не нравится конечный результат. Программа готова, но оказывается ненужной пользователям, которые считают ее плохой по тем или иным причинам. У пользователя одна цель: выполнить поставленную задачу и заняться другими делами. Если они видят, что приложение отнимает у них время, они стараются его избегать.

У незавершенных приложений также могут быть уязвимости, через которые недобросовестные пользователи смогут получить доступ к вашей сети. Не важно, насколько хорошо работает защитное ПО, если вы оставляете дверь открытой для любого, кто проходит мимо. Избавиться от непрошенных гостей практически невозможно, и даже если вы перекроете обнаруженные "дыры", данные уже будут скомпрометированы. Восстановление после взлома — достаточно непростая задача, порой слишком болезненная. Короче говоря, никогда не тестируйте приложение на производственном сервере, поскольку риски слишком высоки.

Инсталляция приложений с помощью PyInstaller

Пользователям не хочется тратить много времени на инсталляцию приложения, каким бы полезным оно ни было. В некоторых случаях недостаточно опытный пользователь может вообще не суметь установить приложение. Короче говоря, необходим надежный способ передачи приложения из вашей системы в систему пользователя. Для этого существуют установщики, такие как PyInstaller (`http://www.pyinstaller.org/`). Они преобразуют приложение в пакет, который может быть легко инсталлирован пользователем.

Утилита PyInstaller работает на всех платформах, поддерживающих Python, поэтому является универсальным решением. Кроме того, при необходимости можно подключать инструменты, характерные для конкретной платформы. Например, в Windows можно создавать подписанные исполняемые файлы. Разработчики Mac оценят то, что PyInstaller гарантирует поддержку пакетов. С другой стороны, подобных вещей лучше избегать. Если задействуются специфичные для платформы средства, установка пройдет успешно только на этой конкретной платформе.

ВНИМАНИЕ!

Многие утилиты-установщики, доступные в Интернете, зависят от платформы. Если вы ищите установщик, который создает исполняемые файлы, убедитесь в том, что они не будут зависеть от применяемой платформы (или, по крайней мере, будут соответствовать целевой платформе). Важно получить продукт, который будет работать везде, где вы хотите. Универсального языка программирования недостаточно, если установочный пакет фактически препятствует процессу установки.

НЕ ИСПОЛЬЗУЙТЕ ПРОГРАММЫ, ЛИШЕННЫЕ ПОДДЕРЖКИ

Некоторые инструменты Python, доступные в Интернете, являются *орфанными*, т.е. перестали активно поддерживаться их создателями. Разработчики приложений Python могут продолжать пользоваться этими инструментами, поскольку им нравятся реализованные в них функции. Но поступать так слишком рискованно, поскольку нет гарантии, что программа будет совместима с последней версией Python. Лучше выбирать те инструменты, которые полностью поддерживаются их создателями.

Если вам абсолютно необходим такой инструмент (например, только он позволяет решить поставленную задачу), убедитесь в том, что он хотя бы имеет

поддержку со стороны сообщества пользователей. Вы по крайней мере сможете получить нужную информацию, когда вам понадобится поддержка. В противном случае можно потратить много времени в бесплодных попытках добиться результата от программы, которая никогда не будет работать должным образом.

Создание документации для разработчиков с помощью утилиты pdoc

С приложениями обычно связаны два вида документации: пользовательская и разработчика. Первая содержит информацию об использовании приложения, тогда как вторая демонстрирует, каким образом устроено и работает приложение. Для библиотеки требуется только документация разработчика, а для настольного приложения может понадобиться лишь пользовательская документация. Сервисная служба может потребовать оба вида документации в зависимости от того, кто ее использует и как она была создана. Большая часть документации относится к разработчикам, и утилита `pdoc` помогает ее сформировать (`https://github.com/BurntSushi/pdoc`).

Работа утилиты `pdoc` основана на информации, которая включается в код в виде строк документации и комментариев. На выходе формируется текстовый файл или HTML-документ. Можно также подключить ее к веб-серверу, чтобы можно было просматривать документацию непосредственно в браузере. По сути, это замена старой утилиты `epydoc`, которая больше не поддерживается.

СТРОКИ ДОКУМЕНТАЦИИ

Это особый вид комментария, который берется в тройные кавычки, например:

```
"""Это строка документации."""
```

Строки документации связываются с различными элементами кода, такими как пакеты, функции, классы и методы. Любой объект, создаваемый в Python, может включать строку документации, которая описывает его назначение.

Самый простой способ увидеть строку документации — указать после имени объекта специальный метод `__doc__()`. Например, если ввести **`print (MyClass.__doc__())`**, отобразится строка документации для класса `MyClass`. Доступ к строке документации можно также получить с помощью метода `help()`, например `help(MyClass)`. Хорошие строки документации содержат сведения о том, что именно делает объект, а не как он это делает.

Сторонние утилиты также могут задействовать строки документации. Имея подходящую утилиту, можно составить документацию для всей библиотеки, не написав ни одного дополнительного предложения. Утилита найдет строки документации в коде библиотеки и сформирует из них документацию. Следовательно, несмотря на то что строки документации и комментарии используются для разных целей, они одинаково важны в коде Python.

Разработка кода приложения с помощью Komodo Edit

В книге рассматривалась работа в интерактивных средах разработки (IDE), но не было дано конкретных рекомендаций (если не считать описания возможностей Jupyter Notebook). Выбор IDE зависит от задач, стоящих перед разработчиком, уровня его навыков и типов создаваемых приложений. Одни IDE предпочтительнее, чем другие, если речь идет о конкретных видах приложений. Одно из наилучших универсальных решений для начинающих разработчиков — редактор Komodo Edit, который можно загрузить по следующему адресу:

```
http://komodoide.com/komodo-edit/
```

Программа доступна бесплатно. Она включает множество инструментов, которые упростят процесс программирования по сравнению с IDLE. Ниже перечислены основные преимущества программы:

- поддержка множества языков программирования;
- автоматическое дополнение ключевых слов;
- проверка отступов;
- поддержка проектов, благодаря чему приложения частично кодируются еще до того, как пользователь приступит к программированию;
- превосходный уровень поддержки.

