

Мэтт
Харрисон



КАК УСТРОЕНЫ PYTHON

ГИД ДЛЯ РАЗРАБОТЧИКОВ, ПРОГРАММИСТОВ И ИНТЕРЕСУЮЩИХСЯ



Illustrated Guide to Python 3

A Complete Walkthrough of Beginning Python with Unique Illustrations Showing how Python Really Works

Matt Harrison

Technical Editors: Roger A. Davidson, Andrew McLaughlin



БИБЛИОТЕКА
ПРОГРАММИСТА

Мэтт Харрисон

КАК УСТРОЕН PYTHON

ГИД ДЛЯ РАЗРАБОТЧИКОВ, ПРОГРАММИСТОВ И ИНТЕРЕСУЮЩИХСЯ



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

ББК 32.973.2-018.1
УДК 004.43
Х21

Харрисон Мэтт

Х21 Как устроен Python. Гид для разработчиков, программистов и интересующихся. — СПб.: Питер, 2019. — 272 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0906-7

Python в моде! Это самый популярный язык программирования. Вакансии для Python-разработчиков входят в список самых высокооплачиваемых, а благодаря бурному развитию обработки данных, знание Python становится одним из самых востребованных навыков в среде аналитиков.

Python — невероятный язык, популярный во многих областях. Он используется для автоматизации простых и сложных задач, цифровой обработки, веб-разработки, игр... Независимо от того, перешли ли вы на Python с другого языка, руководите группой программистов, работающих на Python, или хотите расширить свое понимание, имеет смысл подойти к изучению Python со всей серьезностью.

Готовы начать карьеру питониста? Не теряйте времени на поиск информации, перелопачивая блоги и сайты, списки рассылок и группы. Мэтт Харрисон использует Python с 2000 года. Он занимался научными исследованиями, сборкой и тестированием, бизнес-аналитикой, хранением данных, а теперь делится своими знаниями как с простыми пользователями, так и с крупными корпорациями. Приобщитесь к передовому опыту и узнайте секреты внутренней кухни Python, доступные только профи, работающим с этим языком на протяжении многих лет.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1977921758 англ.
ISBN 978-5-4461-0906-7

© 2017 Matt Harrison
© Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Библиотека программиста», 2019

Краткое содержание

Предисловие	13
От издательства	14
Глава 1. Почему Python?.....	15
Глава 2. Какая версия Python?.....	17
Глава 3. Интерпретатор	21
Глава 4. Запуск программ.....	28
Глава 5. Запись и чтение данных	35
Глава 6. Переменные	38
Глава 7. Подробнее об объектах	51
Глава 8. Числа	62
Глава 9. Строки.....	71
Глава 10. dir, help и pdb	80
Глава 11. Строки и методы	85
Глава 12. Комментарии, логические значения и None	96
Глава 13. Условия и отступы	105
Глава 14. Контейнеры: списки, кортежи и множества	113
Глава 15. Итерации.....	130
Глава 16. Словари.....	140
Глава 17. Функции	152
Глава 18. Индексирование и срезы	163
Глава 19. Операции ввода/вывода с файлами.....	169

Глава 20. Юникод	180
Глава 21. Классы	192
Глава 22. Создание подклассов.....	211
Глава 23. Исключения.....	219
Глава 24. Импортирование библиотек.....	235
Глава 25. Библиотеки: пакеты и модули	244
Глава 26. Полноценный пример	250
Глава 27. В начале пути.....	264
Приложение А. Перемещение по файлам.....	265
Приложение Б. Полезные ссылки	267
Об авторе.....	268
Научные редакторы.....	269

Оглавление

Предисловие	13
От издательства	14
Глава 1. Почему Python?.....	15
Глава 2. Какая версия Python?.....	17
2.1. Установка Python	17
2.2. Какой редактор?	19
2.3. Итоги	19
2.4. Упражнения	20
Глава 3. Интерпретатор	21
3.1. REPL	22
3.2. Пример использования REPL.....	24
3.3. Итоги	27
3.4. Упражнения	27
Глава 4. Запуск программ.....	28
4.1. Запуск программ из IDLE	29
4.2. Усовершенствования для UNIX.....	31
4.3. Итоги	33
4.4. Упражнения	34
Глава 5. Запись и чтение данных	35
5.1. Простой вывод.....	35
5.2. Получение данных от пользователя.....	36
5.3. Итоги	37
5.4. Упражнения	37
Глава 6. Переменные	38
6.1. Изменение и состояние.....	38
6.2. Переменные Python как метки.....	39
6.3. Бирки.....	41
6.4. Повторное связывание переменных	43
6.5. Имена переменных	45
6.6. Дополнительные рекомендации по назначению имен	46
6.7. Итоги	49
6.8. Упражнения	50

Глава 7. Подробнее об объектах	51
7.1. Идентификатор	51
7.2. Тип	53
7.3. Изменяемость	55
7.4. Использование IDLE	57
7.5. Итоги	60
7.6. Упражнения	61
Глава 8. Числа	62
8.1. Сложение	63
8.2. Вычитание	65
8.3. Умножение	65
8.4. Деление	66
8.5. Остаток	66
8.6. Возведение в степень	68
8.7. Порядок операций	69
8.8. Другие операции	69
8.9. Итоги	69
8.10. Упражнения	70
Глава 9. Строки	71
9.1. Форматирование строк	74
9.2. Синтаксис форматных строк	74
9.3. Примеры форматирования	77
9.4. F-строки	78
9.5. Итоги	79
9.6. Упражнения	79
Глава 10. dir, help и pdb	80
10.1. Специальные методы	81
10.2. help	82
10.3. pdb	82
10.4. Итоги	84
10.5. Упражнения	84
Глава 11. Строки и методы	85
11.1. Основные строковые методы	89
11.2. endswith	89
11.3. find	91
11.4. format	91
11.5. join	92
11.6. lower	93
11.7. startswith	93

11.8. strip.....	93
11.9. upper	94
11.10. Другие методы	94
11.11. Итоги	94
11.12. Упражнения	94
Глава 12. Комментарии, логические значения и None	96
12.1. Комментарии	96
12.2. Логические значения	97
12.3. None	101
12.4. Итоги	103
12.5. Упражнения	103
Глава 13. Условия и отступы	105
13.1. Объединение условных выражений.....	107
13.2. Команды if	109
13.3. Команды else.....	109
13.4. Множественный выбор.....	110
13.5. Пробелы	110
13.6. Итоги	112
13.7. Упражнения	112
Глава 14. Контейнеры: списки, кортежи и множества	113
14.1. Списки	113
14.2. Индексы.....	114
14.3. Вставка в список	115
14.4. Удаление из списка	116
14.5. Сортировка списков	116
14.6. Полезные советы по работе со списками.....	117
14.7. Кортежи	122
14.8. Множества	125
14.9. Итоги	128
14.10. Упражнения	128
Глава 15. Итерации	130
15.1. Перебор с индексом	131
15.2. Выход из цикла	134
15.3. Пропуск элементов в цикле.....	134
15.4. Оператор in может использоваться для проверки принадлежности	135
15.5. Удаление элементов из списков при переборе	135
15.6. Блок else	137
15.7. Циклы while.....	137
15.8. Итоги	139
15.9. Упражнения	139

Глава 16. Словари	140
16.1. Присваивание в словарях.....	140
16.2. Выборка значений из словаря.....	142
16.3. Оператор in.....	143
16.4. Сокращенный синтаксис словарей.....	144
16.5..setdefault.....	144
16.6. Удаление ключей.....	147
16.7. Перебор словаря.....	147
16.8. Итоги.....	150
16.9. Упражнения.....	150
Глава 17. Функции	152
17.1. Вызов функций.....	155
17.2. Область видимости.....	156
17.3. Множественные параметры.....	158
17.4. Параметры по умолчанию.....	159
17.5. Правила выбора имен для функций.....	161
17.6. Итоги.....	162
17.7. Упражнения.....	162
Глава 18. Индексирование и срезы	163
18.1. Индексирование.....	163
18.2. Срезы.....	164
18.3. Приращения в срезах.....	166
18.4. Итоги.....	168
18.5. Упражнения.....	168
Глава 19. Операции ввода/вывода с файлами	169
19.1. Открытие файлов.....	169
19.2. Чтение текстовых файлов.....	171
19.3. Чтение двоичных файлов.....	172
19.4. Перебор при работе с файлами.....	173
19.5. Запись файлов.....	174
19.6. Закрытие файлов.....	175
19.7. Проектирование на основе файлов.....	177
19.8. Итоги.....	178
19.9. Упражнения.....	178
Глава 20. Юникод	180
20.1. Историческая справка.....	180
20.2. Основные этапы в Python.....	183
20.3. Кодирование.....	185

20.4. Декодирование	187
20.5. Юникод и файлы.....	188
20.6. Итоги	190
20.7. Упражнения	190
Глава 21. Классы	192
21.1. Планирование класса.....	195
21.2. Определение класса	196
21.3. Создание экземпляра класса	201
21.4. Вызов метода	204
21.5. Анализ экземпляра.....	205
21.6. Приватный и защищенный доступ.....	206
21.7. Простая программа, моделирующая поток посетителей	207
21.8. Итоги	209
21.9. Упражнения	209
Глава 22. Создание подклассов.....	211
22.1. Подсчет остановок	214
22.2. super	215
22.3. Итоги	217
22.4. Упражнения	218
Глава 23. Исключения.....	219
23.1. «Посмотри, прежде чем прыгнуть»	220
23.2. «Проще просить прощения, чем разрешения»	221
23.3. Несколько возможных исключений	223
23.4. finally	225
23.5. Секция else	227
23.6. Выдача исключений	228
23.7. Упаковка исключений.....	229
23.8. Определение собственных исключений.....	232
23.9. Итоги	233
23.10. Упражнения	234
Глава 24. Импортирование библиотек.....	235
24.1. Способы импортирования	236
24.2. Конфликты имен при импортировании.....	239
24.3. Массовое импортирование	240
24.4. Вложенные библиотеки.....	241
24.5. Организация импортирования.....	241
24.6. Итоги	243
24.7. Упражнения	243

Глава 25. Библиотеки: пакеты и модули	244
25.1. Модули	244
25.2. Пакеты.....	244
25.3. Импортирование пакетов	245
25.4. PYTHONPATH.....	246
25.5. sys.path.....	247
25.6. Итоги	248
25.7. Упражнения	249
Глава 26. Полноценный пример	250
26.1. cat.py	250
26.2. Что делает этот код?.....	253
26.3. Типичная структура	254
26.4. #!	255
26.5. Строка документации.....	256
26.6. Импортирование	257
26.7. Метаданные и глобальные переменные.....	257
26.8. Операции с журналом	259
26.9. Другие глобальные переменные	259
26.10. Реализация	259
26.11. Тестирование	259
26.12. if __name__ == '__main__':	260
26.13. __name__	261
26.14. Итоги	262
26.15. Упражнения	263
Глава 27. В начале пути.....	264
Приложение А. Перемещение по файлам.....	265
А.1. Mac и UNIX	265
А.2. Windows	266
Приложение Б. Полезные ссылки	267
Об авторе.....	268
Научные редакторы.....	269

Предисловие

Готовы начать свою карьеру программиста Python? Эта книга вооружит вас знаниями, которые накапливались годами, и практическим опытом, представленными в простом и доступном формате. Вместо того чтобы месяцами мониторить блоги и сайты, искать информацию по спискам рассылки и группам, это пособие позволит программисту быстро освоиться с Python.

Программирование — интересное занятие, а с Python оно еще приносит и удовольствие. Базовый синтаксис Python не только прост, но и доступен для любого возраста. Я преподавал язык программирования Python ученикам младшей школы, подросткам, «отраслевым» профессионалам и пенсионерам. Если вы готовы читать и набирать код на клавиатуре, то стоите в начале прекрасного пути. Как далеко вы сможете зайти — зависит только от того, насколько серьезно вы готовы потрудиться.

Существует несколько уровней владения Python. Базовый синтаксис Python компактен и легко изучается. После того как вы освоите его, перед вами откроются все пути. Вы сможете читать разнообразный код Python и будете понимать его. С этого момента вы сможете изучать более сложные темы и конкретные инструментариумы, участвовать в проектах с открытым кодом на Python или использовать эти знания для изучения других языков программирования.

Мы рекомендуем применять следующий подход к изучению языка: прочитайте главу, сядьте за компьютер и введите примеры этой главы. Python позволяет легко взяться за программирование, избавляя разработчика от многих хлопот, связанных с запуском программ в других языках. У вас возникло искушение просто прочитать книгу? Если вы возьметесь за дело и будете действительно выполнять примеры на компьютере, вы узнаете намного больше, чем при простом чтении.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Почему Python?

Python в моде! Это самый популярный язык, которому учат в университетах. Вакансии для разработчиков Python входят в число самых высокооплачиваемых. Из-за бурного развития теории обработки данных знание Python быстро становится одним из самых желанных навыков для аналитиков. Операционные отделы также осваивают Python для управления подсистемами баз данных. Они осознают то, что давно известно веб-разработчикам, уже использующим Python, а именно то, что Python делает их работу более продуктивной.

В жизни Python наступил переломный момент. Его область применения уже не ограничивается небольшими, динамичными стартапами. Стремясь извлечь пользу из его мощи и эффективности, крупные предприятия также переходят на Python. За последний год я преподавал Python сотням матерых разработчиков с многолетним опытом работы в крупных компаниях, переходивших на Python.

Python повышает производительность программирования. Я перешел на Python из мира Perl. На работе меня включили в команду с коллегой, хорошо владевшим Tcl. Ни один из нас не хотел переходить в другой лагерь, хотя мы оба были заинтересованы в изучении Python. За три дня наш прототип был готов — намного быстрее, чем ожидалось, и мы оба моментально забыли свои предыдущие «goto-языки». Меня привлекло в Python прежде всего то, что этот язык казался мне абсолютно логичным. Я твердо уверен, что каждый, у кого есть хоть какой-то опыт программирования, сможет изучить основы Python всего за несколько дней.

Python легок в освоении. Для начинающих программистов Python станет отличным трамплином. Научиться писать простые программы очень легко, однако Python также хорошо масштабируется для сложных «корпоративных» систем. Наконец, Python подойдет для любого возраста — я сам видел, как люди в возрасте от 7 до 80+ лет изучали основы программирования на примере Python.

2

Какая версия Python?

Эта книга написана на основе Python 3. Версия Python 2 верно служила нам много лет. Фонд Python Software Foundation, управляющий выпуском новых версий, заявил, что эпоха Python 2 подошла к концу. Соответственно, после 2020 года язык поддерживаться не будет.

Версия Python 3 существует уже в течение некоторого времени, и, как выяснилось, она не обладает полной обратной совместимостью с линейкой 2. Если разработка начинается с нуля, беритесь за Python 3. Если вам приходится иметь дело с унаследованными системами, написанными на Python 2, не огорчайтесь. Большая часть материала книги идеально подходит для Python 2. Если же вы захотите сосредоточиться на Python 2, найдите предыдущее издание этой книги.

2.1. Установка Python

Python 3 не устанавливается по умолчанию на большинстве платформ. Некоторые дистрибутивы Linux включают Python 3, но пользователям Windows и Mac придется установить его отдельно.

Если вы используете Windows, откройте раздел загрузок на сайте Python¹ и найдите ссылку **Python 3.6 Windows Installer**. По ссылке загружается файл **.msi**, который устанавливает Python на машину с системой Windows. За-

¹ <https://www.python.org/download>

грузите файл, откройте его двойным щелчком и выполните инструкции, чтобы завершить установку.

ПРИМЕЧАНИЕ

В установочной программе для Windows имеется флажок «Add Python to PATH» (Добавить Python в переменную PATH). Проследите за тем, чтобы этот флажок был установлен. В этом случае при запуске из режима командной строки система будет знать, где найти исполняемый файл Python. Если флажок все же не будет установлен, откройте свойства системы (нажмите клавиши WIN+Pause или выполните команду `environ` из меню Пуск), откройте вкладку **Дополнительно** и щелкните на кнопке **Переменные среды**. Обновите переменную PATH и добавьте следующие пути:

`C:\Program Files\Python 3.6;C:\Program Files\Python 3.6\Scripts`

Если в вашей системе Windows включен механизм UAC (User Account Control), то путь будет выглядеть так:

`C:\Users\<имя_пользователя>\AppData\Local\Programs\Python\Python36`

Пользователи Mac загружают с сайта Python установочную программу для Mac.

ПРИМЕЧАНИЕ

Другой способ установки Python основан на использовании дистрибутива Anaconda¹. Он работает в Windows, Mac и Linux, а также предоставляет много заранее построенных двоичных файлов для выполнения научных вычислений. Традиционно устанавливать эти библиотеки было утомительно, потому что в них были упакованы библиотеки, написанные на C и Fortran, и это требовало дополнительной настройки для компиляции.

Пользователи Mac также могут присмотреться к Homebrew-версии². Если вы уже знакомы с Homebrew, проблема решается простой командой `brew install python3`.

¹ <https://www.anaconda.com/download/>

² <https://brew.sh>

2.2. Какой редактор?

Кроме установки Python вам понадобится текстовый редактор. В нем вы будете писать код. Настоящий мастер не жалеет времени на то, чтобы как следует изучить свои инструменты, и это время не пропадет даром. Умение пользоваться всеми возможностями текстового редактора упростит вашу работу. Во многих современных редакторах предусмотрена некоторая степень поддержки Python.

Если же вы только делаете первые шаги в изучении Python и у вас еще нет особого опыта в работе с текстовыми редакторами, в большинство установок Python включается среда разработки IDLE, которая работает в Windows, Mac и Linux.

При выборе редактора следует обратить внимание на интеграцию со средой Python REPL¹. Вскоре мы рассмотрим пример для IDLE. Желательно, чтобы выбранный вами редактор обладал сходной функциональностью.

Среди популярных редакторов с достойной поддержкой Python можно выделить Emacs, Vim, Atom, Visual Studio Code и Sublime Text. Если вас интересуют более мощные редакторы со встроенной поддержкой рефакторинга и автозавершения, обратите внимание на популярные PyCharm и Wing IDE.

2.3. Итоги

Python 3 — актуальная версия языка Python. Если только вы не работаете над унаследованным кодом, вам стоит отдать предпочтение именно этой версии. Новейшую версию можно загрузить на сайте Python.

Во многих современных редакторах реализована некоторая степень поддержки Python. Разные редакторы и среды разработки предоставляют разную функциональность. Если вы только начинаете осваивать программирование, опробуйте редактор IDLE. На первых порах это именно то, что нужно.

¹ REPL — сокращение от Read, Evaluate, Print Loop (цикл «чтение-вычисление-вывод»). Вскоре мы рассмотрим пример использования REPL.

2.4. Упражнения

1. Установите Python 3 на своем компьютере. Убедитесь в том, что Python успешно запускается.
2. Если вы привыкли работать в конкретном редакторе, узнайте, реализована ли в нем поддержка Python. В частности, умеет ли он:
 - автоматически выделять элементы синтаксиса Python;
 - выполнять код Python в REPL;
 - осуществлять пошаговое выполнение кода Python в отладчике.

3

Интерпретатор

Python традиционно относится к семейству *интерпретируемых* языков (другой термин для описания интерпретируемого языка — *язык сценариев*). Чтобы программа могла выполняться на компьютерном процессоре, она должна существовать в формате, понятном для этого процессора — а именно в *машинном коде*. Интерпретируемые языки не *компилируются* в машинный код напрямую; вместо этого в системе существует промежуточная прослойка — *интерпретатор*, — которая выполняет эту функцию.

У такого подхода есть как достоинства, так и недостатки. Как нетрудно понять, трансляция «на ходу» может занимать много времени. Интерпретируемый код — такой, как программы Python, — может работать в 10–100 раз медленнее программ на языке C. С другой стороны, написание кода на Python оптимизирует время разработки. Программы на языке Python нередко получаются в 2–10 раз короче своих аналогов на языке C. Кроме того, этап компиляции может занимать довольно много времени и отвлекать программиста между разработкой и отладкой.

Многие разработчики и компании охотно идут на этот компромисс. Небольшие программы (то есть содержащие меньше строк кода) быстрее пишутся и создают меньше проблем с отладкой. Труд программистов обходится дорого — если удастся переложить часть работы на оборудование, это может обойтись дешевле, чем привлечение дополнительных специалистов. Отладить 10 строк кода проще, чем отладить 100 строк кода. Исследования показали, что количество ошибок в коде пропорционально количеству строк. Следовательно, если язык позволяет написать меньше строк кода для решения некоторой задачи, то, скорее всего, программа

будет содержать меньше ошибок. Иногда скорость выполнения программы не столь важна, и во многих практических случаях Python работает достаточно быстро. Кроме того, были предприняты проекты, направленные на ускорение работы интерпретатора Python, например PyPy¹.



Рис. 3.1. Различия между компилируемым и интерпретируемым языком. Компилятор обрабатывает программный код и создает исполняемый файл. Интерпретатор создает исполняемый файл, который загружает программный код и управляет его выполнением

3.1. REPL

Для Python также существует *интерактивный интерпретатор*, который называется *REPL* (Read Evaluate Print Loop — цикл «чтение-вычисление-вывод»). REPL в цикле ожидает, пока появятся входные данные, читает

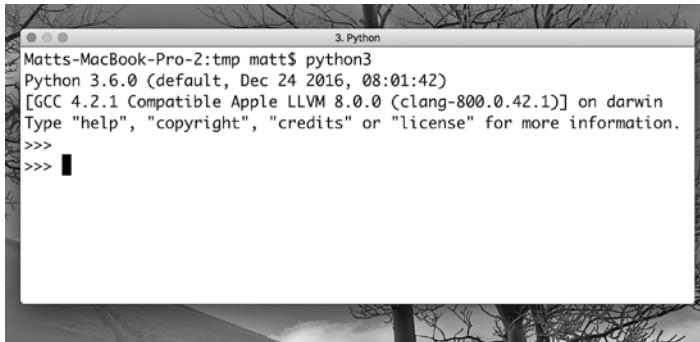
¹ <https://www.pypy.org>

и обрабатывает (интерпретирует) их, после чего выводит результат. Запуская исполняемый файл `python3`, вы запускаете интерактивный интерпретатор Python. Другие среды, например IDLE, также содержат встроенный интерактивный интерпретатор.

ПРИМЕЧАНИЕ

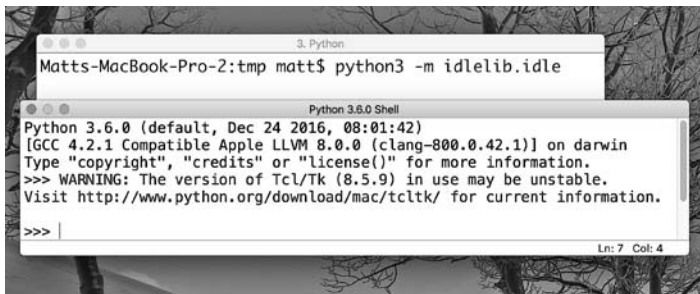
В этой книге Python 3 обычно запускается командой `python3`. В системе Windows исполняемому файлу присвоено имя `python`. Если вы работаете в Windows, замените имя `python3` именем `python`. В системе UNIX менять ничего не нужно.

При запуске интерпретатор выводит версию Python, информацию о сборке и несколько подсказок по использованию. Наконец, интерпретатор выдает приглашение `>>>`.



```
3. Python
Matts-MacBook-Pro-2:tmp matt$ python3
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> █
```

Рис. 3.2. Чтобы запустить REPL, введите в приглашении командной строки команду `python3`. Команда открывает сеанс Python



```
3. Python
Matts-MacBook-Pro-2:tmp matt$ python3 -m idlelib.idle
Python 3.6.0 Shell
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
>>> █
```

Рис. 3.3. Чтобы запустить REPL из IDE, щелкните на значке IDLE или введите команду `python3 -m idlelib.idle`

IDLE (редактор, включенный в поставку Python) также можно запустить командой `python3 -m idlelib.idle`.

ПРИМЕЧАНИЕ

Некоторые дистрибутивы Linux включают не все библиотеки из стандартной библиотеки Python. Это неприятно, но на то есть своя причина: на сервере не нужны библиотеки для создания клиентских приложений. По этой причине Ubuntu и Arch (среди прочих) в установке по умолчанию не включают библиотеки графического интерфейса, необходимые для IDLE.

Если вы увидите ошибку, которая выглядит примерно так:

```
$ python3 -m idlelib.idle
** IDLE can't import Tkinter.
Your Python may not be configured for Tk. **
```

это означает, что в системе отсутствует библиотека `tkinter`.

В Ubuntu следует выполнить команду:

```
$ sudo apt-get install tk-dev
```

В Arch эта команда выглядит так:

```
$ sudo pacman -S tk
```

3.2. Пример использования REPL

Следующий пример показывает, почему интерактивный интерпретатор REPL получил свое название. Введите команду `python3` в командной строке¹ или запустите IDLE; вы увидите приглашение `>>>`.

Введите `2 + 2`, как показано ниже, и нажмите клавишу `Enter`:

```
$ python3
>>> 2 + 2
4
>>>
```

¹ Чтобы вызвать приглашение командной строки, в меню Пуск системы Windows введите `cmd` (или нажмите клавиши `Win+R`). Чтобы быстро вызвать окно терминала на компьютере Mac, нажмите `Command+Space`, введите `Terminal` и нажмите `Return`. Если вы установили Python 3, то теперь сможете запустить его на любой платформе командой `python3`.

В этом примере мы ввели команду `python3`, которая запустила интерпретатор. Первое приглашение `>>>` можно рассматривать как первую часть названия (R — чтение): Python ожидает входных данных. Мы ввели `2 + 2`, интерпретатор прочитал и *обработал* (E — обработка) их. Далее *выводится* (P — вывод) результат этого выражения — 4. Второе приглашение `>>>` относится к циклу (L — цикл): интерпретатор ожидает новых входных данных.

REPL по умолчанию направляет результат выражения в стандартный вывод (если результат отличен от `None`, но об этом позднее). Такое поведение отличается от обычных программ Python, в которых для вывода данных необходимо вызвать функцию `print`. В REPL это экономит несколько нажатий клавиш.

ПРИМЕЧАНИЕ

Приглашение `>>>` используется только в первой строке входных данных. Если команда, введенная в REPL, занимает более одной строки, следует приглашение `...`:

```
>>> sum([1, 2, 3, 4, 5,
... 6, 7])
```

Эти приглашения определяются в модуле `sys`:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

О том, что такое модули, будет рассказано в одной из следующих глав. А пока знайте, что внешний вид приглашений определяется специальными переменными.

REPL — весьма полезный инструмент. При помощи интерактивного интерпретатора можно писать небольшие функции, тестировать фрагменты кода и даже выполнять вычисления, как на калькуляторе. А еще интереснее пойти в другом направлении: запустите свой код Python в REPL. Код будет выполнен, а вы сможете проверить его состояние в REPL (скоро мы покажем, как сделать это в IDLE).

Символы >>> образуют *приглашение*. Здесь вы вводите свою программу. Введите после >>> команду `print("hello world")` и нажмите клавишу Enter. Проследите за тем, чтобы перед словом `print` не было ни пробелов, ни табуляций. Результат должен выглядеть так:

```
>>> print("hello world")
hello world
```

Если все получилось именно так — поздравляем, вы написали свою первую программу на Python. Считайте, что отныне официально приобрелись к миру программирования. Вы только что запустили программу «hello world» — классическую программу, которую многие по традиции пишут в начале знакомства с новым языком. Чтобы выйти из REPL в терминале, введите `quit()`. Пользователи UNIX также могут нажать клавиши `Ctrl+D`.

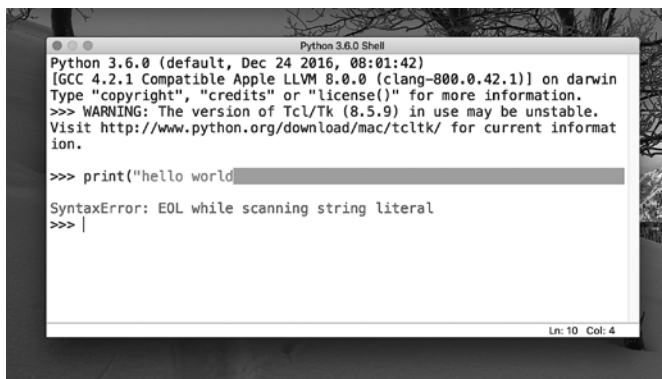


Рис. 3.4. IDLE пытается указать, где произошла ошибка. Цветовое выделение после world обозначает место, в котором ошибка была обнаружена

ПРИМЕЧАНИЕ

Программирование требует точности. Если при вводе `print("hello world")` пропустить всего один символ, результат может оказаться совсем другим, например, таким:

```
>>> print("hello world
      File "<stdin>", line 1
        print("hello world
              ^
SyntaxError: EOL while scanning string literal
```

Компьютер мыслит логично, и если ваши требования ему непонятны, он может предупредить вас, действовать иррационально (или, по крайней мере, вам так покажется) или вообще зависнуть. Не принимайте это близко к сердцу; помните, что в языках есть правила, и весь код, который вы пишете, должен эти правила соблюдать. В предыдущем примере было нарушено правило, требующее, чтобы весь текст, который вы хотите вывести на экран, начинался и заканчивался кавычками. На этот раз Python смутила пропущенная кавычка в конце строки.

3.3. Итоги

Так как Python является интерпретируемым языком, программисты могут использовать REPL для интерактивного исследования возможностей Python. Вам не нужно писать код, компилировать и запускать его — достаточно запустить REPL и начать эксперименты с кодом.

Пользователям, работавшим с компилируемыми языками, такой подход может показаться неожиданным. Не торопитесь с выводами и попробуйте; он может сделать разработку простой и быстрой. Кроме того, не бойтесь экспериментировать с кодом в REPL. По моему опыту, начинающие пользователи Python почему-то обходят REPL стороной. Не бойтесь REPL!

Также существуют другие разновидности REPL для Python. Один из популярных вариантов — Jupyter¹, REPL на базе веб-технологий. Начав с REPL, вскоре вы сможете перейти к другим, более мощным разновидностям.

3.4. Упражнения

1. Откройте REPL для Python 3 и выполните программу «hello world». Если вы забыли, как выглядит эта однострочная программа, просмотрите эту главу.
2. Откройте REPL из IDLE и выполните программу «hello world».

¹ <https://jupyter.org>

4

Запуск программ

Хотя интерактивный интерпретатор может принести пользу в ходе разработки, вам (и другим разработчикам) неизбежно потребуется запускать программы вне REPL. В Python и это делается просто. Чтобы запустить программу Python с именем `hello.py`, откройте окно терминала, перейдите в каталог с программой и введите команду

```
$ python3 hello.py
```

ПРИМЕЧАНИЕ

В этой книге перед командой, при ее выполнении из командной строки, будет ставиться символ `$`. По этому символу команды можно отличить от входных данных интерпретатора (`>>>` или `...`) и содержимого файлов (без префикса).

ПРИМЕЧАНИЕ

Скорее всего, попытка выполнения приведенной выше команды `python3 hello.py` завершится ошибкой, если в системе нет файла с именем `hello.py`.

В главе 3 мы использовали REPL для выполнения программы «hello world»; как выполнить ту же программу в автономном режиме? Создайте файл с именем `hello.py` в своем любимом текстовом редакторе.

Включите в файл `hello.py` следующую команду:

```
print("hello world")
```

Сохраните файл, перейдите в каталог с этим файлом и *выполните* файл (в данном случае термины «выполнить» и «запустить» считаются эквивалентными; иначе говоря, введите `python3` перед именем файла, и интерпретатор Python обработает код за вас).

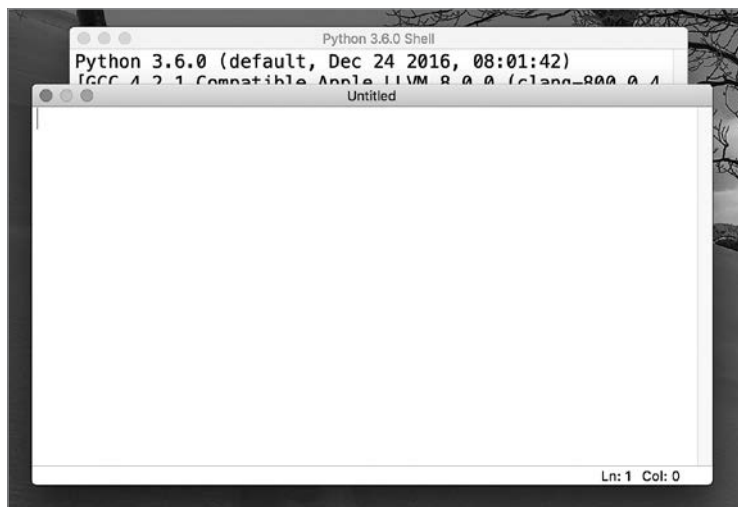


Рис. 4.1. Среда IDLE запущена, и в ней открыто новое окно редактора (с именем Untitled, потому что файл еще не был сохранен)

ПРИМЕЧАНИЕ

Команда `python3` без параметров запускает интерпретатор. Команда `python3 имя_файла.py` выполняет этот файл.

Если вам удалось запустить файл `hello.py`, команда выведет текст `hello world`.

4.1. Запуск программ из IDLE

Программы также можно редактировать и запускать в IDLE. Сначала запустите IDLE: либо щелкните на значке приложения на своем компьютере, либо выполните команду

```
$ python3 -m idlelib.idle
```

При запуске IDLE вы увидите окно *оболочки* (shell) — еще одной разновидности REPL языка Python. Если ввести код, Python немедленно выполнит его. Чтобы создать программу, необходимо создать файл, содержащий код Python. Для этого откройте меню File и выберите команду New File. На экране появляется новое окно — причем это не окно оболочки, а окно *редактора*. Обратите внимание: окно пустое и в нем нет приглашения Python. В этом окне вводится код Python.

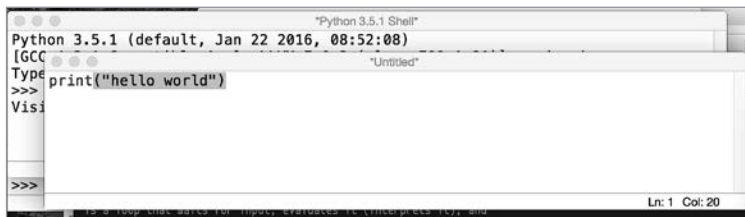


Рис. 4.2. Код, введенный в окне редактора

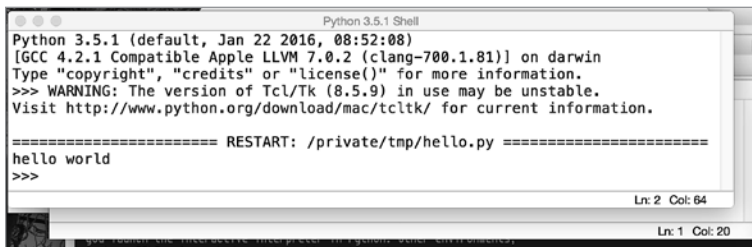


Рис. 4.3. Результат выполнения программы «hello world» в IDLE

Введите код в этом окне. Так как мы рассматриваем пример с программой «hello world», введите команду

```
print("hello world")
```

Вы заметите, что IDLE выделяет `print` и «hello world» разными цветами. Это выделение, называемое *цветовым выделением синтаксиса*, делает код более понятным. Теперь код нужно запустить. Проще всего сделать это нажатием клавиши F5. Другой вариант — открыть меню Run и выбрать команду Run Module. IDLE предложит сохранить файл. Сохраните его под именем `hello.py`. IDLE передает фокус окну оболочки, выводит строку с сообщением о перезапуске оболочки, а затем выводит сообщение `hello world`.

Результат вроде бы тривиальный, но окно оболочки теперь связано с состоянием вашего кода. В данном случае программа только выводит текст на экран, так что особого состояния у нее нет. Но в будущих примерах мы покажем, как воспользоваться механизмом интеграции редактора с оболочкой и создать среду, в которой можно быстро опробовать код, посмотреть результаты и проанализировать вывод программы.

Если вам понадобится информация о навигации в режиме командной строки, обратитесь к приложению А.

4.2. Усовершенствования для UNIX

На платформах UNIX (в частности, к этой категории относятся Linux и OS X) такие файлы, как `hello.py`, часто называются *сценариями* (scripts). Сценарий тоже является программой, этот термин часто используется для того, чтобы отличать интерпретируемый код от машинного. В данном случае сценарии являются *интерпретируемым* кодом, а результат компиляции такого языка, как C, представляет собой *машинный* код.

ПРИМЕЧАНИЕ

Также часто приходится слышать о «сценариях командной строки», «сценариях Perl», «сценариях Python» и т. д. Чем сценарий Python отличается от программы Python? Ничем, дело только в семантике. Сценарием Python обычно называется программа Python, запускаемая из командной строки, тогда как программой Python называется любая программа, написанная на Python (от простейших однострочных до эффектных приложений с графическим интерфейсом или служб корпоративного уровня).

В средах UNIX предусмотрен удобный механизм самостоятельного запуска сценариев. Если разместить в первой строке файла символы `#!`, за которыми указывается путь к интерпретатору, а для файла установить *бит исполнения*, такой файл можно будет запускать сам по себе, то есть без ручного запуска интерпретатора.

Чтобы сценарий выполнялся с интерпретатором Python, указанным в переменной среды, файл `hello.py` должен выглядеть так:

```
#!/usr/bin/env python3
print("hello world")
```

ПРИМЕЧАНИЕ

Новая первая строка приказывает командному интерпретатору, который выполняет файл, выполнить остаток файла с исполняемым файлом `#!/usr/bin/env python3`. (Сценарии командного интерпретатора обычно начинаются с `#!/bin/bash` или `#!/bin/sh`.) Сохраните файл `hello.py` с новой первой строкой.

СОВЕТ

Запись `#!/usr/bin/env` — удобное обозначение первого исполняемого файла `python3`, найденного в переменной среды `PATH`. Так как исполняемый файл `python3` хранится в разных местах на разных платформах, такое решение получается кросс-платформенным. В системе Windows эта строка игнорируется. Если только у вас нет полной уверенности в том, что вы хотите запустить конкретную версию Python, вероятно, стоит использовать запись `#!/usr/bin/env`.

Жестко закодированные пути вида

- `#!/bin/python3`
- `#!/usr/bin/python3.3`

могут нормально работать на вашем компьютере, но если вы передадите свой сценарий другим разработчикам, на машинах которых `python3` не хранится в указанном вами месте, начнутся проблемы. Если для вашего сценария нужна конкретная версия Python, об этом обычно упоминается в файле `README`.

Теперь файл необходимо назначить исполняемым. Откройте терминал, перейдите в каталог с файлом `hello.py` и разрешите исполнение файла следующей командой:

```
$ chmod +x hello.py
```

Команда устанавливает для файла *бит исполнения*. В среде UNIX существуют разные разрешения (устанавливаемые при помощи соответствующих битов) для чтения, записи и исполнения файла. Если для файла установлен бит исполнения, то при запуске файла среда UNIX проверяет его первую строку и выполняет файл так, как указано.