Главная особенность, отличающая Komodo Edit от других IDE, заключается в наличии расширенного варианта среды. Если окажется, что Komodo Edit не соответствует стоящим перед вами задачам, можно обновиться до уровня Komodo IDE (`http://komodoide.com/`). Это среда, которая содержит множество инструментов уже профессионального уровня, таких как профилирование кода (проверка скорости работы приложения) и обозреватель баз данных (упрощает работу с источниками данных).

Отладка приложения с помощью утилиты pydbgr

Профессиональная среда разработки, такая как Komodo IDE, поставляется с полноценным отладчиком. Даже в редакторе Komodo Edit имеется простой отладчик. Но если вы используете среду попроще, то отладчика может вообще не оказаться. *Отладчик* — это программа, которая помогает обнаруживать ошибки в приложении и исправлять их. Чем лучше отладчик, тем меньше усилий потребуется для выявления и устранения ошибок. Если применяемая среда не содержит отладчик, воспользуйтесь сторонним решением, например утилитой `pydbgr`, доступной по следующему адресу:

```
https://code.google.com/p/pydbgr/
```

ЗАПОМНИ!

Это достаточно хороший отладчик, который обеспечивает стандартные возможности, такие как цветовая разметка кода (выделение цветом определенных элементов синтаксиса, например ключевых слов), но также поддерживает ряд нестандартных функций, которые присущи только ему. Ниже перечислены ключевые преимущества отладчика `pydbgr`, которые делают его хорошим выбором, если ваш редактор не поставляется с отладчиком.

- **Улучшенная команда `eval`.** Команда `eval` позволяет увидеть, что произойдет при выполнении строки кода, еще до того, как она будет запущена. Это дает возможность выполнять анализ "что, если" в процессе отладки приложения.
- **Внепроцессная отладка**. Обычно отладка приложений выполняется на одном компьютере. Фактически отладчик является частью процесса приложения, т.е. потенциально может влиять на сам процесс отладки. Использование внепроцессной отладки означает, что отладчик не влияет на приложение, и вам не нужно запускать приложение на одном компьютере с отладчиком.
- **Тщательная проверка байт-кода.** Просмотр того, как написанный вами код приложения превращается в *байт-код* (который, собственно, и обрабатывается интерпретатором Python), иногда помогает в решении сложных проблем.
- **Фильтрация и отслеживание событий.** При запуске приложения в отладчике генерируется ряд событий, помогающих отладчику ориентироваться в происходящем. Например, при переходе к следующей строке кода генерируется одно событие, при возврате из

функции возникает другое событие и т.п. Это позволяет контролировать, каким образом отладчик выполняет трассировку приложения и на какие события он реагирует.

Переход в интерактивную среду с помощью IPython

Стандартная оболочка Python отлично подходит для решения большого числа интерактивных задач. Но, поработав с ней, вы поймете, что она имеет ряд недостатков. Самый большой недостаток заключается в том, что она является чисто текстовой средой, в которую нужно вводить команды. Усовершенствованная оболочка, такая как IPython (`http://ipython.org/`), предлагает более дружественную интерактивную среду с графическим интерфейсом, что позволяет не запоминать синтаксис сложных команд.

ЗАПОМНИ!

IPython — это не просто оболочка, а среда, в которой можно взаимодействовать с Python по-новому, например отображать графики с результатами работы созданных с помощью Python формул. Кроме того, IPython разработана как интерфейс для взаимодействия с другими языками программирования. Фактически IPython отправляет команды реальной оболочке в фоновом режиме, поэтому можно использовать оболочки для других языков программирования, таких как Julia и Haskell.

Одной из наиболее интересных особенностей IPython является возможность функционировать в параллельных вычислительных средах. Обычно оболочка является однопоточной, поэтому выполнять параллельные вычисления невозможно, нельзя даже сформировать многопоточную среду. Одно это делает оболочку IPython заслуживающей внимания.

Тестирование приложений Python с помощью PyUnit

Рано или поздно вы обязательно столкнетесь с тестированием приложений, ведь необходимо убедиться в том, что они работают так, как задумано. Тестирование можно выполнять, поочередно вводя команды и проверяя результаты либо автоматизировав весь процесс. Очевидно, что автоматизированный подход предпочтительнее, ведь это намного быстрее. Программные продукты,

такие как PyUnit (`https://wiki.python.org/moin/PyUnit`), значительно упрощают процесс модульного тестирования.

Интересной особенностью PyUnit является то, что в данном случае для тестирования фактически создается код Python. Получаемый в результате сценарий — это специализированное приложение, которое проверяет основное приложение на наличие ошибок.

ЗАПОМНИ!

Казалось бы, именно сценарии, а не профессионально написанные приложения, должны содержать ошибки. Однако сценарий тестирования спроектирован чрезвычайно просто, благодаря чему даже незначительные ошибки сценария достаточно заметны. Конечно, ошибки могут быть и в самом сценарии (что случается крайне редко), поэтому, если проблему нельзя соотнести с приложением, необходимо проверить сценарий.

Приведение кода в порядок с помощью утилиты `isort`

Код на Python может стать слишком запутанным, особенно если не включать инструкции импорта в верхней части файла. В некоторых ситуациях довольно трудно выяснить, что же происходит с кодом, если не привести его в порядок. Утилита `isort`, доступная по адресу `http://timothycrosley.github.io/isort/`, выполняет, казалось бы, скромную задачу по сортировке инструкций импорта, собирая их в верхней части файла исходного кода. Такое незначительное усовершенствование может существенно улучшить понимание исходного кода.

Знание того, какие именно модули необходимы для конкретного модуля, упрощает выявление потенциальных проблем. Например, если у вас каким-то образом оказалась более старая версия необходимого модуля, понимание того, какие именно модули нужны приложению, поможет упростить поиск необходимого модуля.

Кроме того, информация о том, какие модули требуются приложению, важна при распространении приложения среди пользователей. Сведения о том, что пользователь располагает необходимыми модулями, помогает обеспечить правильную работу приложения.

Управление версиями с помощью Mercurial

Приложения, которые вы создавали в процессе работы с книгой, не слишком сложны. Для них управление версиями вряд ли понадобится. Но когда вы начнете работать в корпоративной среде, создавая реальные пользовательские приложения, управление версиями станет необходимым. *Управление версиями* — это процесс отслеживания изменений, происходящих в приложении между релизами в производственной среде. Когда вы говорите, что используете программу MyApp 1.2, вы ссылаетесь на версию 1.2 программы MyApp. Контроль версий позволяет понять, какая именно версия приложения используется после внесения очередных изменений.

Для Python доступны многочисленные системы управления версиями. Одно из наиболее интересных решений — Mercurial (`www.mercurial-scm.org`). Эта программа доступна практически для любой платформы, на которой будет выполняться Python, поэтому не придется беспокоиться о смене продукта при смене платформы. (Если для платформы нет исполняемой версии, всегда можно создать ее из исходных кодов, предоставляемых на сайте программы.)

В отличие от многих других систем, Mercurial распространяется бесплатно. Даже если впоследствии вам понадобится более продвинутая система, вы получите полезный опыт, поработав с Mercurial над одним-двумя проектами.

ЗАПОМНИ!

Хранение каждой версии приложения в отдельном месте для того, чтобы изменения в случае необходимости можно было откатить или повторить, называется *управлением исходным кодом* (Source Code Management — SCM). Взаимодействие с любой версией исходного кода для конкретного приложения крайне важно, если необходимо возвращаться назад для устранения проблем, которые возникли в новой версии.

Удобно то, что пользователям Mercurial предоставляется превосходное онлайн-руководство, доступное по следующему адресу:

```
https://www.mercurial-scm.org/wiki/Tutorial
```

Желательно самому выполнить упражнения, приведенные в этом руководстве, чтобы лучше изучить процесс управления версиями, но даже просто прочитать такое пособие будет полезным. Конечно, первый урок посвящен установке Mercurial. Затем описывается процедура формирования хранилища (места, где хранятся версии приложений), а также использования хранилища в процессе разработки приложения. В результате вы получите четкое представление о том, как должна функционировать система управления версиями и зачем ее используют при разработке приложений.

Глава 20

Десять (или около того) библиотек, о которых стоит знать

В ЭТОЙ ГЛАВЕ...

- Защита данных с помощью криптографии
- Управление данными и их отображение
- Работа с графикой
- Поиск необходимой информации
- Получение доступа к коду Java

Python предоставляет пользователю обширную поддержку при создании обычных приложений. Но многие приложения нуждаются в расширенных средствах, и здесь на помощь приходят библиотеки, которые расширяют функциональные возможности Python. Если, допустим, вам нужно построить статистический график или воспроизвести звук, то такого рода задачи требуют применения одной или нескольких библиотек.

СОВЕТ

Один из наилучших сайтов для поиска библиотек в Интернете — UsefulModules:

`https://wiki.python.org/moin/UsefulModules`

Найти библиотеки можно и в других местах. Например, в статье "7 Python Libraries you should know about", доступной по указанному ниже адресу, приводится относительно полное описание семи библиотек:

http://www.lleess.com/2013/03/7-pythonlibraries-you-should-know-about.html

Есть сайты, посвященные конкретным платформам, например Unofficial Windows Binaries for Python Extension Packages:

http://www.lfd.uci.edu/~gohlke/pythonlibs/

По большому счету списки библиотек можно найти где угодно.

Задача данной главы не сводится к тому, чтобы расширить уже имеющийся у вас список библиотек. Вашему вниманию предлагаются десять библиотек, которые работают на всех платформах и предлагают базовые средства, необходимые любому пользователю. Будем считать, что эта глава — источник сведений об основных группах библиотек, которые вам понадобятся в будущем.

Создание безопасной среды с помощью PyCrypto

Защита данных — важный аспект программирования. Приложения полезны тем, что позволяют легко манипулировать данными любых типов. Поэтому приложение должно защищать данные, с которыми работает, иначе все усилия будут напрасными. Именно данные имеют наибольшую ценность для бизнеса, а приложение — это всего лишь инструмент. Защита данных предполагает предотвращение кражи или несанкционированного использования данных. Здесь на помощь придут криптографические библиотеки типа PyCrypto, которая доступна по следующему адресу:

https://www.dlitz.net/software/pycrypto/

ЗАПОМНИ!

Основная задача этой библиотеки — преобразование данных таким образом, чтобы ни у кого не было возможности получить к ним доступ до тех пор, пока они находятся в долговременном хранилище. Такой способ целенаправленной модификации данных называется *шифрованием*. При считывании данных в память подпрограмма-*дешифровщик* преобразует зашифрованные данные в их оригинальную форму, чтобы дать возможность приложению поработать с ними. Главным элементом этого процесса является *ключ*, используемый для шифрования и дешифрования. Обеспечение безопасности ключа также должно учитываться при разработке приложения. Пользователь, у которого есть ключ, сможет прочитать данные, но кроме него никто не получит к ним доступа.

Взаимодействие с базами данных с помощью SQLAlchemy

База данных — это способ хранения повторяющихся или структурированных данных на диске. К примеру, записи (отдельные строки в базе данных) клиентов относятся к разряду повторяющихся в силу одинаковой структуры информации по каждому клиенту (имя, адрес, номер телефона и т.п.). Точный способ хранения данных зависит от типа используемой базы данных. Одни базы данных хранят информацию в текстовом виде, другие — в табличном, третьи — в неструктурированном. Для этого может использоваться древовидная структура или плоский файл. В процессе знакомства с технологией СУБД (система управления базами данных) вы столкнетесь с множеством новых, незнакомых терминов. Большая их часть понятна только администраторам баз данных и не должна волновать рядового пользователя.

ЗАПОМНИ!

Самый распространенный тип баз данных — это реляционные СУБД. В них используются таблицы, содержащие записи и поля. Каждое поле — это часть столбца с однотипной информацией, например с именем клиента. Таблицы связываются между собой разными способами, в результате чего возможно создание сложных межтабличных отношений. Например, в таблице заказов у каждого клиента может быть одна или несколько записей, поэтому таблица клиентов и таблица заказов взаимосвязаны.