СОВЕТ

Если вы захотите узнать, что делает команда `chmod`, вызовите справку по ней командой `man`:

```
$ man chmod
```

Теперь для выполнения файла достаточно ввести его имя в терминале и нажать `Enter`. Введите

```
$ ./hello.py
```

Команда должна запустить вашу программу (или сценарий). Обратите внимание на символы `./` перед именем программы. Обычно при вводе команды в терминале среда ищет исполняемый файл в `PATH` (переменная среды, определяющая, в каких каталогах хранятся исполняемые файлы). Если только переменная `PATH` не содержит `.` (или родительский каталог `hello.py`), перед именем необходимо указать `./` (или полный путь к исполняемому файлу). В противном случае вы получите сообщение об ошибке:

```
$ hello.py
bash: hello.py: command not found
```

Да, вся эта работа была проделана только для того, чтобы вам не приходилось вводить команду `python3 hello.py`. Почему? Прежде всего потому, что вашей программе было бы лучше называться `hello` (без суффикса `.py`). А может быть, вы захотите разместить программу в `PATH`, чтобы ее можно было запустить в любой момент. Назначив файл исполняемым и добавив символы `#!`, можно создать файл, который выглядит как обычный исполняемый файл. Такому файлу не нужно расширение `.py`, а для его выполнения не придется вводить команду `python3`.

4.3. Итоги

Запускать программы Python несложно. Запуск не требует долгого этапа компиляции. От вас потребуется лишь сообщить Python программу, которую вы хотите запустить. Во многих редакторах также предусмотрена возможность выполнения кода Python. Вам стоит выяснить, как это делается в вашем любимом редакторе. В IDLE это просто: достаточно нажать клавишу `F5`.

4.4. Упражнения

1. Создайте файл `hello.py`, содержащий код из этой главы.
2. Запустите `hello.py` из терминала.
3. Запустите `hello.py` из IDLE.
4. Если вы предпочитаете работать в другом редакторе, запустите `hello.py` из него.
5. Если вы работаете на платформе UNIX, создайте файл с именем `hello`. Добавьте в него код «hello world» и внесите необходимые изменения, чтобы код можно было выполнить командой

`./hello`

5

Запись и чтение данных

У программ обычно имеются входные и выходные данные. В этой главе вы узнаете, как вывести значения на экран и как получить значение от пользователя. В языке Python обе задачи решаются тривиально.

5.1. Простой вывод

Чтобы вывести данные для пользователя, проще всего воспользоваться функцией `print`, которая записывает данные в *стандартный вывод* — поток, в который компьютер направляет выходные данные. Если вы работаете в терминале, то стандартный вывод направляется на терминал:

```
>>> print('Hello there')
Hello there
```

Чтобы вывести несколько значений, разделите их запятыми. Python автоматически вставляет пробелы между ними. При вызове функции `print` можно указывать строки и числа:

```
>>> print('I am', 10, 'years old')
I am 10 years old
```

Позднее в этой главе строки будут рассмотрены более подробно. Вы узнаете, как отформатировать их, чтобы придать выходным данным нужный вид.

5.2. Получение данных от пользователя

Встроенная функция `input` читает текст с терминала. Эта функция получает текст подсказки, который выводится на экран, а затем ожидает, пока пользователь введет что-либо в *стандартном вводе* и нажмет `Enter`. Стандартный ввод представляет собой поток, из которого компьютер получает входные данные. В терминале стандартный ввод может читаться из символов, вводимых вами с клавиатуры:

```
>>> name = input('Enter your name:')
```

Если ввести эту команду в интерпретаторе (пробелы по обе стороны от = не обязательны, но их рекомендуется вводить, чтобы код лучше читался), может показаться, что ваш компьютер завис. На самом деле Python ожидает, когда вы что-нибудь введете и нажмете `Enter`. После нажатия `Enter` введенные данные будут сохранены в переменной `name`. Введите имя `Matt` и нажмите клавишу `Enter`. Если теперь вывести значение `name`, программа выведет только что введенное вами значение:

```
>>> print(name)
Matt
```

ПРИМЕЧАНИЕ

Значение, введенное в терминале при вызове `input`, всегда представляет собой строку. Если вы попытаетесь выполнить с ним математические операции, результат может оказаться не тем, на который вы рассчитывали:

```
>>> value = input('Enter a number:')
3
>>> other = input('Enter another:')
4
```

Если теперь попытаться сложить `value` с `other`, будет выполнена конкатенация (сцепление строк), потому что в обеих переменных хранятся строки:

```
>>> type(value)
<class 'str'>
>>> value + other
'34'
```

Если вы хотите сложить числа, содержащиеся в строках, их необходимо преобразовать из строкового типа к числовому. Для преобразования строк

к другому типу, например целому или вещественному, используются функции `int` и `float` соответственно.

Чтобы сложить значения `value` и `other` в числовом виде, преобразуйте их в числа функцией `int`:

```
>>> int(value) + int(other)
7
```

В следующей главе числовые и строковые типы будут рассмотрены более подробно.

5.3. Итоги

Python предоставляет две функции для простого вывода данных на экран и получения данных от пользователя. Это функции `print` и `input`. Помните, что при вызове функции `input` вы всегда получаете строку.

5.4. Упражнения

1. Напишите код Python, который запрашивает у пользователя его имя, после чего выводит `Hello` и введенное имя.
2. Напишите программу, которая запрашивает у пользователя его возраст. Выведите сообщение, в котором говорится, сколько лет будет пользователю в следующем году.

6

Переменные

Итак, вы научились запускать программы в интерпретаторе (или REPL) и в командной строке. Теперь можно осваивать основы программирования. *Переменные* — основные структурные элементы компьютерных программ.

Переменные играют важную роль в программах Python, потому что в мире Python нет ничего, кроме *объектов*. (На самом деле это не совсем так — ключевые слова объектами не являются.) Переменные позволяют назначать объектам имена, чтобы к ним можно было обращаться из программного кода.

6.1. Изменение и состояние

В программировании существует две важные концепции: *состояния* и *изменения*. *Состояние* связано с цифровым представлением модели. Например, если вы хотите моделировать лампочку, можно хранить в программе ее текущий статус — включена лампочка или нет? Среди других интересных вариантов состояния можно хранить тип лампочки (люминесцентная, лампа накаливания), потребляемую мощность, размер, возможность регулировки яркости и т. д.

Концепция *изменения* связана с переходом в новое состояние. Скажем, в примере с лампочкой было бы полезно иметь переключатель, который переводит лампочку в противоположное состояние: если лампочка выключена, то она включается, и наоборот.

Какое отношение все это имеет к переменным? Вспомните, что в Python нет ничего, кроме объектов. Объекты обладают состоянием, которое может изменяться. Для хранения информации об объектах используются переменные.

Когда в программе появляются объекты с состоянием, которое может изменяться, перед вами открывается целый мир новых возможностей. В программе можно смоделировать практически все, что угодно, — если вы сможете определить, каким состоянием должна обладать модель и какие действия (или изменения) к ней должны применяться.

6.2. Переменные Python как метки

Наборы переменных используются для хранения состояния. *Переменную* можно представлять себе как своего рода метку или бирку: важная информация помечается именем переменной. Продолжая предыдущий пример, предположим, что вы хотите запомнить состояние лампочки. Хранить данные бессмысленно, если к ним нельзя обратиться. Если же вы хотите обращаться к данным и хранить их состояние, нужно создать *переменную*, связанную с этими данными. В следующем фрагменте состояние лампочки хранится в переменной с именем `status`:

```
>>> status = "off"
```

На этом моменте стоит остановиться подробнее, потому что здесь происходит много всего интересного. Справа находится слово `"off"`, заключенное в кавычки. Это *строковый литерал*, или встроенный тип данных Python со специальным синтаксисом. Кавычки сообщают Python, что объект является строкой, поэтому Python создает объект со строковым содержимым. Строки используются для хранения текстовых данных — в данном случае символов `off`.

Этот объект обладает рядом интересных свойств. Во-первых, у него есть *идентификатор*. Можно считать, что идентификатор указывает на место, в котором Python хранит этот объект в памяти. У объекта также имеется *тип* (в данном случае это строка). Наконец, у объекта есть *значение* — здесь оно состоит из символов `off`, так как это строка.

Знак `=` представляет *оператор присваивания* во многих языках программирования. Не бойтесь технических терминов — они проще, чем кажется

на первый взгляд. Оператор присваивания связывает имя переменной с ее объектом. Он показывает, что имя слева от него является переменной, в которой будет храниться объект из правой части. В данном случае переменной присваивается имя `status`.

Создавая переменную, Python приказывает объекту увеличить свой *счетчик ссылок*. Если на объект указывают переменные или другие объекты, то у этого объекта счетчик ссылок положителен. Когда переменные перестают существовать (например, при выходе из функции переменные из этой функции исчезают), счетчик ссылок уменьшается. Когда счетчик уменьшается до 0, интерпретатор Python делает вывод, что объект больше никому не нужен, и подвергает его *уборке мусора*. Это означает, что объект удаляется из памяти, чтобы ваши программы не выходили из-под контроля и не занимали всю память на компьютере.

ПРИМЕЧАНИЕ

Если вы захотите узнать значение счетчика ссылок для объекта, вызовите для него функцию `sys.getrefcount`:

```
>>> import sys
>>> names = []
>>> sys.getrefcount(names)
2
```

Обратите внимание: значение счетчика может показаться слишком высоким, но в документации этой функции говорится:

«...Возвращает счетчик ссылок объекта. Возвращаемое значение обычно на 1 выше ожидаемого, потому что оно включает (временную) ссылку для аргумента `getrefcount()`».

Хотя Python и предоставляет такую возможность, обычно счетчик ссылок не представляет интереса для разработчика. Лучше доверить хлопоты с уничтожением объектов интерпретатору Python.

Обычно Python делает все это за вас автоматически, без подсказок со стороны пользователя. Во многих других языках программисту

приходится вручную приказывать программе выделять и освобождать память.

Для пущей наглядности попробуйте теперь снова прочитать код — на этот раз слева направо. `Status` — переменная, с которой связывается объект, созданный за нас Python. У этого объекта есть тип (строка), и он содержит значение `"off"`.

6.3. Бирки

У моего дедушки было животноводческое ранчо, поэтому я приведу некомпьютерную аналогию. Любому владельцу ранчо необходим хороший гуртовщик, который следит за скотом (то есть за хозяйскими капиталовложениями).

Чтобы не перепутать коров, часто используются специальные бирки. Такая бирка прикрепляется к уху каждой коровы, и по ней можно узнать каждое конкретное животное.

А теперь вернемся к программированию: вместо ранчо вы управляете различными аспектами своей программы. Программа может хранить много разных блоков информации, которые тоже нужно как-то отличать друг от друга: это информация *состояния*. Например, если вам нужно хранить информацию о людях, в этой информации может присутствовать имя, возраст и адрес человека.

Подобно тому, как скотоводы помечают бирками своих животных, программисты создают переменные для отслеживания данных. Еще раз взгляните на наш пример:

```
>>> status = "off"
```

Эта команда приказывает Python создать *строку*, содержащую текст `off`. Программа создает переменную с именем `status` и связывает ее с этой строкой. Позднее, когда вам потребуется узнать информацию состояния, можно приказать программе вывести ее на экран — вот так:

```
>>> print(status)
off
```

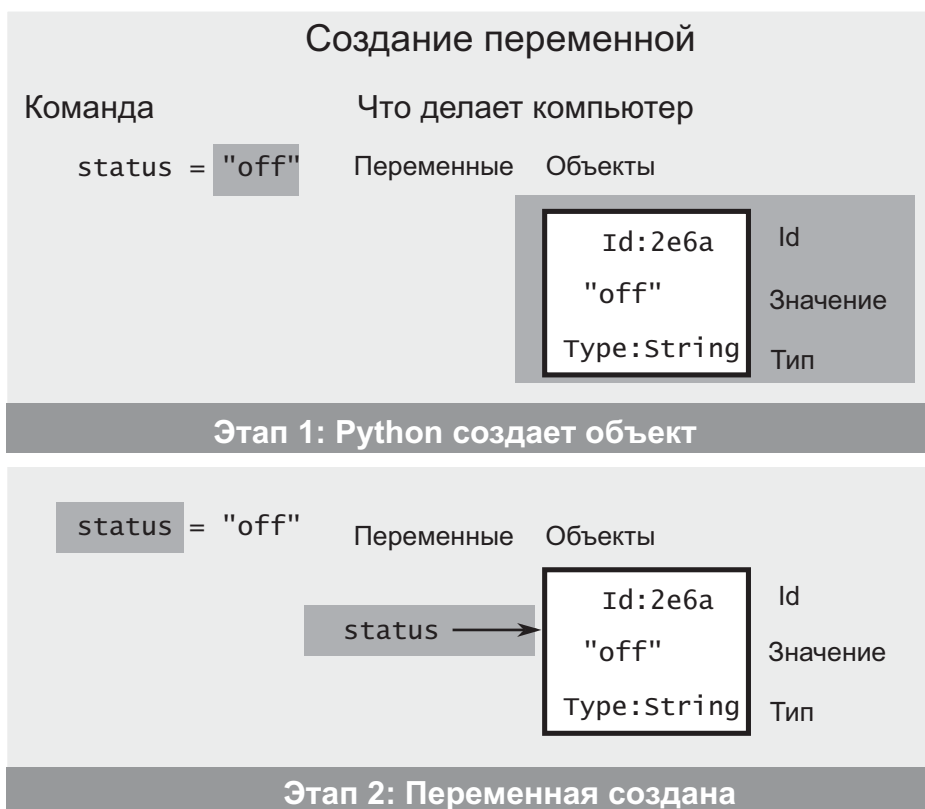


Рис. 6.1. Два этапа присваивания литерала. Сначала Python создает объект. У этого объекта есть значение "off", тип (строка) и идентификатор (местонахождение объекта в памяти). После того как объект будет создан, Python ищет переменную с именем status. Если такая переменная существует, Python изменяет объект, на который указывает эта переменная; в противном случае Python создает переменную и связывает ее с объектом

Ничто не мешает вам создать *состояние* и потерять его, если вы забыли сохранить его в переменной. Создавать объекты, которые вы не используете, немного странно; тем не менее это возможно. Предположим, вы хотите отслеживать для объекта лампочки потребляемую мощность. Команда

```
>>> "120 watt"
```

приказывает Python создать строковый объект, содержащий текст `120 watt`. Проблема в том, что этот объект не присваивается никакой пере-

менной. Теперь вы никак не сможете использовать его в своей программе. Python позволяет обращаться только к данным, хранящимся в переменных, поэтому теперь объект стал недоступным. Программа обращается к объектам по именам переменных. Если эта информация должна использоваться в программе, правильнее было бы использовать команду вида

```
>>> wattage = "120 watt"
```

Позднее в своей программе вы можете обратиться к переменной `wattage`, вывести ее значение, даже присвоить его другой переменной или присвоить `wattage` новое значение (допустим, ваша лампа накаливания разбилась и вы заменили ее светодиодной):

```
>>> incandescent = wattage
>>> wattage = "25 watt"
>>> print(incandescent, wattage)
120 watt 25 watt
```

Управление состоянием — один из основных аспектов программирования, а переменные — один из механизмов этого управления.

6.4. Повторное связывание переменных

Переменные, как и бирки на коровах, остаются связанными со своими объектами в течение какого-то времени, но не навсегда. Python позволяет легко изменить содержимое переменной:

```
>>> num = 400
>>> num = '400' # теперь num содержит строку
```

В этом примере переменная `num` сначала была связана с целым числом, но потом программа связала ее со строкой.

ПРИМЕЧАНИЕ

Для переменной тип неважен. В языке Python типом обладает объект, а не переменная.

Переменную можно изменять сколько угодно раз. Тем не менее будьте внимательны и не изменяйте переменную, если вам все еще нужны старые

данные. После того как все переменные будут отсоединены от объекта, вы фактически приказываете Python уничтожить объект при первой возможности, чтобы освободить занимаемую им внутреннюю память (в программировании это называется уборкой мусора).

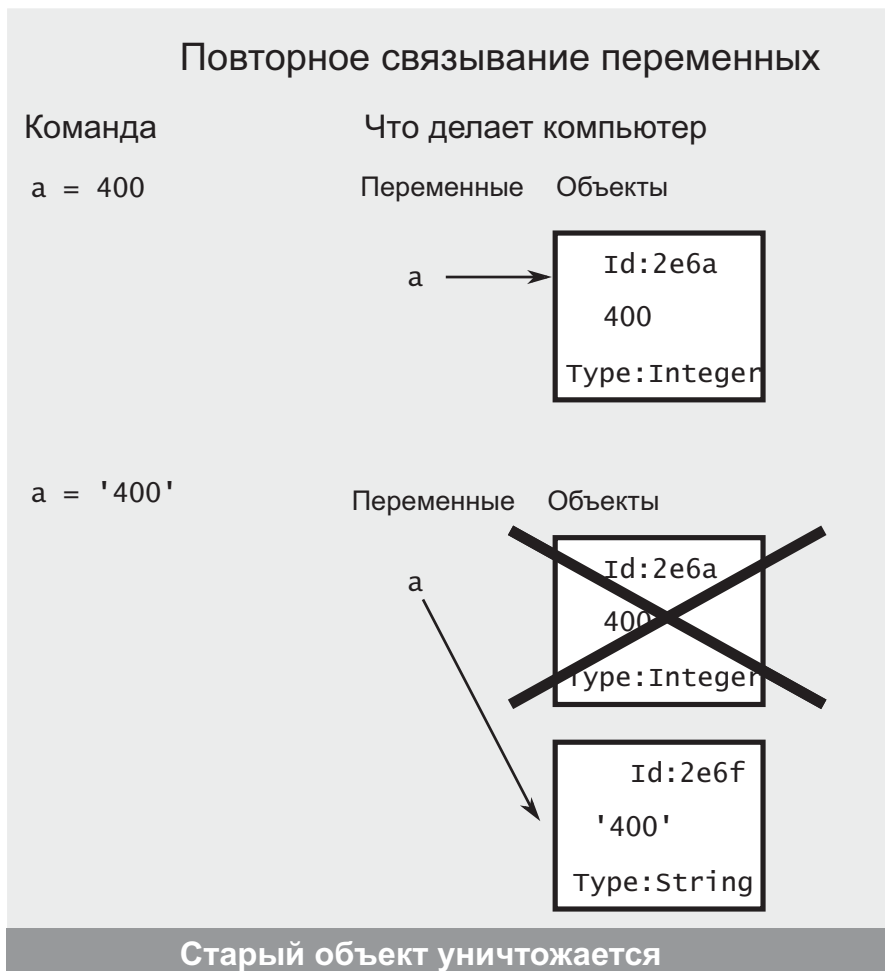


Рис. 6.2. Повторное связывание переменных. Переменная может быть заново связана с любым типом; Python не пытается этому как-то помешать и не выдает предупреждений. Если с объектом не связана ни одна переменная, Python уничтожает этот объект в процессе уборки мусора

СОВЕТ

Это один из тех случаев, когда Python позволяет вам что-то сделать, но это вовсе не означает, что это стоит делать в реальной жизни. Да, переменную можно заново связать с другим типом, но не нужно. Изменение типа переменной затруднит чтение кода. Кроме того, программа только запутает других разработчиков, которые используют ваш код. Не используйте одну переменную для работы со значениями разных типов!

Начинающие разработчики Python нередко допускают одну и ту же ошибку: они заново используют одну переменную в своем коде, полагая, что они тем самым экономят память. Как вы уже видели, это не так. Сами переменные памяти почти не занимают — она расходуется для хранения объекта. Повторное использование переменной не изменит затраты памяти на хранение объекта, но собьет с толку тех, кто будет читать код в будущем.

6.5. Имена переменных

В Python имена переменных не выбираются полностью произвольно: существуют конкретные соглашения, которые выполняются большинством программистов. Одни из этих соглашений обязательны к выполнению, другие — нет. Например, интерпретатор требует, чтобы имя переменной *не совпадало с ключевым словом языка*. Скажем, `break` — ключевое слово, поэтому ваша переменная не может называться `break`. При попытке использовать `break` как имя переменной, вы получите синтаксическую ошибку. И хотя следующий код выглядит вполне нормально, Python сообщит об ошибке:

```
>>> break = 'foo'
      File "<stdin>", line 1
        break = 'foo'
            ^
SyntaxError: invalid syntax
```

Если вы столкнетесь с синтаксической ошибкой (`SyntaxError`) в совершенно нормальном на первый взгляд коде Python, убедитесь в том, что имя переменной не совпадает с ключевым словом.

ПРИМЕЧАНИЕ

Ключевые слова зарезервированы для языковых конструкций Python, поэтому при попытке использования их в качестве переменных Python приходит в замешательство.

Модуль `keyword` содержит атрибут `kwlist` со списком всех ключевых слов Python:

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

Также для просмотра ключевых слов в REPL можно вызвать метод `help()`. Он переводит REPL в режим справочной информации, в котором можно вводить команды для получения справки (в отличие от команд Python). Введите `keywords` и нажмите `Enter`. Если ввести любое ключевое слово, программа выведет документацию и сопутствующие разделы справки. Чтобы выйти из режима справки, просто нажмите `Enter`.

6.6. Дополнительные рекомендации по назначению имен

Кроме уже упоминавшегося правила о том, что имена переменных не могут совпадать с ключевыми словами, существует ряд рекомендаций, поддерживаемых сообществом Python. Эти рекомендации просты:

- имена переменных должны состоять из символов нижнего регистра;
- слова должны разделяться символом подчеркивания;
- имена переменных не могут начинаться с цифр;
- имена переменных не могут переопределять *встроенные* функции.

Несколько примеров имен переменных — как хороших, так и плохих:

```
>>> good = 4
>>> bAd = 5 # плохо - верхний регистр
>>> a_longer_variable = 6
```

Не рекомендуется

```
>>> badLongerVariable = 7
```

Плохо - начинается с цифры

```
>>> 3rd_bad_variable = 8
```

```
File "<stdin>", line 1
```

```
    3rd_bad_variable = 8
```

```
    ^
```

```
SyntaxError: invalid syntax
```

Плохо - ключевое слово

```
>>> for = 4
```

```
File "<stdin>", line 1
```

```
    for = 4
```

```
    ^
```

```
SyntaxError: invalid syntax
```

Плохо - встроенная функция

```
>>> compile = 5
```

СОВЕТ

Правила и соглашения назначения имен в Python перечислены в документе «PEP 8 — Style Guide for Python Code»¹. Сокращение PEP означает «Python Enhancement Proposal», то есть «Предложение по улучшению Python» — так называется процесс документирования функции, усовершенствования или передовой практики для Python. Документы PEP доступны на сайте Python.

ПРИМЕЧАНИЕ

Хотя Python не разрешает использовать ключевые слова в качестве имен переменных, *встроенные* имена могут использоваться как переменные.

¹ <https://www.python.org/dev/peps/pep-0008/>

Встроенные имена (built-ins) — имена функций, классов или переменных, которые Python загружает автоматически, чтобы вам было проще обращаться к ним. В отличие от ключевых слов, Python позволяет использовать встроенное имя в качестве названия переменной без малейших возражений. Тем не менее поступать так не рекомендуется — это считается признаком плохого стиля.

Использование встроенного имени в качестве названия переменной приводит к *замещению* встроенного имени. Новое имя переменной не позволит получить доступ к исходному встроенному имени. Фактически вы берете встроенную переменную и заимствуете ее для собственного использования. В результате доступ к исходному встроенному имени будет возможен только через модуль `__builtins__`. Гораздо лучше с самого начала избегать замещения.

Ниже перечислены *встроенные* имена Python, которые не следует использовать в качестве имен переменных:

```
>>> dir(__builtins__)

['ArithmeticError', 'AssertionError',
'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError',
'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning',
'RuntimeError', 'RuntimeWarning',
'StopAsyncIteration', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True',
```

```
'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError',
'Warning', 'ZeroDivisionError', '_',
'__build_class__', '__debug__', '__doc__',
'__import__', '__loader__', '__name__',
'__package__', '__spec__', 'abs', 'all', 'any',
'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr',
'dict', 'dir', 'divmod', 'enumerate', 'eval',
'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int',
'isinstance', 'issubclass', 'iter', 'len',
'license', 'list', 'locals', 'map', 'max',
'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

СОВЕТ

Несколько встроенных имен, которые выглядят особенно соблазнительно в качестве имен переменных: `dict`, `id`, `list`, `min`, `max`, `open`, `range`, `str`, `sum` и `type`.

6.7. Итоги

В Python все сущности являются объектами. Объекты обладают состоянием, которое также называется значением. Для работы с объектами используются переменные. Переменные Python напоминают бирки: они связываются с объектом и обладают именем. Однако объект содержит важные данные: значение и тип данных.

В этой главе также рассматривается повторное связывание переменных с объектами. Python допускает такую возможность, но вы должны быть

внимательны и не должны изменять тип переменной, потому что это усложнит понимание кода для тех, кто будет его читать. Наконец, в этой главе рассматриваются соглашения об именах переменных Python.

6.8. Упражнения

1. Создайте переменную `pi`, которая указывает на приближенное значение числа π . Создайте переменную `r` для представления радиуса круга; эта переменная должна быть равна 10. Вычислите площадь круга ($\pi \times \text{квадрат радиуса}$). Умножение выполняется оператором `*`, а возведение числа в квадрат — оператором `**`. Например, результат `3**2` равен 9.
2. Создайте переменную `b`, которая указывает на ширину прямоугольника со значением 10. Создайте переменную `h`, которая указывает на высоту прямоугольника со значением 2. Вычислите длину периметра прямоугольника. Измените ширину, присвоив ей значение 6, и снова вычислите периметр.

7

Подробнее об объектах

В этой главе объекты будут рассмотрены более подробно. Речь пойдет о трех важных свойствах объектов:

- идентификатор;
- тип;
- значение.

7.1. Идентификатор

На самом низком уровне *идентификатор* определяет местонахождение объекта в памяти компьютера. В Python существует встроенная функция `id`, которая возвращает идентификатор объекта:

```
>>> name = "Matt"
>>> id(name)
140310794682416
```

При выполнении этого кода идентификатор строки `"Matt"` выводится в форме `140310794682416` (это число определяет адрес оперативной памяти вашего компьютера). Обычно идентификатор изменяется в зависимости от компьютера и сеанса командного интерпретатора, однако на протяжении жизненного цикла программы идентификатор объекта остается неизменным.

Одну корову можно пометить двумя бирками; точно так же две переменные могут указывать на один объект. Если вы хотите, чтобы другая

переменная (скажем, с именем `first`) тоже указывала на объект, на который указывает `name`, выполните следующую команду:

```
>>> first = name
```

Команда приказывает Python назначить переменной `first` такой же идентификатор, как у `name`. При выполнении функции `id` для двух переменных будут получены одинаковые значения:

```
>>> id(first)
140310794682416
>>> id(name)
140310794682416
```



Рис. 7.1. На диаграмме показано, что происходит при связывании переменной с существующей переменной. Обе переменные указывают на один объект.

Обратите внимание: переменная при этом не копируется! Также обратите внимание на то, что объект обладает значением "Matt", типом и идентификатором

Какая польза от идентификатора, спросите вы? Вообще-то пользы немного. При программировании на Python вас обычно не интересуют такие низкоуровневые подробности, как адрес объекта в памяти. Однако при помощи идентификатора можно показать время создания объектов и возможно ли их изменение. Кроме того, идентификаторы косвенно используются для *проверки тождественности* оператором `is`.

Оператор `is` проверяет тождественность, то есть он проверяет, указывают ли две переменные на один и тот же объект:

```
>>> first is name
True
```

Если вывести в REPL `first` и `name`, будут выведены одинаковые значения, потому что оба имени указывают на один объект:

```
>>> print(first)
Matt
>>> print(name)
Matt
```

Бирку можно снять с одной коровы и повесить на другую. Точно так же и переменную можно перевести с одного объекта на другой. Например, мы можем заставить переменную `name` указывать на новый объект. Вы увидите, что идентификатор `name` изменился, тогда как идентификатор `first` остался прежним:

```
>>> name = 'Fred'
>>> id(name)
140310794682434
>>> id(first)
140310794682416
```

7.2. Тип

Еще одно свойство объекта — его *тип*. В программах чаще всего используются такие типы, как *строки*, *целые числа*, *числа с плавающей точкой* и *логические значения*. Существует много других типов, вдобавок вы можете создавать собственные. Под «типом объекта» обычно подразумевается *класс* объекта. Класс определяет состояние данных, хранящихся в объекте, а также *методы* (или действия), которые может выполнять объект. В языке Python тип объекта легко проверяется встроенной функцией `type`:

```
>>> type(name)
<class 'str'>
```

В данном случае функция `type` сообщает, что переменная `name` указывает на строку (`str`).

В следующей таблице перечислены типы различных объектов Python.

Объект	Тип
Строка	<code>str</code>
Целое число	<code>int</code>
Число с плавающей точкой	<code>float</code>
Список	<code>list</code>
Словарь	<code>dict</code>
Кортеж	<code>tuple</code>
Функция	<code>function</code>
Класс, определяемый пользователем	<code>type</code>
Экземпляр класса	<code>class</code>
Встроенная функция	<code>builtin_function_or_method</code>
<code>type</code>	<code>type</code>

Из-за применения *утиной типизации* функция `type` используется не так часто. Вместо того чтобы проверять, относится ли объект к типу, поддерживающему некоторую операцию, обычно вы просто пытаетесь выполнить эту операцию.

Однако иногда в программе имеются данные, которые нужно преобразовать к другому типу. Такая ситуация типична для чтения данных из стандартного ввода: данные обычно поступают в виде строки, и эту строку нужно преобразовать в число. Python предоставляет встроенные классы `str`, `int`, `float`, `list`, `dict` и `tuple`, которые при необходимости выполняют преобразование к соответствующему типу:

```
>>> str(0)
'0'
```

```
>>> tuple([1,2])
(1, 2)
>>> list('abc')
['a', 'b', 'c']
```

ПРИМЕЧАНИЕ

Термин «утиная типизация» происходит от поговорки XVIII века, в которой говорилось о механической утке: «Если оно выглядит как утка, ходит как утка и крикает как утка, то это утка».

Впрочем, лично я предпочитаю сцену из фильма «Монти Пайтон и Священный Грааль», где в одной женщине распознают ведьму, потому что она весит столько же, сколько весит утка. (Если вам это кажется странным, посмотрите фильм — мне он нравится.) Идея в том, что женщина обладает такими же характеристиками (вес), как и утка, поэтому ее следует считать ведьмой.

Python действует аналогично. Если вы хотите использовать объект для перебора элементов, вы помещаете его в цикл `for`. Вы не проверяете, является ли он списком или подклассом списка, а просто выполняете перебор. Если вы хотите выполнить операцию сложения, вы не проверяете, является ли объект числом или строкой (или другим типом, поддерживающим сложение). Если операция завершится неудачей — нормально, это всего лишь говорит о том, что вы предоставили неправильный тип.

Если вы знакомы с объектно-ориентированным программированием, «утиная типизация» ослабляет жесткие требования создания подклассов. Вместо того чтобы наследовать от нескольких классов, чтобы воспользоваться предоставляемым ими поведением, вы должны реализовать протоколы (обычно для этого нужно определить один-два метода). Например, чтобы создать класс, который может использоваться для сложения, необходимо реализовать метод `.__add__`. Любой класс может определить этот метод и реагировать на операцию сложения.

7.3. Изменяемость

Другое интересное свойство объекта — его *изменяемость*. Многие объекты являются *изменяемыми*, другие *неизменяемы*. Изменяемые объекты допускают изменение своего значения «на месте» — иначе говоря, их состояние можно изменить, но их идентификатор останется неизменным. Неизменяемые объекты не позволяют изменить свое значение. Вместо этого их переменная связывается с новым объектом, но это также приводит к изменению идентификатора переменной.

В языке Python словари и списки являются изменяемыми типами. Строки, кортежи, целые числа и числа с плавающей точкой относятся

к неизменяемым типам. Следующий пример демонстрирует, что идентификатор переменной, содержащей целое число, изменится при изменении значения. Сначала программа присваивает целое число переменной `age` и проверяет идентификатор:

```
>>> age = 1000
>>> id(age)
140310794682416
```

Обратите внимание: если теперь изменить значение целого числа, то у него появится новый идентификатор:

```
>>> age = age + 1
>>> id(age)
140310793921824
```

А теперь рассмотрим пример изменения списка. Начнем с пустого списка и проверим его идентификатор. Обратите внимание: даже после добавления элемента в список идентификатор списка остается неизменным, поскольку список является изменяемым типом. Сначала мы создаем список и проверяем идентификатор:

```
>>> names = []
>>> id(names)
140310794682432
```

Теперь добавим в список строку. Здесь есть ряд моментов, заслуживающих внимания. Возвращаемое значение метода `.append` не выводится (то есть метод не возвращает новый список). Но если просмотреть переменную `names`, выясняется, что новое имя успешно занесено в список. Также идентификатор списка остался неизменным — список был успешно изменен:

```
>>> names.append("Fred")
>>> names
['Fred']
>>> id(names)
140310794682432
```

Изменяемые объекты не должны использоваться в качестве ключей в словарях. Кроме того, они могут создать проблемы при использовании в качестве параметров по умолчанию в функциях.

7.4. Использование IDLE

Приведенный пример было бы неплохо опробовать в IDLE (или в вашем любимом редакторе с интегрированным интерпретатором REPL). Так как IDLE включает REPL, вы можете ввести приведенный код и проанализировать его прямо в REPL. Однако вы также можете написать код, запустить его и проанализировать из REPL. Чтобы опробовать эту возможность, откройте новый файл и включите в него следующий код:

```
name = "Matt"
first = name
age = 1000
print(id(age))
age = age + 1
print(id(age))
names = []
print(id(names))
names.append("Fred")
print(id(names))
```

Сохраните код в файле с именем `iden.py`. Запустите файл. В IDLE для этого следует нажать клавишу F5. В REPL будут выведены четыре числа. Первые два будут разными; это говорит о том, что целое число является неизменяемым. Последние два числа совпадают. Это объясняется тем, что несмотря на изменение списка `names` идентификатор остался прежним. В общем-то в этом факте нет ничего принципиально нового.

Теперь самое интересное: если ввести в REPL команду `dir()`, она выведет список переменных. Вы увидите, что глобальные переменные из `iden.py` теперь стали доступны.

REPL в IDLE открывает доступ ко всем глобальным переменным. Вы можете просмотреть `name` и `names` и даже вызывать функции или методы — например, `names.append("George")`.

Имея возможность изучить результаты только что выполненного кода, вы можете быстро проанализировать код и поэкспериментировать с ним. Опытные разработчики Python нередко пишут код в REPL, вставляют его в файл, снова запускают файл, пишут новый код в REPL и продолжают писать код таким способом.

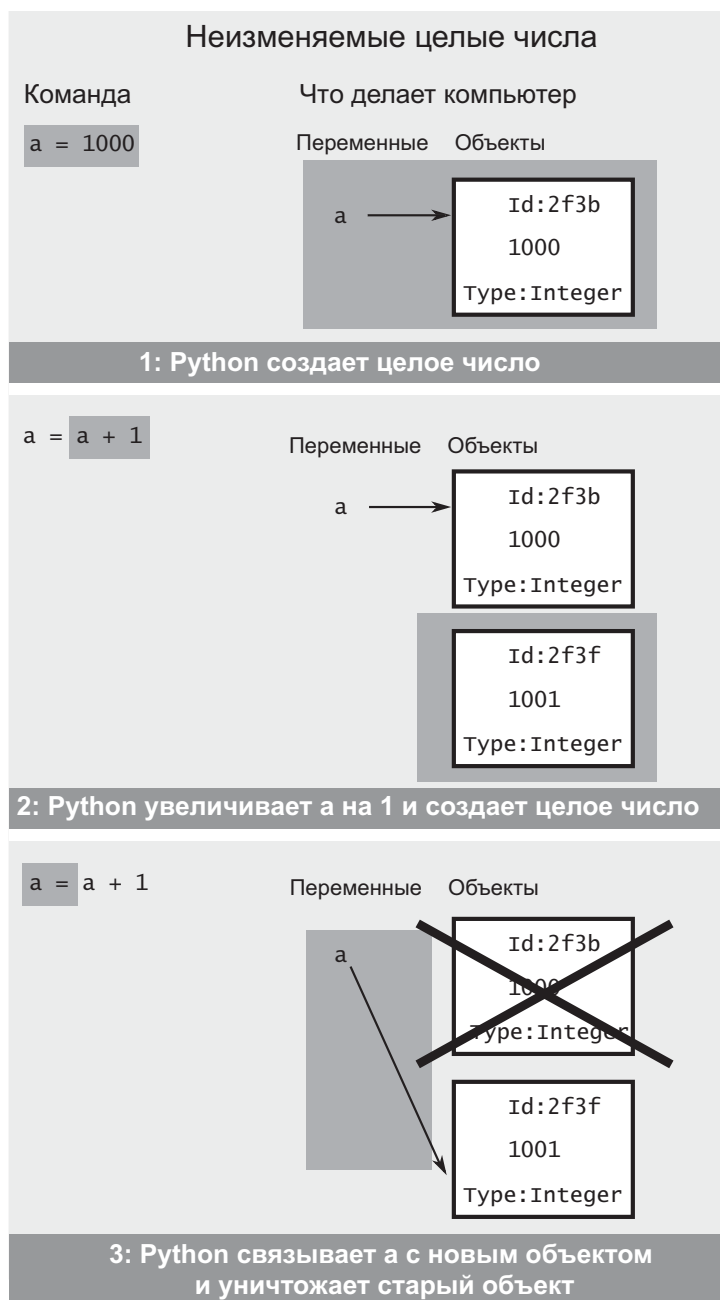


Рис. 7.2. Попытка изменить целое число неизбежно приводит к созданию нового целого числа. Целые числа неизменяемы, и их значение должно создаваться заново

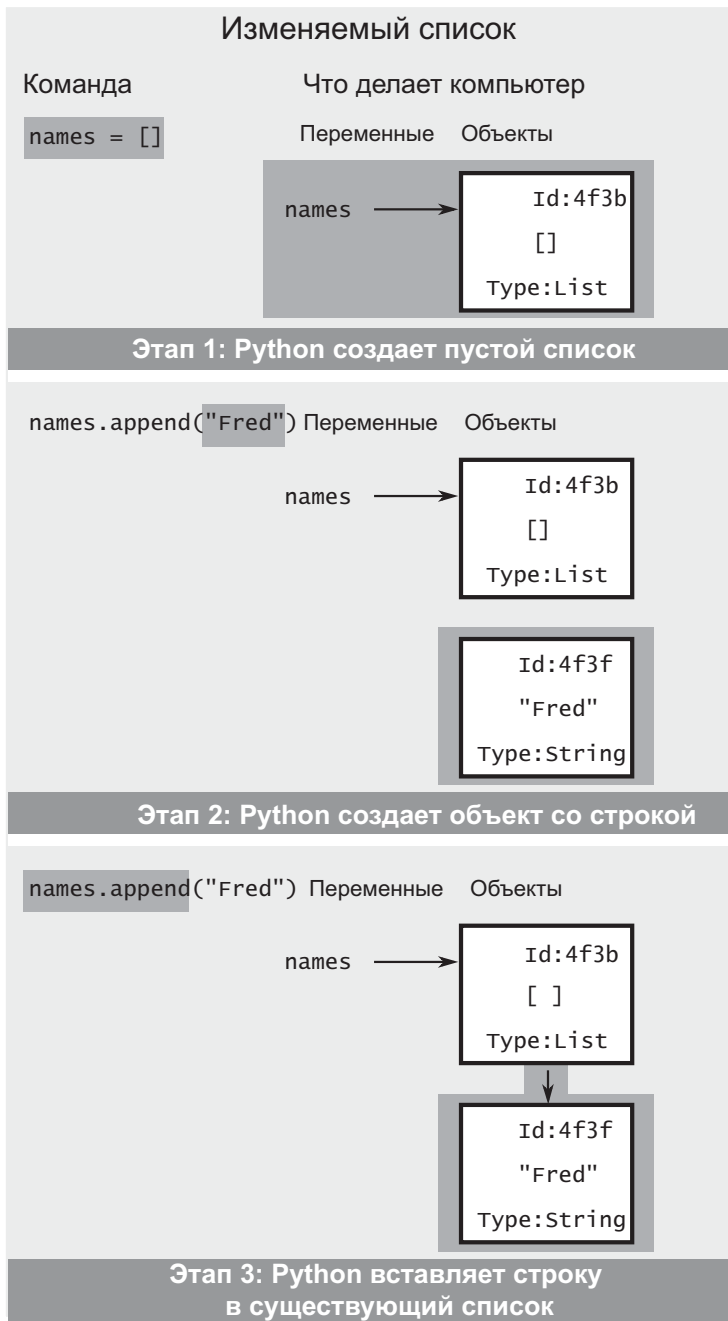
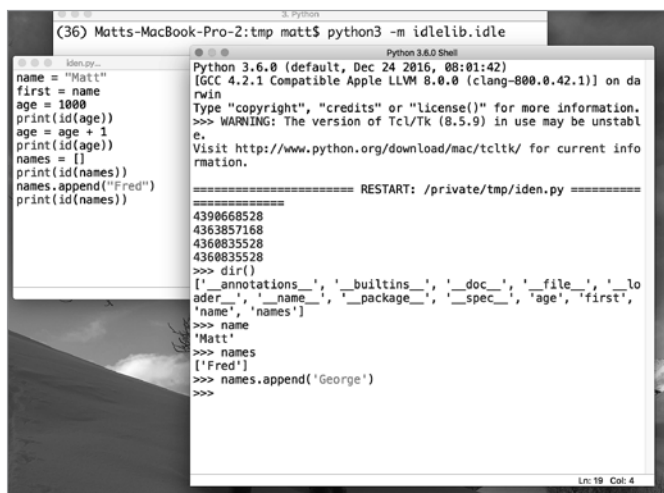


Рис. 7.3. При присоединении объекта к списку изменяется значение списка. При добавлении и удалении элементов идентификатор списка не изменяется



The screenshot shows the IDLE Python IDE. On the left, a file named `iden.py` contains the following Python code:

```
name = "Matt"
first = name
age = 1000
print(id(age))
age = age + 1
print(id(age))
names = []
print(id(names))
names.append("Fred")
print(id(names))
```

On the right, the Python 3.6.0 Shell (REPL) displays the output of the script execution:

```
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on da
rwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstabl
e.
Visit http://www.python.org/download/mac/tcltk/ for current info
rmation.