В реляционных СУБД используется SQL (Structured Query Language — язык структурированных запросов), который обеспечивает доступ к отдельным записям. Для работы с СУБД нужны специальные инструменты, и здесь на помощь приходит библиотека SQLAlchemy (`www.sqlalchemy.org`). Благодаря ей упрощается создание различных запросов к базам данных, таких как получение данных конкретного клиента, создание записи для нового клиента, обновление записи существующего клиента и удаление записи о старом клиенте.

Смотрим на мир с помощью Google Maps

Геокодирование (нахождение географических координат, таких как широта и долгота, на основе географических данных, таких как адрес объекта) в современном мире имеет множество применений. Эту информацию можно использовать, например, для поиска ближайшего ресторана, пропавшего в горах

альпиниста и т.п. Для работы с географическими данными предназначено приложение Google Maps, доступное по следующему адресу:

```
https://pypi.python.org/pypi/googlemaps/
```

Google Maps используется не только для прокладывания маршрутов, но и в геоинформационных системах (Geographic Information System — GIS). По большому счету задача GIS сводится к поиску местоположения того или иного объекта и анализу карт местности. Другими словами, Google Maps позволяет приложению видеть внешний мир.

Добавление графического интерфейса пользователя с помощью TkInter

Пользователи предпочитают работать с графическим интерфейсом в силу его удобства и простоты по сравнению с интерфейсом командной строки. Графические интерфейсы в Python можно создавать с помощью различных продуктов. Один из наиболее популярных инструментов — TkInter, доступный по следующему адресу:

```
https://wiki.python.org/moin/TkInter
```

Этот пакет полюбился разработчикам за простоту. В действительности это интерфейс к библиотеке Tcl/Tk (Tool Command Language — Tool kit), которая доступна на сайте `www.tcl.tk`. Во многих языках Tcl/Tk служит основой для создания графического интерфейса пользователя.

СОВЕТ

Вряд ли вы воздержитесь от соблазна добавить графический интерфейс в свое приложение. Пользователям нравятся графические интерфейсы, и если вы хотите, чтобы ваше приложение завоевало популярность, вам придется учитывать пожелания пользователей. Но помните о том, что разработка таких интерфейсов — это длительный процесс, а кроме того, производительность графического приложения может оказаться ниже.

Создание презентации с табличными данными с помощью PrettyTable

Табличные данные важно отображать так, чтобы они были понятны пользователям. Из примеров, приведенных в книге, вы уже знаете, что табличные

данные хранятся в формате, удобном для целей программирования. Однако пользователям нужно что-то наглядное и визуально привлекательное. Библиотека PrettyTable, которая доступна по адресу `https://pypi.python.org/pypi/PrettyTable`, позволяет создавать красивые табличные презентации в приложениях командной строки.

Добавление звука с помощью PyAudio

Звук — это удобный способ передачи информации определенного рода. Но его следует применять с осторожностью, поскольку глухонемые его не слышат, а тех, кто слышит, обилие звукового сопровождения зачастую просто раздражает. Как бы там ни было, звук порой оказывается важным средством коммуникации (или просто добавляет в приложение определенную изюминку).

Одной из наилучших межплатформенных библиотек, предназначенных для работы со звуком в приложениях Python, является PyAudio:

`http://people.csail.mit.edu/hubert/pyaudio/`

Эта библиотека позволяет записывать и воспроизводить звуки. Например, можно записать аудиозаметки для выполняемых задач и затем при необходимости воспроизвести список заметок.

МЕТОДИКИ РАБОТЫ СО ЗВУКОМ В PYTHON

Со звуком можно работать по-разному. Основные мультимедийные службы Python (см. документацию, доступную по адресу `https://docs.python.org/3/library/mm.html`) реализуют базовую функциональность воспроизведения звука. Можно также записывать определенные типы аудиофайлов, хотя выбор файловых форматов ограничен. Кроме того, некоторые пакеты типа `winsound` (`https://docs.python.org/3/library/winsound.html`) являются платформенно-зависимыми, из-за чего не могут использоваться в универсальных приложениях.

Более продвинутый вариант, предлагающий расширенную функциональность работы со звуком, — библиотеки типа PyAudio. Перечень таких библиотек доступен по следующей ссылке:

`https://wiki.python.org/moin/Audio`

Эти библиотеки обычно ориентируются на коммерческие приложения, обеспечивая, например, запись и воспроизведение голосовых заметок. В их задачу не входит воспроизведение высококачественного звука.

Для геймеров важна поддержка звуковых эффектов, таких как шаги приближающегося монстра. Соответствующие инструменты содержатся в библиотеках типа PyGame (`http://www.pygame.org/news.html`). Их использование предполагает наличие высококлассного оборудования, а кроме того, вам придется потратить немало времени на разработку аудиофункций приложения. Перечень таких библиотек можно найти по следующему адресу:

`https://wiki.python.org/moin/PythonGameLibraries`

СОВЕТ

Работа со звуком имеет свои плюсы и минусы. Например, платформенно-независимая библиотека не сможет задействовать специальные инструменты, предлагаемые той или иной платформой. Более того, она может не поддерживать многие файловые форматы, используемые конкретными платформами. Задача платформенно-независимой библиотеки сводится к обеспечению базовой поддержки звука на всех системах, с которыми она может взаимодействовать.

Построение графиков с помощью PyQtGraph

Мы привыкли воспринимать информацию визуально. Если пользователю сначала показать данные в виде таблицы, а затем — в виде графика, то ему больше запомнится второй вариант. Благодаря графикам люди видят тренды и понимают, почему данные меняются в ту или иную сторону. Тем не менее передать на экране табличную информацию в виде графиков не так-то просто. Для этого применяется библиотека PyQtGraph (`www.pyqtgraph.org`).

Несмотря на то что библиотека предназначена для выполнения инженерных, математических и научных вычислений, ее можно применять и в других целях. PyQtGraph поддерживает вывод двух- и трехмерных диаграмм, и с ее помощью можно строить графики на основе числовых таблиц. Результат полностью интерактивен, поэтому пользователь может выбрать область диаграммы для редактирования. Кроме того, библиотека располагает широким набором разнообразных графических элементов, или *виджетов* (например, кнопок, которые можно отображать на экране), упрощающих процесс кодирования.

ЗАПОМНИ!