===== RESTART: /private/tmp/iden.py =====
4390668528
4363857168
4360835528
4360835528
>>> dir()
['_annotations_', '__builtins__', '__doc__', '__file__', '__lo
ader__', '__name__', '__package__', '__spec__', 'age', 'first',
'name', 'names']
>>> name
'Matt'
>>> names
['Fred']
>>> names.append('George')
>>>
```

Рис. 7.4. Выполнение кода и анализ результатов в REPL в IDLE.
Если вы не используете IDLE, узнайте, как это делается в вашем редакторе

7.5. Итоги

В Python нет ничего, кроме объектов. Объекты обладают тремя свойствами:

- Тип — определяет класс для объекта.
- Значение — данные, содержащиеся в объекте. Проверяя два объекта на равенство (`==`), вы сравниваете их значения.
- Идентификатор — уникальный идентификатор объекта. В версии Python на сайте www.python.org идентификатор фактически определяет местонахождение объекта в памяти, которое является уникальным значением. Когда вы проверяете два объекта на тождественность (оператором `is`), вы проверяете, совпадают ли их идентификаторы.

Вы также узнали, что изменяемые объекты (например, списки) могут изменять свое значение, тогда как неизменяемые объекты (например, строки и списки) изменяться «на месте» не могут.

7.6. Упражнения

1. Создайте переменную, которая указывает на число с плавающей точкой. Проанализируйте идентификатор, тип и значение этого числа. Обновите переменную, увеличив ее на 20. Снова проанализируйте идентификатор, тип и значение. Изменился ли идентификатор? Изменилось ли значение?
2. Создайте переменную, которая указывает на пустой список. Проанализируйте идентификатор, тип и значение списка. Присоедините к списку число 300. Снова проанализируйте идентификатор, тип и значение. Изменился ли идентификатор? Изменилось ли значение?

8

Числа

В этой главе рассматриваются операции с числами в языке Python. В Python поддерживаются *целые числа* (такие, как -1 , 5 или 2000) и *числа с плавающей точкой* (приближенное компьютерное представление таких чисел, как $0,333$, $0,5$ или $-1000,234$); они позволяют легко выполнять числовые операции. Изначально в Python предусмотрена поддержка операций сложения, вычитания, умножения, деления, возведения в степень, вычисления остатка и многих других операций!

В отличие от других языков, в Python существуют только объекты — это относится и к числам. Целые числа относятся к классу `int`:

```
>>> type(1)
<class 'int'>
```

Числа с плавающей точкой относятся к классу `float`:

```
>>> type(2.0)
<class 'float'>
```

Точность представления нецелых чисел ограничена. Пользователь должен сам определить, достаточно ли ее для его текущих вычислений. Во внутреннем представлении чисел с плавающей точкой Python использует двоичное представление (соответствующее стандарту IEEE 754 для чисел с плавающей точкой). У чисел с плавающей точкой существует определенный порог точности, поэтому возможна погрешность округления. Собственно, разработчик всегда должен ожидать, что вычисления выполняются с погрешностью округления. (Если вам потребуется более

высокая точность, модуль `decimal` предоставляет более точную, хотя и более медленную реализацию.)

Рассмотрим точность представления на простом примере. Что происходит при выполнении простой на первый взгляд операции вычитания?

```
>>> print(1.01 - .99)
0.0200000000000000018
```

СОВЕТ

Если вам захочется больше узнать о числах с плавающей точкой и их представлении в памяти компьютера, в Википедии¹ имеется более подробная информация по теме.

8.1. Сложение

Python REPL может использоваться как простейший калькулятор. Чтобы сложить два целых числа, введите выражение:

```
>>> 2 + 6
8
```

ПРИМЕЧАНИЕ

В этом примере результат не связывается с переменной. Для вывода результата простейшего вычисления на терминал этого может быть достаточно. Если результат вдруг понадобится вам позднее, интерпретатор Python сохраняет последний результат в переменной с именем `_`:

```
>>> 2 + 6
8
>>> result = _
>>> result
8
```

При сложении двух целых чисел будет получено целое число. Аналогичным образом можно сложить два числа с плавающей точкой:

```
>>> .4+.01
0.41000000000000003
```

¹ https://ru.wikipedia.org/wiki/Число_с_плавающей_запятой

Этот пример снова показывает, почему при работе с числами с плавающей точкой необходима осторожность: операции могут приводить к потере точности (настоящий результат равен 0,41).

А что произойдет при сложении целого числа с числом с плавающей точкой?

```
>>> 6 + .2
6.2
```

Python решает, что, поскольку вы складываете целое число с числом с плавающей точкой, вам нужна арифметическая операция с плавающей точкой. В данном случае Python автоматически *преобразует* 6 в число с плавающей точкой перед тем, как складывать его с .2. Python возвращает результат в формате с плавающей точкой.

ПРИМЕЧАНИЕ

Если в операции используются два числовых операнда, преобразование обычно работает правильно. Для операций, в которых задействовано целое число и число с плавающей точкой, целое число преобразуется в формат с плавающей точкой.

ПРИМЕЧАНИЕ

Преобразования между строками и числами не выполняются для большинства математических операций. У правила есть два исключения: оператор форматирования строки и оператор умножения.

При использовании % со строкой в левой части (*левый операнд*) и любым объектом (включая числа) в правой части (*правый операнд*) Python выполняет операцию форматирования:

```
>>> print('num: %s' % 2)
num: 2
```

Если левый операнд является строкой, то при применении оператора умножения * Python выполняет повторение:

```
>>> 'Python!' * 2
'Python!Python!'

>>> '4' * 2
'44'
```

ПРИМЕЧАНИЕ

С классами `int` и `float` могут выполняться явные преобразования (несмотря на внешнее сходство с функциями, в действительности это классы):

```
>>> int(2.3)
2

>>> float(3)
3.0
```

8.2. Вычитание

Вычитание имеет много общего с умножением. При вычитании двух целых чисел или двух чисел с плавающей точкой будет получено целое число или число с плавающей точкой соответственно. Для смешанных числовых типов перед вычитанием выполняется преобразование операндов:

```
>>> 2 - 6
-4

>>> .25 - 0.2
0.04999999999999999

>>> 6 - .2
5.8
```

8.3. Умножение

Во многих языках программирования символ `*` (звездочка) используется для умножения. Вероятно, вы и сами догадаетесь, что произойдет при умножении двух целых чисел:

```
>>> 6 * 2
12
```

Если вы внимательно читали, то уже знаете, что происходит при умножении двух чисел с плавающей точкой:

```
>>> .25 * 12.0
3.0
```

А при смешанных типах в произведении будет получен результат с плавающей точкой:

```
>>> 4 * .3
1.2
```

Результат с плавающей точкой в этих примерах выглядит правильным. Тем не менее будьте внимательны и помните о проблемах с погрешностью представления — не стоит полагать, что вам всегда будет везти.

8.4. Деление

В Python (как и во многих языках) знак / обозначает деление:

```
>>> 12 / 4
3.0
```

В Python 3 также решена проблема, которую в предыдущих версиях Python многие считали дефектом. Результат деления двух целых чисел является числом с плавающей точкой:

```
>>> 3 / 4
0.75
```

Прежде в Python выполнялось целочисленное деление. Если вам нужен именно такой вариант поведения, используйте оператор //. Какое целое число использует Python? Результат *округления в меньшую сторону*:

```
>>> 3 // 4
0
```

8.5. Остаток

Оператор % вычисляет остаток от целочисленного деления. Например, с его помощью можно проверить число на четность (или узнать, были ли обработаны 1000 элементов списка):

```
# Остаток от деления 4 на 3
>>> 4 % 3
1
>>> 3 % 2 # нечетно, если результат равен 1
1
>>> 4 % 2 # четно, если результат равен 0
0
```

СОВЕТ

Будьте внимательны при использовании оператора % с отрицательными числами. Поведение оператора изменяется в зависимости от того, какой из операндов отрицателен. Понятно, что при отсчете в обратном направлении остаток должен повторяться с определенной периодичностью:

```
>>> 3 % 3
0
>>> 2 % 3
2
>>> 1 % 3
1
>>> 0 % 3
0
```

Какой результат должен быть получен при вычислении $-1 \% 3$? Так как отсчет ведется в обратном направлении, результат снова должен быть равен 2:

```
>>> -1 % 3
2
```

Но если изменить знак делителя, происходит нечто странное:

```
>>> -1 % -3
-1
```

Python гарантирует, что знак результата совпадает со знаком делителя (или равен 0). Чтобы окончательно запутать вас, еще один пример:

```
>>> 1 % -3
-2
```

Мораль: пожалуй, вам не стоит вычислять остаток с отрицательным делителем — только если вы твердо уверены в том, что вам действительно нужно именно это.

8.6. Возведение в степень

Python также поддерживает оператор *возведения в степень* `**` (две звездочки). Если вы хотите возвести 4 в квадрат (4 — основание, 2 — показатель степени), это делается так:

```
>>> 4 ** 2
16
```

Результаты операции возведения в степень обычно стремительно увеличиваются. Допустим, вы хотите возвести 10 в 100-ю степень:

[illegible]

Для хранения целых чисел программам требуется определенный объем памяти. Так как целые числа обычно относительно невелики, Python оптимизирует затраты памяти, чтобы избежать ее непроизводительного расходования. Во внутренней реализации «системные» целые числа могут быть преобразованы в длинные целые для хранения больших чисел. Python 3 делает это за вас автоматически.

Представьте шкалу пружинных весов. Если вы всегда взвешиваете мелкие предметы, вам будут удобны весы с короткой шкалой. Если вы торгуете песком, вероятно, песок будет удобно насыпать в мешки, чтобы с ними было удобнее работать. Вы будете держать под рукой небольшие мешки, чтобы использовать их для взвешивания. Но если время от времени вам приходится взвешивать большие предметы, выходящие за короткую шкалу, придется доставать большие весы и большой мешок. Было бы неразумно использовать большой мешок и большие весы для многих мелких предметов.

Точно так же Python старается оптимизировать память для хранения целых чисел, ориентируясь на небольшие размеры. Если целое число не помещается в малый блок памяти (мешок), Python *преобразует* его в *большое целое*. На самом деле это желательное поведение, потому что в некоторых средах *ошибка переполнения* приводит к аварийному завершению программы (или РасМан отказывается переходить на уровни выше 255, потому что счетчик уровня хранится в виде 8-разрядного числа).

ПРИМЕЧАНИЕ

Python включает модуль `operator` с функциями для основных математических операций. В частности, он пригодится вам при использовании таких расширенных возможностей Python, как *лямбда-функции* или *генераторы списков*:

```
>>> import operator
>>> operator.add(2, 4) # same as 2 + 4
6
```

8.7. Порядок операций

При выполнении математических вычислений не все операции выполняются слева направо. Например, операции сложения и вычитания выполняются после операций умножения и деления. Компьютеры работают по тем же правилам. Если вы хотите, чтобы операции сложения (или вычитания) выполнялись в первую очередь, используйте круглые скобки для обозначения порядка операций:

```
>>> 4 + 2 * 3
10
>>> (4 + 2) * 3
18
```

Как следует из примера, выражение в круглых скобках вычисляется в первую очередь.

8.8. Другие операции

Справочный раздел REPL весьма полезен. В нем имеется тема `NUMBERMETHODS`, объясняющая, как работают все числовые операции.

8.9. Итоги

В Python имеется встроенная поддержка основных математических операций. Разумеется, язык поддерживает сложение, вычитание, умножение и деление. Кроме того, доступны операции возведения в степень и вычис-

ления остатка. Если вам понадобится управлять порядком выполнения операций, заключите ту операцию, которая должна выполняться первой, в круглые скобки. Если вам нужно провести простые вычисления, вместо запуска приложения-калькулятора можно выполнить их в Python. Возможности языка более чем достаточны для большинства задач.

8.10. Упражнения

1. В эту неделю вы спали 6,2, 7, 8, 5, 6,5, 7,1 и 8,5 часа. Вычислите среднюю продолжительность сна в часах.
2. Делится ли число 297 без остатка на 3?
3. Чему равен результат возведения 2 в десятую степень?
4. В Википедии¹ високосный год определяется следующим образом:

«...високосным оставался год, номер которого кратен четырем, но исключение делалось для тех, которые были кратны 100. Такие годы были високосными только тогда, когда делились еще и на 400... Так, годы 1700, 1800 и 1900 не являются високосными, так как они кратны 100 и не кратны 400. Годы 1600 и 2000 — високосные».

Напишите код Python, который определяет, являются ли високосными годы 1800, 1900, 1903, 2000 и 2002.

¹ https://ru.wikipedia.org/wiki/Високосный_год

9

Строки

Строки представляют собой неизменяемые объекты для хранения символьных данных. Строка может содержать один символ, слово, последовательность слов, абзац, несколько абзацев, хотя может не содержать ни одного символа.

В Python строки заключаются между символами ' (одинарные кавычки), " (двойные кавычки), """ (тройные двойные кавычки) или ''' (тройные одинарные кавычки). Несколько примеров:

```
>>> character = 'a'
>>> name = 'Matt'
>>> with_quote = "I ain't gonna"
>>> longer = """This string has
... multiple lines
... in it"""
>>> latin = '''Lorum ipsum
... dolor'''
>>> escaped = 'I ain\'t gonna'
>>> zero_chars = ''
>>> unicode_snake = "I love \N{SNAKE}"
```

Обратите внимание: строки всегда начинаются и завершаются кавычками одного типа. Как показано в примере `with_quote`, одинарные кавычки могут находиться внутри строк, заключенных в двойные кавычки, и наоборот. Кроме того, если вам понадобится включить в строку кавычку того же типа, ее можно *экранировать* символом \ (обратная косая черта). При выводе экранированного символа обратная косая черта игнорируется.

ПРИМЕЧАНИЕ

У наблюдательного читателя может возникнуть вопрос — как включить в строку символ обратной косой черты? Чтобы включить обратную косую черту в обычную строку, ее необходимо экранировать... да, все верно — еще одной обратной косой чертой:

```
>>> backslash = '\\'  
>>> print(backslash)  
\  

```

ПРИМЕЧАНИЕ

Несколько стандартных способов экранирования символов в Python:

Экранирующая последовательность	Вывод
\\	Обратная косая черта
\'	Одинарная кавычка
\"	Двойная кавычка
\b	ASCII-символ Backspace
\n	Новая строка
\t	Табуляция
\u12af	16-разрядный символ Юникода
\U12af89bc	32-разрядный символ Юникода
\N{SNAKE}	Символ Юникода
\o84	Символ в восьмеричной кодировке
\xFF	Шестнадцатеричный символ

СОВЕТ

Если вы не хотите применять экранирование, используйте *необработанные* (raw) строки, поставив перед строкой префикс `r`. Необработанные строки обычно встречаются в двух местах. Они используются в *регулярных выражениях*, в которых обратная косая черта также используется как экранирующий символ. Регулярные выражения применяются для поиска символов в тексте по шаблону (например, телефонных номеров, имен и т. д.). Модуль `re` из стандартной библиотеки Python предоставляет поддержку

регулярных выражений. Кроме того, необработанные строки также используются в путях Windows, где обратная косая черта интерпретируется как разделитель.

В необработанных строках символы интерпретируются буквально (то есть без всякого экранирования). Следующий пример демонстрирует различия между необработанными и обычными строками:

```
>>> slash_t = r'\tText \\'
>>> print(slash_t)
\tText \

>>> normal = '\tText \\'
>>> print(normal)
Text \
```

В Python также предусмотрен механизм тройных кавычек для определения строк. Тройные кавычки удобны для создания строк, разделенных на абзацы. Кроме того, строки в тройных кавычках часто используются в *строках документации*. Строки документации будут рассматриваться в главе, посвященной функциям. Ниже приведен пример строки в тройных кавычках:

```
>>> paragraph = """Lorem ipsum dolor
... sit amet, consectetur adipisicing
... elit, sed do eiusmod tempor incididunt
... ut labore et dolore magna aliqua. Ut
... enimad minim veniam, quis nostrud
... exercitation ullamco laboris nisi ut
... aliquip ex ea commodo consequat. Duis
... aute irure dolor in reprehenderit in
... voluptate velit esse cillum dolore eu
... fugiat nulla pariatur. Excepteur sint
... occaecat cupidatat non proident, sunt
... in culpa qui officia deserunt mollit
... anim id est laborum."""
```

Строки в тройных кавычках удобны тем, что в них можно вставлять одинарные и двойные кавычки без экранирования:

```
>>> """This string has double " and single
... quotes ' inside of it"""
'This string has double " and single\nquotes \' inside of it'
```

Впрочем, если кавычки приходятся на конец строки, последнюю кавычку в тексте необходимо экранировать:

```
>>> """He said, "Hello"""
  File "<stdin>", line 1
    """He said, "Hello"""
                                ^
SyntaxError: EOL while scanning string literal
```

```
>>> """He said, "Hello\""""
'He said, "Hello"'
```

9.1. Форматирование строк

Хранить строки в переменных удобно, но необходимо также иметь возможность собирать строки из других строк и выполнять с ними нужные манипуляции. Для этой цели можно воспользоваться механизмом форматирования строк.

В Python 3 предпочтительным способом форматирования считается использование метода `.format`. В следующем примере мы приказываем Python заменить `{}` (заполнитель) содержимым переменной `name`, то есть строкой `Matt`:

```
>>> name = 'Matt'
>>> print('Hello {}'.format(name))
Hello Matt
```

Другое полезное свойство форматирования заключается в том, что форматировать также можно нестроковые объекты — например, числа:

```
>>> print('I:{} R:{} S:{}'.format(1, 2.5, 'foo'))
I:1 R:2.5 S:foo
```

9.2. Синтаксис форматных строк

В форматных строках существует специальный синтаксис для *полей-заполнителей*. Если форматной строке передается объект, для поиска атрибутов может использоваться синтаксис `.имя_атрибута`. Также поддерживается возможность извлечения индексируемых элементов с использованием записи `[индекс]`. В документации Python они называются

именами полей. Имена полей могут быть пустыми, содержать имя аргумента с ключевым словом, номер позиционного аргумента или индекс в списке или словаре (в квадратных скобках):

```
>>> 'Name: {}'.format('Paul')
'Name: Paul'

>>> 'Name: {name}'.format(name='John')
'Name: John'

>>> 'Name: {[name]}'.format({'name': 'George'})
'Name: George'
```

В фигурные скобки ({ и }) также может быть заключено целое число. Оно определяет позицию аргумента, переданного `.format` (нумерация начинается с нуля). В следующем примере используются номера позиционных аргументов. Первый аргумент, `.format, 'Paul'`, находится в позиции 0; второй, `'George'`, находится в позиции 1; наконец, `'John'` находится в позиции 2:

```
>>> 'Last: {2} First: {0}'.format('Paul', 'George',
...                               'John')
'Last: John First: Paul'
```

Существует целый язык форматирования строк. Поставив двоеточие после имени поля, вы сможете передать дополнительную форматную информацию. Формат описан ниже. Все, что заключено в квадратные скобки, не является обязательным.

```
: [[fill]align][sign][#][0][width][grouping_option][.precision][type]
```

В следующей таблице перечислены поля и их значения.

Поле	Описание
fill	Символ, используемый для заполнения из align (пробел по умолчанию)
align	Тип выравнивания вывода: < (по левому краю), > (по правому краю), ^ (по центру) или = (вставить дополнение после знака)
sign	Для чисел: + (выводить знак как для положительных, так и для отрицательных чисел), (по умолчанию, выводить знак только для отрицательных чисел) или пробел (начальные пробелы для положительных чисел, знак для отрицательных чисел)

Поле	Описание
#	Префикс для целых чисел: 0b (двоичные), 0o (восьмеричные) или 0x (шестнадцатеричные)
0	Дополняющие нули
width	Минимальная ширина поля
grouping_option	Разделители в числах: , (тысячи разделяются запятыми), _ (тысячи разделяются подчеркиваниями)
.precision	Для чисел с плавающей точкой (количество знаков в дробной части), для нечисловых данных (максимальная длина)
type	Числовой тип или s (строковый формат по умолчанию); см. таблицы форматирования для целых чисел и чисел с плавающей точкой

В следующих таблицах перечислены различные средства форматирования целых чисел и чисел с плавающей точкой.

Целые типы	Описание
b	Двоичный
c	Символьный — преобразуется в символ Юникода
d	Десятичный (по умолчанию)
n	Десятичный с разделителями для локального контекста
o	Восьмеричный
x	Шестнадцатеричный (нижний регистр)
X	Шестнадцатеричный (верхний регистр)

Типы с плавающей точкой	Описание
e/E	Экспоненциальная запись (буква «e» в нижнем/верхнем регистре)
f	Фиксированная точка
g/G	Универсальная запись. Фиксированная или экспоненциальная запись в зависимости от числа (по умолчанию g)
n	g с разделителями для локального контекста
%	Проценты (с умножением на 100)

9.3. Примеры форматирования

Ниже приведены некоторые примеры использования `.format`. Чтобы отформатировать строку в центре поля из 12 символов, окруженную символами `*`, используйте приведенный ниже код. Здесь `*` — символ-заполнитель, `^` — поле выравнивания, а `12` — поле ширины:

```
>>> "Name: {:^12}".format("Ringo")
'Name: ***Ringo****'
```

Затем отформатируем число в процентах с шириной 10, одним знаком в дробной части и знаком перед дополнением ширины. Здесь `=` — поле выравнивания, `+` гарантирует, что в отформатированном значении всегда выводится знак (как для отрицательных, так и для положительных чисел), `10.1` — поля ширины и точности, а `%` — тип для преобразования числа в проценты:

```
>>> "Percent: {:=+10.1%}".format(-44/100)
'Percent: - 44.0%'
```

Далее следуют примеры двоичного и шестнадцатеричного преобразования. Полю *типа целого числа* присваиваются значения `b` и `x` соответственно:

```
>>> "Binary: {:b}".format(12)
'Binary: 1100'
```

```
>>> "Hex: {:x}".format(12)
'Hex: c'
```

ПРИМЕЧАНИЕ

Метод `.format` для строк предоставляет альтернативу для оператора `%`, сходную с функцией `C printf`. Оператор `%` также доступен, и некоторые пользователи предпочитают именно его, поскольку в простых случаях запись получается короче, а также из-за его сходства с `C`. Заполнители `%s`, `%d` и `%x` заменяются своими строковыми, целыми и шестнадцатеричными значениями соответственно. Несколько примеров:

```
>>> "Num: %d Hex: %x" % (12, 13)
'Num: 12 Hex: d'

>>> "%s %s" % ('hello', 'world')
'hello world'
```

СОВЕТ

Отличный источник информации о форматировании — встроенная справочная документация, доступная в REPL. Введите команду

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

Команда включает режим справочной информации и выводит приглашение `help>`. Введите команду

```
help> FORMATTING
```

Просмотрите вывод — вы найдете в нем много примеров. Если просто нажать `Enter` в приглашении `help>`, вы вернетесь к обычному приглашению.

Также немало полезной информации можно найти на сайте <https://pyformat.info/>. Здесь представлено много примеров форматирования как с использованием `.format`, так и с более старым оператором `%`.

Информация о форматировании строк также содержится в разделе **STRINGS** справочной информации.

9.4. F-строки

В Python 3.6 появилась новая разновидность строк — так называемые *f-строки*. Если поставить перед строкой префикс `f`, в заполнителе можно будет включить код. Простой пример:

```
>>> name = 'matt'
>>> f'My name is {name}'
'My name is matt'
```

Python просматривает заполнитель и вычисляет содержащийся там код. В заполнитель можно включить вызовы функций, вызовы методов или любой другой произвольный код:

```
>>> f'My name is {name.capitalize()}'
'My name is Matt'
```

Также можно передать форматные строки после двоеточия (`:`):

```
>>> f'Square root of two: {2**.5:5.3f}'
'Square root of two: 1.414'
```

9.5. Итоги

В этой главе мы представили строки. Строки могут определяться с разными ограничителями. В отличие от других языков, в которых строки, заключенные в двойные кавычки " и одинарные кавычки ', могут работать по-разному, в Python их поведение не различается. Впрочем, строки в тройных кавычках могут состоять из нескольких физических строк (абзацев).

Мы также рассмотрели метод `.format` и разобрали несколько примеров форматирования строк. В завершение главы была представлена одна из новых возможностей Python 3.6 — f-строки.

9.6. Упражнения

1. Создайте переменную `name`, указывающую на ваше имя. Создайте другую переменную — `age` — с целым числом, соответствующим вашему возрасту. Выведите отформатированную строку с обоими значениями. Например, для имени Fred и возраста 23 должна быть выведена строка:

```
Fred is 23
```

2. Создайте переменную `paragraph`, которая содержит следующий текст:

```
"Python is a great language!", said Fred. "I don't
ever remember having this much fun before."
```

3. Посетите сайт <https://unicode.org> и найдите в греческой таблице символов букву «омега». Создайте строку, которая содержит символ «омега», используя как кодовый пункт в Юникоде (форма `\u`), так и имя символа в Юникоде (форма `\N`). Кодовый пункт — шестнадцатеричное число в таблице, имя выводится жирным шрифтом после кодового пункта. Например, букве «тэта» соответствует кодовый пункт `03f4` и имя `GREEK CAPITAL THETA SYMBOL`.

4. Создайте переменную `item`, которая указывает на строку `"car"`. Создайте переменную `cost`, которая указывает на число `13499.99`. Выведите строку, в которой значение `item` выравнивается по левому краю поля из 10 символов, а `cost` — по правому краю поля из 10 символов с 2 цифрами в дробной части и разделением тысяч запятыми. Результат должен выглядеть так (без кавычек):

```
'car'           13,499.99'
```

10

dir, help и pdb

Наше знакомство со строками было в лучшем случае поверхностным, но сейчас нужно ненадолго прерваться для обсуждения двух важных функций и одной библиотеки, включенных в поставку Python. Первая функция — `dir` — показывает, каким мощным и полезным инструментом является REPL. Функция `dir` возвращает атрибуты объекта. Если у вас открыт интерпретатор Python и вы хотите узнать атрибуты строки, эта функция выдает следующий результат:

```
>>> dir('Matt')

['_add_', '__class__', '__contains__',
 '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__',
 '__getitem__', '__getnewargs__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'capitalize',
 'casefold', 'center', 'count', 'encode', 'endswith',
 'expandtabs', 'find', 'format', 'format_map',
 'index', 'isalnum', 'isalpha', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric',
 'isprintable', 'isspace', 'istitle', 'isupper',
 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase',
 'title', 'translate', 'upper', 'zfill']
```


`dir` выводит все атрибуты переданного объекта. Так как функции `dir` была передана строка `'Matt'`, функция выводит атрибуты строки. Эта удобная возможность языка Python демонстрирует его философию по принципу «батарейки прилагаются»: Python предоставляет простой механизм для получения атрибутов любого объекта. В других языках для получения доступа к такой функциональности могут потребоваться специальные сайты, документация или IDE, но в Python благодаря наличию REPL эту информацию можно получить быстро и легко.

Список атрибутов упорядочен по алфавиту; на первую пару атрибутов, начинающихся с `__`, обычно можно не обращать внимания. Позднее вам встретятся такие атрибуты, как `capitalize` (запись строки с прописной буквы), `format` (форматирование строк, продемонстрированное выше) или `lower` (преобразование строки к нижнему регистру). Эти атрибуты являются *методами* — функциями, связанными с объектами. Чтобы *вызвать* функцию, поставьте точку после объекта, укажите имя метода и поставьте круглые скобки.

Три примера вызова методов:

```
>>> print('matt'.capitalize())
Matt

>>> print('Hi {}'.format('there'))
Hi there

>>> print('YIKES'.lower())
yikes
```

10.1. Специальные методы

Возможно, вас интересуют атрибуты, начинающиеся с `__`. Как было сказано выше, эти атрибуты соответствуют методам; люди называют их *специальными, волшебными* или *дандер-методами* (dunder methods). Специальные методы определяют, что происходит во внутренней реализации при выполнении операций с объектами. Например, при применении оператора `+` или `%` к строке будет вызван метод `.__add__` или `.__mod__` соответственно. Неопытные программисты Python обычно забывают о специальных методах. Но когда вы начнете реализовывать собственные классы и захотите, чтобы они реагировали на такие операции, как `+` или `%`, вы можете определить их.

СОВЕТ

В документации `help()` имеется тема `SPECIALMETHODS` с описаниями этих методов.

Также описания доступны на сайте Python. Откройте раздел Documentation, Language Reference, Data Model — они находятся здесь.

10.2. help

`help` — другая встроенная функция, используемая в сочетании с REPL. В книге ранее уже упоминался вызов `help()` без аргументов для вызова справочной документации.

Функция `help` также выводит документацию для метода, модуля, класса или функции, переданной в аргументе. Например, если вас интересует, что делает атрибут `upper` у строк, информация может быть получена следующим вызовом:

```
>>> help('some string'.upper)
Help on built-in function upper:

upper(...) method of builtins.str instance
    S.upper() -> str

    Return a copy of S converted to uppercase.
```

Функция `help` в сочетании с REPL позволяет читать документацию, не открывая браузера, и даже без доступа к интернету. Даже если вы попали на необитаемый остров, все равно сможете изучать Python — для этого понадобится лишь компьютер с установленной версией Python и источник питания.

10.3. pdb

В Python также включен отладчик для пошагового выполнения кода. Он находится в модуле с именем `pdb`. Эта библиотека построена по образцу библиотеки `gdb` языка C. Чтобы выйти в отладчик в любой точке программы Python, вставьте в нее следующий код:

```
import pdb; pdb.set_trace()
```

На самом деле здесь две команды, но я обычно размещаю их в одной строке, разделив точкой с запятой (;) — так после завершения отладки обе команды можно будет удалить одним нажатием клавиши в редакторе. Пожалуй, это единственная ситуация, в которой я использую точку с запятой в коде Python (две команды в одной строке).

При выполнении в этой строке открывается приглашение (pdb), напоминающее приглашение REPL. В нем также можно выполнять код, просматривать объекты и переменные. Кроме того, вы сможете установить точки прерывания для последующего анализа.

Ниже приведена таблица со списком полезных команд pdb:

Команда	Описание
h, help	Вывести список доступных команд
n, next	Выполнить следующую строку
c, cont, continue	Продолжать выполнение до следующей точки прерывания
w, where, bt	Вывести трассировку стека с информацией о текущей позиции выполнения
u, up	Подняться на уровень вверх в стеке
d, down	Опуститься на уровень вниз в стеке
l, list	Вывести исходный код текущей строки

ПРИМЕЧАНИЕ

В книге «Programming Pearls» Джон Бентли (Jon Bentley) пишет:

«Когда мне приходится отлаживать маленький алгоритм где-то в недрах большой программы, я иногда использую средства отладки... хотя команды отладочного вывода обычно быстрее реализуются и работают эффективнее хитроумных отладчиков».

Я слышал, как Гвидо ван Россум, создатель Python, высказывает то же мнение: он предпочитает *отладочный вывод*. Принцип отладочного вывода прост: вы вставляете функции `print`, которые поясняют, что происходит в программе. Часто этого оказывается достаточно для выявления проблемы. Не забудьте удалить команды отладки или преобразовать их в команды записи в журнал перед выпуском кода.

Если потребуется более серьезный анализ, вы всегда можете воспользоваться модулем `pdb`.

10.4. Итоги

Python предоставляет много вспомогательных инструментов, упрощающих жизнь разработчика. Если вы научились работать с REPL, то сможете пользоваться ими. Функция `dir` выводит информацию о том, какими атрибутами обладает объект. Функция `help` поможет проанализировать эти атрибуты для документации.

В этой главе также был представлен модуль `pdb`. Этот модуль позволяет выполнять код в пошаговом режиме, что может быть полезно при отладке.

10.5. Упражнения

1. Откройте REPL и создайте переменную `name` со своим именем. Выведите атрибуты строки. Выведите справочную документацию для методов `.find` и `.title`.
2. Откройте REPL и создайте переменную `age` со своим возрастом. Выведите атрибуты целого числа. Выведите справочную документацию для метода `.numerator`.

11

Строки и методы

В главе 10 вы узнали о встроенной функции `dir` и некоторых методах, которые можно вызвать для строковых объектов. Так как строки являются неизменяемыми, эти методы не изменяют строку, а возвращают новую строку или новый результат. Операции со строками позволяют создать новую версию, преобразованную к верхнему регистру, вернуть отформатированную строку, создать строку в нижнем регистре и т. д. Для выполнения этих операций разработчик вызывает *методы*.

Методы представляют собой функции, вызываемые для экземпляра типа. Что это означает? Чтобы вызвать метод для строкового объекта, поставьте точку (.) и укажите имя объекта прямо за именем переменной, хранящей данные (или самими данными). За именем объекта должны следовать круглые скобки со списком аргументов.

ПРИМЕЧАНИЕ

В этой книге я ставлю точку перед именами методов. Это делается для того, чтобы напомнить вам о том, что перед методом должен быть указан объект. Скажем, вместо метода `capitalize` в тексте будет упоминаться метод `.capitalize`. Вызов его для объекта `text` выглядит так:

```
text.capitalize()
```

В этом отношении методы отличаются от функций (таких, как `help`), которые вызываются самостоятельно (без объекта или точки):

```
help()
```

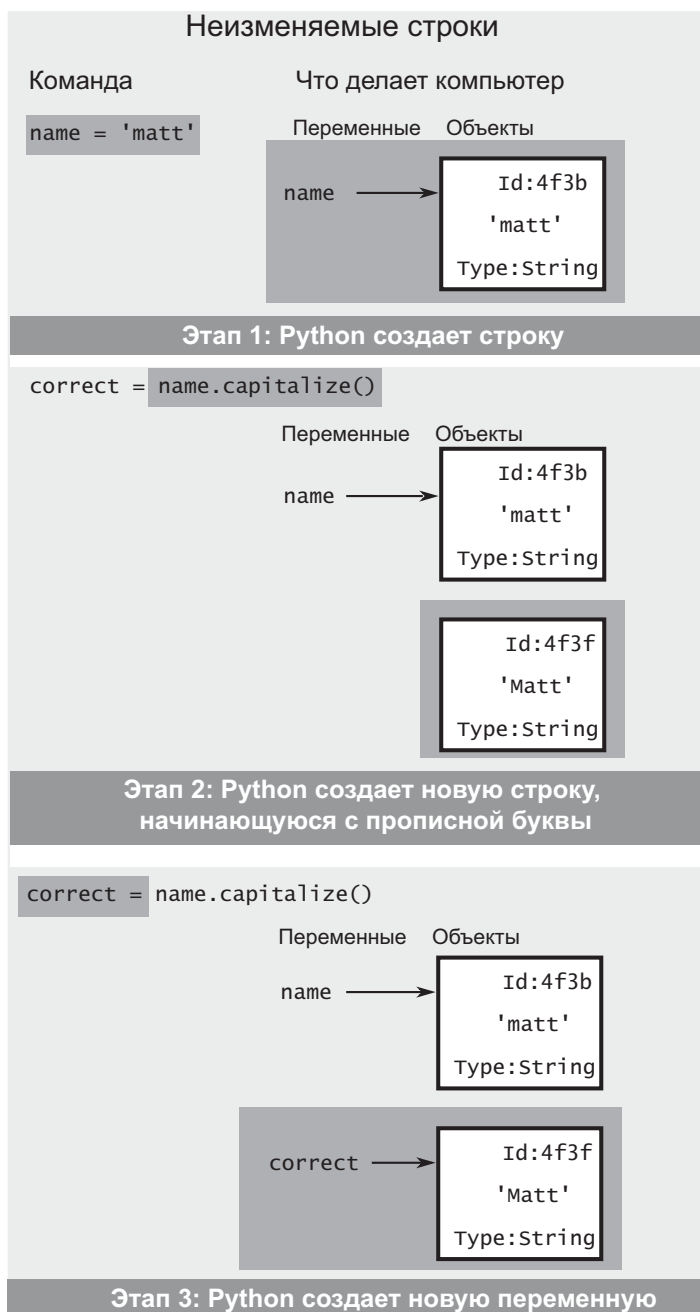


Рис. 11.1. Вызов метода для строки. Метод не изменяет саму строку, потому что строки неизменяемы. Вместо этого метод возвращает новую строку

В следующем примере метод `.capitalize` вызывается для переменной, указывающей на строку, и для строкового литерала. Обратите внимание: объект, для которого вызывается метод, при этом не изменяется. Так как строки неизменяемы, результатом метода является новый объект со значением, записанным с прописной буквы:

```
>>> name = 'matt'

# вызывается для переменной
>>> correct = name.capitalize()
>>> print(correct)
Matt
```

Обратите внимание: `name` при этом не изменяется:

```
>>> print(name)
matt
```

Метод `.capitalize` не обязательно вызывать для переменной. Его также можно вызвать прямо для строкового литерала:

```
>>> print('fred'.capitalize())
Fred
```

В Python методы и функции являются *полноправными объектами*. Как упоминалось ранее, в Python вообще нет ничего, кроме объектов. Если опустить круглые скобки, Python не выдаст сообщения об ошибке, а выведет ссылку на метод, который является объектом:

```
>>> print('fred'.capitalize)
<built-in method capitalize of str object at
0x7ff648617508>
```

Существование методов и функций как полноправных объектов позволяет использовать такие нетривиальные возможности, как *замыкания* и *декораторы* (они рассматриваются в моем учебнике Python промежуточного уровня).

ПРИМЕЧАНИЕ

А у целых чисел и чисел с плавающей точкой тоже есть методы? Да, еще раз повторю: в Python нет ничего, кроме объектов, и у объектов есть методы. В этом легко убедиться, вызвав `dir` для целого числа (или переменной, в которой хранится целое число):

```
>>> dir(4)
['__abs__', '__add__', '__and__',
 '__class__',
 ...
 '__subclasshook__', '__truediv__',
 '__trunc__', '__xor__', 'conjugate',
 'denominator', 'imag', 'numerator',
 'real']
```

Вызов метода для числа несколько усложняется использованием точки для обозначения вызова метода. Так как точка часто встречается в числах с плавающей точкой, ее использование для вызова методов для чисел может привести Python в замешательство.

Например, метод `.conjugate` возвращает комплексное сопряжение целого числа. Но при попытке вызвать его для целого числа вы получите ошибку:

```
>>> 5.conjugate()
Traceback (most recent call last):
...
5.conjugate()
      ^
SyntaxError: invalid syntax
```

Одно из возможных решений — заключить число в круглые скобки:

```
>>> (5).conjugate()
5
```

Также можно присвоить переменную 5 и вызвать метод для переменной:

```
>>> five = 5
>>> five.conjugate()
5
```

Впрочем, на практике вызов методов для чисел встречается довольно редко.

11.1. Основные строковые методы

Некоторые методы строк часто используются в программах и обнаруживаются в стороннем коде. Информацию о других можно найти при помощи `dir` и `help` (или электронной документации).

11.2. endswith

Допустим, у вас имеется переменная с именем файла и вы хотите проверить расширение. Задача легко решается методом `.endswith`:

```
>>> x1 = 'Oct2000.xls'
>>> x1.endswith('.xls')
True
>>> x1.endswith('.xlsx')
False
```

ПРИМЕЧАНИЕ

Обратите внимание: при вызове метода вы должны были передать *параметр* (или *аргумент*) `'xls'`. У методов имеется *сигнатура* — этот пугающий термин означает, что они должны вызываться с правильным количеством (и типами) параметров. Для метода `.endswith` вполне логично, что, если вы хотите узнать, кончается ли строка другой строкой, нужно сообщить Python, какое именно завершение нужно проверить. Для этого методу передается строка-завершитель.

СОВЕТ

И снова всю информацию такого рода можно легко найти при помощи `help`. В документации должно быть указано, какие параметры являются обязательными, а также должны быть перечислены все необязательные параметры. Справка для `endswith` выглядит так:

```
>>> help(x1.endswith)
Help on built-in function endswith:

endswith(...)
    S.endswith(suffix[, start[, end]]) -> bool

    Return True if S ends with the specified
```

```
suffix, False otherwise. With optional
start, test S beginning at that position.
With optional end, stop comparing S at
that position. suffix can also be a
tuple of strings to try1.
```

Обратите внимание на строку

```
S.endswith(suffix[, start[, end]]) -> bool
```

S представляет строку (*экземпляр*), для которой вызывается метод; в нашем случае это переменная `x1`. Имя метода — `.endswith`. В круглые скобки () заключены параметры. *Обязательным* параметром является `suffix`. Метод `.endswith` сообщит об ошибке, если этот параметр не будет передан при вызове:

```
>>> x1.endswith()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: endswith() takes at least 1 argument
(0 given)
```

Параметры в квадратных скобках [] являются *необязательными*. В данном случае параметры `start` и `end` позволяют проверить только часть строки. Например, если вы хотите убедиться в том, что часть строки с 0 и до 3 символов завершается символами «Oct», это можно сделать так:

```
>>> x1.endswith('Oct', 0, 3)
True
```

У строк также имеется парный метод `.startswith`, так что, если вы захотите узнать, начинается ли строка с 'Oct', это делается так:

```
>>> x1.startswith('Oct')
True
```

¹ Вернуть True, если S заканчивается `suffix`, в противном случае — False. При дополнительном запуске проверка S начинается с этой позиции. С необязательным параметром `end` прекратите сравнение S в этом положении. `suffix` также может быть кортежем строк.

11.3. find

Метод `.find` предназначен для поиска подстрок в других строках. Он возвращает *индекс* (смещение, начинающееся с 0) совпадающей подстроки. Если подстрока не найдена, метод возвращает `-1`:

```
>>> word = 'grateful'

# 0 is g, 1 is r, 2 is a
>>> word.find('ate')
2
>>> word.find('great')
-1
```

11.4. format

Метод `format` позволяет легко создавать новые строки, объединяя существующие переменные. Этот метод рассматривался в главе 9:

```
>>> print('name: {}, age: {}'.\
...       format('Matt', 10))
name: Matt, age: 10
```

ПРИМЕЧАНИЕ

В приведенном примере функция `print` занимает две строки. Последовательность символов `.\` сообщает Python, что на следующей строке находится продолжение. Если открыта левая круглая скобка `(`, вы также можете размещать аргументы в нескольких строках без `\`:

```
>>> print("word".
...       find('ord'))
1
>>> print("word".find(
...       'ord'))
1
```

Чтобы код лучше читался, строки продолжения снабжаются отступом. Отступ из четырех пробелов сообщает всем читателям кода, что вторая строка является продолжением предыдущей команды:

```
>>> print("word".\
...      find('ord'))
1
>>> print("word".find(
...      'ord'))
1
```

Зачем разбивать код, который мог бы поместиться в одной строке, на несколько строк? Чаще всего это делается при использовании стандартов оформления кода, требующих, чтобы длина строки не превышала 80 символов. Если метод получает несколько аргументов, выдержать ограничение в 80 символов может быть непросто. (Для Python длина строки не важна, но читатели вашего кода — другое дело.) На практике каждый аргумент метода нередко размещается в отдельной строке:

```
>>> print('{} {} {} {} {}'.format(
...      'hello',
...      'to',
...      'you',
...      'and',
...      'you'
... ))
hello to you and you
```

11.5. join

Нередко имеется список (см. далее в книге), и вы хотите что-то вставить между существующими элементами. Метод `.join` создает новую строку из последовательности, вставляя строку между каждой парой элементов списка:

```
>>> ', '.join(['1', '2', '3'])
'1, 2, 3'
```

СОВЕТ

В большинстве интерпретаторов Python `.join` работает быстрее, чем многократная конкатенация с использованием оператора `+`. Приведенная идиома является стандартной.

11.6. lower

Метод `.lower` возвращает копию строки, преобразованную к нижнему регистру. Такое преобразование часто бывает полезно для проверки совпадения ввода с некоторой строкой. Например, одни программы записывают расширения файлов в верхнем регистре, другие — нет. Если вы хотите проверить, имеет ли файл расширение `TXT` или `txt`, это можно сделать так:

```
>>> fname = 'readme.txt'
>>> fname.endswith('txt') or fname.endswith('TXT')
True
```

Версия, более традиционная для Python, выглядит так:

```
>>> fname.lower().endswith('txt')
True
```

11.7. startswith

Метод `.startswith` аналогичен `.endswith`, но он проверяет, начинается ли строка с другой строки:

```
>>> 'Book'.startswith('B')
True
>>> 'Book'.startswith('b')
False
```

11.8. strip

Метод `.strip` возвращает новую строку, из которой удалены начальные и конечные *пропуски* (пробелы, табуляции, символы новой строки). Например, это может быть полезно для нормализации или разбора данных, введенных пользователем (или загруженных из интернета).

```
>>> ' hello there '.strip()
'hello there'
```

Обратите внимание: из строки были удалены три начальных пробела и два пробела в конце. При этом два пробела между словами остались на своих местах. Если вам нужно удалить только начальные или конечные пропуски, эти задачи решаются методами `lstrip` и `rstrip` соответственно.

11.9. upper

Метод `.upper` аналогичен `.lower`. Он возвращает копию строки, в которой все буквы преобразованы к верхнему регистру:

```
>>> 'yell'.upper()
'YELL'
```

11.10. Другие методы

У строк есть и другие методы, но они используются реже. Познакомьтесь с ними самостоятельно — прочитайте документацию и опробуйте на практике. В приложении приведен полный список этих методов.

ПРИМЕЧАНИЕ

Часть `STRINGMETHODS` в разделе справки из REPL содержит документацию по всем строковым методам, а также ряд примеров.

11.11. Итоги

Эта глава была посвящена методам. Методы всегда вызываются с указанием объекта и точки перед именем метода. Также были рассмотрены наиболее типичные методы для строк. Следует помнить, что строки являются неизменяемыми. Если вы захотите изменить значение строки, для этого необходимо создать новую строку.

11.12. Упражнения

1. Создайте строку `school` с названием вашего учебного заведения. Просмотрите методы, доступные для этой строки. Воспользуйтесь функцией `help` для просмотра документации.

2. Создайте строку `country` со значением `'usa'`. Создайте новую строку `correct_country` со значением, преобразованным к верхнему регистру, с использованием строкового метода.
3. Создайте строку `filename` со значением `'hello.py'`. Проверьте, завершается ли имя файла суффиксом `'.java'`. Определите индекс подстроки `'py'`. Проверьте, начинается ли строка с подстроки `'world'`.
4. Откройте REPL. Войдите в режим справочной информации и просмотрите раздел `STRINGMETHODS`.

12

Комментарии, логические значения и None

В этой главе рассматриваются комментарии, логические значения и None. Комментарии делают ваш код более понятным. Логические значения и тип None очень часто используются в коде Python.

12.1. Комментарии

Комментарии не являются типом, потому что Python их игнорирует. Они должны что-то напомнить или пояснить программисту. Существуют разные мнения по поводу того, как нужно писать комментарии, их цели и полезности. Спектр мнений широк: от тех, кто против любых комментариев, до тех, кто комментирует почти каждую строку кода. Если вы участвуете в проекте, старайтесь соблюдать принятую в нем схему комментирования. Как правило, хороший комментарий должен объяснять *почему*, а не *как* (для ответа на вопрос *как* должно быть достаточного самого кода).

Чтобы создать комментарий в Python, начните строку с символа #. Все, что следует после этого символа, игнорируется:

```
>>> # This line is ignored by Python
```

Также комментарий может располагаться в конце строки:

```
>>> num = 3.14 # PI
```

СОВЕТ

Комментарии также используются для временной блокировки кода в процессе редактирования. Если ваш редактор поддерживает такую возможность, иногда бывает проще закомментировать код вместо того, чтобы удалять его полностью.

Тем не менее перед тем, как передавать код для распространения, закомментированные участки лучше удалить.

В других языках поддерживаются многострочные комментарии, но в Python их нет. Единственный способ создать комментарий из нескольких строк — начинать каждую строку с #.

СОВЕТ

Возможно, у вас появилась соблазнительная мысль создать комментарий из нескольких строк при помощи строки, заключенной в тройные кавычки. Такие конструкции получаются уродливыми и непонятными, не используйте их.

12.2. Логические значения

Логические значения представляют понятия «истина» (True) и «ложь» (False). Вы уже видели их в приведенных ранее примерах кода — например, в результате `.startswith`:

```
>>> 'bar'.startswith('b')
True
```

Такие значения можно присваивать переменным:

```
>>> a = True
>>> b = False
```

ПРИМЕЧАНИЕ

В языке Python логические значения относятся к классу `bool`:

```
>>> type(True)
<class 'bool'>
```

Бывает полезно преобразовывать другие типы к логическим значениям. В языке Python это можно сделать при помощи класса `bool`. Впрочем, обычно прямые преобразования типов оказываются излишними из-за неявного преобразования, которое выполняет Python при проверке условных конструкций. При обработке условий это преобразование будет выполнено за вас.

В терминологии Python нередко приходится слышать о «квазиистинном» или «квазиложном» поведении объектов — это означает, что нелогические типы могут неявно вести себя так, словно являются логическими. Если вы не уверены в том, как себя поведет ваш тип, выполните явное преобразование с использованием класса `bool`.

Для строк пустая строка обладает «квазиложным» поведением, тогда как непустые значения интерпретируются как `True`:

```
>>> bool('')
False
>>> bool('0') # Строка содержит 0
True
```

Так как непустая строка обладает квазиистинным поведением, вы можете проверить, содержит ли строка какие-либо данные. В следующем фрагменте значение `name` задается в коде, но представьте, что оно должно вводиться пользователем:

```
>>> name = 'Paul'
>>> if name:
...     print("The name is {}".format(name))
... else:
...     print("Name is missing")
The name is Paul
```

Необязательно проверять длину `name`. Не делайте так:

```
>>> if len(name) > 0:
...     print("The name is {}".format(name))
```

И так тоже поступать не нужно:

```
>>> if bool(name):
...     print("The name is {}".format(name))
```

потому что Python вычислит содержимое команды `if` и преобразует его в логическое значение за вас. Поскольку непустая строка интерпретируется как `True`, достаточно использовать конструкцию вида

```
>>> if name:
...     print("The name is {}".format(name))
```

ПРИМЕЧАНИЕ

Встроенные типы `int`, `float`, `str` и `bool` являются классами. Хотя из-за регистра символов (нижнего) они похожи на функции, на самом деле это классы. В этом легко убедиться вызовом `help(str)`:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
|   str(object='') -> str
|   str(bytes_or_buffer[, encoding[, errors]]) -> str
|
```

Здесь проявляется небольшая непоследовательность языка Python: классы, определяемые пользователем, обычно подчиняются правилам PEP8, которые рекомендуют «верблюжий» регистр в именах классов.

Для чисел ноль интерпретируется как `False`, тогда как другие числа обладают поведением `True`:

```
>>> bool(0)
False
>>> bool(4)
True
```

Хотя явное преобразование функцией `bool` доступно, обычно оно излишне, потому что переменные неявно преобразуются в логические значения при использовании в условных командах. Например, типы-контейнеры (такие, как *списки* и *словари*), не содержащие элементов, обладают «квазиложным» поведением. С другой стороны, при появлении элементов они интерпретируются как «квазиистинные».

СОВЕТ

Будьте внимательны при разборе содержимого, которое должно быть преобразовано в логические значения. Непустые строки интерпретируются как

True. Например, неожиданный сюрприз может преподнести строка 'False', которая интерпретируется как True:

```
>>> bool('False')
True
```

Ниже приведена таблица квазиистинных и квазиложных значений.

Квазиистинность	Квазиложность
True	False
Большинство объектов	None
1	0
3.2	0.0
[1, 2]	[] (пустой список)
{'a': 1, 'b': 2}	{ } (пустой словарь)
'string'	"" (пустая строка)
'False'	
'0'	

СОВЕТ

Не проверяйте логические значения, чтобы узнать, равны ли они True. Если у вас имеется переменная `done`, содержащая логическое значение, достаточно следующей конструкции:

```
if done:
    # ...
```

С другой стороны, следующая проверка будет излишней:

```
if done == True:
    # ...
```

Как и эта проверка:

```
if bool(done):
    # ...
```

Аналогичным образом, если у вас имеется список и вам потребуется выполнить разные действия для пустого и непустого списка, достаточно будет следующего фрагмента:

```
members = []
if members:
    # Действия, если members
    # содержит значения
else:
    # Список members пуст
```

А следующая проверка будет избыточной. Не нужно проверять квазиистинность списка по его длине:

```
if len(members) > 0:
    # Действия, если members
    # содержит значения
else:
    # Список members пуст
```

ПРИМЕЧАНИЕ

Если вы хотите определить неявную квазиистинность для объектов, определяемых пользователем, это поведение определяется методом `.__bool__`. Метод может возвращать `True` или `False`. Если этот специальный метод не определен, то метод `.__len__` проверяется на ненулевое значение. Если и этот метод не определен, объект по умолчанию интерпретируется как `True`:

```
>>> class Nope:
...     def __bool__(self):
...         return False

>>> n = Nope()
>>> bool(n)
False
```

Если все это покажется вам непонятным, вернитесь к примеру после того, как прочитаете об этих классах.

12.3. None

`None` является экземпляром `NoneType`. В других языках также существуют аналогичные конструкции, такие как *nil*, *NULL* или *undefined*. Присваивая переменной `None`, вы тем самым показываете, что переменной еще

не было присвоено реальное значение. В логическом контексте `None` интерпретируется как `False`:

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

ПРИМЕЧАНИЕ

Функция Python при отсутствии команды `return` по умолчанию возвращает `None`:

```
>>> def hello():
...     print("hi")

>>> result = hello()
hi
>>> print(result)
None
```

ПРИМЕЧАНИЕ

`None` является одиночным (*синглетным*) экземпляром (то есть в интерпретаторе Python всегда хранится только одна копия `None`). Идентификатор этого значения всегда остается одним и тем же:

```
>>> a = None
>>> id(a)
140575303591440
>>> b = None
>>> id(b)
140575303591440
```

Таким образом, любая переменная, содержащая `None`, указывает на тот же объект, что и любая другая переменная, содержащая `None`. Обычно для таких переменных используется проверка *тождественности* оператором `is` вместо проверки *равенства* оператором `==`:

```
>>> a is b
True
>>> a is not b
False
```

Оператор `is` работает быстрее `==` и напоминает программисту, что сравниваются идентификаторы, а не значения. Также выражение `is` можно поместить в команду `if`:

```
>>> if a is None:
...     print("A is not set!")
A is not set!
```

Так как `None` в логическом контексте интерпретируется как `False`, также возможно такое решение:

```
>>> if not a:
...     print("A is not set!")
A is not set!
```

Тем не менее будьте осторожны с другими значениями, которые также интерпретируются как `False` — такими, как `0`, `[]` или `' '` (пустая строка). Проверка на `None` должна осуществляться явно.

12.4. Итоги

В этой главе вы узнали о комментариях в языке Python. Комментарии начинаются с символа `#`, а все последующие символы до конца строки игнорируются. Многострочные комментарии не поддерживаются.

Также в этой главе обсуждаются значения `True`, `False` и преобразование к логическому типу. Многие значения интерпретируются как `True` в логическом контексте (например, при использовании в команде `if`). Как `False` интерпретируются нули, `None` и пустые последовательности.

Глава завершается описанием объекта `None`. Это одиночный объект, обозначающий переменные, значение которых должно быть присвоено в будущем. Он также является результатом функции, которая не возвращает значение явно.

12.5. Упражнения

1. Создайте переменную `age` и присвойте ей свой возраст. Создайте другую переменную — `old`, которая использует условие для проверки того, что вы старше 18 лет. Значение `old` должно быть равно `True` или `False`.
2. Создайте переменную `name` и присвойте ей свое имя. Создайте другую переменную — `second_half`, которая проверяет, находится ли

первая буква `name` во второй половине алфавита. Что необходимо для того, чтобы выполнить сравнение?

3. Создайте список `names` с именами людей в классе. Напишите код, который выводит сообщение `'The class is empty!'` или `'Class has enrollments.'` в зависимости от того, содержит ли `names` хоть одно значение. (В этой главе вы найдете подсказки.)
4. Создайте переменную `car` и присвойте ей `None`. Напишите код, который выводит сообщение `'Taxi for you!'` или `'You have a car!'` в зависимости от того, содержит ли `car` хоть одно значение. (`None` не является названием автомобиля.)

13

Условия и отступы

В этой главе вы больше узнаете о сравнениях в Python. В коде часто приходится принимать решения по выбору выполняемого пути, поэтому сейчас мы посмотрим, как это делается.

Кроме логических литералов `True` и `False` для получения логических значений в Python также можно использовать выражения. Если у вас имеется два числа, вы можете сравнить их и проверить, что первое больше второго, или наоборот. Эта задача решается операторами `>` и `<` соответственно:

```
>>> 5 > 9
False
```

В следующей таблице перечислены операторы сравнения для создания логических значений:

Оператор	Описание
<code>></code>	Больше
<code><</code>	Меньше
<code>>=</code>	Больше или равно
<code><=</code>	Меньше или равно
<code>==</code>	Равно
<code>!=</code>	Не равно
<code>is</code>	Объекты тождественны
<code>is not</code>	Объекты не тождественны

Эти операции могут использоваться с большинством типов. А если вы создадите собственный класс, определяющий соответствующие специальные методы, то ваш класс тоже сможет использовать их:

```
>>> name = 'Matt'
>>> name == 'Matt'
True
>>> name != 'Fred'
True
>>> 1 > 3
False
```

ПРИМЕЧАНИЕ

Специальные методы сравнения `__gt__`, `__lt__`, `__ge__`, `__le__`, `__eq__` и `__ne__` соотносятся с `>`, `<`, `>=`, `<=`, `==` и `!=` соответственно. Определять все эти методы довольно утомительно и однообразно. Для классов, в которых обычно используются эти сравнения, декоратор класса `functools.total_ordering` предоставит всю функциональность сравнений при условии определения методов `__eq__` and `__le__`. Декоратор автоматически сгенерирует остальные методы сравнения. В противном случае необходимо реализовать все шесть методов:

```
>>> import functools
>>> @functools.total_ordering
... class Abs(object):
...     def __init__(self, num):
...         self.num = abs(num)
...     def __eq__(self, other):
...         return self.num == abs(other)
...     def __lt__(self, other):
...         return self.num < abs(other)

>>> five = Abs(-5)
>>> four = Abs(-4)
>>> five > four # Случай "меньше" не используется!
True
```

Декораторы считаются темой средней сложности. В книге начального уровня они не рассматриваются.

СОВЕТ

Операторы `is` и `is not` предназначены для сравнения *тождественности* (то есть того, что два объекта имеют одинаковые идентификаторы и фактически являются одним объектом (а не только имеют одинаковые значения)). Так как `None` является одиночным объектом и имеет только один идентификатор, `is` и `is not` могут использоваться с `None`:

```
>>> if name is None:
...     # Инициализация name
```

13.1. Объединение условных выражений

Условные выражения могут объединяться *логическими* операторами `and`, `or` и `not`.

Логический оператор	Описание
<code>x and y</code>	Выражение истинно только в том случае, если истинны оба операнда
<code>x or y</code>	Выражение истинно, если истинным является хотя бы один из операндов
<code>not x</code>	Логическое отрицание <code>x</code> (<code>True</code> превращается в <code>False</code> , и наоборот)

В следующем простом примере оценка присваивания осуществляется на основании количества набранных баллов — выражение проверяет, лежит ли количество набранных баллов в заданном числовом диапазоне:

```
>>> score = 91
>>> if score > 90 and score <= 100:
...     grade = 'A'
```

ПРИМЕЧАНИЕ

В таких случаях Python позволяет использовать *диапазонную проверку*:

```
>>> if 90 < score <=100:
...     grade = 'A'
```

Работают оба стиля проверок, но в других языках программирования диапазонные проверки поддерживаются редко.

Следующий пример проверяет, входит ли имя `name` в заданную группу:

```
>>> name = 'Paul'
>>> beatle = False
>>> if name == 'George' or \
...     name == 'Ringo' or \
...     name == 'John' or \
...     name == 'Paul':
...     beatle = True
... else:
...     beatle = False
```

ПРИМЕЧАНИЕ

В этом примере знак `\` в конце строки `'George' or \` указывает, что команда продолжается на следующей строке. Как и во многих языках программирования, в Python условные выражения могут заключаться в круглые скобки. Так как в Python круглые скобки не являются обязательными, многие разработчики опускают их, если они не являются строго необходимыми для определения приоритета операторов. Однако у круглых скобок есть еще один нюанс: они подсказывают интерпретатору, что команда еще не завершена и будет продолжена в следующей строке. Это означает, что в следующем примере символ `\` не нужен:

```
>>> name = 'Paul'
>>> beatle = False
>>> if (name == 'George' or
...     name == 'Ringo' or
...     name == 'John' or
...     name == 'Paul'):
...     beatle = True
... else:
...     beatle = False
```

СОВЕТ

Ниже приведена версия проверки принадлежности, которая считается одной из базовых идиом Python. Чтобы проверить, входит ли значение в набор, объедините элементы набора в множество и воспользуйтесь оператором `in`:

```
>>> beatles = {'George', 'Ringo', 'John', 'Paul'}
>>> beatle = name in beatles
```

Множества более подробно рассматриваются в одной из последующих глав.

Пример использования ключевого слова `not` в условной команде:

```
>>> last_name = 'unknown'
>>> if name == 'Paul' and not beatle:
...     last_name = 'Revere'
```

13.2. Команды if

Логические значения (`True` и `False`) часто используются в *условных* командах. Условная команда, по сути, означает: «если это условие истинно, выполнить блок кода, а если нет — выполнить другой код». Условные команды часто используются в Python. Иногда «команда `if`» проверяет значения, содержащие логические значения; в других случаях проверяются выражения, результат которых интерпретируется как логическое значение. Другая распространенная проверка связана с неявным преобразованием к «квазиистинным» или «квазиложным» значениям:

```
>>> debug = True
>>> if debug: # Проверка логического значения
...     print("Debugging")
Debugging
```

13.3. Команды else

Команда `else` может использоваться в сочетании с командой `if`. Тело `else` выполняется только в том случае, если условие `if` дает результат `False`. Ниже приведен пример объединения команды `else` с `if`:

```
>>> score = 87
>>> if score >= 90:
...     grade = 'A'
... else:
...     grade = 'B'
```

Python вычисляет *выражение* `score >= 90` и получает результат `False`. Так как условие `if` было ложным, выполняются команды блока `else`, а переменной `grade` присваивается значение `'B'`.

13.4. Множественный выбор

Выбор далеко не всегда ограничивается двумя вариантами. При необходимости вы можете добавить промежуточные шаги при помощи ключевого слова `elif` (сокращение от «else if»). Ниже приведена схема выбора с несколькими вариантами:

```
>>> score = 87
>>> if score >= 90:
...     grade = 'A'
... elif score >= 80:
...     grade = 'B'
... elif score >= 70:
...     grade = 'C'
... elif score >= 60:
...     grade = 'D'
... else:
...     grade = 'F'
```

Каждая из команд `if`, `elif` и `else` имеет собственный блок кода. Python начинает перебор сверху, пытаясь найти условие с результатом `True`. Обнаружив такое условие, Python выполняет блок, а затем переходит к выполнению кода, следующего за всеми блоками `elif` и `else`. Если ни одно из условий `if` и `elif` не дает результата `True`, выполняется блок команды `else`.

ПРИМЕЧАНИЕ

Команда `if` может содержать ноль и более команд `elif`. Наличие команды `else` не обязательно. Если команда `else` присутствует, то в команде `if` она может быть только одна.

13.5. Пробелы

Возможно, вы обратили внимание на одну странность: за логическим выражением в команде `if` следует двоеточие (`:`). Строки, следующие непосредственно после команды `if`, снабжаются отступом из четырех пробелов. Строки с отступами образуют *блок* кода, который выполняется в том случае, если выражение `if` дает результат `True`.

В других языках команда `if` могла бы выглядеть так:

```
if (score >= 90) {  
    grade = 'A';  
}
```

В таких языках границы блоков в командах `if` обозначаются фигурными скобками (`{ }`). Весь код между фигурными скобками выполняется в том случае, если значение `score` больше либо равно 90.

В отличие от таких языков, в Python для обозначения блоков используются два признака:

- двоеточие (`:`);
- отступы.

Если вы программировали на других языках, понять правила использования отступов в Python будет нетрудно. Все, что от вас потребуется, — заменить левую фигурную скобку (`{`) двоеточием (`:`) и последовательно применять отступы до конца блока.

СОВЕТ

Что считать «последовательным применением отступов»? Обычно для создания отступов в коде используются либо символы табуляции, либо пробелы. Интерпретатор Python обращает внимание только на единство стиля отступов на уровне файла. Проект может состоять из нескольких файлов, использующих разные схемы отступов, но это было бы глупо.

Согласно PEP 8, в Python рекомендуется использовать отступы из четырех пробелов. Если вы начнете смешивать табуляции с пробелами в одном файле, вскоре возникнут проблемы. И хотя пробелы считаются предпочтительными, если вы работаете над кодом, в котором уже используются символы табуляции, лучше действовать последовательно и продолжить использовать символы табуляции в коде.

Если вы начнете смешивать табуляции с пробелами, исполняемый файл python3 сообщит об ошибке `TabError`.

13.6. Итоги

В этой главе рассматривается команда `if`. Эта команда может использоваться для создания сколь угодно сложных условий посредством объединения выражений операторами `and`, `or` и `not`.

Также в главе рассматривались блоки, отступы и табуляции/пробелы. При первом знакомстве с Python правило обязательных отступов может показаться досадной помехой. Я сталкивался с людьми, которые так считают, на своих учебных курсах. Но когда я их спрашивал, включают ли они отступы в код на других языках, они отвечали: «Конечно, так код лучше читается». В Python удобочитаемости кода уделяется большое внимание, и правило об отступах помогает в этом.

13.7. Упражнения

1. Напишите команду `if`, которая проверяет, является ли год, хранящийся в переменной, високосным. Правила определения високосных годов приведены в упражнениях главы 8.
2. Напишите команду `if` для проверки переменной, содержащей целое число, на четность.
3. Напишите команду `if`. Найдите блок с отступом и проверьте, какие символы использует ваш редактор при создании отступов — табуляции или пробелы. Если отступы создаются символами табуляции, настройте свой редактор, чтобы он использовал пробелы. В некоторых редакторах табуляции отображаются иначе; если в вашем редакторе они отображаются одинаково, для проверки можно подвести курсор к отступу. Если курсор переместится сразу на 4 или 8 символов, значит, редактор вставил символ табуляции.

14

Контейнеры: списки, кортежи и множества

Многие типы, рассматривавшиеся до настоящего момента, были *скалярными величинами*, содержащими одно значение. Целые числа, числа с плавающей точкой и логические значения — все это скалярные значения.

Контейнеры способны содержать не один, а несколько объектов (скалярные типы и даже другие контейнеры). В этой главе рассматриваются некоторые из таких типов — списки, кортежи и множества.

14.1. Списки

Списки, как следует из самого названия, используются для хранения списков объектов. В Python список может содержать элементы произвольного типа, в том числе и элементы разных типов. Впрочем, на практике в списках чаще хранятся элементы только одного типа. Список также можно представить себе как упорядоченную последовательность элементов. Списки относятся к *изменяемым типам*; это означает, что вы можете добавлять, удалять и изменять их содержимое без создания нового объекта. Существует два способа создания пустых списков; вызов класса `list` и с использованием синтаксиса литералов в квадратных скобках — `[]`:

```
>>> names = list()
>>> other_names = []
```

Чтобы список при создании уже был заполнен, перечислите нужные значения в квадратных скобках с использованием *синтаксиса литералов*:

```
>>> other_names = ['Fred', 'Charles']
```

ПРИМЕЧАНИЕ

Класс `list` также может использоваться для создания заранее заполненных списков, но синтаксис получается довольно громоздким, потому что ему должен передаваться список:

```
>>> other_names = list(['Fred', 'Charles'])
```

Обычно этот класс используется для объединения других типов последовательностей в списки. Например, строка является последовательностью символов. Если передать строку при вызове `list`, вы получите список отдельных символов:

```
>>> list('Matt')
['M', 'a', 't', 't']
```

У списков, как и у других типов, имеются методы, которые можно для них вызывать (чтобы просмотреть полный перечень методов, используйте вызов `dir([])`). Например, добавление элементов в конец списка осуществляется методом `.append`:

```
>>> names.append('Matt')
>>> names.append('Fred')
>>> print(names)
['Matt', 'Fred']
```

Помните, что списки являются изменяемыми. Python не возвращает новый список при присоединении элементов. Обратите внимание: вызов `.append` не возвращает список (REPL ничего не выводит). Вместо этого возвращается `None`, а список обновляется на месте. В языке Python функция или метод по умолчанию возвращает `None`. Невозможно создать метод, который ничего не возвращает.

14.2. Индексы

Список — один из *типов последовательностей* в языке Python. В последовательностях хранятся упорядоченные наборы объектов. Чтобы

эффективно работать с последовательностями, необходимо понимать, что такое *индекс*. С каждым элементом списка связан индекс, описывающий его местоположение в списке. Например, при изготовлении картофельных чипсов используется картофель, масло и соль, которые обычно перечисляются именно в таком порядке. Картофель находится на первом месте списка, масло на втором, а соль на третьем.

Во многих языках программирования первому элементу последовательности присваивается индекс 0, второму — индекс 1, третьему — индекс 2, и т. д. Нумерация индексов начинается с нуля.

Для обращения к элементу списка указывается индекс этого элемента, заключенный в квадратные скобки:

```
>>> names[0]
'Matt'

>>> names[1]
'Fred'
```

14.3. Вставка в список

Чтобы вставить элемент в позицию с определенным индексом, используйте метод `.insert`. При вызове `.insert` все элементы, следующие за этим индексом, сдвигаются вправо:

```
>>> names.insert(0, 'George')
>>> print(names)
['George', 'Matt', 'Fred']
```

Для замены элемента с заданным индексом применяется синтаксис с квадратными скобками:

```
>>> names[1] = 'Henry'
>>> print(names)
['George', 'Henry', 'Fred']
```

Чтобы присоединить элементы в конец списка, воспользуйтесь методом `.append`:

```
>>> names.append('Paul')
>>> print(names)
['George', 'Henry', 'Fred', 'Paul']
```

ПРИМЕЧАНИЕ

В Python списки в действительности реализуются в виде массива указателей. Такая реализация обеспечивает быстрый произвольный доступ к элементам по индексам. Кроме того, операции присоединения и удаления в конце списка выполняются быстро ($O(1)$), тогда как операции вставки и удаления в середине списка выполняются медленнее ($O(n)$). Если окажется, что вам часто приходится вставлять и извлекать элементы в начале списка, лучше использовать структуру данных `collections.deque`.