В отличие от других библиотек, представленных в этой главе, PyQtGraph не является автономной. Это означает, что для ее использования должны быть установлены другие приложения, что неудивительно, поскольку PyQtGraph выполняет большой объем вычислений. Библиотека требует наличия следующих компонентов:

- Python версии 2.7 или выше;
- PyQt версии 4.8 или выше (`https://wiki.python.org/moin/PyQt`) либо PySide (`https://wiki.python.org/moin/PySide`);
- numpy (`http://www.numpy.org/`);
- scipy (`http://www.scipy.org/`);
- PyOpenGL (`http://pyopengl.sourceforge.net/`).

Поиск информации с помощью IRLib

Когда объем информации достигает внушительных размеров, найти ее бывает очень трудно. Представьте, что жесткий диск — это огромная древовидная база данных свободной структуры, в которой отсутствует индексация. При расширении и увеличении такой структуры данные в ней просто теряются. (Попробуйте, например, найти на диске фотографии с прошлогоднего отпуска, и вы поймете, о чем идет речь.) Поэтому важно встраивать в приложение возможности поиска, чтобы пользователи могли находить файлы или нужные данные.

ВНИМАНИЕ!

Для Python доступен целый ряд поисковых библиотек. Проблема в том, что их установка требует определенных усилий, и к тому же они не обеспечивают достаточную платформенную поддержку. По сути, некоторые из них работают только на одной или двух платформах. Тем не менее библиотека IRLib (`https://github.com/gr33ndata/irlib`) написана исключительно на Python, а это гарантия того, что она будет работать на всех платформах. Если ее окажется недостаточно, удостоверьтесь, что новый продукт сможет обеспечить необходимую поисковую функциональность на всех выбранных платформах и что его установка не потребует больших усилий.

Принцип работы IRLib заключается в создании поискового индекса для любого типа информации, с которой вы хотите работать. Этот индекс можно сохранить на диске для последующего использования. Поисковый механизм работает на основе метрик — записей, наиболее точно соответствующих критериям поиска.

Создание интероперабельной среды Java с помощью JPype

Python предоставляет доступ к огромному массиву библиотек, которых должно быть более чем достаточно. Но что если вы хотите использовать библиотеку Java, которая идеально подходит для ваших задач? Напрямую обратиться к ней из приложения Python невозможно. В таком случае на помощь придет библиотека JPype (`http://jpype.sourceforge.net/`), позволяющая получить доступ ко многим (но не ко всем) библиотекам Java непосредственно из Python. Она создает своего рода мост между двумя языками на уровне байт-кода. Вам не придется делать ничего сверхъестественного для работы с Java.

ПРЕОБРАЗОВАНИЕ ПРИЛОЖЕНИЯ PYTHON В JAVA

Добиться интероперабельности двух языков можно разными способами. Один из них — создание моста, например с помощью JPype. Другой способ — преобразование кода, созданного для одного языка, в код другого языка. Этот подход применяется в Jython (`https://wiki.python.org/jython/`). Данная утилита преобразует код Python в код Java, позволяя задействовать в приложении весь функционал Java и в то же время сохранять все полезные наработки Python.

Какое бы решение вы ни выбрали, вам придется столкнуться с определенными ограничениями. Например, в JPype нет доступа к некоторым библиотекам Java. Более того, при использовании этой библиотеки выполнение приложения может сильно замедлиться, поскольку мост JPype постоянно преобразует вызовы и данные. Проблема, присущая Jython, заключается в том, что вы теряете возможность модифицировать код после преобразования. Любые изменения будут вызывать несовместимость между оригинальным кодом Python и его Java-аналогом. Другими словами, не существует идеального решения по совмещению всего самого лучшего из двух языков в одном приложении.

Доступ к локальным сетевым ресурсам с помощью Twisted Matrix

В зависимости от настроек сети может потребоваться доступ к файлам и другим ресурсам, который нельзя получить с помощью встроенных инструментов платформы. В таком случае воспользуйтесь библиотекой Twisted Matrix

(`https://twistedmatrix.com/trac/`). Она содержит функции, позволяющие устанавливать подключения независимо от типа используемого протокола.

Особенностью библиотеки является то, что она управляется событиями. Это означает, что приложение не "зависает" в ожидании ответа сети. Более того, поддержка событий позволяет реализовывать асинхронное взаимодействие (при котором запрос отправляется одной процедурой, а затем обрабатывается совершенно другой).

Доступ к интернет-ресурсам с помощью библиотек

Несмотря на то что такие продукты, как Twisted Matrix, позволяют обрабатывать сетевые запросы, оптимальным решением при работе в Интернете является получение специализированной библиотеки протокола HTTP. Это объясняется тем, что специализированная библиотека производительнее и функциональнее. Если требуется явная поддержка HTTP или HTTPS, воспользуйтесь библиотекой `httplib2`, доступной по следующему адресу:

```
https://github.com/jcgregorio/httplib2
```

Эта библиотека, написанная исключительно на Python, существенно упрощает обработку таких HTTP-запросов, как Keep-Alive. (Keep-Alive — это значение, определяющее, как долго должен быть открыт порт в ожидании ответа, чтобы приложению не приходилось каждый раз повторно создавать соединение, тратя при этом ресурсы и время.) Она обеспечивает полную поддержку запросов `GET` и `POST`, а также стандартных методов сжатия, применяемых при передаче данных в Интернете, таких как `deflate` и `gzip`.