14.4. Удаление из списка

Для удаления элементов из списка используется метод `.remove`:

```
>>> names.remove('Paul')
>>> print(names)
['George', 'Henry', 'Fred']
```

Также возможно удаление по индексу с использованием синтаксиса с квадратными скобками:

```
>>> del names[1]
>>> print(names)
['George', 'Fred']
```

14.5. Сортировка списков

Одна из самых распространенных операций со списками — *сортировка*. Метод `.sort` упорядочивает значения в списке, при этом сортировка осуществляется «*на месте*». Метод не возвращает новую отсортированную копию списка, а обновляет список с измененным порядком элементов:

```
>>> names.sort()
>>> print(names)
['Fred', 'George']
```

Если предыдущий порядок элементов важен, скопируйте список перед сортировкой. Впрочем, для сортировки последовательностей также

можно воспользоваться более общим решением с функцией `sorted`. Функция `sorted` работает с любой последовательностью и возвращает новый список с упорядоченными элементами:

```
>>> old = [5, 3, -2, 1]
>>> nums_sorted = sorted(old)
>>> print(nums_sorted)
[-2, 1, 3, 5]
>>> print(old)
[5, 3, -2, 1]
```

Будьте внимательны с тем, что вы сортируете. Python требует, чтобы вы явно выражали свои намерения. В Python 3 при попытке отсортировать список с разнотипными элементами может произойти ошибка:

```
>>> things = [2, 'abc', 'Zebra', '1']
>>> things.sort()
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

Как метод `.sort`, так и функция `sorted` позволяют управлять сортировкой, для чего в параметре `key` передается произвольная функция сортировки. Параметр `key` может содержать функцию (а также класс или метод), которая получает один элемент и возвращает нечто пригодное для сравнения.

В следующем примере, где в параметре `key` передается `str`, все элементы списка сортируются по правилам сортировки строк:

```
>>> things.sort(key=str)
>>> print(things)
['1', 2, 'Zebra', 'abc']
```

14.6. Полезные советы по работе со списками

Обычно списки также поддерживают другие методы. Попробуйте открыть интерпретатор Python и ввести несколько примеров. Не забывайте о своих друзьях `dir` и `help`.

СОВЕТ

Встроенная функция `range` строит целочисленные последовательности. Следующий фрагмент создает последовательность из чисел от 0 до 4:

```
>>> nums = range(5)
>>> nums
range(5)
```

Python 3 не любит лишней работы. Функция `range` не материализует список, а предоставляет итератор, который будет возвращать эти числа при переборе. Передав результат `list`, вы сможете увидеть сгенерированные числа:

```
>>> list(nums)
[0, 1, 2, 3, 4]
```

Обратите внимание: сгенерированная последовательность не включает число 5. Многие функции Python, использующие конечные индексы, означают «до, но не включая» (другой пример такого рода — срезы — будет представлен позднее).

Если последовательность должна начинаться с ненулевого числа, функции `range` можно передать два параметра. В этом случае первый параметр определяет начало последовательности (включительно), а второй — конец (не включая):

```
# Числа от 2 до 5
>>> nums2 = range(2, 6)
>>> nums2
range(2, 6)
```

У функции `range` также имеется необязательный третий параметр — *приращение*. Если приращение равно 1 (по умолчанию), следующее число в последовательности, возвращаемой `range`, будет на 1 больше предыдущего. С приращением 2 будет возвращаться каждое второе число. В следующем примере возвращаются только четные числа, меньшие 11:

```
>>> even = range(0, 11, 2)
>>> even
range(0, 11, 2)

>>> list(even)
[0, 2, 4, 6, 8, 10]
```

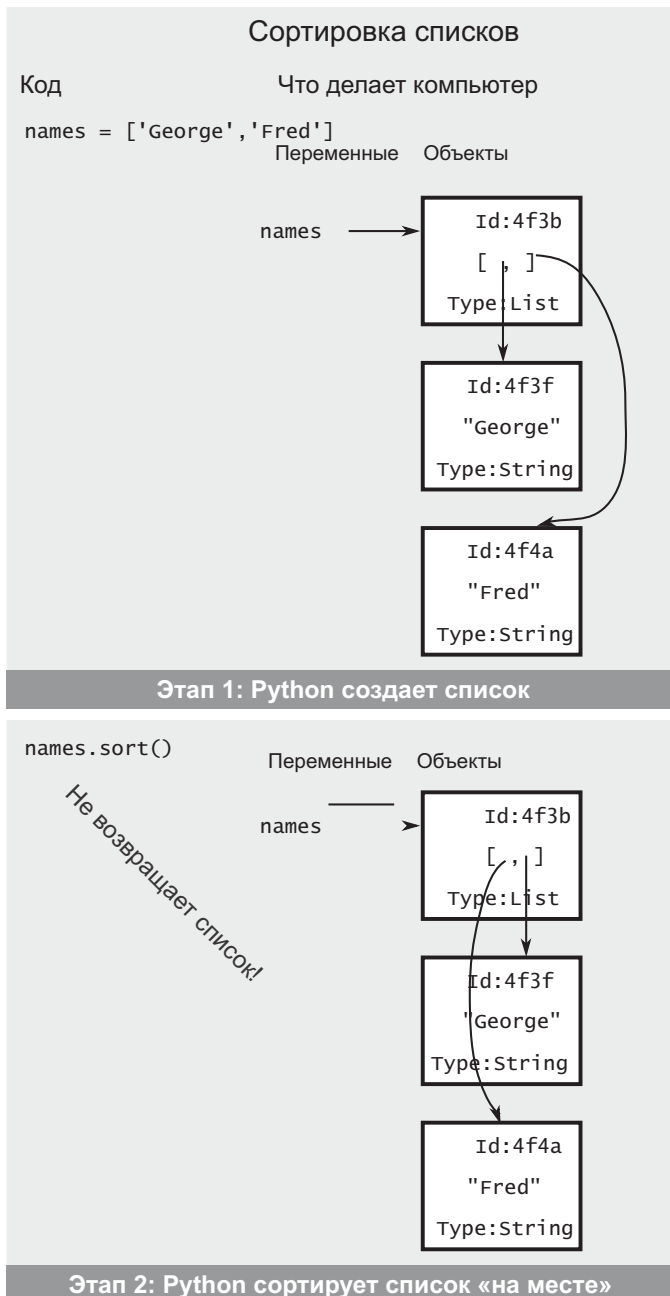


Рис. 14.1. Сортировка списка методом `.sort`. Помните, что список сортируется «на месте». В результате вызова метода `.sort` список изменяется, а сам метод возвращает `None`

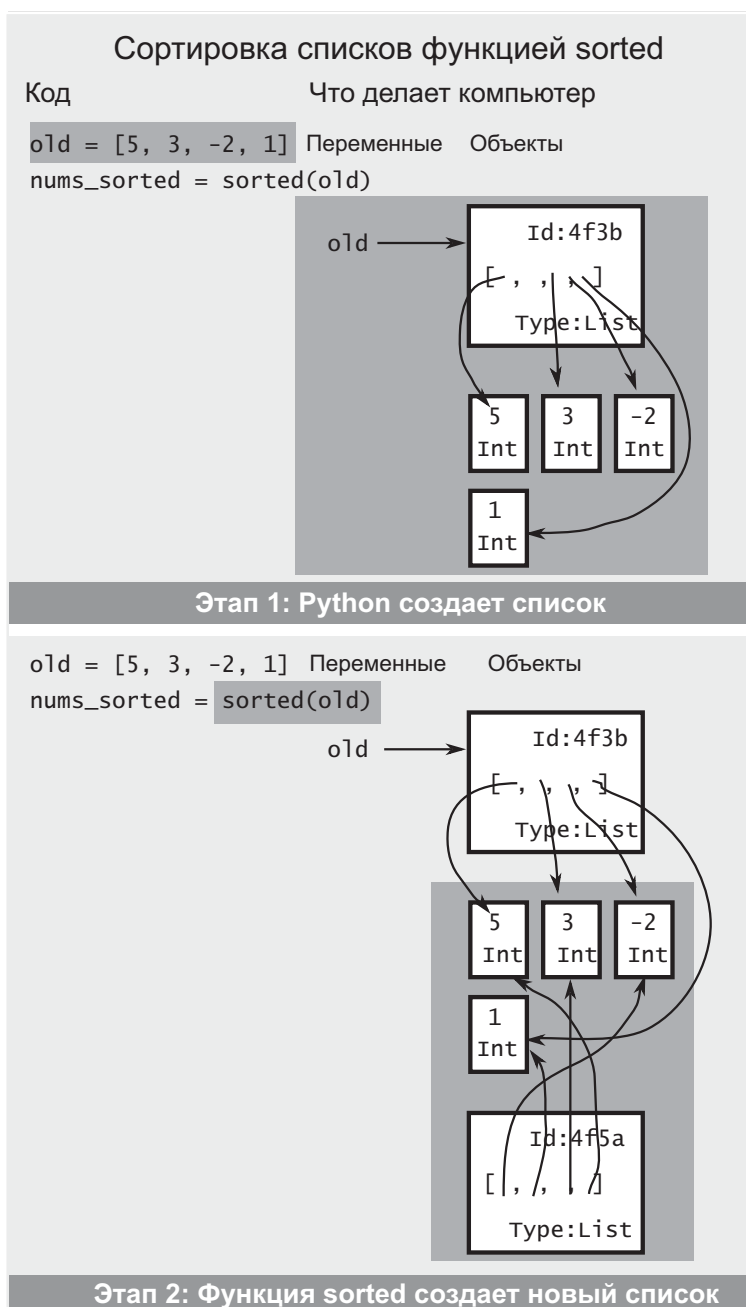


Рис. 14.2. Сортировка списка функцией `sorted`. Обратите внимание: список не изменяется, а при вызове возвращается новый список. Кроме того, Python повторно использует элементы списка, не создавая новых элементов

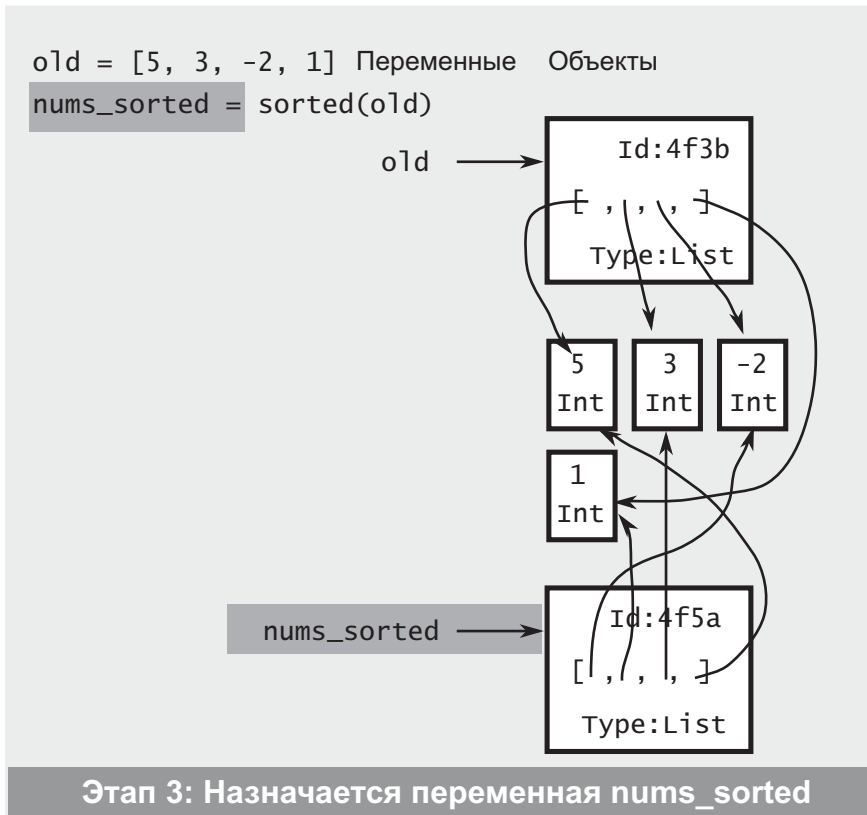


Рис. 14.3. На последнем этапе в результате операции присваивания переменная указывает на новый список. Обратите внимание: функция `sorted` работает с любыми последовательностями, не только со списками

ПРИМЕЧАНИЕ

Конструкция «до, но не включая» более формально называется *полуоткрытым интервалом*. Обычно полуоткрытые интервалы используются при определении последовательностей натуральных чисел. Они обладают рядом удобных свойств:

- Длина серии последовательных чисел равна разности между концом и началом интервала.
- Сращивание двух подпоследовательностей обходится без перекрытия.

В Python идиома полуоткрытого интервала применяется достаточно часто. Пример:

```
>>> a = range(0, 5)
>>> b = range(5, 10)
>>> both = list(a) + list(b)
>>> len(both) # 10 - 0
10

>>> both
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Если вы часто работаете с числовыми последовательностями, вам стоит поступать так же — особенно при определении API.

Знаменитый теоретик в области программирования Эдгар Дейкстра (Edsger Dijkstra) размышлял об индексировании с 0 и о том, почему этот вариант правилен¹. Он завершает свои рассуждения так:

«Многие языки программирования проектировались без должного внимания к этой подробности».

К счастью, Python к числу таких языков не относится.

14.7. Кортежи

Кортежи (tuples) представляют собой *неизменяемые* последовательности. Их можно рассматривать как упорядоченные записи данных. После того как кортеж будет создан, изменить его не удастся. Чтобы создать кортеж с использованием *синтаксиса литерала*, заключите компоненты в круглые скобки и разделите их запятыми. Также имеется класс `tuple`, который может использоваться для построения нового кортежа из существующей последовательности:

```
>>> row = ('George', 'Guitar')
>>> row
('George', 'Guitar')

>>> row2 = ('Paul', 'Bass')
>>> row2
('Paul', 'Bass')
```

¹ <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html>

Можно создать кортеж с нулем или с одним элементом. На практике кортежи используются для хранения записей данных, поэтому такой вариант встречается нечасто. Есть два способа создания пустых кортежей — функция `tuple` и синтаксис литералов:

```
>>> empty = tuple()
>>> empty
()
```

```
>>> empty = ()
>>> empty
()
```

Кортеж с одним элементом можно создать тремя способами:

```
>>> one = tuple([1])
>>> one
(1,)
```

```
>>> one = (1,)
>>> one
(1,)
```

```
>>> one = 1,
>>> one
(1,)
```

ПРИМЕЧАНИЕ

Круглые скобки используются в Python для обозначения вызовов функций или методов. Кроме того, они используются для определения приоритета операторов и еще могут использоваться при создании кортежей. Такое обилие применений может привести к недоразумениям. Запомните простое правило: если в круглые скобки заключен один элемент, то Python рассматривает круглые скобки как обычные (для определения приоритета операторов) — например, как в записи $(2 + 3) * 8$. Если же в скобки заключено несколько элементов, разделенных запятыми, Python рассматривает их как кортеж:

```
>>> d = (3)
>>> type(d)
<class 'int'>
```

На первый взгляд может показаться, что `d` — кортеж, заключенный в круглые скобки, но Python считает `d` целым числом. Для кортежей, состоящих

из одного элемента, следует поставить после элемента запятую (,) или же использовать класс `tuple` со списком, состоящим из одного элемента:

```
>>> e = (3,)
>>> type(e)
<class 'tuple'>
```

Создать кортеж с несколькими элементами можно тремя способами. Обычно последний считается наиболее соответствующим стилю Python. Благодаря наличию круглых скобок проще увидеть, что перед вами кортеж:

```
>>> p = tuple(['Steph', 'Curry', 'Guard'])
>>> p
('Steph', 'Curry', 'Guard')

>>> p = 'Steph', 'Curry', 'Guard'
>>> p
('Steph', 'Curry', 'Guard')

>>> p = ('Steph', 'Curry', 'Guard')
>>> p
('Steph', 'Curry', 'Guard')
```

Так как кортежи являются неизменяемыми, присоединение к ним новых элементов件возможно:

```
>>> p.append('Golden State')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute
'append'
```

ПРИМЕЧАНИЕ

Почему кортежи и списки существуют по отдельности? Почему бы просто не использовать списки, которые на первый взгляд являются надмножеством кортежей?

Главное различие между объектами заключается в изменяемости. Так как кортежи неизменяемы, они могут служить ключами в словарях. Кортежи часто используются для представления записей данных — например, строк

запросов баз данных, которые могут содержать разнородные типы объектов. Например, в кортеже могут храниться имя, адрес и возраст:

```
>>> person = ('Matt', '123 North 456 East', 24)
```

Кортежи используются для возвращения нескольких элементов из функций. Кроме того, они подсказывают разработчику, что этот тип не предназначен для изменения.

Наконец, кортежи используют меньше памяти, чем списки. Если вы работаете с последовательностями, которые не должны изменяться, возможно, вам стоит воспользоваться кортежами ради экономии памяти.

14.8. Множества

Еще одну разновидность контейнеров в Python составляют *множества* (set). Множество представляет собой неупорядоченную совокупность объектов, в которой не может быть дубликатов. Как и кортеж, множество можно создать на базе списка или других последовательностей, элементы которых можно перебирать. Однако, в отличие от списков и кортежей, для множеств неважен порядок элементов. Множества часто используются для двух целей: для удаления дубликатов и для проверки принадлежности. Так как механизм поиска основан на оптимизированной функции хеширования, реализованной для словарей, операция поиска занимает очень мало времени даже для очень больших множеств.

ПРИМЕЧАНИЕ

Так как множества должны вычислять хеш-код для каждого элемента, в них могут храниться только *хешируемые* элементы. Хеш-код представляет собой квазиуникальное число, сгенерированное для заданного объекта. Если объект является хешируемым, для него всегда генерируется одно и то же число.

В языке Python изменяемые элементы не являются хешируемыми. Это означает, что хешировать список или словарь невозможно. Для хеширования пользовательских классов необходимо реализовать методы `__hash__` и `__eq__`.

Множество можно определить передачей последовательности классу `set` (еще один класс преобразования, который выглядит как функция):

```
>>> digits = [0, 1, 1, 2, 3, 4, 5, 6,
...          7, 8, 9]
>>> digit_set = set(digits) # Удаление лишней 1
```

Множества также могут создаваться в синтаксисе литералов с { }:

```
>>> digit_set = {0, 1, 1, 2, 3, 4, 5, 6,
...             7, 8, 9}
```

Как упоминалось выше, множество отлично подходит для удаления дубликатов. При создании множества на базе последовательности все дубликаты удаляются. Так, из `digit_set` был удален лишний элемент 1:

```
>>> digit_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Для проверки принадлежности используется операция `in`:

```
>>> 9 in digit_set
True
```

```
>>> 42 in digit_set
False
```

ПРИМЕЧАНИЕ

В Python существует особый протокол принадлежности. Если класс (множество, список или класс, определяемый пользователем) реализует метод `__contains__` (или протокол перебора), он может использоваться для проверки принадлежности. Вследствие особенностей реализации множеств проверка принадлежности для них может выполняться намного быстрее, чем проверки по списку.

В следующем примере создается множество с именем `odd`. Оно будет использоваться в дальнейших примерах:

```
>>> odd = {1, 3, 5, 7, 9}
```

Множества Python поддерживают классические операции теории множеств, такие как объединение (`|`), пересечение (`&`), вычитание (`-`) и исключающее ИЛИ (`^`).

Оператор *вычитания* (-) удаляет элементы, входящие в одно множество, из другого множества:

```
>>> odd = {1, 3, 5, 7, 9}
```

```
# Вычитание
```

```
>>> even = digit_set - odd
```

```
>>> even
```

```
{0, 8, 2, 4, 6}
```

Обратите внимание на порядок элементов результата: на первый взгляд он кажется произвольным. Если для вас важен порядок элементов, вместо множества лучше использовать другой тип данных.

Операция *пересечения* (&) возвращает элементы, присутствующие в обоих множествах:

```
>>> prime = set([2, 3, 5, 7])
```

```
# В обоих множествах
```

```
>>> prime_even = prime & even
```

```
>>> prime_even
```

```
{2}
```

Операция *объединения* (|) возвращает множество, состоящее из всех элементов обоих множеств (с исключением дубликатов):

```
>>> numbers = odd | even
```

```
>>> print(numbers)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Операция *исключающего ИЛИ* (^) возвращает множество с элементами, присутствующими только в одном из двух множеств:

```
>>> first_five = set([0, 1, 2, 3, 4])
```

```
>>> two_to_six = set([2, 3, 4, 5, 6])
```

```
>>> in_one = first_five ^ two_to_six
```

```
>>> print(in_one)
```

```
{0, 1, 5, 6}
```

СОВЕТ

Когда стоит использовать множество вместо списка? Вспомните, что множества оптимизированы для проверки принадлежности и удаления дубли-

катов. Если вы выполняете операции объединения или вычитания списков, возможно, вам стоит вместо списков использовать множества.

Множества также намного эффективнее работают при проверке принадлежности. Оператор `in` для множеств работает быстрее, чем для списков. Тем не менее за эту скорость приходится расплачиваться: в отличие от списков и кортежей, множества не поддерживают конкретный порядок хранения элементов. Если для вас важен порядок элементов, используйте структуру данных, в которой он сохраняется.

14.9. Итоги

В этой главе описаны некоторые встроенные типы-контейнеры. Сначала были рассмотрены списки — изменяемые упорядоченные последовательности. Помните, что методы списков не возвращают новый список, а обычно изменяют список «на месте». Списки поддерживают вставку произвольных элементов, но обычно в списке хранятся однотипные элементы.

Затем были рассмотрены кортежи. Кортежи тоже упорядочены, как и списки, но в отличие от списков кортежи не поддерживают изменение «на месте». На практике они обычно используются для представления записей данных — например, строк, прочитанных из базы данных. В кортежах, представляющих записи данных, могут храниться разные типы объектов.

В завершение главы были представлены множества. Множества являются изменяемыми, но они не упорядочены. Они используются для удаления дубликатов и проверки принадлежности. Из-за используемого механизма хеширования операции с множествами выполняются быстро и эффективно. Для этого элементы множества должны быть хешируемыми.

14.10. Упражнения

1. Создайте список, сохраните в нем имена ваших коллег и друзей. Изменился ли идентификатор списка? Отсортируйте список. Какой элемент стоит в начале списка? Какой элемент является вторым?

2. Создайте кортеж с вашим именем, фамилией и возрастом. Создайте список `people` и присоедините к нему свой кортеж. Создайте другие кортежи с соответствующей информацией о ваших друзьях и присоедините их к списку. Отсортируйте список. Когда в книге будут рассмотрены функции, вы сможете использовать параметр `key` для сортировки списка по любому полю кортежа: имени, фамилии или возрасту.
3. Создайте список с именами ваших друзей. Создайте список с десятью самыми популярными именами. При помощи операций множеств проверьте, входят ли имена кого-либо из ваших друзей в этот список.
4. Посетите сайт проекта «Гутенберг»¹ и найдите страницу текста из Шекспира. Вставьте текст в строку, заключенную в тройные кавычки. Создайте другую строку с абзацем текста из Ральфа Уолдо Эмерсона. Используйте метод `.split` строк для получения списка слов из каждого текста. При помощи операций с множествами найдите общие слова, встречающиеся в текстах обоих авторов, а также слова, уникальные для каждого автора.
5. Кортежи и списки похожи, но обладают разным поведением. Используйте операции с множествами для нахождения атрибутов объекта списка, отсутствующих у объекта кортежа.

¹ <https://www.gutenberg.org/>

15

Итерации

Одна из стандартных идиом при работе с последовательностями — перебор содержимого последовательности. Например, вы можете отфильтровать один из элементов, применить к элементам функцию, вывести элементы и т. д. Для решения этой задачи можно воспользоваться циклом `for`. В следующем примере выводятся все строки из списка:

```
>>> for letter in ['c', 'a', 't']:
...     print(letter)
c
a
t

>>> print(letter)
t
```

ПРИМЕЧАНИЕ

Обратите внимание: конструкция цикла `for` содержит двоеточие (:), за которым следует код с отступом (*блок цикла for*).

В цикле `for` Python создает новую переменную `letter` для хранения текущего элемента. Обратите внимание: в `letter` хранится не индекс, а строка. Python не очищает переменную после завершения цикла.

15.1. Перебор с индексом

В таких языках, как C, перебор в последовательностях ведется не по элементам последовательности, а по индексам. Используя индексы, вы можете извлекать элементы с указанными индексами. Ниже показан один из вариантов решения этой задачи с использованием встроенных функций Python `range` и `len`:

```
>>> animals = ["cat", "dog", "bird"]
>>> for index in range(len(animals)):
...     print(index, animals[index])
0 cat
1 dog
2 bird
```

Приведенный выше фрагмент содержит код с *душком*: этот термин обычно подразумевает, что вы используете Python не так, как следовало бы. Обычно перебор в последовательностях выполняется для получения доступа к элементам последовательности, а не к индексам. Тем не менее в отдельных случаях индекс тоже бывает нужен. Python предоставляет встроенную функцию `enumerate`, с которой комбинация `range` и `len` становится излишней. Функция `enumerate` возвращает кортеж (*индекс, элемент*) для каждого элемента в последовательности:

```
>>> animals = ["cat", "dog", "bird"]
>>> for index, value in enumerate(animals):
...     print(index, value)
0 cat
1 dog
2 bird
```

Так как кортеж содержит пару из индекса и значения, вы можете воспользоваться *распаковкой кортежа* для создания двух переменных, `index` и `value`, прямо в цикле `for`. Имена переменных должны разделяться запятой. Если длина кортежа совпадает с количеством переменных, включаемых в цикл `for`, Python создаст их за вас.

Создание переменной в цикле for

Команда

Что делает компьютер

```
for letter in ['c', 'a', 't']:  
    print(letter)
```

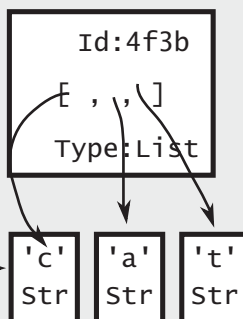
Переменные

Объекты

Вывод

c

letter →



Этап 1: Сначала переменная указывает на c

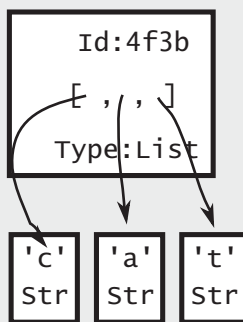
```
for letter in ['c', 'a', 't']:  
    print(letter)
```

Вывод

c

a

letter →



Этап 2: Затем переменная указывает на a

Рис. 15.1. Создание переменной в цикле for. Создается новая переменная letter; сначала она указывает на символ 'c', который выводится вызовом print. Затем цикл переходит на следующую итерацию, и переменная указывает на 'a'

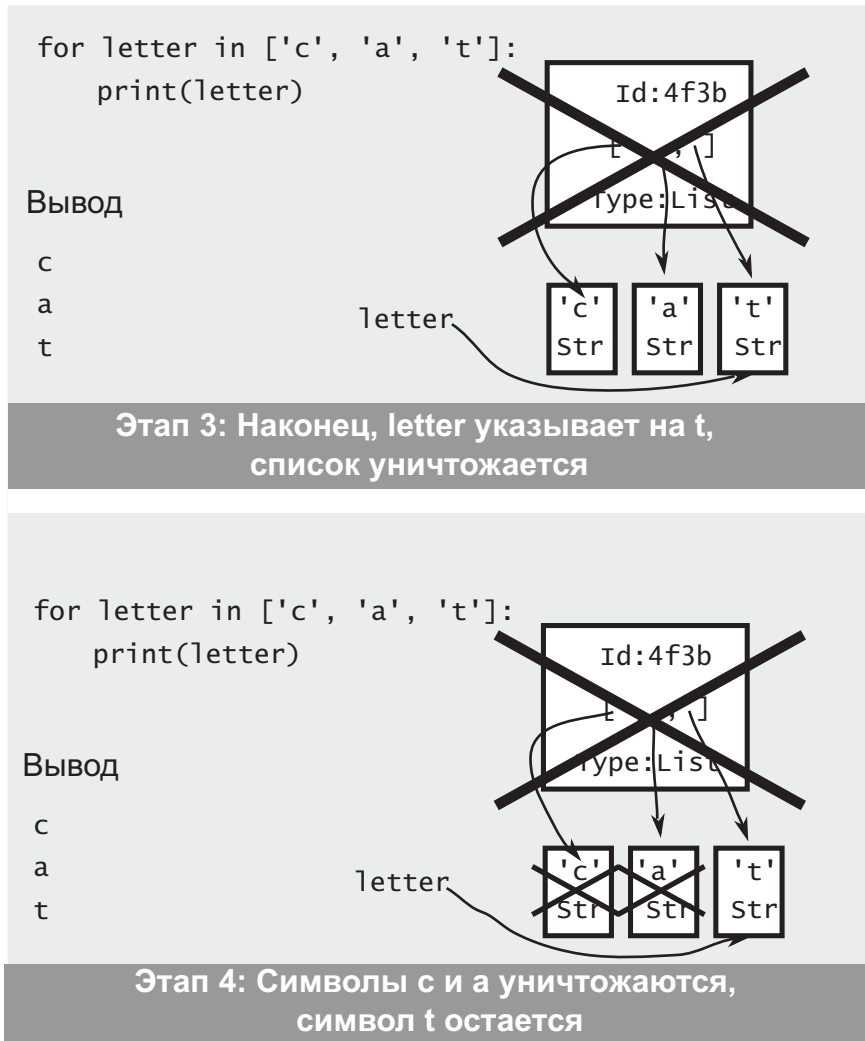


Рис. 15.2. Цикл `for` продолжается, а после вывода `'t'` он завершается. В этот момент литерал списка уничтожается в ходе уборки мусора. Так как на `'c'` и `'a'` указывает только список, они тоже уничтожаются. Однако переменная `letter` продолжает указывать на `'t'`. Python не уничтожает эту переменную, и она продолжит существовать после завершения цикла `for`

15.2. Выход из цикла

Иногда бывает нужно прервать выполнение цикла преждевременно, до перебора всех элементов в цикле. Ключевое слово `break` прерывает ближайший внутренний цикл, в котором вы находитесь. Следующая программа суммирует числа до тех пор, пока не обнаружит отрицательное число. В этом случае выполнение цикла прерывается командой `break`:

```
>>> numbers = [3, 5, 9, -1, 3, 1]
>>> result = 0
>>> for item in numbers:
...     if item < 0:
...         break
...     result += item
>>> print(result)
17
```

ПРИМЕЧАНИЕ

В строке

```
result += item
```

используется так называемое *расширенное присваивание*. Эта команда эквивалентна:

```
result = result + item
```

Расширенное присваивание выполняется чуть быстрее, так как выборка для переменной `result` выполняется всего один раз. Кроме того, команда получается более компактной и быстрее вводится.

ПРИМЕЧАНИЕ

Блок `if` внутри блока `for` снабжен отступом из восьми пробелов. Блоки могут быть вложенными, и каждый уровень должен иметь свой отступ.

15.3. Пропуск элементов в цикле

Другая стандартная идиома циклов — пропуск отдельных элементов при переборе. Если выполнение тела цикла `for` занимает некоторое время, а выполняться оно должно только для некоторых элементов последова-

тельности, вам пригодится ключевое слово `continue`. Команда `continue` приказывает Python прервать обработку текущего элемента цикла и продолжить цикл от начала блока `for` со следующим значением в цикле.

В следующем примере суммируются все положительные числа из списка:

```
>>> numbers = [3, 5, 9, -1, 3, 1]
>>> result = 0
>>> for item in numbers:
...     if item < 0:
...         continue
...     result = result + item
>>> print(result)
21
```

15.4. Оператор `in` может использоваться для проверки принадлежности

Мы использовали команду `in` в цикле `for`. В языке Python эта команда также может использоваться для проверки принадлежности. Если вы хотите узнать, содержит ли список заданный элемент, используйте команду `in` для проверки:

```
>>> animals = ["cat", "dog", "bird"]
>>> 'bird' in animals
True
```

Если вам потребуется узнать индекс, используйте метод `.index`:

```
>>> animals.index('bird')
2
```

15.5. Удаление элементов из списков при переборе

Как уже упоминалось ранее, списки являются изменяемыми. Изменяемость означает, что в них можно добавлять или удалять элементы. Кроме того, списки являются последовательностями, а значит, их содержимое можно перебирать.

Например, если вы захотите отфильтровать список имен так, чтобы в нем остался только элемент 'John' или 'Paul', делать это так было бы неправильно:

```
>>> names = ['John', 'Paul', 'George',
...         'Ringo']
>>> for name in names:
...     if name not in ['John', 'Paul']:
...         names.remove(name)

>>> print(names)
['John', 'Paul', 'Ringo']
```

Что произошло? Python предполагает, что списки не изменяются в процессе перебора. Добравшись до 'George', цикл удаляет имя из списка. Во внутренней реализации Python отслеживает текущий индекс цикла `for`. На этот момент в списке остаются только три элемента: 'John', 'Paul' и 'Ringo'. Однако цикл `for` думает, что текущей является позиция с индексом 3 (четвертый элемент), а четвертого элемента не существует, поэтому цикл останавливается, и элемент 'Ringo' остается на месте.

Существует два альтернативных решения для удаления элементов из списка в процессе перебора. В первом варианте удаляемые элементы отбираются при первом проходе по списку. Следующий цикл перебирает только те элементы, которые подлежат удалению (`names_to_remove`), и удаляет их из исходного списка (`names`):

```
>>> names = ['John', 'Paul', 'George',
...         'Ringo']
>>> names_to_remove = []
>>> for name in names:
...     if name not in ['John', 'Paul']:
...         names_to_remove.append(name)

>>> for name in names_to_remove:
...     names.remove(name)

>>> print(names)
['John', 'Paul']
```


Другое решение — перебор по копии списка. Оно довольно легко реализуется конструкцией копирования среза [:], которая будет рассмотрена в главе, посвященной срезам:

```
>>> names = ['John', 'Paul', 'George',
... 'Ringo']
>>> for name in names[:]: # copy of names
...     if name not in ['John', 'Paul']:
...         names.remove(name)

>>> print(names)
['John', 'Paul']
```

15.6. Блок else

Цикл `for` также может содержать блок `else`. Любой код в блоке `else` будет выполнен в том случае, если цикл `for` не достиг команды `break`. Следующий пример проверяет, являются ли числа из цикла положительными:

```
>>> positive = False
>>> for num in items:
...     if num < 0:
...         break
... else:
...     positive = True
```

Команды `continue` не влияют на выполнение блока `else`.

Имя команды `else` выглядит несколько странно. Для цикла `for` она показывает, что была обработана вся последовательность. Блок `else` в цикле `for` часто применяется для обработки отсутствия элементов.

15.7. Циклы while

Python позволяет многократно выполнять блок кода, пока некоторое условие остается истинным. Такая конструкция называется *циклом while*, а для ее создания используется команда `while`. За циклом `while` следует

выражение, результат которого равен `True` или `False`, а за выражением идет двоеточие. Помните, что следует за двоеточием (`:`) в Python? Да, блок кода с отступом. Этот блок продолжит выполняться, пока результат выражения остается равным `True`. В программе может легко возникнуть бесконечный цикл.

Бесконечные циклы обычно нежелательны, потому что ваша программа «зависает» в цикле без возможности выхода. Впрочем, у правила есть исключения: например, сервер в бесконечном цикле принимает и обрабатывает запросы. Другое исключение, встречающееся в коде Python более высокого уровня, — бесконечный генератор. Генератор ведет себя как отложенный список, который создает значения только тогда, когда они будут задействованы в переборе. Если вы знакомы с обработкой потоков, генератор можно рассматривать как поток. (Генераторы в этой книге не рассматриваются, но я опишу их в книге более высокого уровня.)

Как правило, если у вас имеется объект, поддерживающий перебор, для перебора элементов используется цикл `for`. Циклы `while` используются при отсутствии простого доступа к объекту, поддерживающему перебор.

Типичный пример использования цикла `while` — обратный отсчет:

```
>>> n = 3
>>> while n > 0:
...     print(n)
...     n = n - 1
3
2
1
```

Для выхода из цикла `while` также может использоваться команда `break`:

```
>>> n = 3
>>> while True:
...     print(n)
...     n = n - 1
...     if n == 0:
...         break
```

15.8. Итоги

В этой главе рассматривается использование циклов `for` для перебора элементов последовательности. Вы видели, что в цикле можно вести перебор по спискам; также возможен перебор по строкам, кортежам, словарям и другим структурам данных. Более того, вы можете определять собственные классы, поддерживающие перебор в циклах `for`, реализуя метод `__iter__`.

Цикл `for` создает переменную при переборе. Эта переменная не уничтожается после цикла, а продолжает существовать. Если цикл `for` выполняется внутри функции, переменная будет уничтожена при выходе из функции.

Также в этой главе была представлена функция `enumerate`. Функция возвращает последовательность кортежей (пар «индекс, значение») для переданной последовательности. Если вам понадобится получить при переборе как индекс, так и значение, используйте `enumerate`.

Наконец, вы узнали, как прервать выполнение цикла, перейти к следующему элементу или использовать команду `else`. Все эти конструкции позволяют адаптировать логику циклов для конкретных задач.

15.9. Упражнения

1. Создайте список с именами друзей и коллег. Вычислите среднюю длину имен в списке.
2. Создайте список с именами друзей и коллег. Проведите поиск имени `John` в списке в цикле `for`. Если имя не найдено, выведите соответствующее сообщение (подсказка: используйте `else`).
3. Создайте список кортежей из имени, фамилии и возраста ваших друзей и коллег. Если возраст неизвестен, занесите значение `None`. Вычислите средний возраст, пропустив все значения `None`. Выведите каждое имя, за которым следует строка `old` (возраст выше среднего) или `Young` (возраст ниже среднего).

16

Словари

Словари (dictionaries) — высокооптимизированный встроенный тип в языке Python. Словарь Python отчасти напоминает обычный словарь, состоящий из слов и определений. Задача словаря — обеспечить быстрый поиск определения по ключевому слову. В обычном словаре для ускорения поиска можно воспользоваться методом бинарного поиска (открыть словарь на середине, определить, в какой половине находится искомое слово, и повторить).

Словарь Python тоже состоит из слов и определений, но они называются *ключами* и *значениями* соответственно. Цель словаря — обеспечить быстрый поиск по ключу и извлечь значение, связанное с этим ключом. Как и в обычном словаре, в котором нахождение слова по определению займет много времени (если вы заранее не знаете искомое слово), поиск по значению в словаре Python происходит очень медленно.

В Python 3.6 у словарей появилась новая особенность: ключи теперь сортируются по порядку вставки. Если вы пишете код Python, который должен работать в предыдущих версиях, вы должны запомнить, что до версии 3.6 порядок ключей был произвольным (что позволяло Python выполнять быстрый поиск, но было не особо полезно для конечного пользователя).

16.1. Присваивание в словарях

Словари связывают *ключ* со *значением* (в других языках программирования они могут называться *хешами*, *хеш-картами*, *картами* или *ассо-*

цитативными массивами). Допустим, вы хотите сохранить информацию о человеке. Вы уже видели, как использовать кортеж для представления записи. Словари тоже могут использоваться для решения этой задачи. Так как словари встроены в Python, для их создания можно воспользоваться синтаксисом литералов. В следующем словаре хранится имя и фамилия:

```
>>> info = {'first': 'Pete', 'last': 'Best'}
```

ПРИМЕЧАНИЕ

Также словарь можно создать при помощи встроенного класса `dict`. Если передать классу список пар кортежей, он вернет словарь:

```
>>> info = dict([('first', 'Pete'),  
...              ('last', 'Best')])
```

При вызове `dict` также можно использовать именованные параметры:

```
>>> info = dict(first='Pete', last='Best')
```

Если вы используете именованные параметры, они должны быть допустимыми именами переменных Python, тогда они будут преобразованы в строки.

Для вставки значений в словарь можно воспользоваться индексными операциями:

```
>>> info['age'] = 20  
>>> info['occupation'] = 'Drummer'
```

В этом примере ключами являются `'first'`, `'last'`, `'age'` и `'occupation'`. Например, `'age'` — *ключ*, которому соответствует *значение*: целое число 20. Чтобы быстро найти значение, соответствующее `'age'`, выполните поиск индексной операцией:

```
>>> info['age']  
20
```

С другой стороны, если вам потребуется определить, какой ключ соответствует значению 20, такая операция будет слишком медленной. В приведенном примере продемонстрирован синтаксис литералов для создания изначально заполненного словаря. Пример также показывает, как квадратные скобки (индексные операции) используются для вставки

элементов в словарь и их извлечения. Индексная операция связывает ключ со значением при использовании в сочетании с *оператором присваивания* (=). Если индексная операция не содержит присваивания, она находит значение, соответствующее заданному ключу.

16.2. Выборка значений из словаря

Как вы уже видели, синтаксис литералов с квадратными скобками может извлечь значение из словаря при использовании квадратных скобок без присваивания:

```
>>> info['age']
20
```



Рис. 16.1. Создание словаря. В данном случае в качестве значения используется существующая переменная. Обратите внимание: словарь не копирует переменную, но создает указатель на нее (увеличивая ее счетчик ссылок)

Впрочем, будьте осторожны — при попытке обратиться к ключу, отсутствующему в словаре, Python выдаст исключение:

```
>>> info['Band']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Band'
```

Поскольку для Python характерно явное выражение намерений программиста и быстрое выявление сбоев, выдачу исключений можно только приветствовать. Вы знаете, что ключ 'Band' отсутствует в словаре. Было бы гораздо хуже, если бы Python возвращал случайное значение для ключа, отсутствующего в словаре. Это привело бы к дальнейшему распространению ошибки в программе и маскировке логического дефекта.

16.3. Оператор in

Python предоставляет *оператор* `in`, который позволяет быстро проверить наличие ключа в словаре:

```
>>> 'Band' in info
False
>>> 'first' in info
True
```

Как вы уже видели, `in` также работает с последовательностями. Оператор `in` может использоваться со списком, множеством или строкой для проверки принадлежности:

```
>>> 5 in [1, 3, 6]
False
>>> 't' in {'a', 'e', 'i'}
False
>>> 'P' in 'Paul'
True
```

ПРИМЕЧАНИЕ

В Python 3 был исключен метод `.has_key`, который предоставляет ту же функциональность, что и команда `in`, но работает только со словарями. Любые действия, ведущие к большей универсальности, можно только приветствовать!

Команда `in` работает с большинством контейнеров. Кроме того, вы можете определить собственные классы, поддерживающие эту команду. Ваш объект должен поддерживать перебор или определять метод `.__contains__`.

16.4. Сокращенный синтаксис словарей

Метод `.get` словаря получает значение, соответствующее ключу. Метод `.get` также может получать необязательный параметр для значения по умолчанию, если ключ не найден. Если вы хотите, чтобы `genre` по умолчанию присваивалось значение `'Rock'`, это можно сделать так:

```
>>> genre = info.get('Genre', 'Rock')
>>> genre
'Rock'
```

СОВЕТ

Метод `.get` словарей — один из способов обойти исключение `KeyError` при попытке использования синтаксиса с квадратными скобками для выборки по ключу, отсутствующему в словаре.

16.5. `setdefault`

У словарей есть полезный метод с неудачно выбранным именем `.setdefault`. Этот метод имеет такую же сигнатуру, как `.get`, и поначалу работает так же: он возвращает значение по умолчанию, если ключ не существует. Кроме того, он также связывает с ключом значение по умолчанию, если ключ не найден. Так как `.setdefault` возвращает значение, если инициализировать его изменяемым типом (таким, как словарь или список), результат можно будет изменять на месте.

Метод `.setdefault` может использоваться для создания накопителя или счетчика ключей. Например, если вы хотите подсчитать количество людей с одним именем, это можно сделать так:

```
>>> names = ['Ringo', 'Paul', 'John',
...         'Ringo']
>>> count = {}
>>> for name in names:
```

```
...     count.setdefault(name, 0)
...     count[name] += 1
```

Без метода `.setdefault` потребовалось бы немного больше кода:

```
>>> names = ['Ringo', 'Paul', 'John',
...          'Ringo']
>>> count = {}
>>> for name in names:
...     if name not in count:
...         count[name] = 1
...     else:
...         count[name] += 1

>>> count['Ringo']
2
```

COBET

Подобные операции подсчета встречаются настолько часто, что позднее в стандартную библиотеку Python был добавлен класс `collections.Counter`. Этот класс позволяет выполнять такие операции в более компактном виде:

```
>>> import collections
>>> count = collections.Counter(['Ringo', 'Paul',
...                               'John', 'Ringo'])
>>> count
Counter({'Ringo': 2, 'Paul': 1, 'John': 1})
>>> count['Ringo']
2
>>> count['Fred']
0
```

Ниже приведен несколько искусственный пример, демонстрирующий изменение результата `.setdefault`. Предположим, имеется словарь, связывающий имя музыканта с названием группы, в которой он играл. Если человек по имени Paul участвует в двух группах, результат должен связывать имя Paul со списком, содержащим обе группы:

```
>>> band1_names = ['John', 'George',
...                'Paul', 'Ringo']
>>> band2_names = ['Paul']
>>> names_to_bands = {}
>>> for name in band1_names:
```

```
...     names_to_bands.setdefault(name,
...     []).append('Beatles')
>>> for name in band2_names:
...     names_to_bands.setdefault(name,
...     []).append('Wings')
>>> print(names_to_bands['Paul'])
['Beatles', 'Wings']
```

В развитие темы: без `setdefault` этот код получился бы более длинным:

```
>>> band1_names = ['John', 'George',
... 'Paul', 'Ringo']
>>> band2_names = ['Paul']
>>> names_to_bands = {}
>>> for name in band1_names:
...     if name not in names_to_bands:
...         names_to_bands[name] = []
...     names_to_bands[name].\
...         append('Beatles')
>>> for name in band2_names:
...     if name not in names_to_bands:
...         names_to_bands[name] = []
...     names_to_bands[name].\
...         append('Wings')
>>> print(names_to_bands['Paul'])
['Beatles', 'Wings']
```

СОВЕТ

Модуль `collections` из стандартной библиотеки Python содержит удобный класс — `defaultdict`. Этот класс своим поведением напоминает словарь, но также позволяет задать для ключа в качестве значения по умолчанию произвольную фабрику (`factory`). Если фабрика по умолчанию отлична от `None`, она инициализируется и вставляется как значение каждый раз, когда отсутствует ключ. Предыдущий пример может быть переписан с `defaultdict` в следующем виде:

```
>>> from collections import defaultdict
>>> names_to_bands = defaultdict(list)
>>> for name in band1_names:
...     names_to_bands[name].\
...         append('Beatles')
>>> for name in band2_names:
```

```
...     names_to_bands[name].\
...         append('Wings')
>>> print(names_to_bands['Paul'])
['Beatles', 'Wings']
```

Код с `defaultdict` получается чуть более понятным, чем с использованием `setdefault`.

16.6. Удаление ключей

Другая стандартная операция со словарями — удаление ключей и соответствующих им значений. Для удаления элемента из словаря используется команда `del`:

```
# Удаление 'Ringo' из словаря
>>> del names_to_bands['Ringo']
```

СОВЕТ

Python не позволяет добавлять и удалять данные из словаря во время его перебора. В таких ситуациях Python выдает исключение `RuntimeError`:

```
>>> data = {'name': 'Matt'}
>>> for key in data:
...     del data[key]
Traceback (most recent call last):
...
RuntimeError: dictionary changed size during iteration
```

16.7. Перебор словаря

Словари также поддерживают перебор командой `for`. По умолчанию при переборе по словарю вы получаете ключи:

```
>>> data = {'Adam': 2, 'Zeek': 5, 'Fred': 3}
>>> for name in data:
...     print(name)
Adam
Zeek
Fred
```

ПРИМЕЧАНИЕ

У словарей имеется метод `.keys`, который тоже перебирает ключи из словаря. В Python 3 метод `.keys` возвращает *представление* (view) — «окно» для просмотра ключей, находящихся в настоящий момент в словаре. Представление, как и список, может использоваться для перебора. Однако в отличие от списка, он не является копией этих ключей. Если позднее ключ будет удален из словаря, это изменение отразится в представлении — но не в списке.

Чтобы перебрать значения в словаре, проведите перебор по методу `.values`:

```
>>> for value in data.values():  
...     print(value)  
2  
5  
3
```

Результат вызова `.values` также является представлением. Он отражает текущее состояние значений, находящихся в словаре.

Чтобы получать при переборе как ключ, так и значение, используйте метод `.items`, который возвращает представление:

```
>>> for key, value in data.items():  
...     print(key, value)  
Adam 2  
Zeek 5  
Fred 3
```

Если материализовать представление в список, вы увидите, что список является последовательностью кортежей (ключ, значение) — то, что получает `dict` для создания словаря:

```
>>> list(data.items())  
[('Adam', 2), ('Zeek', 5), ('Fred', 3)]
```

СОВЕТ

Помните, что словарь упорядочивается в порядке вставки ключей. Если вам нужен другой порядок, последовательность перебора необходимо отсортировать.

Встроенная функция `sorted` возвращает новый отсортированный список для заданной последовательности:

```
>>> for name in sorted(data.keys()):
...     print(name)
Adam
Fred
Zeek
```

У функции `sorted` есть необязательный аргумент `reverse` для перехода на противоположный порядок вывода:

```
>>> for name in sorted(data.keys(),
...     reverse=True):
...     print(name)
Zeek
Fred
Adam
```

ПРИМЕЧАНИЕ

Ключи могут относиться к разным типам. Единственное требование к ключам — *хешируемость*. Например, список не является хешируемым, потому что он может изменяться, и Python не может сгенерировать для него хеш, соответствующий текущему содержимому. Если использовать список в качестве ключа, а затем изменить ключ, какое значение должен вернуть словарь — для старого списка, для нового списка, оба сразу? Python не берется гадать и заставляет программиста использовать неизменяемые ключи.

Вы можете вставлять элементы в словарь, используя в качестве ключей целые числа и строки:

```
>>> weird = {1: 'first',
...     '1': 'another first'}
```

Обычно смешивать разнотипные ключи не рекомендуется, потому что это делает код менее понятным и усложняет сортировку. Python 3 не сортирует списки со смешанными типами без функции `key`, которая явно указывает Python, как следует сравнивать разные типы. Это один из тех случаев, когда Python позволяет что-то сделать, но это не значит, что вам стоит пользоваться этой возможностью. Как говорил один из основных разработчиков Python Рэймонд Хеттингер (Raymond Hettinger): «Многие вопросы типа “Могут ли

я сделать `x` на языке Python?” равнозначны вопросу “Могу ли я остановить машину на рельсах, если поблизости нет поезда?” Да, можете. Нет, не стоит».

@raymondh

16.8. Итоги

Эта глава была посвящена словарям. Эта структура данных играет важную роль, потому что она является одним из «строительных блоков» Python. Классы, пространства имен и модули — все это во внутренней реализации Python строится на базе словарей.

Словари обеспечивают быстрый поиск или вставку по ключу. Также возможен поиск по значению, но он выполняется медленно. Если вам часто приходится выполнять эту операцию со словарями, это верный признак кода «с душком» — подумайте об использовании другой структуры данных.

Также были представлены способы изменения словаря. В словаре можно вставлять и удалять ключи, а также проверять принадлежность при помощи команды `in`.

Мы рассмотрели некоторые нетривиальные конструкции, использующие `.setdefault` для вставки и возвращения значений за одну операцию. Также были представлены специализированные классы словарей `Counter` и `defaultdict` из модуля `collections`.

Поскольку словари изменяемы, вы можете удалять из них ключи командой `del`. Также возможен перебор ключей словаря в цикле `for`.

В Python 3.6 появилось упорядочение элементов в словарях. Ключи упорядочиваются в порядке вставки, а не по алфавиту и не в числовом порядке.

16.9. Упражнения

1. Создайте словарь `info`, в котором хранится ваше имя, фамилия и возраст.

2. Создайте пустой словарь `phone` с подробной информацией о вашем телефоне. Добавьте в словарь данные о размере экрана, объеме памяти, ОС, фирме-производителе и т. д.
3. Напишите абзац текста в строке, заключенной в тройные кавычки. Используйте метод `.split` для создания списка слов. Создайте словарь для хранения счетчика вхождений каждого слова в абзаце.
4. Посчитайте, сколько раз используется каждое слово в абзаце текста Ральфа Уолдо Эмерсона.
5. Напишите код для вывода анаграмм (слов, состоящих из тех же букв в другом порядке) из абзаца текста.
6. Алгоритм *PageRank* использовался для создания поисковой системы Google. Алгоритм назначает каждой странице ранг, основанный на количестве входящих ссылок. Он получает одно входное значение: список страниц, ссылающихся на другие страницы. Каждой странице изначально назначается ранг 1. Выполняется несколько итераций алгоритма — обычно 10. Для каждой итерации:
 - страница передает свой ранг, разделенный на количество исходящих ссылок, каждой странице, с которой она связана ссылкой;
 - перенесенный ранг умножается на *коэффициент затухания*, обычно равный 0,85.

Напишите код для выполнения 10 итераций этого алгоритма со списком кортежей входных и выходных ссылок:

```
links = [('a', 'b'), ('a', 'c'), ('b', 'c')]
```

17

Функции

Мы прошли долгий путь без обсуждения функций, которые являются основными структурными элементами программ Python. Функции представляют собой фрагменты кода, выделенные в отдельный блок. В примерах уже использовались такие встроенные функции, как `dir` и `help` (а также классы, которые ведут себя как функции преобразования типа — `float`, `int`, `dict`, `list` и `bool`).

Функцию можно рассматривать как «черный ящик», которому передаются входные данные (хотя их наличие не обязательно). Затем «черный ящик» выполняет серию операций и возвращает результат (если функция завершается без вызова `return`, она неявно возвращает `None`). Главным преимуществом функций является возможность повторного использования кода. После того как функция будет определена, вы сможете вызывать ее снова и снова. Если в программе имеется код, который должен выполняться многократно, то вместо копирования/вставки вы можете один раз оформить его в виде функции, а затем вызывать эту функцию. При этом сокращается объем кода, а сама программа становится более понятной. Кроме того, становится проще вносить изменения (и исправлять ошибки), поскольку это делается в одном месте.

Рассмотрим простой пример функции. Эта функция с именем `add_2` получает на входе число, прибавляет к нему 2 и возвращает результат:

```
>>> def add_2(num):  
...     '''  
...     return 2 more than num  
...     '''  
...     result = num + 2  
...     return result
```

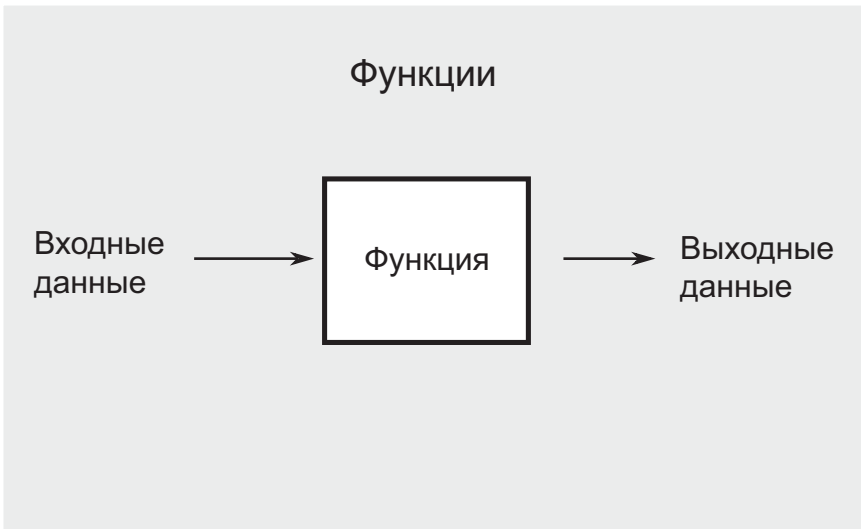



Рис. 17.1. Функция работает как «черный ящик»: она получает входные данные и выдает результат (выходные данные). Функции можно передавать при вызове других функций, а также использовать повторно

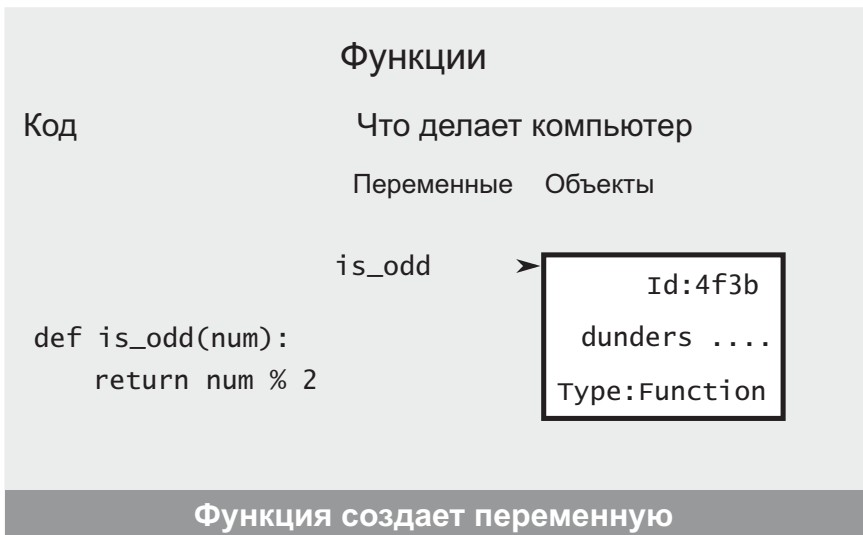


Рис. 17.2. Создание функции. Обратите внимание: Python создает новый объект функции, а затем сохраняет указатель на него в переменной с именем функции. Для просмотра атрибутов только что созданной функции можно вызвать функцию `dir` для имени функции

Из каких частей состоит функция? Весь фрагмент кода называется *определением функции*. Определение начинается с команды `def` (сокращение от *define*, то есть «определение»). За `def` следует обязательный пробел (достаточно одного) и имя функции — `add_2`. Это имя будет использоваться для *вызова* функции (то есть ее выполнения). При создании функции Python создаст новую переменную, имя которой совпадает с именем функции.

За именем функции следует открывающая круглая скобка, за которой следует `num` и закрывающая круглая скобка. Имена между скобками (их может быть сколько угодно, хотя в данном случае имя только одно) определяют входные *параметры* — объекты, передаваемые функции.

За круглыми скобками следует двоеточие (`:`). Когда вы видите двоеточие в Python, за ним почти наверняка следует блок с отступом — по аналогии с телом цикла `for` (см. выше). Весь код с отступом образует *тело* функции.

В теле функции содержится вся логика. Вероятно, вы узнаете в первых трех строках кода с отступом конструкцию строки в тройных кавычках. Это не комментарий, хотя выглядит похоже, это строка. Python позволяет разместить непосредственно после `:` строку, которая называется *строкой документации*. Строка документации используется исключительно для документирования кода. В ней должно содержаться описание блока кода, следующего за ней. Строка документации никак не влияет на логику функции.

СОВЕТ

Функция `help` неоднократно упоминалась в этой книге. Важно отметить, что эта функция получает информацию из *строки документации* переданного объекта. Если вызвать `help` для `add_2`, вы получите следующий результат (при условии, что вы ввели код `add_2`, приведенный выше):

```
>>> help(add_2)
Help on function add_2 in module
__main__:
add_2()
    return 2 more than num
(END)
```

Строки документации напоминают вам, что делает ваш код. Содержательные и точные, они предоставят бесценную информацию каждому, кто захочет использовать ваш код.

После строки документации (обратите внимание: строка документации не является обязательной) следует логика функции. Здесь вычисляется результат. Наконец, команда `return` сообщает, что у функции имеется результат, то есть выходное значение. Команда `return` не является обязательной, и при ее отсутствии функция по умолчанию возвращает `None`. Функция может содержать несколько команд `return`, и они даже не обязаны находиться в конце функции. Например, условная команда может содержать две команды `return`: в блоке `if` и в блоке `else`.

Подведем итог. Основные части функции:

- Ключевое слово `def`.
- Имя функции.
- Параметры функции в круглых скобках.
- Двоеточие (`:`).
- Отступ:
 - Строка документации.
 - Логика.
 - Команда `return`.

Создать функцию несложно. Функции позволяют повторно использовать код, отчего код становится более коротким и понятным. Функции помогают исключить глобальное состояние, заменяя его переменными с коротким сроком жизни в теле функции. Используйте функции для улучшения структуры кода.

17.1. Вызов функций

В Python функции вызываются по имени функции, за которым следуют круглые скобки. В следующем фрагменте вызывается только что определенная функция `add_2`:

```
>>> add_2(3)
5
```

Чтобы вызвать функцию, укажите ее имя, за которым следует открывающая круглая скобка, входные параметры и закрывающая круглая скобка.

Количество параметров должно соответствовать количеству параметров в объявлении функции. Обратите внимание: REPL выводит результат вызова — целое число 5 (то, что возвращает команда `return`).

Функции `add_2` можно передать произвольный объект. Но если этот объект не поддерживает сложение с числами, будет выдано исключение. При передаче строки выдается исключение `TypeError`:

```
>>> add_2('hello')
Traceback (most recent call last):
...
TypeError: must be str, not int
```

17.2. Область видимости

Python ищет переменные в разных местах. Эти места называются *областями видимости* или *пространствами имен*. При поиске переменной (не забывайте, что функции в Python также являются переменными — как и классы, модули и т. д.), Python выполняет поиск в следующих местах и в следующем порядке:

- *Локальная область видимости* — переменные, определенные внутри функций.
- *Глобальная область видимости* — переменные, определяемые на глобальном уровне.
- *Встроенная область видимости* — переменные, заранее определенные в Python.

В следующем коде поиск переменных осуществляется по всем трем областям видимости:

```
>>> x = 2 # Глобальная
>>> def scope_demo():
...     y = 4 # Локальная для scope_demo
...     print("Local: {}".format(y))
...     print("Global: {}".format(x))
...     print("Built-in: {}".format(dir))

>>> scope_demo()
Local: 4
Global: 2
Built-in: <built-in function dir>
```

После вызова `scope_demo` локальная переменная `y` уничтожается в ходе уборки мусора и становится недоступной в глобальной области видимости:

```
>>> y
Traceback (most recent call last):
...
NameError: name 'y' is not defined
```

Переменные, определяемые внутри функции или метода, являются локальными. В общем случае стоит избегать глобальных переменных, потому что они усложняют понимание кода. Глобальные переменные часто встречаются в учебниках, блогах и документации, потому что их использование сокращает объем кода и помогает сосредоточиться на концепциях, не отвлекаясь на упаковку переменных в функциях. Функции и классы помогают избавиться от глобальных переменных, улучшают модульность кода и упрощают его понимание.

ПРИМЕЧАНИЕ

Python позволяет замещать (переопределять) переменные в глобальной и встроенной области видимости. На глобальном уровне вы можете определить собственную переменную с именем `dir`. В этот момент встроенная функция `dir` *замещается* глобальной переменной. То же самое можно сделать внутри функции и создать локальную переменную, которая замещает глобальную или встроенную переменную:

```
>>> def dir(x):
...     print("Dir called")

>>> dir('')
Dir called
```

Команда `del` может использоваться для удаления переменных в локальной или глобальной области видимости. Однако на практике лучше с самого начала избегать замещения встроенных имен:

```
>>> del dir
>>> dir('')
['_add_', '__class__', '__contains__', ... ]
```

ПОДСКАЗКА

Функции `locals` и `globals` используются для вывода содержимого этих областей видимости. Они возвращают словари с текущим содержимым области видимости:

```
>>> def foo():
...     x = 1
...     print(locals())

>>> foo()
{'x': 1}
```

Переменная `__builtins__` выводит имена из встроенной области видимости. Ее атрибут `__dict__` выдает такой же словарь, как для глобальных и локальных имен.

17.3. Множественные параметры

Функции могут получать несколько параметров. Следующая функция получает два параметра и возвращает их сумму:

```
>>> def add_two_nums(a, b):
...     return a + b
```

Так как Python является динамическим языком, указывать типы параметров не нужно. Эта функция может суммировать два целых числа:

```
>>> add_two_nums(4, 6)
10
```

А может суммировать числа с плавающей точкой:

```
>>> add_two_nums(4.0, 6.0)
10.0
```

И строки тоже:

```
>>> add_two_nums('4', '6')
'46'
```

Обратите внимание: для строк используется операция `+` для выполнения *конкатенации* (сцепления двух строк).

Но если вы попытаете сложить строку с числом, Python сообщит об ошибке:

```
>>> add_two_nums('4', 6)
Traceback (most recent call last):
...
TypeError: Can't convert 'int' object to str implicitly
```

Это одна из тех ситуаций, когда Python требует более точно описать нужную операцию и не пытается угадывать за вас. Если вы хотите сложить строковый тип с числом, возможно, сначала нужно преобразовать их к числовому формату (при помощи `float` или `int`). С другой стороны, если нужно выполнить операцию конкатенации, следует преобразовать числа в строки. Python не выбирает выполняемую операцию автоматически. Вместо этого выдается ошибка, которая заставляет программиста разрешить неоднозначность.

17.4. Параметры по умолчанию

Одна из удобных особенностей функций Python — *параметры по умолчанию*. Как следует из названия, они позволяют задать значения по умолчанию для параметров функций. Параметры по умолчанию не являются обязательными, хотя при необходимости их можно переопределить.

Следующая функция похожа на `add_two_nums`, но если при вызове второе число не указано, по умолчанию прибавляется 3:

```
>>> def add_n(num, n=3):
...     """default to
...     adding 3"""
...     return num + n

>>> add_n(2)
5
>>> add_n(15, -5)
10
```

Чтобы создать для параметра значение по умолчанию, поставьте после параметра знак равенства (=) и нужное значение.

ПРИМЕЧАНИЕ

Параметры по умолчанию должны объявляться после обычных параметров, в противном случае Python выдаст ошибку `SyntaxError`:

```
>>> def add_n(num=3, n):
...     return num + n
Traceback (most recent call last):
...
SyntaxError: non-default argument follows
default argument
```

Python требует, чтобы обязательные параметры были объявлены ранее не-обязательных. Приведенный выше код не будет работать для вызова вида `add_n(4)`, потому что отсутствует обязательный параметр.

СОВЕТ

Не используйте изменяемые типы (списки, словари) в качестве параметров по умолчанию — разве что вы очень хорошо понимаете, что делаете. Из-за особенностей работы Python параметры по умолчанию создаются только один раз — во время определения функции, а не во время ее выполнения. Если вы используете изменяемое значение по умолчанию, то при каждом вызове функции будет заново использован тот же экземпляр параметра по умолчанию:

```
>>> def to_list(value, default=[]):
...     default.append(value)
...     return default

>>> to_list(4)
[4]
>>> to_list('hello')
[4, 'hello']
```

Тот факт, что параметры по умолчанию создаются в момент генерирования функции, многие программисты считают дефектом. Это связано с тем, что такое поведение чревато разными неожиданностями. Обходное решение заключается в том, чтобы вынести создание значений по умолчанию из фазы определения функции (которая выполняется всего один раз) в фазу выполнения функции (чтобы новое значение создавалось при каждом выполнении функции).

Модифицируйте изменяемые параметры по умолчанию, чтобы им присваивалось значение `None`. Затем создайте экземпляр нужного изменяемого типа в теле функции, если значение по умолчанию равно `None`:


```
>>> def to_list2(value, default=None):
...     if default is None:
...         default = []
...     default.append(value)
...     return default

>>> to_list2(4)
[4]
>>> to_list2('hello')
['hello']
```

Следующий код:

```
... if default is None:
...     default = []
```

можно записать в одну строку с использованием *условного выражения*:

```
... default = default if default is not None else []
```

17.5. Правила выбора имен для функций

Правила выбора имен функций имеют много общего с правилами выбора имен переменных (и они также находятся в документе PEP 8). В именах используется так называемый *змеиный регистр*, который проще читается. Имена функций:

- должны записываться в нижнем регистре;
- слова должны разделяться подчеркиваниями;
- не должны начинаться с цифр;
- не должны переопределять встроенные имена;
- не должны совпадать с ключевыми словами.

В таких языках, как Java, используется так называемый «верблюжий регистр». По этой схеме создаются имена переменных вида `sectionList` или `hasTimeOverlap`. В Python переменным были бы присвоены имена `section_list` и `has_time_overlap` соответственно. Хотя код Python должен следовать соглашениям PEP 8, в PEP 8 также принимается в расчет единство стиля. Если в коде, над которым вы работаете, используются разные

схемы назначения имен, следуйте примеру и используйте схему, применяемую в существующем коде. Собственно, в модуле `unittest` из стандартной библиотеки до сих пор применяется схема в стиле Java (потому что изначально этот модуль был импортирован из библиотеки Java `junit`).

17.6. Итоги

Функции позволяют инкапсулировать изменения и побочные эффекты в своем теле. В этой главе вы узнали, что функции могут получать ввод и возвращать результат. Входных параметров может быть несколько, и им можно назначать значения по умолчанию.

Вспомните, что в Python нет ничего, кроме объектов, а при создании функции вы также создаете переменную с именем функции, которая указывает на эту функцию.

Функции также могут включать строку документации, которая записывается непосредственно после объявления. Эти строки образуют документацию, которая выводится при вызове `help` для функции.

17.7. Упражнения

1. Напишите функцию `is_odd`, которая получает целое число и возвращает `True` для нечетных чисел или `False` для четных.
2. Напишите функцию `is_prime`, которая получает целое число и возвращает `True` для простых чисел или `False` для чисел, не являющихся простыми.
3. Напишите функцию бинарного поиска. Функция должна получать отсортированную последовательность и искомый элемент и возвращать индекс найденного элемента. Если элемент не найден, функция должна возвращать `-1`.
4. Напишите функцию, которая получает строки в «верблюжьем регистре» (`ThisIsCamelCased`) и преобразует их в «змеиный регистр» (`this_is_camel_cased`). Измените функцию, добавив в нее аргумент `separator`, чтобы функция также могла выполнять преобразование к «кебаб-регистру» (`this-is-camel-case`).

18

Индексирование и срезы

Python предоставляет две конструкции для извлечения данных из последовательностей (списки, кортежи и даже строки). Речь идет о конструкциях индексирования и срезах. Индексирование позволяет извлекать отдельные элементы из последовательности, а срезы предназначены для извлечения подпоследовательностей.

18.1. Индексирование

Индексирование уже было продемонстрировано ранее для списков. Например, если у вас имеется список с названиями животных, вы сможете выбирать элементы по индексу:

```
>>> my_pets = ["dog", "cat", "bird"]
>>> my_pets[0]
'dog'
```

СОВЕТ

Напомним, что в Python индексирование начинается с 0. Чтобы извлечь первый элемент, используйте индекс 0, а не 1.

В Python предусмотрена удобная возможность обращения к элементам по отрицательным индексам. Индекс -1 обозначает последний элемент, -2 — предпоследний и т. д. Эта запись чаще всего используется для получения последнего элемента списка:

```
>>> my_pets[-1]
'bird'
```

Гвидо ван Россум, создатель Python, в своем твите объяснил, как следует понимать отрицательные значения индексов:

«...Правильный подход [к отрицательному индексированию] — интерпретировать `a[-X]` как `a[len(a)-X]`»

@gvanrossum

Операции индексирования также можно выполнять с кортежами и строками:

```
>>> ('Fred', 23, 'Senior')[1]
23
```

```
>>> 'Fred'[0]
'F'
```

Некоторые типы, например множества, не поддерживают операции индексирования. Если вы хотите определить собственный класс, поддерживающий операции индексирования, реализуйте метод `__getitem__`.



Рис. 18.1. Положительные и отрицательные значения индексов

18.2. Срезы

Кроме извлечения одного элемента по целочисленному индексу вы можете воспользоваться *срезом* (slice) для извлечения подпоследовательности. Срез может содержать начальный индекс, необязательный

конечный индекс и необязательное приращение (все значения разделяются двоеточиями).

Срез для извлечения первых двух элементов списка:

```
>>> my_pets = ["dog", "cat", "bird"] # список
>>> print(my_pets[0:2])
['dog', 'cat']
```

Напомним, что в Python используются *полуоткрытые интервалы*. Список доходит до конечного индекса, но не включает его. Как упоминалось ранее, функция `range` также аналогично ведет себя со вторым параметром.

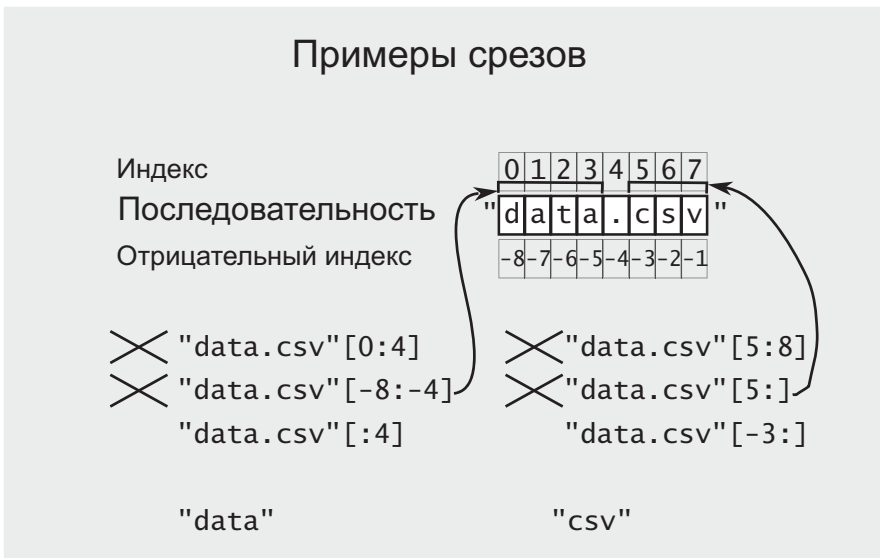


Рис. 18.2. Выделение первых четырех символов строки. Показаны три варианта; предпочтительным считается последний. Нулевой индекс не указан, так как он используется по умолчанию. Решение с отрицательными индексами выглядит просто глупо. Также продемонстрирован срез трех последних символов. И снова последний вариант считается идиоматическим решением. Первые два варианта предполагают, что длина строки равна 8 символам, а последний будет работать с любой строкой, содержащей не менее 3 символов

При определении среза с двоеточием (:) первый индекс не является обязательным. Если первый индекс не указан, то срез по умолчанию начинается с первого элемента списка (нулевой элемент):

```
>>> print(my_pets[:2])  
['dog', 'cat']
```

В срезах также могут использоваться отрицательные индексы. Отрицательной может быть как начальная, так и конечная позиция. Индекс -1 представляет последний элемент. Если срез распространяется до последнего элемента, вы получите все, кроме этого элемента:

```
>>> my_pets[0:-1]  
['dog', 'cat']  
>>> my_pets[:-1] # defaults to 0  
['dog', 'cat']  
  
>>> my_pets[0:-2]  
['dog']
```

Последний индекс также не является обязательным. Если последний индекс отсутствует, срез по умолчанию распространяется до конца списка:

```
>>> my_pets[1:]  
['cat', 'bird']  
>>> my_pets[-2:]  
['cat', 'bird']
```

Наконец, для начального и конечного индекса могут использоваться значения по умолчанию. Если оба индекса отсутствуют, то возвращаемый срез проходит от начала до конца (и содержит копию списка). Эта конструкция может использоваться для быстрого копирования списков в Python:

```
>>> print(my_pets[:])  
['dog', 'cat', 'bird']
```

18.3. Приращения в срезах

После начального и конечного индекса срез также может получать приращение. Если приращение не задано, по умолчанию используется значение 1. Приращение 1 означает, что из последовательности извлекается каждый элемент между индексами. С приращением 2 берется каждый второй элемент, с приращением 3 — каждый третий и т. д.:

```
>>> my_pets = ["dog", "cat", "bird"]
>>> dog_and_bird = my_pets[0:3:2]
>>> print(dog_and_bird)
['dog', 'bird']

>>> zero_three_six = [0, 1, 2, 3, 4, 5, 6][::3]
>>> print(zero_three_six)
[0, 3, 6]
```

ПРИМЕЧАНИЕ

Функция `range` также поддерживает третий параметр, задающий приращение:

```
>>> list(range(0, 7, 3))
[0, 3, 6]
```

Приращение может быть отрицательным. Приращение `-1` означает, что вы двигаетесь в обратном направлении справа налево. Чтобы использовать отрицательное приращение, укажите значение начального индекса больше конечного. Исключением является ситуация, в которой опущен как начальный, так и конечный индекс: приращение `-1` переставляет элементы последовательности в обратном порядке:

```
>>> my_pets[0:2:-1]
[]
>>> my_pets[2:0:-1]
['bird', 'cat']

>>> print([1, 2, 3, 4][::-1])
[4, 3, 2, 1]
```

Когда в следующий раз на собеседовании вам предложат переставить символы строки в обратном порядке, это можно сделать в одной строке:

```
>>> 'emerih'[::-1]
'hireme'
```

Конечно, от вас, скорее всего, потребуют сделать это на C. Просто скажите, что вы хотите программировать на Python!

18.4. Итоги

Операции индексирования используются для извлечения отдельных значений из последовательностей. Например, они позволяют легко получить символ из строки или элемент из списка либо кортежа.

Если вам нужна подпоследовательность, используйте синтаксическую конструкцию среза. Срезы соблюдают принцип полуоткрытых интервалов и дают последовательность до конечного индекса (не включая его). Если вы передадите необязательное приращение, то сможете пропускать элементы при формировании среза.

В Python предусмотрена удобная возможность использования отрицательных значений для индексирования или создания среза относительно конца последовательности. Это позволяет выполнять операции относительно длины последовательности, так что вам не приходится беспокоиться о вычислении длины и вычитании смещений из полученного результата.

18.5. Упражнения

1. Создайте переменную с вашим именем, хранящимся в формате строки. Используйте операции индексирования для получения первого символа. Извлеките последний символ. Будет ли ваш код для извлечения последнего символа работать с именем произвольной длины?
2. Создайте переменную `filename`. Предполагая, что за именем файла следует трехбуквенное расширение, найдите расширение с использованием операции среза. Так, для файла `README.txt` должно быть получено расширение `txt`. Будет ли ваш код работать с именами файлов произвольной длины?
3. Создайте функцию `is_palindrome` для проверки того, что переданное слово одинаково читается в обоих направлениях.

19

Операции ввода/вывода с файлами

Операции чтения и записи файлов часто встречаются в программировании. Python в значительной мере упрощает выполнение этих операций. Вы можете читать как текстовые, так и двоичные файлы; чтение может осуществляться по байтам, строкам и даже по всему содержимому файла. Аналогичные возможности доступны и для записи файлов.

19.1. Открытие файлов

Функция Python `open` возвращает объект файла. Эта функция может получать несколько необязательных параметров:

```
open(filename, mode='r', buffering=-1, encoding=None,
      errors=None, newline=None, closefd=True, opener=None)
```

ПОДСКАЗКА

В системе Windows могут возникнуть проблемы из-за символа `\`, используемого в качестве разделителя путей. Строки Python также используют `\` как экранирующий символ. Если у вас имеется каталог с именем `test` и вы используете запись `"C:\test"`, Python интерпретирует `\t` как символ табуляции.

Проблема решается использованием необработанных строк для представления путей Windows. Поставьте символ `r` перед строкой:

```
r"C:\test"
```

Обычно интерес представляют только первые два или три параметра. Первый параметр — имя файла — является обязательным. Второй параметр определяет, какая операция выполняется с файлом (чтение или запись) и является файл текстовым или двоичным. Режимы перечислены в таблице на рис. 19.1.

Режим	Описание
'r'	Чтение текстового файла (используется по умолчанию)
'w'	Запись текстового файла (перезапись, если файл существует)
'x'	Запись текстового файла (исключение <code>FileExistsError</code> , если файл существует)
'a'	Присоединение к текстовому файлу (запись в конец)
'rb'	Чтение двоичного файла
'wb'	Запись двоичного файла (перезапись)
'w+b'	Открытие двоичного файла для чтения и записи
'xb'	Запись двоичного файла (исключение <code>FileExistsError</code> , если файл существует)
'ab'	Присоединение к двоичному файлу (запись в конец)

Рис. 19.1. Режимы работы с файлами

Чтобы получить подробную информацию о параметрах функции `open`, вызовите для нее `help`. Эта функция содержит чрезвычайно подробную документацию.

Если вы работаете с текстовыми файлами, обычно в параметре `themode` передается `'r'` для *чтения* файла или `'w'` для *записи*. В следующем примере файл `/tmp/a.txt` открывается для записи. Python создаст этот файл или заменит его, если он уже существует:

```
>>> a_file = open('/tmp/a.txt', 'w')
```

Объект файла, возвращаемый при вызове `open`, содержит различные методы для чтения и записи. В этой главе рассматриваются наиболее часто используемые методы. Чтобы ознакомиться с полной документацией по объекту файла, передайте его при вызове функции `help`. Функция выведет список всех методов и покажет, что они делают.