Предметный указатель

A

Anaconda 76
  кеш пакетов 223
  утилита Prompt 233, 245
ASCII 252

C

C# 33
CRUD 53, 335

F

FIFO 291

G

Google Maps 399

H

HTML 367
HTTP 405

I

IDE 54, 73, 378
IDLE 54, 74, 378
IPython 116, 393
IRLib 403

J

Java 33
JPype 404
Jupyter Notebook 73, 83, 101
  закрытие 98
  справочная система 114
Jython 404

K

Komodo Edit 391

L

LIFO 291
localhost 357

M

Mercurial 395
MIME 362

N

NumPy 30

P

Perl 34
PIL 377
POP3 356
PrettyTable 400
PyAudio 401
PyCrypto 398
Pydoc 244
PyInstaller 389
PyQtGraph 402
PyUnit 393

R

Roundup Issue Tracker 386

S

SciPy 30
SMTP 354
  сервер 366
SOAP 377
SQLAlchemy 399

T

TkInter 400

U

Unicode 382

V

VirtualEnv 387

X

XML 30, 379

А

Аргумент 56, 147
  именованный 149
  позиционный 148
  список 321
Арифметическая операция 137
Атрибут 220, 241

Б

Баг 74
База данных 399
Байт-код 392
Библиотека 219, 397
Блок кода 158, 220
Блокнот 76
  импорт 89
  создание 86
  удаление 89
  уровень доверия 108
  экспорт 87
Булево значение 130

В

Версия Python 35
Виджет 378, 402
Виртуальная среда 387
Вложенная инструкция 166
Выражение 135

Г

Геокодирование 399
Геттер 326
Грамотное программирование 76

Д

Дата и время 131
Двухсторонняя очередь 305
Дерево выбора 166
Документация 374
  строка 390

И

Императивное программирование 61
Импорт 219
Индекс 275
  отрицательный 276
Инструкция
  break 174
  continue 176
  from...import 225, 228
  if 156
  if...elif 163
  if...else 161
  import 225, 226
  pass 177
  raise 209
  return 151
  switch 297
  вложенная 166
  составная 161
Интегрированная среда разработки 38
Исключение 159, 185
  аргументы 198
  вложенное 206
  встроенное 192
  генерирование 208
  иерархическая обработка 203
  настраиваемое 212
  перехват 191

К

Кеш пакетов 223
Класс 310
  дочерний 329
  родительский 329
  создание 325
  сохранение на диске 326
Ключ словаря 294
Коллекция 289
Команда 59
Командная строка 54
  выход 69
Комментарий 95
  многострочный 97
Комплексное число 128
Конкатенация 95, 258

Конструктор 317
Контрольная точка 103
  восстановление 104
  сохранение 104
Кортеж 291

М

Магическая функция 116
Мантисса 128
Метод 312
  аргумент self 316
  класса 314
  экземпляра 316
Модуль 219

Н

Наследование 311, 328
Номер строки 110

О

Оболочка 69
Объект 312
  Counter 285
Объектно-ориентированное
      программирование 61
Операнд 136
Оператор 136
  арифметический 137
  логический 139, 159
  перегрузка 312, 323
  побитовый 139
  позиционного расширения 284
  приоритет 142
  присваивания 140
  сравнения 138, 157
  тернарный 136
  тождественности 142
  унарный 137
  членства 141, 280
Отладка 75
Отладчик 187, 392
Отступы 94
Очередь 291, 303
  двухсторонняя 305
Ошибка 187
  выполнения 189
  компиляции 188
  логическая 191
  семантическая 191
  синтаксическая 190

П

Пакет 219
  встроенный 222
  документация 244
  загрузка 232
  импорт 225
  кеш 223
  обновление 237
  поиск 231
  пользовательский 223
  просмотр содержимого 240
  удаление 238
  установка 235
Папка 336
Переменная 124, 135
  класса 310, 319
  среды 58
    ERRORLEVEL 69
  экземпляра 311, 321
Перехват исключения 186
Платформа 35
Побитовая операция 139
Повторение 258
Подсветка синтаксиса 379
Порт 356, 358
Порядок числа 128
Последовательность 270, 290
Предложение 161
  else 178
  except 195
  finally 213
Приложение 23
Присваивание 140
Производительность 382
Процедура 23
Процедурное программирование 61
Путь 337

Р

Расширение 336
Репозиторий кода 84

С

Сеттер 326
Система счисления 126
Словарь 290, 294
  ключ 294
Сокет 357
Спагетти-код 309
Специальный символ 254
Спецификация формата 265
Список 270
  изменение 276
  индекс 275
  печать 283
  повторение элементов 280
  поиск 280
  получение доступа 274
  создание 272
  сортировка 282
  циклический обход 276
Справочная система 62
Сравнение 138, 157
Среда разработки 38
Стек 291, 299
Строка 130, 253
  документации 390
  обработка 259
  поиск значения 262
  форматирование 265

Т

Тег 113
Тернарный оператор 136
Тестирование 393
Тип данных 125
  bool 130
  float 127
  int 126
  с плавающей точкой 127
  целочисленный 126
Тождественность 142

У

Указатель вставки 255
Унарный оператор 137
Управление версиями 395
Управляющая последовательность 255
Управляющий символ 255
Утилита 385
  conda 223, 233
  isort 394
  pdoc 390
  pip 233, 240
  pydbgr 392

Ф

Файл 336
  изменение содержимого 347
  создание 341
  удаление 351
  чтение содержимого 344
Функциональное программирование 61
Функция 143
  append() 277, 306
  appendleft() 306
  capitalize() 259
  center() 259
  clear() 277
  copy() 277
  count() 262, 290
  dir() 273
  empty() 304
  endswith() 263
  exit() 69
  expandtabs() 259
  extend() 277, 306
  extendleft() 306
  find() 263
  float() 130
  format() 265
  full() 304
  get() 304
  help() 63
  index() 263, 290
  input() 153
  insert() 277
  int() 130

isalnum() 259
isalpha() 259
isdecimal() 259
isdigit() 260
islower() 260
isnumeric() 260
isspace() 260
istitle() 260
isupper() 260
join() 260, 285
len() 260, 278
ljust() 260
lower() 260
lstrip() 260
max() 260
min() 260
open() 342
ord() 130
os.exit() 70
os.remove() 351
os.rmdir() 351
os.rmtree() 351
pop() 277, 306
popleft() 306
put() 304
quit() 69
range() 183
remove() 277, 306
replace() 263
reverse() 283
rfind() 263
rindex() 263
rjust() 260, 284
rstrip() 261
sort() 283
split() 232, 261
splitlines() 261
startswith() 264
str() 130
strip() 261
swapcase() 261
sys.exit() 70
title() 261
type() 129
upper() 261
zfill() 261