ПРИМЕЧАНИЕ

В системах UNIX временные файлы хранятся в каталоге `/tmp`. Если вы работаете в Windows, просмотрите переменную среды `Temp` следующей командой:

```
с:\> ЕCHO %Temp%
```

Обычно результат выглядит так:

```
C:\Users\<username>\AppData\Local\Temp
```

19.2. Чтение текстовых файлов

Python предоставляет разные способы чтения данных из файлов. Открыв файл в текстовом режиме (используется по умолчанию), вы сможете читать или записывать в него строки. Если же файл открыт в двоичном режиме (`'rb'` для чтения, `'wb'` для записи), то вы будете читать и записывать *байтовые* строки.

Для чтения отдельной строки из существующего текстового файла используется метод `.readline`. Если режим не задан, Python по умолчанию использует режим чтения текстового файла (режим `r`):

```
>>> passwd_file = open('/etc/passwd')
>>> passwd_file.readline()
'root:x:0:0:root:/root:/bin/bash'
```

Будьте осторожны — при попытке открыть для чтения несуществующий файл Python выдает ошибку:

```
>>> fin = open('bad_file')
Traceback (most recent call last):
...
IOError: [Errno 2] No such file or
directory: 'bad_file'
```

СОВЕТ

Функция `open` возвращает экземпляр *объекта файла*. У этого объекта есть методы для чтения и записи данных.

Стандартные имена переменных для объектов файлов — `fin` (file input), `fout` (file output), `fp` (file pointer — используется для ввода или вывода),

или такие имена, как `passwd_file`. Такие имена, как `fin` и `fout`, удобны тем, что они показывают, что файл используется для чтения или записи соответственно.

Как показано выше, метод `.readline` возвращает одну строку файла. Вы можете многократно вызывать его для получения каждой строки или же воспользоваться методом `.readlines` для получения списка, содержащего все строки.

Чтобы прочитать все содержимое файла в одну строку, используйте метод `.read`:

```
>>> passwd_file = open('/etc/passwd')
>>> print(passwd_file.read())
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
```

Всегда закрывайте файлы после завершения работы с ними; для этого следует вызвать метод `.close`:

```
>>> passwd_file.close()
```

Как видите, закрыть файл совсем несложно. А немного позднее вы поймете, почему это так важно.

19.3. Чтение двоичных файлов

Чтобы прочитать данные из двоичного файла, передайте в `themode` строку `'rb'` (Read Binary). При чтении данных из двоичного файла вы получаете не обычные строки, а *байтовые строки*. Не беспокойтесь, интерфейс для работы с байтовыми строками очень похож на интерфейс работы с обычными строками. Ниже приведены первые 8 байтов PNG-файла. Чтобы прочитать 8 байтов, передайте значение 8 методу `.read`:

```
>>> bfin = open('img/dict.png', 'rb')
>>> bfin.read(8)
b'\x89PNG\r\n\x1a\n'
```

Обратите внимание на `b` в начале строки — это признак байтовой строки. Вы также можете вызвать `.readline` для двоичного файла; метод будет читать данные до тех пор, пока не достигнет `b'\n'`. Сравнение двоичных и обычных строк будет представлено в одной из следующих глав.

19.4. Перебор при работе с файлами

Перебор в последовательностях уже упоминался ранее. В языке Python вы можете легко провести перебор по строкам файла. При работе с текстовым файлом можно провести перебор по методу `.readlines` для того, чтобы получать строки одну за одной:

```
>>> fin = open('/etc/passwd')
>>> for line in fin.readlines():
...     print(line)
```

Но поскольку `.readlines` возвращает список, Python придется прочитать весь файл для создания списка, а это может создать проблемы. Если, допустим, файл содержит записи журнала на сервере, он может израсходовать всю свободную память. Впрочем, у Python для таких случаев имеется один прием: для перебора строк файла Python позволяет провести перебор по экземпляру файла. При прямом переборе по файлу Python не выполняет работу заранее и читает строки текста только по мере надобности:

```
>>> fin = open('/etc/passwd')
>>> for line in fin:
...     print(line)
```

Как происходит перебор прямо по экземпляру файла? У Python имеется специальный метод `.__iter__`, который определяет поведение для перебора по экземпляру. Для класса файла метод `.__iter__` перебирает строки файла.

Метод `.readlines` следует применять только в том случае, если вы уверены, что файл поместится в памяти, а вам потребуется обращаться к строкам неоднократно. В противном случае прямой перебор по файлу является предпочтительным.

19.5. Запись файлов

Чтобы записать данные в файл, необходимо сначала открыть файл в режиме *записи*. Если выбрать режим `'w'`, файл будет открыт для записи текстовых данных:

```
>>> fout = open('/tmp/names.txt', 'w')
>>> fout.write('George')
```

Этот вызов попытается перезаписать файл `/tmp/names.txt` в том случае, если он существует; в противном случае файл будет создан. Если у вас нет разрешения для обращения к файлу, выдается ошибка `Permission Error`. Если путь указан некорректно, вы получите ошибку `FileNotFoundError`.

Для размещения данных в файле используются два метода — `.write` и `.writelines`. Метод `.write` получает строку в параметре и записывает данные в файл. Метод `.writelines` получает последовательность со строковыми данными и записывает ее в файл.

ПРИМЕЧАНИЕ

Чтобы включить в файл символы новой строки, вы должны явно передать их методам файла. На платформах UNIX строки, передаваемые `.write`, должны завершаться комбинацией `\n`. Аналогичным образом все строки последовательности, передаваемой `.writelines`, тоже должны завершаться `\n`. В системе Windows признаком новой строки служит комбинация `\r\n`.

Чтобы вы могли писать кроссплатформенный код, строка `linesep` из модуля `os` определяет правильную комбинацию новой строки для платформы:

```
>>> import os
>>> os.linesep # Платформа UNIX
'\n'
```

СОВЕТ

Если вы опробовали приведенный выше пример в интерпретаторе, возможно, вы заметили, что файл `/tmp/names.txt` остался пустым, хотя вы и приказали Python записать в него имя «George». Что произошло?

Файловый вывод *буферизуется* операционной системой. Чтобы оптимизировать запись на носители информации, операционная система записывает данные только при достижении некоторого порога. В системах Linux этот порог обычно составляет 4 Кбайт.

Чтобы осуществить принудительную запись данных, вызовите метод `.flush`, который сбрасывает незаписанные данные на носитель информации.

Более радикальный механизм, гарантирующий, что данные будут записаны, основан на вызове метода `.close`. Вызов этого метода сообщает Python, что запись в файл завершена:

```
>>> fout2 = open('/tmp/names2.txt',  
...               'w')  
>>> fout2.write('John\n')  
>>> fout2.close()
```

19.6. Заккрытие файлов

Как упоминалось ранее, вызов `.close` записывает файловые буферы на носитель информации. Правила Python требуют всегда закрывать файлы после завершения работы с ними (как для чтения, так и для записи).

Есть несколько причин, по которым файлы должны явно закрываться в программах:

- Если файл хранится в глобальной переменной, он никогда не будет автоматически закрыт во время выполнения программы.
- сPython автоматически закрывает файлы, если они уничтожаются в ходе уборки мусора. Другие реализации Python могут этого не делать.
- Без вызова `.flush` вы не знаете, когда ваши данные будут записаны.
- Вероятно, в вашей операционной системе установлено ограничение количества открытых файлов на процесс.
- Некоторые операционные системы не позволяют удалить открытый файл.

Python обычно берет на себя уборку мусора, и программисту не приходится беспокоиться об уничтожении объектов. Операции открытия и закрытия файлов являются исключениями. Python автоматически закрывает файл при выходе объекта файла из области видимости. Тем не менее в этом случае не стоит полагаться на уборку мусора. Явно выражайте свои намерения и прибирайте за собой — обязательно закрывайте файлы!

В Python 2.5 появилась команда `with`. Она используется в сочетании с *менеджерами контекста* для контроля условий, которые происходят до и после выполнения блока. Функция `open` также служит менеджером контекста для обеспечения того, что файл будет открыт перед входом в блок и будет закрыт при выходе из блока. Пример:

```
>>> with open('/tmp/names3.txt', 'w') as fout3:  
...     fout3.write('Ringo\n')
```

Этот фрагмент эквивалентен следующему:

```
>>> fout3 = open('/tmp/names3.txt', 'w')  
>>> fout3.write('Ringo\n')  
>>> fout3.close()
```

Обратите внимание на то, что строка с `with` завершается двоеточием. Когда строка кода Python завершается двоеточием, последующий код всегда снабжается отступом. Код с отступом, следующий за двоеточием, называется *блоком*, или *телом*, менеджера контекста. В следующем примере блок состоит из операции записи текста `Ringo` в файл. После этого блок завершается. В приведенном примере этого не видно, но о завершении блока `with` можно судить по тому, что у кода пропадает отступ. В этот момент менеджер контекста вступает в действие и выполняет логику выхода. Логика выхода файлового менеджера контекста приказывает Python автоматически закрыть файл при завершении блока.

СОВЕТ

Используйте конструкцию `with` для чтения и записи файлов. Закрывать файлы — полезная привычка, и если вы используете команду `with` при работе с файлами, вам не придется беспокоиться об их закрытии. Команда `with` автоматически закрывает файл за вас.

19.7. Проектирование на основе файлов

Вы уже видели, как использовать функции для организации и структурирования сложных программ. Одно из преимуществ использования функций заключается в том, что функции могут повторно использоваться в коде. Приведем полезный совет по организации функций, работающих с файлами.

Предположим, вы хотите написать код, который получает имя файла и создает последовательность строк из файла, причем перед каждой строкой вставляется ее номер. На первый взгляд может показаться, что API (программный интерфейс, Application Programming Interface) ваших функций должен получать имя файла, содержимое которого вы хотите изменить:

```
>>> def add_numbers(filename):
...     results = []
...     with open(filename) as fin:
...         for num, line in enumerate(fin):
...             results.append(
...                 '{0}-{1}'.format(num, line))
...     return results
```

Такой код нормально работает. Но что произойдет, если потребуется вставить номера перед строками, взятыми не из файла, а из другого источника? Если потребуется протестировать код, теперь придется обращаться к файловой системе. Одно из решений — рефакторинг функции `add_numbers`, чтобы она только открывала файл в менеджере контекста, а затем вызывала другую функцию — `add_nums_to_seq`. Новая функция содержит логику, которая работает с последовательностью, а не зависит от имени файла. Так как файл ведет себя как последовательность строк, исходная функциональность будет сохранена:

```
>>> def add_numbers(filename):
...     with open(filename) as fin:
...         return add_nums_to_seq(fin)

>>> def add_nums_to_seq(seq):
...     results = []
...     for num, line in enumerate(seq):
...         results.append(
```

```
... '{0}-{1}'.format(num, line))  
...     return results
```

Теперь у вас имеется более общая функция `add_nums_to_seq`, более простая для тестирования и повторного использования, потому что вместо зависимости от имени файла она зависит от последовательности. Вы можете передать список строк или создать фиктивный файл, который будет передаваться функции.

ПОДСКАЗКА

Существуют и другие типы, которые тоже реализуют интерфейс, сходный с интерфейсом файлов (чтение и запись). В любой момент, когда вам приходится использовать имя файла при программировании, спросите себя, можно ли применить эту логику к другим видам последовательностей. Если это возможно, используйте приведенный выше пример рефакторинга для получения кода, который создает меньше проблем с повторным использованием и тестированием.

19.8. Итоги

Python предоставляет единую функцию `open` для взаимодействия как с текстовыми, так и с двоичными файлами. Указывая режим, вы сообщаете Python, какие операции должны выполняться с файлом — чтение или запись. При работе с текстовыми файлами читаются и записываются строки. При работе с двоичными файлами читаются и записываются байтовые строки.

Не забывайте закрывать файлы. Идиоматическим способом закрытия считается использование команды `with`. Наконец, следите за тем, чтобы ваши функции могли работать с последовательностями данных вместо имен файлов, так как это сделает ваш код более универсальным.

19.9. Упражнения

1. Напишите функцию для записи файлов данных, разделенных запятыми (CSV, Comma Separated Values). Функция должна получать в параметрах имя файла и список кортежей. Кортежи должны содержать имя, адрес и возраст. Файл должен создать строку

заголовка, за которой следует строка для каждого кортежа. Если функции будет передан следующий список кортежей:

```
[('George', '4312 Abbey Road', 22),  
( 'John', '54 Love Ave', 21)]
```

в файл должны быть записаны следующие данные:

```
name,address,age  
George,4312 Abbey Road,22  
John,54 Love Ave,21
```

2. Напишите функцию для чтения CSV-файлов. Она должна возвращать список словарей, интерпретируя первую строку как имена ключей, а каждую последующую строку как значения этих ключей. Для данных в приведенном примере будет возвращен следующий результат:

```
[{'name': 'George', 'address': '4312 Abbey Road', 'age': 22},  
{ 'name': 'John', 'address': '54 Love Ave', 'age': 21}]
```

20

Юникод

Строки уже неоднократно встречались вам в этой книге, но мы еще не обсуждали одно из самых больших изменений Python 3 — строки Юникода. В Python 2 строки Юникода поддерживались, но их нужно было специально создавать. Теперь ситуация изменилась, все строки хранятся в Юникоде.

20.1. Историческая справка

Что такое Юникод (Unicode)? Это стандарт представления глифов (символы, входящие в большинство письменных языков, а также знаки и эмодзи). Спецификацию стандарта можно найти на сайте Юникода¹, причем стандарт часто обновляется. Стандарт состоит из различных документов и диаграмм, связывающих *кодовые пункты* (шестнадцатеричные числа, такие как 0048 или 1F600) с глифами (Н или ☺) и именами (LATIN CAPITAL Н и GRINNING FACE). Кодовые пункты и имена уникальны, хотя многие глифы кажутся очень похожими.

А теперь небольшая историческая справка. С широким распространением компьютеров разные организации стали предлагать разные схемы для отображения двоичных данных на строковые данные. Одна из таких *кодировок* — ASCII — использует 7 битов данных для отображения на 128 знаков и управляющих символов. Такая кодировка нормально работает в средах, ориентированных на латинский алфавит, — напри-

¹ <https://unicode.org>

мер, в английском языке. Наличие 128 разных глифов предоставляет достаточно места для символов нижнего регистра, символов верхнего регистра, цифр и знаков препинания.

Со временем поддержка языков, отличных от английского, стала более распространенной, и кодировки ASCII оказались недостаточно. В семействе Windows до Windows 98 поддерживается кодировка Windows-1252 с поддержкой различных символов с диакритикой и знаков (например, знака евро).

Все эти схемы кодирования обеспечивают однозначное отображение байтов на символы. Для поддержки китайской, корейской и японской письменности потребуется много более 128 символов. Четырехбайтовая кодировка позволяет поддерживать свыше 4 миллиардов символов. Тем не менее такая универсальность не дается даром. Для большинства людей, использующих только символы из кодировки ASCII, четырехкратное увеличение затрат памяти при таком же объеме данных выглядит колоссальными потерями памяти.

Чтобы иметь возможность поддержки всех символов без лишних затрат памяти, пришлось пойти на компромисс — отказаться от кодирования символов последовательностью битов. Вместо этого символы были абстрагированы. Каждый символ отображался на уникальный *кодový пункт* (обладающий шестнадцатеричным значением и уникальным именем). Затем различные кодировки отображали кодовые пункты на битовые кодировки. Юникод устанавливает соответствие между символом и кодовым пунктом, а не конкретным представлением. В разных контекстах альтернативные представления могут обеспечивать более эффективные характеристики.

Одна из кодировок, UTF-32, использует 4 байта для хранения информации о символе. Такое представление удобно для низкоуровневых программистов благодаря тривиальным операциям индексирования. С другой стороны, она расходует вчетверо больше памяти, чем ASCII, для кодирования серии букв латинского алфавита.

Концепция кодировок переменной ширины также помогала бороться с затратами памяти. Одной из таких кодировок является UTF-8, в которой для представления символа используется от 1 до 4 байтов. Кроме того, UTF-8 обладает обратной совместимостью с ASCII. UTF-8 — самая

Кодовые таблицы Юникода

Глиф
 Кодовый пункт
 Имя

	1F60	1F61	1F62	1F63	1F64
0					
	1F600	1F610	1F620	1F630	1F640

Смайлики-эмоджи

Смайлики упорядочены по форме рта для упрощения поиска символов в кодовой таблице.

Лица

1F600		УЛЫБАЮЩЕЕСЯ ЛИЦО
1F601		ЛИЦО С УХМЫЛКОЙ
1F602		СМЕХ ДО СЛЕЗ
1F603		СМЕХ

→ 263A  довольный смайлик (светлая улыбка)

Код

'\N{GRINNING FACE}' **Имя**

'\u0001f600' **Кодовый пункт**

'' **Глиф**

b'\xf0\x9f\x98\x80', decode('utf8') **UTF-8**

Рис. 20.1. Чтение кодовых таблиц на сайте unicode.org. В таблицах перечислены глифы с соответствующими шестнадцатеричными кодовыми пунктами. За этой таблицей следует другая таблица с кодовым пунктом, глифом и названием. Вы можете использовать глиф, имя или кодовый пункт. Если кодовый пункт содержит более 4 цифр, используйте прописную букву *U* и дополните код слева нулями до 8 цифр. Если же цифр 4 и меньше, необходимо использовать строчную букву *u* и дополнять код слева нулями до 4 цифр. Также приведен пример декодирования байтовой строки UTF-8 в соответствующий глиф

распространенная кодировка в интернете — отлично подходит для кодирования символов, так как она поддерживается многими приложениями и операционными системами.

СОВЕТ

Чтобы узнать предпочтительную кодировку для вашей машины, выполните следующий код (приведен результат для моего 2015 Mac):

```
>>> import locale
>>> locale.getpreferredencoding(False)
'UTF-8'
```

Еще раз проясним: UTF-8 — кодировка байтов кодовых пунктов Юникода. Заявить, что UTF-8 и Юникод — одно и то же, в лучшем случае неточность, а в худшем — демонстрация непонимания способа кодирования символов. Более того, само название происходит от слов «Unicode Transformation Format — 8 bit», то есть «формат преобразования Юникода — 8-разрядный», то есть это формат для Юникода.

Рассмотрим пример. Символ с именем `SUPERSCRIPT TWO` определяется в стандарте Юникода как кодовый пункт `U+00b2`. Его глиф (печатное представление) имеет вид ². В ASCII этот символ представить невозможно. В Windows-1252 такое представление существует, символ кодируется байтом `b2` в шестнадцатеричной записи (это представление совпадает с кодовым пунктом, хотя такое совпадение не гарантируется). В UTF-8 ему соответствует кодировка `c2`. Аналогичным образом в UTF-16 используется кодировка `ffe b2 00`. Таким образом, у этого кодового пункта Юникода существует несколько разных кодировок.

20.2. Основные этапы в Python

В Python можно создавать строки Юникода. Собственно, вы делали это с первых страниц книги: напомним, что в Python 3 все строки кодируются в Юникоде. Но что делать, если вы хотите создать строку с символами, не входящими в ASCII? Ниже приведен код создания строки « x^2 ». Если вы найдете глиф, который должен использоваться в строке, скопируйте символ в код:

```
>>> result = 'x²'
```

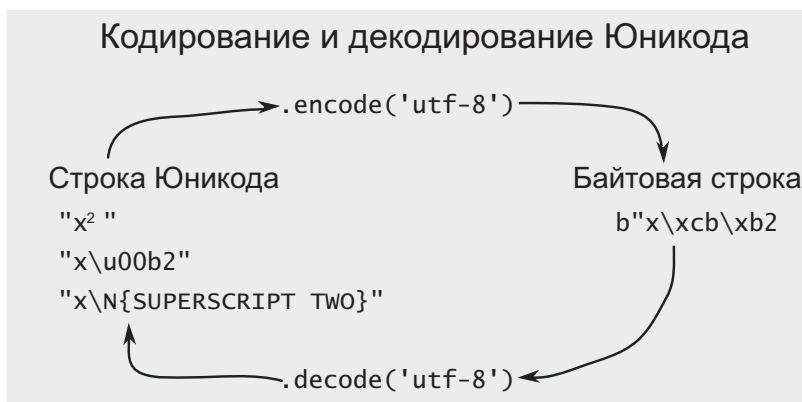


Рис. 20.2. Кодирование (в данном случае с использованием UTF-8) строки Юникода в байтовое представление и последующее декодирование той же байтовой строки в Юникод (тоже с использованием UTF-8). Также при декодировании следует предельно четко выражать свои намерения, потому что неточность приведет к появлению ошибочных данных или искажению символов

Такой способ обычно работает, хотя могут возникнуть проблемы, если ваш шрифт не поддерживает указанный глиф. Тогда вместо него будет отображаться так называемый «тофу» (пустой прямоугольник или ромб с вопросительным знаком). Также для включения символов, не входящих в ASCII, можно включить шестнадцатеричный кодовый пункт Юникода после префикса `\u`:

```
>>> result2 = 'x\u00b2'
```

Обратите внимание: эта строка тождественна предыдущей строке:

```
>>> result == result2
True
```

Наконец, можно использовать в строке имя кодового пункта, заключив его в фигурные скобки в конструкции `\N{}`:

```
>>> result3 = 'x\N{SUPERSCRIPT TWO}'
```

Все эти способы работают. И все они возвращают одну и ту же строку Юникода:

```
>>> print(result, result2, result3)
x² x² x²
```


Третий вариант получается менее компактным. Но если вы не помните нужный кодовый пункт или глиф не поддерживается шрифтом, пожалуй, этот вариант оказывается самым удобочитаемым.

СОВЕТ

В документации Python имеется раздел, посвященный Юникоду. Введите `help()`, а затем `UNICODE`. В этом разделе обсуждается не столько Юникод, сколько различные способы создания строк Python.

20.3. Кодирование

Пожалуй, один из ключей к пониманию Юникода в Python — понимание того, что *строка Юникода кодируется в байтовую строку*. Байтовые строки никогда не кодируются, но могут декодироваться в строку Юникода. Аналогичным образом *строки Юникода не декодируются*. Также на процессы кодирования и декодирования можно взглянуть под другим углом: кодирование преобразует понятное или осмысленное для человека представление в абстрактное представление, предназначенное для хранения (Юникод в байты или буквы в байты), а декодирование преобразует это абстрактное представление обратно в форму, удобную для человека.

Для заданной строки Юникода можно вызвать метод `.encode`, чтобы просмотреть ее представление в различных кодировках. Начнем с UTF-8:

```
>>> x_sq = 'x\u00b2'
>>> x_sq.encode('utf-8')
b'x\xc2\xb2'
```

Если Python не поддерживает кодировку, будет выдана ошибка `UnicodeEncodeError`. Это означает, что кодовые пункты Юникода не поддерживаются в этой кодировке. Например, ASCII не поддерживает символ возведения в квадрат:

```
>>> x_sq.encode('ascii')
Traceback (most recent call last):
...
UnicodeEncodeError: 'ascii' codec can't encode character
'\xb2' in position 1: ordinal not in range(128)
```

Если вы достаточно давно работаете с Python, вероятно, вы сталкивались с этой ошибкой. Она означает, что указанное кодирование не позволит представить все символы. Если вы уверены в том, что хотите заставить Python провести кодирование, то можете передать параметр `errors`. Чтобы проигнорировать символы, которые Python не может представить, передайте параметр `errors='ignore'`:

```
>>> x_sq.encode('ascii', errors='ignore')
b'x'
```

Если передать параметр `errors='replace'`, Python вставит вопросительные знаки вместо неподдерживаемых байтов:

```
>>> x_sq.encode('ascii', errors='replace')
b'x?'
```

ПРИМЕЧАНИЕ

Модуль `encodings` содержит отображения для кодировок, поддерживаемых Python. В Python 3.6 поддерживаются 99 кодировок. Многие современные приложения стараются ограничиваться UTF-8, но если вам нужна поддержка других кодировок, вполне возможно, что они поддерживаются в Python.

Таблица кодировок находится в `encodings.aliases.aliases`. Кроме того, таблица кодировок приведена в документации модуля на сайте Python¹.

Несколько возможных вариантов кодирования для этой строки:

```
>>> x_sq.encode('cp1026') # Турецкий
b'\xa7\xea'
>>> x_sq.encode('cp949') # Корейский
b'x\xa9\xf7'
>>> x_sq.encode('shift_jis_2004') # Японский
b'x\x85K'
```

Хотя Python поддерживает много кодировок, они все реже используются на практике. Как правило, они встречаются только в унаследованных приложениях. В наши дни большинство приложений использует UTF-8.

¹ <https://docs.python.org/3/library/codecs.html>

20.4. Декодирование

Термин «декодирование» имеет особый смысл в Python. Он означает получение последовательности байтов и создание строки Юникода. В Python байты никогда не кодируются, а только декодируются (собственно, метода `.encode` для байтов не существует). Если вы запомните это правило, оно избавит вас от многих проблем при работе с символами, не входящими в ASCII. Предположим, у вас имеется последовательность байтов UTF-8 для `x²`:

```
>>> utf8_bytes = b'x\xc2\xb2'
```

Работая с символьными данными, представленными в виде байтов, постарайтесь как можно быстрее преобразовать их в строку Юникода. Обычно в приложении вам придется иметь дело только со строками, а байты использовать только как механизм сериализации (то есть при сохранении файлов или их передаче по сети). Если вы получите эти байты из какого-то источника и используемый фреймворк или библиотека не преобразует их в строку, можно поступить так:

```
>>> text = utf8_bytes.decode('utf-8')
>>> text
'x²'
```

Другой момент, на который следует обратить внимание, — вы не сможете угадать, в каком виде была закодирована последовательность байтов. Вы знаете, что некая последовательность в коде закодирована в UTF-8, потому что вы только что сами ее создали; но если неопознанные байты были получены вами из другого источника, они вполне могут иметь другую кодировку.

ПРИМЕЧАНИЕ

Вы должны знать кодировку символов; в противном случае вы можете попытаться провести декодирование с неверной кодировкой и получить неправильные данные. Такие неверно декодированные строки называются «модзибакэ» (японское слово, означающее «искажение символов», — но «модзибакэ» звучит круче).

Несколько примеров ошибочного декодирования:

```
>>> b'x\xc2\xb2'.decode('cp1026') # Турецкая
'İBŞ'

>>> b'x\xc2\xb2'.decode('cp949') # Корейская
'x뽕'

>>> b'x\xc2\xb2'.decode('shift_jis_2004') # Японская
'x ヲ イ'
```

Некоторые кодировки поддерживают не все последовательности байтов. Если вы проведете декодирование с ошибочной кодировкой, то вместо искаженных символов вы получите исключение. Например, ASCII не поддерживает следующую последовательность байтов:

```
>>> b'x\xc2\xb2'.decode('ascii')
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc2
in position 1: ordinal not in range(128)
```

Ошибка `UnicodeDecodeError` означает, что вы пытались преобразовать байтовую строку в строку Юникода и кодировка не позволила создать строки Юникода для всех байтов. Как правило, это указывает на использование неправильной кодировки. Для получения оптимальных результатов попробуйте определить и использовать правильную кодировку.

Если кодировка неизвестна, можно передать параметр `errors='ignore'`, но это приведет к потере данных. Эта мера должна использоваться только в самом крайнем случае:

```
>>> b'x\xc2\xb2'.decode('ascii', errors='ignore')
'x'
```

20.5. Юникод и файлы

При чтении текстового файла Python возвращает строки Юникода. Python использует кодировку по умолчанию (`locale.getpreferredencoding(False)`). Если вы хотите закодировать текстовый файл в другой кодировке, передайте эту информацию с параметром `encoding` функции `open`.

Кроме того, кодировку можно задать при открытии файла для записи. Не забывайте, что кодировки следует рассматривать как формат сериализации (используемый для передачи информации по интернету или сохранения данных в файле). Ниже приведены два примера записи в файл. С первым файлом кодировка не определяется, поэтому по умолчанию используется UTF-8 (кодировка по умолчанию для системы). Во втором примере в параметре `encoding` назначается кодировка CP949 (корейская):

```
>>> with open('/tmp/sq.utf8', 'w') as fout:
...     fout.write('x²')

>>> with open('/tmp/sq.cp949', 'w', encoding='cp949') as fout:
...     fout.write('x²')
```

Просмотрите файлы в терминале UNIX. Вы увидите, что они имеют разное содержимое, потому что используют разные кодировки:

```
$ hexdump /tmp/sq.utf8
00000000 78 c2 b2
00000003

$ hexdump /tmp/sq.cp949
00000000 78 a9 f7
00000003
```

Данные можно прочесть из файлов:

```
>>> data = open('/tmp/sq.utf8').read()
>>> data
'x²'
```

Помните, что в Python лучше *явно выразить намерения, чем подразумевать*. При работе с кодировками необходима конкретность. При чтении файла Python в системе с UTF-8 попытается декодировать из UTF-8. Это нормально, если файл кодировался в UTF-8, но если нет — вы получите либо модзибакэ, либо ошибку:

```
>>> data = open('/tmp/sq.cp949').read()
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf-8' codec can't decode byte
0xa9 in position 1: invalid start byte
```

Если вы явно выразите свои намерения и сообщите Python, что файл был закодирован в CP949, то получите из файла правильные данные:

```
>>> data = open('/tmp/sq.cp949', encoding='cp949').read()
>>> data
'x²'
```

Если вы работаете с текстовыми файлами, содержащими символы, не входящие в ASCII, обязательно явно укажите их кодировку.

20.6. Итоги

Юникод определяет соответствие между кодовыми пунктами и глифами. В языке Python строки могут содержать глифы Юникода. Строку Юникода можно закодировать в байтовую строку с применением разных кодировок. К строкам Python никогда не применяется декодирование.

При чтении текстового файла можно задать кодировку, чтобы полученная строка Юникода содержала правильные символы. При записи в текстовый файл можно задать параметр `encoding`, чтобы явно задать используемую кодировку.

UTF-8 — самая популярная из всех современных кодировок. Если только у вас нет веских причин для использования другой кодировки, используйте принятую по умолчанию кодировку UTF-8.

20.7. Упражнения

1. Посетите сайт <http://unicode.org> и загрузите таблицу с кодовыми пунктами. Выберите символ, не входящий в набор символов ASCII, и напишите код Python для вывода этого символа как по кодовому пункту, так и по имени.
2. Существуют различные символы Юникода, которые выглядят как перевернутые версии ASCII-символов. Найдите таблицу отображения этих символов (это будет несложно). Напишите функцию, которая получает строку ASCII-символов и возвращает перевернутую версию этой строки.

3. Запишите в файл свое имя, записанное перевернутыми символами. Приведите примеры кодировок, которые поддерживали бы такую запись вашего имени. Приведите примеры кодировок, которые такую запись не поддерживают.
4. Умные (или закругленные) кавычки не поддерживаются в ASCII. Напишите функцию, которая получает строку ASCII и возвращает строку, в которой двойные кавычки заменяются умными кавычками. Например, строка *Python comes with "batteries included"* должна превратиться в *Python comes with “batteries included”* (если присмотреться повнимательнее, вы увидите, что открывающие кавычки закруглены не так, как закрывающие).
5. Напишите функцию, которая получает текст со смайликами в старом стиле (:), :P и т. д.) Используя таблицу эмодзи¹, добавьте в свою функцию код для замены текстовых смайликов их Юникод-версиями из таблицы.

¹ <http://unicode.org/emoji/charts/full-emoji-list.html>

21

Классы

Строки, словари, файлы и целые числа — все это объекты. Даже функции представляют собой объекты. В Python почти все является объектом. Есть и исключения: ключевые слова (например, `in`) объектами не являются. Кроме того, имена переменных объектами не являются, но указывают на них. В этой главе мы углубимся в тему и разберемся, что же собой представляет объект.

Термин «объект» многозначен. Одно из определений объектно-ориентированного программирования подразумевает использование структур для группировки данных (состояния) и методов (функций для изменения состояния). Многие объектно-ориентированные языки, такие как C++, Java и Python, используют *классы* для определения состояния, которое может храниться в объекте, и методов для изменения этого состояния. Если классы являются определениями состояния и методов, *экземпляры* являются воплощениями этих классов. Как правило, когда разработчик говорит об объектах, он имеет в виду экземпляры классов.

В языке Python `str` — имя класса, используемого для хранения строк. Класс `str` определяет методы строк.

Чтобы создать экземпляр класса `str` с именем `b`, используйте синтаксис строковых литералов Python:

```
>>> b = "I'm a string"
>>> b
"I'm a string"
```


Если программисты начнут обсуждать `b`, вы можете услышать самые разные термины. «`b` — это строка», «`b` — это объект», «`b` — это экземпляр строки»... Пожалуй, последняя формулировка самая конкретная. Однако `b` при этом не является классом строки.

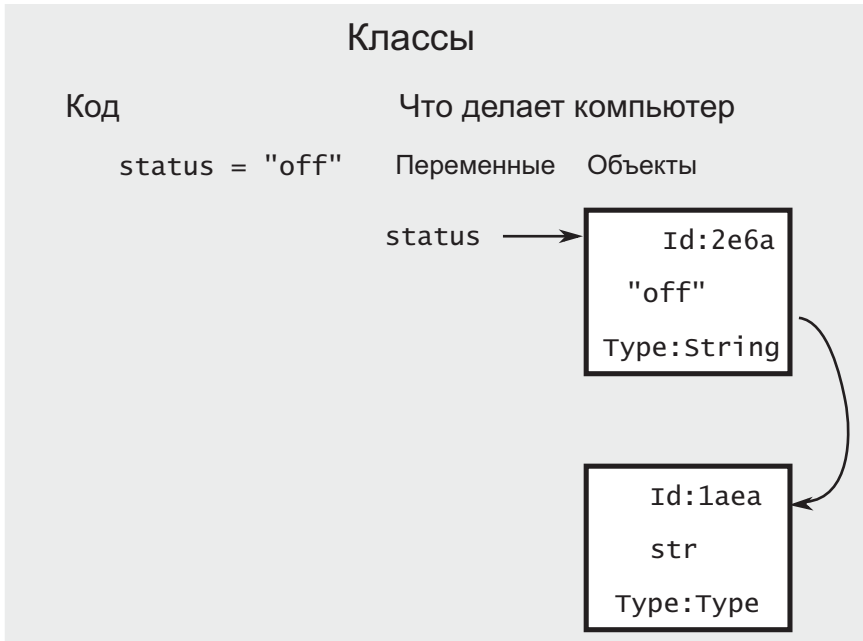


Рис. 21.1. Объект строки. У каждого объекта имеется тип, которым в действительности является класс объекта. В данном случае это класс `str`

ПРИМЕЧАНИЕ

Класс `str` также может использоваться для создания строк, но чаще используется для преобразования типа. Строки встроены в язык, поэтому передавать строковый литерал классу `str` было бы лишним. Иначе говоря, не нужно использовать запись

```
>>> c = str("I'm a string"),
```

так как Python автоматически создает строку при заключении символов в кавычки. Термин «*литерал*» в данном случае означает специальный синтаксис создания строк, встроенный в Python.

С другой стороны, если у вас имеется число, которое нужно преобразовать в строку, можно вызвать `str`:

```
>>> num = 42
>>> answer = str(num)
>>> answer
'42'
```

Говорят, что Python «поставляется с батарейками» — вместе с библиотеками и классами, заранее определенными для разработчика. Эти классы обычно получают достаточно универсальными. Вы можете определять собственные классы, специализированные для вашей предметной области, специализированные объекты с состоянием и логику для изменения этого состояния.

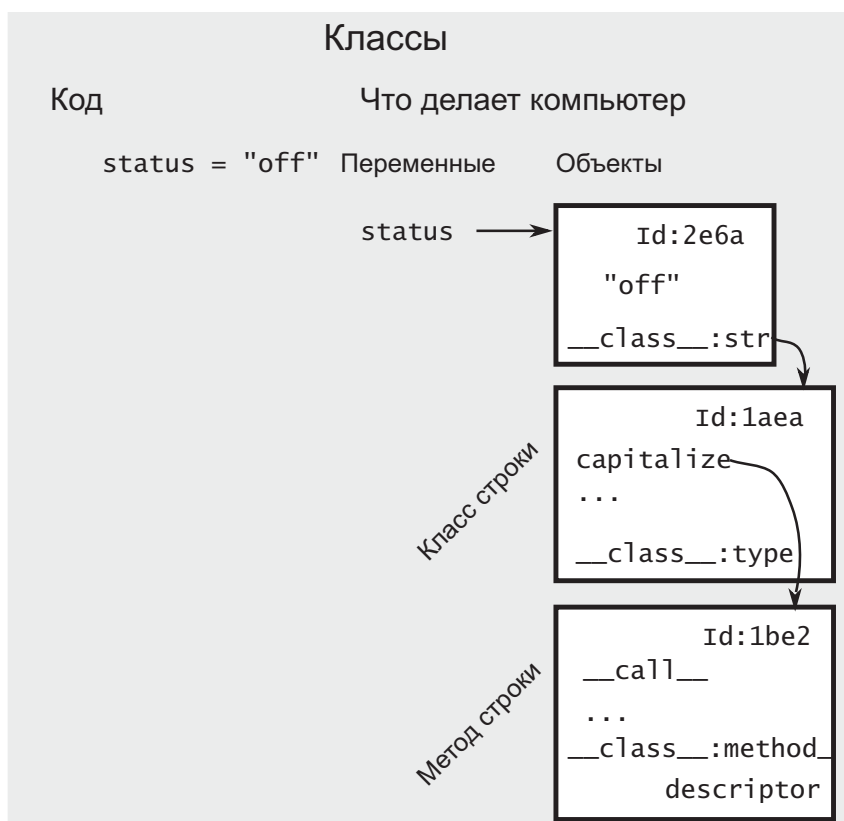


Рис. 21.2. Обновленная версия объекта строки. Тип изменился на `__class__`, потому что при анализе объекта атрибут `__class__` указывает на класс объекта. Этот класс содержит разные методы (на схеме показан только метод `capitalize`, но есть много других). Методы тоже являются объектами, как видно из диаграммы

21.1. Планирование класса

Прежде всего заметим, что классы не всегда необходимы в Python. Подумайте, нужно ли определять класс или же хватит функции (или группы функций). Класс может быть полезен для программного представления физических или концептуальных объектов. Понятия, являющиеся описаниями, — скорость, температура, среднее арифметическое, цвет — не являются хорошими кандидатами для классов.

Если вы решили, что хотите моделировать что-либо с помощью класса, задайте себе следующие вопросы:

- ☐ У него есть имя?
- ☐ Какими свойствами он обладает?
- ☐ Присущи ли эти свойства всем экземплярам класса? А именно:
 - Какие свойства являются общими для класса в целом?
 - Какие из этих свойств уникальны для каждого экземпляра?
- ☐ Какие операции он выполняет?