аргумент 147
возврат значения 150
определение 145
перегрузка 311

Х

Хост 357

Ц

Цикл 172
for 172
while 179
бесконечный 180
вложенный 182

Ч

Число
комплексное 128
с плавающей точкой 127
целое 126
Членство 141

Ш

Шифрование 398

Э

Экземпляр 311
Экспонента 128
Экспоненциальная запись 127
Электронная почта 353

Ю

Юзабилити 388

Я

Ядро 113
Язык программирования 26
Ячейка 90
выполнение содержимого 107
добавление 105
документирующая 92
объединение 106
перемещение 106
разделение 106

BESTSELLING BOOK SERIES

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Python для чайников®

2-е издание

Шпаргалка

Основные операторы Python

Оператор	Тип	Описание	Пример
-	Арифметический	Вычитание правого операнда из левого	`5 - 2 = 3`
-	Унарный	Выполняет операцию отрицания для исходного значения, в результате чего положительные значения становятся отрицательными и наоборот	Результат выражения `-(-4)` будет равен 4, а выражения `-4` — –4
-=	Присваивание	Вычитание значения правого операнда из значения левого операнда и присваивание результата левому операнду	`MyVar -= 2`
!=	Сравнение	Проверяет, будут ли два значения не равны. В некоторых устаревших версиях Python вместо оператора != допускается использовать оператор <>. В современных версиях Python это приведет к ошибке	`1! = 2` дает `True`
%	Арифметический	Остаток от деления левого операнда на правый	`5 % 2 = 1`
%=	Присваивание	Деление значения левого операнда на значение правого операнда и присваивание остатка от деления левому операнду	`MyVar %= 2`
& (И)	Побитовый	Выполняет поразрядное сравнение двух операндов, устанавливая бит результата в 1, если оба исходных бита равны 1	`0b1100 & 0b0110 = 0b0100`
*	Арифметический	Умножение правого операнда на левый	`5 * 2 = 10`
**	Арифметический	Возведение левого операнда в степень, задаваемую правым операндом	`5 ** 2 = 25`
**=	Присваивание	Возведение значения левого операнда в степень, заданную правым операндом, и присваивание результата левому операнду	`MyVar **= 2`
*=	Присваивание	Умножение значения правого операнда на значение левого операнда и присваивание результата левому операнду	`MyVar *= 2`
/	Арифметический	Деление левого операнда на правый	`5 / 2 = 2.5`
//	Арифметический	Целочисленное деление левого операнда на правый (с округлением вниз)	`5 // 2 = 2`
//=	Присваивание	Деление значения левого операнда на значение правого операнда и присваивание целочисленного результата левому операнду	`MyVar //= 2`
/=	Присваивание	Деление значения левого операнда на значение правого операнда и присваивание результата левому операнду	`MyVar /= 2`

BESTSELLING BOOK SERIES

Python для чайников®

2-е издание

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Оператор	Тип	Описание	Пример
^ (исключающее ИЛИ)	Побитовый	Выполняет поразрядное сравнение двух операндов, устанавливая бит результата в 1, если ровно один из исходных битов равен 1. Если же оба бита равны 1 или оба бита равны 0, то результирующий бит устанавливается равным 0	`0b1100 ^ 0b0110 = 0b1010`
| (ИЛИ)	Побитовый	Выполняет поразрядное сравнение двух операндов, устанавливая бит результата в 1, если один из исходных битов равен 1	`0b1100 | 0b0110 = 0b1110`
~	Унарный	Инвертирует биты числа, в результате чего все биты 0 превращаются в биты 1 и наоборот	Результат выражения ~4 будет равен –5
~ (НЕ)	Побитовый	Инвертирует все разряды операнда	`~0b1100 = -0b1101` `~0b0110 = -0b0111`
+	Арифметический	Сложение двух операндов	`5 + 2 = 7`
+	Унарный	Поддерживается исключительно ради полноты. Возвращает значение операнда	Результат выражения +4 будет равен 4
+=	Присваивание	Добавление значения правого операнда к значению левого операнда и присваивание результата левому операнду	`MyVar += 2`
<	Сравнение	Проверяет, будет ли значение левого операнда меньше значения правого операнда	`1 < 2` дает `True`
<< (сдвиг влево)	Побитовый	Сдвигает биты левого операнда влево на число позиций, задаваемых правым операндом. Все новые биты устанавливаются равными 0, а все отсекаемые биты теряются	`0b00110011 << 2 = 0b11001100`
<=	Сравнение	Проверяет, будет ли значение левого операнда меньше или равно значению правого операнда	`1 <= 2` дает `True`
=	Присваивание	Присваивание левому операнду значения правого операнда	`MyVar = 5`
==	Сравнение	Проверяет, будут ли два значения равны. Обратите внимание на удвоение знака равенства. Ошибка многих новичков заключается в использовании одного знака равенства, что приводит к присваиванию одного значения другому	`1 == 2` дает `False`
>	Сравнение	Проверяет, будет ли значение левого операнда больше значения правого операнда	`1 > 2` дает `False`
>=	Сравнение	Проверяет, будет ли значение левого операнда больше или равно значению правого операнда	`1 >= 2` дает `False`

BESTSELLING BOOK SERIES

Python для чайников®

2-е издание

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Оператор	Тип	Описание	Пример
>> (сдвиг вправо)	Побитовый	Сдвигает биты левого операнда вправо на число позиций, задаваемых правым операндом. Все новые биты устанавливаются равными 0, а все отсекаемые биты теряются	`0b00110011 >> 2 = 0b00001100`

`and`	Логический	Проверяет, являются ли оба операнда истинными (`True`)	`True and True` равно `True`
			`True and False` равно `False`
			`False and True` равно `False`
			`False and False` равно `False`

`in`	Членство	Проверяет, содержится ли значение левого операнда в последовательности, заданной правым операндом	`"Привет" in "Привет Пока"` равно `True`
`is`	Тождественность	Возвращает `True`, если тип значения или выражения правого операнда соответствует типу значения или выражения левого операнда	`type(2) is int` равно `True`
`is not`	Тождественность	Возвращает `True`, если тип значения или выражения правого операнда не соответствует типу значения или выражения левого операнда	`type(2) is not int` равно `False`
`not`	Логический	Инвертирует булево значение операнда: истинное значение становится ложным, а ложное — истинным	`not True` равно `False` `not False` равно `True`
`not in`	Членство	Проверяет, отсутствует ли значение левого операнда в последовательности, заданной правым операндом	`"Привет" not in "Привет Пока"` равно `False`

`or`	Логический	Проверяет, является ли один из операндов истинным (`True`)	`True or True` равно `True`
			`True or False` равно `True`
			`False or True` равно `True`
			`False or False` равно `False`

BESTSELLING BOOK SERIES

Python для чайников®

2-е издание

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Приоритет операторов Python

Оператор	Описание
`()`	С помощью скобок можно группировать выражения и переопределять заданный по умолчанию приоритет. В результате операция с более низким приоритетом (например, сложение) получает преимущество над операцией с более высоким приоритетом (такой, как умножение)
`**`	Возведение левого операнда в степень, задаваемую правым операндом
`~ + -`	Унарные операторы применяются к одной переменной или одному выражению
`* / % //`	Умножение, деление, остаток от деления (деление по модулю) и целочисленное деление
`+ -`	Сложение и вычитание
`>> <<`	Побитовый сдвиг вправо и влево
`&`	Побитовое И
`^ |`	Побитовое исключающее ИЛИ и простое ИЛИ
`<= < > >=`	Операторы сравнения
`== !=`	Операторы равенства
`= %= /= //= -= += *= **=`	Операторы присваивания
`is` `is not`	Операторы тождественности
`in` `not in`	Операторы членства
`not or and`	Логические операторы

BESTSELLING BOOK SERIES

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Python для чайников®

2-е издание

Шпаргалка

Распространенные магические функции в Jupyter Notebook и IPython

Магическая функция	Поддерживает ли статус при отдельном вводе	Описание
`%alias`	Да	Назначает или отображает псевдоним для системной команды
`%autocall`	Да	Позволяет вызывать функции без ввода скобок. Имеет следующие настройки: `Off`, `Smart` (по умолчанию) и `Full`. В случае указания параметра `Smart` скобки нужны только при наличии аргумента
`%automagic`	Да	Позволяет вызывать магические функции без ввода знака процента (`%`). Имеет следующие настройки: `False` (по умолчанию) и `True`
`%autosave`	Да	Отображает или изменяет значение интервала между автоматическими сохранениями блокнота. По умолчанию сохранение происходит каждые 120 секунд
`%cd`	Да	Меняет текущий каталог. С помощью этой команды можно перемещаться по истории каталогов или переходить к закладке
`%cls`	Нет	Выполняет очистку экрана
`%colors`	Нет	Задает цветовую схему для отображения текста, который ассоциируется с подсказками, информационной системой и обработчиками исключений. Доступные варианты: `NoColor` (черно-белая схема), `Linux` (по умолчанию) и `LightBG`
`%config`	Да	Позволяет конфигурировать IPython
`%dhist`	Да	Выводит список каталогов, к которым выполнялось обращение в ходе текущего сеанса
`%file`	Нет	Выводит имя файла, содержащего исходный код данного объекта
`%hist`	Да	Отображает список магических функций, которые вызывались во время текущего сеанса
`%install_ext`	Нет	Устанавливает заданный модуль расширения
`%load`	Нет	Загружает код приложения из другого источника, например из Интернета
`%load_ext`	Нет	Загружает расширение Python с указанным именем модуля
`%lsmagic`	Да	Отображает список доступных в настоящий момент магических функций
`%magic`	Да	Отображает экран справки, содержащий информацию о магических функциях

Шпаргалка

Магическая функция	Поддерживает ли статус при отдельном вводе	Описание
`%matplotlib`	Да	Устанавливает модуль построения диаграмм, позволяя отображать графики внутри ячеек блокнота. Возможные значения: `'gtk'`, `'gtk3'`, `'inline'`, `'nbagg'`, `'osx'`, `'qt'`, `'qt4'`, `'qt5'`, `'tk'` и `'wx'`
`%paste`	Нет	Вставляет содержимое буфера обмена в командную строку IPython
`%pdef`	Нет	Показывает, каким образом вызывается данный объект (при условии, что объект является вызываемым)
`%pdoc`	Нет	Отображает строку документации объекта
`%pinfo`	Нет	Выводит подробную информацию об объекте (часто более подробную, чем предлагает справочная система)
`%pinfo2`	Нет	Выводит дополнительную информацию об объекте (если она доступна)
`%reload_ext`	Нет	Повторно загружает предварительно установленный модуль расширения
`%source`	Нет	Отображает исходный код объекта (при условии, что он доступен)
`%timeit`	Нет	Вычисляет наилучшую производительность инструкции
`%%timeit`	Нет	Вычисляет наилучшую производительность для всех инструкций ячейки, кроме той, которая находится в одной строке с магической функцией (т.е. является инструкцией инициализации)
`%unalias`	Нет	Удаляет из списка ранее созданный псевдоним
`%unload_ext`	Нет	Выгружает указанный модуль расширения
`%%writefile`	Нет	Записывает содержимое ячейки в указанный файл

Программирование на Python — это легко и увлекательно!

Python — это мощный язык программирования, на котором можно создавать самые разные приложения, не зависящие от платформы. Он идеально подходит для новичков, особенно если нужно быстро научиться программировать и начать создавать реальные проекты. Благодаря пошаговым инструкциям, приведенным в книге, вы сможете в краткие сроки освоить основы языка. Работая в среде Jupyter Notebook, вы будете применять принципы грамотного программирования для создания смешанного представления кода, заметок, математических уравнений и графиков.

В книге...

- Загрузка и установка интерпретатора Python
- Использование командной строки
- Знакомство со средой Jupyter Notebook
- Основы программирования на Python
- Создание коллекций и списков
- Взаимодействие с пакетами
- Поиск и устранение ошибок

Джон Пол Мюллер — писатель и технический редактор, автор более 100 книг и 600 статей, тематика которых варьируется от сетевых технологий и искусственного интеллекта до управления базами данных и программирования на Python. Как технический редактор помог издать книги более чем 70 авторам. Также занимается консультациями и написанием сертификационных экзаменов.

Изображение на обложке:
©depositphotos.com/84480656
Автор: andriano_cz

ДИАЛЕКТИКА
www.dialektika.com

Языки программирования/Python

ISBN 978-5-907144-26-2

для ЧАЙНИКОВ®