Рассмотрим конкретный пример. Предположим, вы работаете на горнолыжном курорте и хотите моделировать использование подъемников. Один из вариантов основан на создании класса, определяющего кресло на подъемник. Подъемник (если вдруг вы никогда им не пользовались) оснащается множеством кресел. Лыжники встают в очередь у основания горы, чтобы сесть на кресла, а затем сходят с них на вершине.

Разумеется, модель необходимо до определенного уровня абстрагировать. Не нужно моделировать каждое низкоуровневое свойство кресла. Например, для расчета нагрузки неважно, из какого материала сделано кресло — из алюминия, стали или дерева. С другой стороны, это может быть важно для некоторых лыжников.

Есть ли имя у моделируемой сущности? Да, «кресло». Среди свойств кресла можно выделить номер, вместимость, наличие планки безопасности и мягких сидений. Если погрузиться немного глубже, вместимость можно разбить на максимальную вместимость и текущую занятость. Максимальная емкость должна оставаться постоянной, тогда как занятость может изменяться в любой момент времени.

С креслом связан ряд операций — например, добавление людей у подножия горы и выход на вершине. Другим действием может стать позиция планки безопасности. Мы будем обрабатывать загрузку и выгрузку посетителей, но в нашей модели позиция планки безопасности будет игнорироваться.

21.2. Определение класса

Ниже приведен класс Python, представляющий кресло на подъемнике. Сначала рассмотрим простое определение класса; комментарии пронумерованы для последующего обсуждения:

```
>>> class Chair:                                # 1
...     ''' A Chair on a chairlift '''          # 2
...     max_occupants = 4                       # 3
...
...     def __init__(self, id):                  # 4
...         self.id = id                        # 5
...         self.count = 0
...
...     def load(self, number):                  # 6
...         self.count += number
...
...     def unload(self, number):                # 7
...         self.count -= number
```

Ключевое слово `class` в Python определяет класс. Классу должно быть присвоено имя (1), за которым следует двоеточие. Вспомните, что в Python за двоеточием следует блок с отступом (кроме использования в срезах). Также обратите внимание на последовательные отступы под определением класса и на то, что имя класса начинается с прописной буквы.

ПРИМЕЧАНИЕ

Имена классов записываются в «верблюжьем регистре». Так как `Chair` — одно слово, вы могли этого и не заметить. В отличие от функций, в именах которых слова соединяются символами подчеркивания, в «верблюжьем регистре» каждое слово начинается с прописной буквы, а сами слова просто соединяются без разделителей. Обычно в качестве имен классов используются имена

существительные. В Python имена классов не могут начинаться с цифр. Ниже приведены примеры имен классов — как хороших, так и плохих.

- Kitten # хорошо
- jaguar # плохо - начинается со строчной
- SnowLeopard # хорошо - "верблюжий регистр"
- White_Tiger # плохо - содержит подчеркивания
- 9Lives # плохо - начинается с цифры

За дополнительной информацией о выборе имен классов обращайтесь к PEP 8.

Следует заметить, что многие встроенные типы не соблюдают это правило: `str`, `int`, `float` и т. д.

Непосредственно после объявления класса можно вставить строку документации (2). Это самая обычная строка. Обратите внимание: если эта строка заключена в тройные кавычки, она может состоять из нескольких абзацев. Строки документации не обязательны, но они могут пригодиться читателям вашего кода; кроме того, они выводятся функцией `help` при анализе кода в REPL. Используйте строки документации осмотрительно, и она принесет огромную пользу.

Внутри тела класса, снабженного отступами, можно создать *атрибуты класса* (3). Атрибут класса используется для хранения состояния, общего для всех экземпляров класса. В нашем примере любое кресло, которое мы будем создавать, может вмещать до четырех посетителей. У атрибутов классов есть свои преимущества. Так как значение задается на уровне класса, вам не придется повторяться и задавать его при создании каждого нового кресла. С другой стороны, ваши кресла будут жестко запрограммированы так, чтобы они поддерживали только четыре места. Позднее вы узнаете, как переопределить атрибут класса.

Затем следует команда `def` (4). Все выглядит так, словно мы определяем функцию внутри тела класса. Все верно, если не считать того, что функция, определяемая прямо в теле класса, называется *методом*. Так как этот метод имеет специальное имя `__init__`, он называется *конструктором*. Метод получает два параметра, `self` и `id`. У большинства методов первым параметром является `self`. Его можно интерпретировать как экземпляр класса.

Конструктор вызывается при создании экземпляра класса. Если рассматривать класс как «фабрику», которая предоставляет шаблон или чертеж для создания экземпляров, то конструктор инициализирует состояние этих экземпляров. Конструктор получает экземпляр во входных данных (параметр `self`) и обновляет его внутри метода. Python помогает вам и передает экземпляр автоматически. Впрочем, это может породить путаницу, но об этом позднее.

Внутри тела конструктора (5) (оно снабжено отступом, потому что следует за двоеточием) присоединяются два атрибута, которые будут уникальными для экземпляра: `id` и `count`. На большинстве подъемников каждое кресло помечается уникальным номером. Атрибут `id` представляет число. Кроме того, на одном кресле могут ехать несколько лыжников — их количество хранится в атрибуте `count` и инициализируется нулем. Обратите внимание: конструктор ничего не возвращает, но обновляет значения, уникальные для экземпляра.

ПРИМЕЧАНИЕ

В Python есть встроенная функция `id`, но вы также можете использовать это имя как имя атрибута класса. Функция `id` при этом остается доступной. Каждый раз, когда вы хотите обратиться к атрибуту `id`, проводится поиск по экземпляру. Если экземпляру было присвоено имя `chair`, то для получения значения `id` следует использовать запись `chair.id`. Таким образом, встроенная функция не замещается.

О завершении логики конструктора можно судить по исчезновению уровня отступа. Мы видим определение другого метода (6), `load`. Этот метод представляет операцию, которая может выполняться экземпляром класса. В данном случае кресло может загружать пассажиров, а этот метод сообщает экземпляру, что нужно делать в подобных ситуациях. И снова `self` (экземпляр) является первым параметром метода. Второй параметр, `number`, содержит количество людей, садящихся на кресло. Помните, что кресло на подъемнике обычно вмещает несколько (до четырех в нашем примере) человек. Когда лыжник садится на кресло, нужно вызвать метод `load` для кресла, а внутри тела этого метода обновить атрибут `count` экземпляра.

Также существует парный метод `unload` (7), который должен вызываться при спуске лыжника с подъемника на вершине горы.

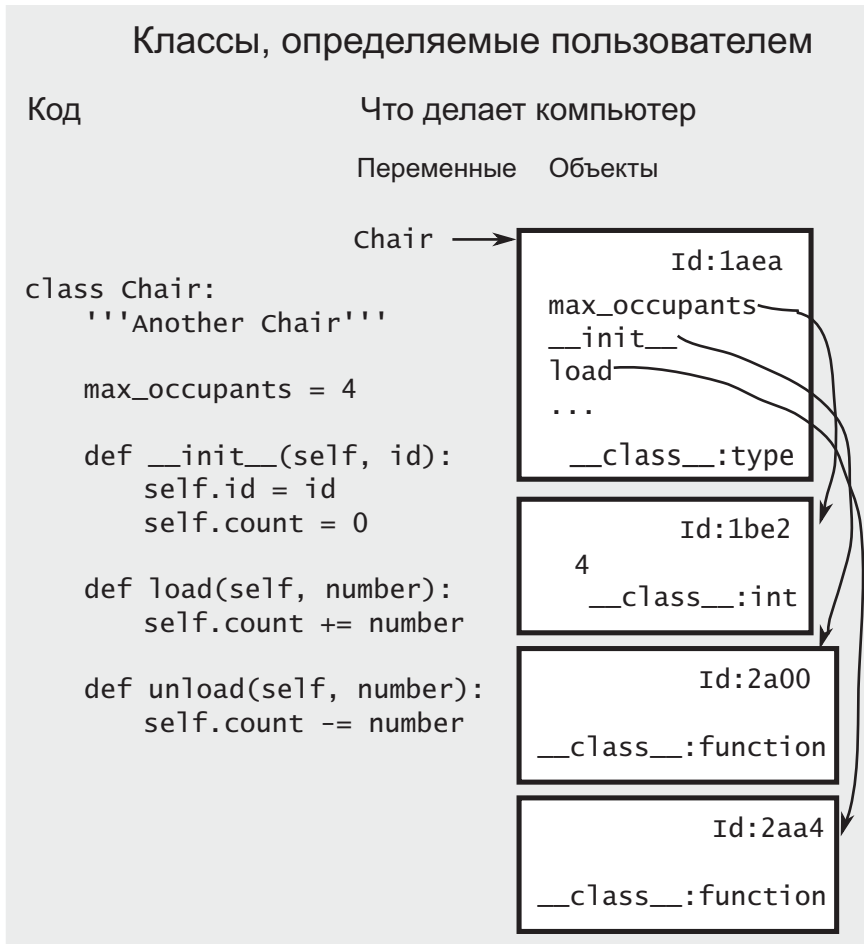


Рис. 21.3. Создание экземпляра класса. Python создает за вас новый тип. Все атрибуты класса или методы будут храниться в атрибутах нового класса. Атрибуты экземпляров (`id` и `count`) в классе отсутствуют, потому что они определяются для экземпляров

ПРИМЕЧАНИЕ

Не бойтесь методов. Вы уже видели многие методы — например, метод `.capitalize`, определенный для строки. Методы представляют собой функции, присоединенные к классу. Метод вызывается не сам по себе, а для конкретного экземпляра класса:

```
>>> 'matt'.capitalize()
'Matt'
```

Таким образом, создать класс несложно. Сначала вы определяете нужные атрибуты. Те атрибуты, которые существуют на уровне класса, размещаются в определении класса. Атрибуты, уникальные для каждого экземпляра, размещаются в конструкторе. Вы также можете определить методы с операциями, изменяющими экземпляр класса. После того как класс будет определен, Python создает переменную с именем класса, которая указывает на класс:

```
>>> Chair
<class '__main__.Chair'>
```

Для получения расширенной информации о классе можно воспользоваться функцией `dir`. Обратите внимание: атрибуты класса определяются в классе, и к ним можно обращаться с указанием имени класса:

```
>>> dir(Chair)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'load', 'max_occupants', 'unload']

>>> Chair.max_occupants
4
```

На диаграмме в этом разделе показано, как хранятся атрибуты и методы класса. Также список атрибутов и методов можно просмотреть из REPL. Так как в Python нет ничего, кроме объектов, все они обладают атрибутом `__class__`:

```
>>> Chair.__class__
<class 'type'>
>>> Chair.max_occupants.__class__
<class 'int'>
>>> Chair.__init__.__class__
<class 'function'>
>>> Chair.load.__class__
<class 'function'>
>>> Chair.unload.__class__
<class 'function'>
```


Методы также определяются на уровне класса, но атрибуты экземпляров — нет. Так как атрибуты экземпляров уникальны для объекта, они хранятся в экземпляре.

Если в вашем классе или его методах определены строки документации, вы можете просмотреть их при помощи `help`:

```
>>> help(Chair)
Help on class Chair in module __main__:

class Chair(builtins.object)
|   A Chair on a chairlift
|
|   Methods defined here:
|
|   __init__(self, id)
|
|   load(self, number)
|
|   unload(self, number)
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
|
|   -----
|   Data and other attributes defined here:
|
|   max_occupants = 4
```

21.3. Создание экземпляра класса

Итак, вы определили класс, моделирующий кресло на подъемнике, и можете создавать *экземпляры* этого класса. Класс `Chair` можно сравнить с фабрикой: он берет заготовки объектов и превращает их в объекты кресел.

Если говорить конкретно, то эта задача решается методом конструктора `__init__`. В первом параметре передается минимальный объект `self`, то есть «заготовка». Python назначает объекту атрибут `__class__` (указывающий на класс `Chair`) перед тем, как передавать его конструктору.

Возможно, другая аналогия поможет вам лучше понять, как работают классы: вспомните, как в мультфильмах изображают рождение детей. Нерожденные дети обитают где-то в облаках. В один прекрасный момент прилетает аист, забирает ребенка с облака и доставляет его в колыбель в родительском доме. При вызове конструктора Python забирает ребенка с облака (получает объект). Он доставляет ребенка в дом, делает его членом семьи (присваивая атрибуту `__class__` значение `Chair` или другое значение, соответствующее вашему классу). Когда ребенок окажется в доме, его можно будет одеть, помыть и т. д. Помните, что объекты хранят состояние, которое может изменяться при помощи операций.

Ниже приведен код создания кресла с номером 21 на языке Python. При вызове класса (то есть указании имени класса с круглыми скобками) вы сообщаете Python о необходимости вызова конструктора. В отличие от некоторых языков, в Python не нужно использовать ключевое слово `new` или указывать тип; достаточно поставить круглые скобки с параметрами конструктора после имени класса:

```
>>> chair = Chair(21)
```

Еще раз уточним терминологию: переменная `chair` указывает на объект, то есть экземпляр. Она не указывает на класс. Объект относится к классу `Chair`. Экземпляр содержит ряд атрибутов, включая `count` и `id`.

Чтобы обратиться к атрибуту экземпляра, следует указать его экземпляр (`chair`):

```
>>> chair.count
0
```

```
>>> chair.id
21
```

В Python используется определенная иерархия поиска атрибутов. Сначала Python ищет атрибут в экземпляре. Если поиск оказывается безуспешным, Python переходит к поиску атрибута в классе (так как экземпляры знают, к какому классу они принадлежат). Если и на этот раз поиск

завершится неудачей, Python выдает ошибку `AttributeError` (атрибут отсутствует). Атрибут `max_occupants` обычно хранится в классе, но к нему также можно обратиться через экземпляр:

```
>>> chair.max_occupants
4
```

Во внутренней реализации Python заменяет это обращение следующим:

```
>>> chair.__class__.max_occupants
4
```

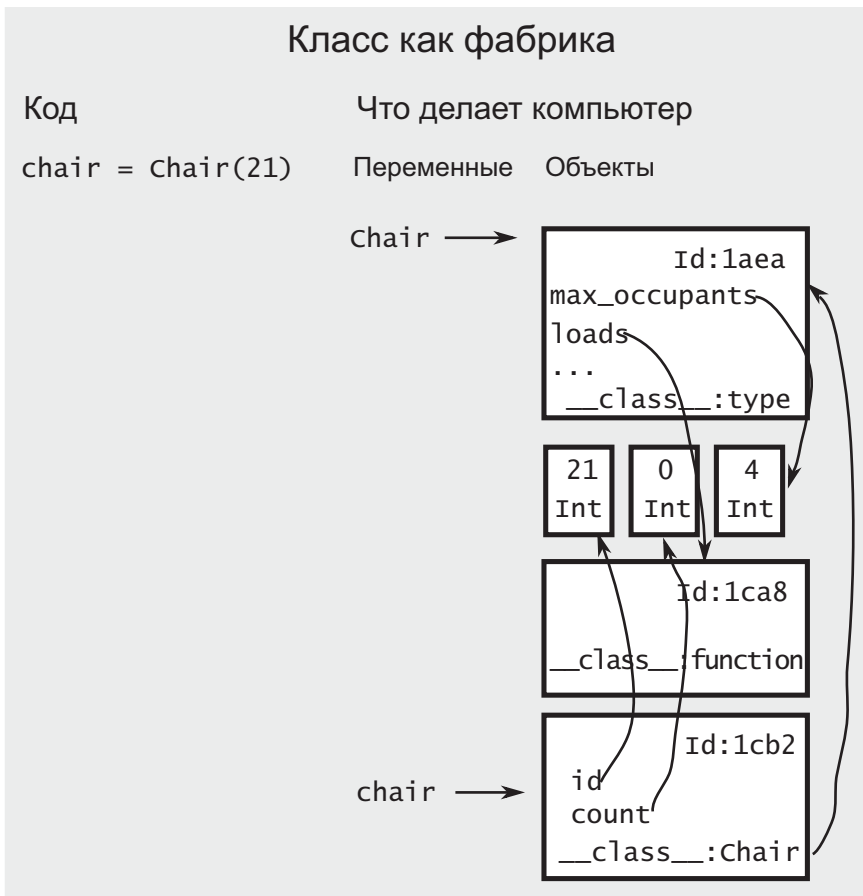


Рис. 21.4. Процесс построения объекта. При вызове конструктора «аист» — Python приносит конструктору «ребенка» — объект (`self`). У этого объекта установлен атрибут `__class__`, но конструктор может свободно изменять экземпляр, добавляя новые атрибуты. Объект превращается в `chair`

Поиск атрибутов отличается от поиска переменных. Вспомните, что Python начинает поиск переменных с локальной области видимости, затем переходит к глобальной области видимости, затем к встроенной — и в итоге выдает ошибку `NameError`, если поиск не дал результатов. Поиск атрибутов начинается с экземпляра, затем переходит к классу, а если атрибут не найден, выдается ошибка `AttributeError`.

21.4. Вызов метода

Если у вас имеется экземпляр класса, для него можно вызвать методы. Методы, как и функции, вызываются с круглыми скобками, в которых перечисляются аргументы. В следующем примере вызывается метод `.load`, добавляющий трех лыжников на кресло:

```
>>> chair.load(3)
```

Кратко разберем синтаксис вызова метода. Сначала указывается экземпляр (`chair`), за которым следует точка. Точка в Python обозначает поиск атрибута (если только она не следует за числовым литералом). Когда вы видите, что за экземпляром следует точка, помните, что Python будет искать то, что идет после точки.

Сначала Python ищет `load` в экземпляре. В экземпляре этот атрибут найти не удастся (вспомните, что в конструкторе для экземпляра были назначены только атрибуты `count` и `id`). Однако экземпляр также содержит ссылку на свой класс. Так как поиск в экземпляре завершился неудачей, Python переходит к поиску этих атрибутов в классе. Метод `.load` определен для класса `Chair`, поэтому Python возвращает его. Круглые скобки обозначают вызов метода, а число 3 передается в параметре метода.

Вспомните, как выглядело объявление `load`:

```
... def load(self, number):           # 6
...     self.count += number
```

В объявлении указаны два параметра, `self` и `number`, а при вызове передается только один параметр 3. Почему количество параметров не совпадает? Параметр `self` представляет экземпляр (`chair` в данном случае). Python вызывает метод `.load`, передавая `chair` в параметре `self` и 3 в па-

парамetre `number`. Фактически Python берет на себя все хлопоты, связанные с параметром `self`, и передает его автоматически.

ПРИМЕЧАНИЕ

Когда вы используете вызов вида

```
chair.load(3),
```

во внутренней реализации используется вызов следующего вида:

```
Chair.load(chair, 3).
```

Вы можете опробовать этот способ и убедиться, что он работает, но поступать так на практике не рекомендуется, потому что такой код хуже читается и занимает больше места.

21.5. Анализ экземпляра

Если у вас имеется экземпляр и вы хотите узнать его атрибуты, есть несколько вариантов. Информацию можно посмотреть в документации (если она существует). Можно прочитать код, в котором определяется класс. Наконец, можно воспользоваться функцией `dir`, которая проанализирует его за вас:

```
>>> dir(chair)
['_class_', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
 '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', 'count', 'id', 'load',
 'max_occupants', 'unload']
```

Напомним, что функция `dir` перечисляет атрибуты объекта. Обратившись к документации `dir`, вы увидите, что приведенное выше определение `dir` не совсем правильно. В документации говорится:

«...возвращает упорядоченный по алфавиту список имен, включающий (некоторые) атрибуты заданного объекта, а также атрибуты, доступные из него».

`help(dir)` (наше выделение)

Функция выводит атрибуты, доступные из объекта. Фактическое состояние экземпляра хранится в атрибуте `__dict__` — словаре, связывающем имена атрибутов со значениями:

```
>>> chair.__dict__
{'count': 3, 'id': 21}
```

Итак, в экземпляре на самом деле хранятся только атрибуты `count` и `id`, а другие атрибуты доступны через класс. Где же хранится класс?

В атрибуте `__class__`:

```
>>> chair.__class__
<class '__main__.Chair'>
```

Важно, чтобы экземпляр знал свой класс, потому что в классе хранятся методы и атрибуты класса.

21.6. Приватный и защищенный доступ

В некоторых языках существует концепция приватных атрибутов и методов. Предполагается, что эти методы являются подробностями реализации и не должны вызываться конечными пользователями. Более того, язык программирования может блокировать доступ к ним.

Python не пытается в чем-то препятствовать пользователям. Предполагается, что вы взрослый человек и способны отвечать за свои действия. Если вы хотите получить к чему-либо доступ, вы сможете это сделать. Тем не менее будьте готовы принять последствия.

Программисты Python понимают, что в объекте может быть удобно хранить состояние и методы, являющиеся подробностями реализации. Чтобы конечный пользователь понимал, что к этим компонентам не стоит обращаться, их имена начинаются с символа подчеркивания. Ниже приведен класс со вспомогательным методом `._check`, который не предназначен для вызова всеми желающими:

```
>>> class CorrectChair:
...     ''' A Chair on a chairlift '''
...     max_occupants = 4
```

```
...
...     def __init__(self, id):
...         self.id = id
...         self.count = 0
...
...     def load(self, number):
...         new_val = self._check(self.count + number)
...         self.count = new_val
...
...     def unload(self, number):
...         new_val = self._check(self.count - number)
...         self.count = new_val
...
...     def _check(self, number):
...         if number < 0 or number > self.max_occupants:
...             raise ValueError('Invalid count:{}'.format(
...                 number))
...         return number
```

Метод `._check` считается приватным — к нему должны обращаться только экземпляры. Приватные методы вызываются методами `.load` и `.unload` класса. При желании вы сможете вызвать их за пределами класса. Тем не менее делать этого не следует — все компоненты с символом подчеркивания считаются подробностями реализации, которые могут отсутствовать в будущих версиях класса.

21.7. Простая программа, моделирующая поток посетителей

Воспользуемся классом для моделирования потока лыжников на горнолыжном курорте. Мы сделаем ряд базовых допущений — например, что на каждом кресле могут с равной вероятностью ехать от 0 до `max_occupants` лыжников. Класс включает подъемник, загружает его и работает в бесконечном цикле. Четыре раза в секунду выводится текущая статистика:

```
import random
import time

class CorrectChair:
    ''' A Chair on a chairlift '''
    max_occupants = 4
```

```
def __init__(self, id):
    self.id = id
    self.count = 0

def load(self, number):
    new_val = self._check(self.count + number)
    self.count = new_val

def unload(self, number):
    new_val = self._check(self.count - number)
    self.count = new_val

def _check(self, number):
    if number < 0 or number > self.max_occupants:
        raise ValueError('Invalid count:{}'.format(
            number))
    return number

NUM_CHAIRS = 100

chairs = []
for num in range(1, NUM_CHAIRS + 1):
    chairs.append(CorrectChair(num))

def avg(chairs):
    total = 0
    for c in chairs:
        total += c.count
    return total/len(chairs)

in_use = []
transported = 0
while True:
    # загрузка
    loading = chairs.pop(0)
    in_use.append(loading)
    in_use[-1].load(random.randint(0, CorrectChair.max_occupants))

    # выгрузка
    if len(in_use) > NUM_CHAIRS / 2:
        unloading = in_use.pop(0)
        transported += unloading.count
        unloading.unload(unloading.count)
        chairs.append(unloading)
```



```
print('Loading Chair {} Count:{} Avg:{:.2} Total:{}'.format
      (loading.id, loading.count, avg(in_use), transported))
time.sleep(.25)
```

Эта программа будет бесконечно выводить количество лыжников на подъемнике. Данные выводятся на терминал, но функция `print` может быть заменена кодом вывода данных в CSV-файл.

Изменив всего два числа (глобальное значение `NUM_CHAIRS` и атрибут класса `CorrectChair.max_occupants`), вы сможете изменить поведение модели для моделирования большего или меньшего подъемника. Вызов `random.randint` можно заменить функцией, которая более точно представляет распределение нагрузки.

21.8. Итоги

В этой главе классы были рассмотрены более подробно. Мы обсудили терминологию, связанную с классами. Можно говорить «объект» или «экземпляр»; эти термины можно считать синонимами. Каждый объект связан с некоторым классом. Класс — своего рода фабрика, определяющая поведение объектов/экземпляров.

Объект создается специальным методом, который называется конструктором. Этому методу присваивается имя `__init__`. Вы также можете определять собственные методы классов.

При создании класса необходимо как следует поразмыслить. Какими атрибутами должен обладать класс? Если атрибут остается постоянным для всех объектов, определите его в классе. Если атрибут уникален для объекта, задайте его в конструкторе.

21.9. Упражнения

1. Представьте, что вы проектируете приложение для банка. Как должна выглядеть модель клиента? Какими атрибутами она должна обладать? Какие методы она должна поддерживать?
2. Представьте, что вы создаете игру из серии Super Mario. Нужно определить класс для представления героя игры Марио. Как он бу-

дет выглядеть? Если вы не знакомы с играми серии Super Mario, используйте свою любимую видео- или настольную игру для моделирования игрока.

3. Создайте класс для моделирования твитов (сообщений в «Твиттере»). Если вы не знаете, что такое «Твиттер», в Википедии приводится следующее определение¹: «[...] социальная сеть для публичного обмена сообщениями при помощи веб-интерфейса, SMS, средств мгновенного обмена сообщениями или сторонних программ-клиентов для пользователей интернета любого возраста».
4. Создайте класс для моделирования бытового электроприбора (то-стер, стиральная машина, холодильник и т. д.).

¹ <https://ru.wikipedia.org/wiki/Твиттер>

22

Создание подклассов

Помимо группировки состояния и операций, классы также обеспечивают повторное использование кода. Если у вас уже имеется класс, а вам нужен другой класс, слегка отличающийся от него поведением, один из способов повторного использования заключается в создании подклассов. Класс, от которого производится создание подклассов, называется суперклассом (другое распространенное название суперкласса — родительский класс).

Предположим, вы хотите создать кресло, способное вмещать шесть лыжников. Чтобы создать класс `Chair6`, моделирующий кресло для шести человек, — более специализированную версию `Chair`, можно воспользоваться созданием подклассов. Подклассы позволяют программисту *наследовать* методы родительских классов и переопределять методы, которые нужно изменить.

Ниже приведен класс `Chair6`, который является подклассом `CorrectChair`:

```
>>> class Chair6(CorrectChair):  
...     max_occupants = 6
```

Обратите внимание: родительский класс `CorrectChair` заключается в круглые скобки после имени класса. Заметим, что `Chair6` не определяет конструктор в своем теле, однако вы можете создавать экземпляры класса:

```
>>> sixer = Chair6(76)
```

Как Python создает объект, если в классе не определен конструктор? Вот что происходит: когда Python ищет метод `__init__`, поиск начинается

Субклассы

Код

Что делает компьютер

```
class Chair6(CorrectChair):
    max_occupants = 6
```

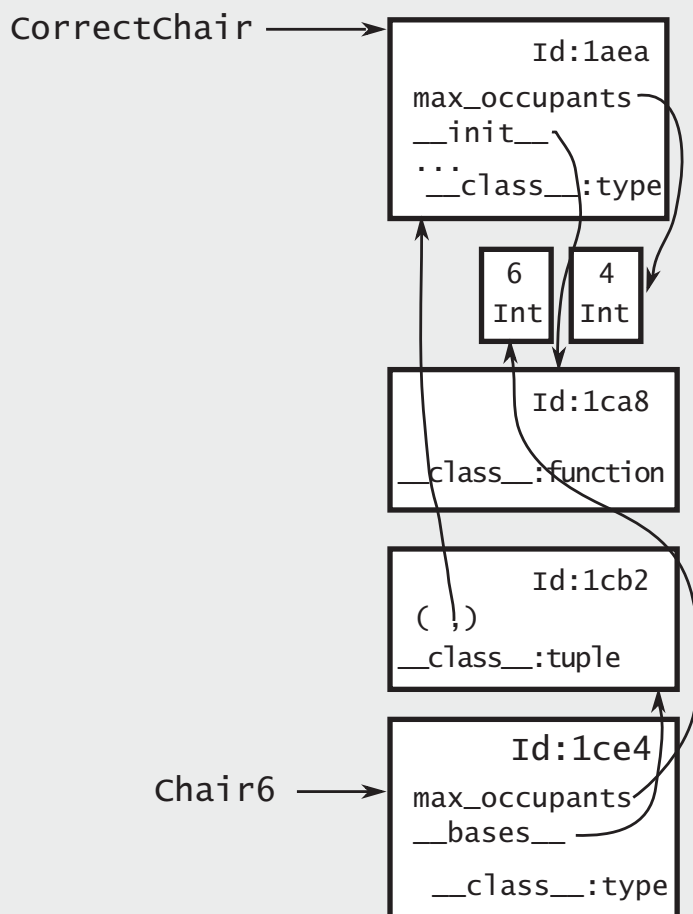


Рис. 22.1. Атрибут `__bases__` в подклассе. Связь между подклассом и его родительскими классами позволяет искать атрибуты в четко определенном порядке. Если атрибут определен в экземпляре подкласса, то используется этот атрибут. Если нет, то после экземпляра поиск продолжается в классе (`__class__`) экземпляра. Если и эта попытка оказывается неудачной, поиск осуществляется в родительских классах (`__bases__`)

с `Chair6`. Так как класс `Chair6` содержит только атрибут `max_occupants`, Python не найдет здесь метод `__init__`. Но поскольку `Chair6` является подклассом `CorrectChair`, он обладает атрибутом `__bases__` с перечислением базовых классов, сведенных в кортеж:

```
>>> Chair6.__bases__  
(__main__.CorrectChair,)
```

Затем Python ищет конструктор в базовых классах. Он находит конструктор в `CorrectChair` и использует его для создания нового класса.

Такой же поиск происходит при вызове `.load` для экземпляра. У экземпляра нет атрибута, соответствующего имени метода, поэтому Python проверяет класс экземпляра. В `Chair6` тоже нет метода `.load`, поэтому Python ищет атрибут в базовом классе `CorrectChair`. Здесь метод `.load` вызывается со слишком большим значением, что приводит к ошибке `ValueError`:

```
>>> sixer.load(7)  
Traceback (most recent call last):  
  File "/tmp/chair.py", line 30, in <module>  
    sixer.load(7)  
  File "/tmp/chair.py", line 13, in load  
    new_val = self._check(self.count + number)  
  File "/tmp/chair.py", line 23, in _check  
    number))  
ValueError: Invalid count:7
```

Python находит метод в базовом классе, но вызов метода `._check` приводит к ошибке `ValueError`.

22.1. Подсчет остановок

Иногда лыжнику не удастся нормально сесть на подъемник. В таких случаях оператор замедляет движение или останавливает подъемник, чтобы помочь лыжнику. Мы можем воспользоваться Python для создания нового класса, который будет подсчитывать количество таких остановок.

Предположим, при каждом вызове `.load` должна вызываться функция, которая возвращает логический признак того, произошла остановка или

нет. В параметрах функции передается количество лыжников и объект кресла.

Ниже приведен класс, который получает функцию `is_stalled` в конструкторе. Эта функция будет вызываться при каждом вызове `.load`:

```
>>> class StallChair(CorrectChair):
...     def __init__(self, id, is_stalled):
...         super().__init__(id)
...         self.is_stalled = is_stalled
...         self.stalls = 0
...
...     def load(self, number):
...         if self.is_stalled(number, self):
...             self.stalls += 1
...         super().load(number)
```

Чтобы создать экземпляр этого класса, необходимо предоставить функцию `is_stalled`. Следующая простая функция генерирует остановки в 10 % случаев:

```
>>> import random
>>> def ten_percent(number, chair):
...     """Return True 10% of time"""
...     return random.random() < .1
```

Теперь можно создать экземпляр, указав функцию `ten_percent` в качестве параметра `is_stalled`:

```
>>> stall42 = StallChair(42, ten_percent)
```

22.2. super

Напомним, что `StallChair` определяет свой собственный метод `.__init__`, который вызывается при создании экземпляра. Обратите внимание: первая строка конструктора выглядит так:

```
super().__init__(id)
```

При вызове `super` внутри метода вы получаете доступ к правильному родительскому классу. Строка в конструкторе позволяет вызвать конструктор `CorrectChair`. Вместо того чтобы повторять логику назначения

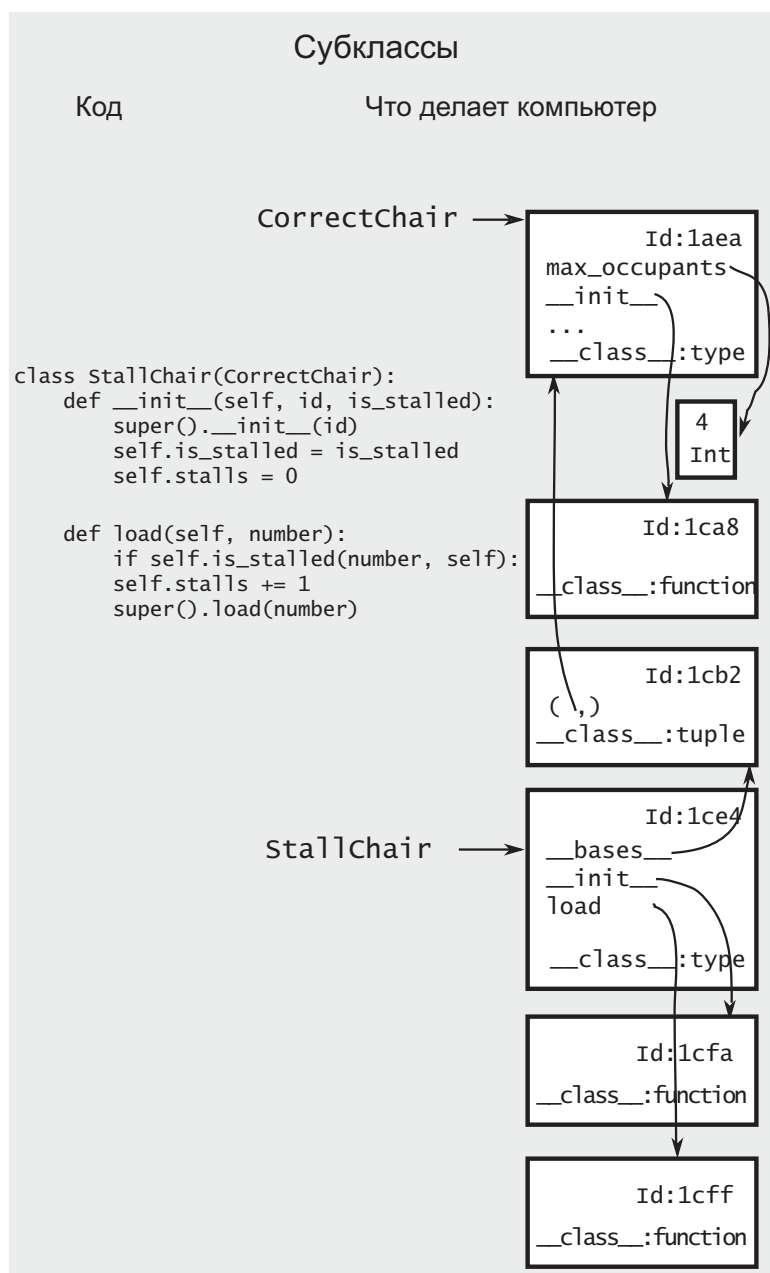


Рис. 22.3. Код создания подкласса с измененными методами. Обратите внимание на использование `super()` для вызова метода родительского класса. Диаграмма показывает, какие объекты создаются при создании класса, который является подклассом

атрибутов `id` и `count`, вы можете использовать логику из родительского класса. Так как `StallChair` имеет дополнительные атрибуты, которые нужно задать для экземпляра, это можно сделать после вызова родительского конструктора.

Метод `.load` тоже содержит вызов `super`:

```
def load(self, number):
    if self.is_stalled(number, self):
        self.stalls += 1
    super().load(number)
```

В методе `.load` вы вызываете функцию `is_stalled`, чтобы определить, останавливался ли подъемник, после чего передаете управление исходной функциональности `.load` из `CorrectChair` при помощи `super`.

Размещение общего кода в одном месте (в базовом классе) сокращает количество ошибок и дублирований кода.

ПРИМЕЧАНИЕ

Ключевое слово `super` особенно полезно в двух случаях. Во-первых, при определении порядка разрешения методов в классах с несколькими родителями `super` гарантирует последовательность этого порядка. Во-вторых, при изменении базового класса `super` самостоятельно определит новый базовый класс, а это упрощает сопровождение кода.

22.3. Итоги

В этой главе рассматриваются подклассы — новые специализированные классы, использующие код из своих базовых классов (также называемых суперклассами или родительскими классами). Для любого метода, не реализованного в подклассе, Python использует функциональность родительского класса. В реализации метода вы можете либо переопределить его полностью, либо включить вызов `super`. При вызове `super` вы получаете доступ к родительскому классу, чтобы использовать содержащуюся в нем функциональность.

22.4. Упражнения

1. Создайте класс, представляющий кошку. Что может делать кошка? Какими свойствами она обладает? Создайте подкласс кошки, представляющий тигра. Как изменится поведение подкласса?
2. В предыдущей главе вы создали класс, представляющий Марио из видеоигры Super Mario Brothers. В последних изданиях игры можно было играть за других персонажей. Все они обладали сходной базовой функциональностью¹, но различались способностями. Создайте базовый класс, представляющий персонажа, после чего реализуйте четыре подкласса — для Марио, Луиджи, Тода и Принцессы.

Имя	Марио	Луиджи	Тод	Принцесса
Скорость	4	3	5	2
Прыжок	4	5	2	3
Сила	4	3	5	2
Специальное умение				Парение в воздухе

¹ https://www.mariowiki.com/Super_Mario_Bros._2#Playable_characters

23

Исключения

Компьютеру можно приказать выполнить действие, которое он выполнить не может, — например, прочитать несуществующий файл или произвести деление на ноль. Python позволяет обработать такие исключительные ситуации, возникающие в программе. В таких случаях Python *выдает*, или *инициирует*, *исключение*.

Обычно при возникновении исключения Python прерывает выполнение и выводит *трассировку стека*, по которой можно определить, где именно возникла проблема. В трассировке стека указывается строка и файл ошибки:

```
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Из этой трассировки следует, что в строке 1 файла `<stdin>` (имя «файла» интерпретатора) произошла ошибка деления на ноль. При возникновении исключения в ходе выполнения программы трассировка стека показывает, в каком файле и строке возникла проблема. Приведенный пример с интерпретатором не особенно полезен, потому что программа состоит всего из одной строки кода. Однако в больших программах возможна многоуровневая иерархия трассировки стека из-за того, что функции вызывают другие функции и методы.

Допустим, файл содержит следующий код:

```
def err():
    1/0

def start():
    return middle()

def middle():
    return more()

def more():
    err()
```

При попытке выполнить его вы получите следующую трассировку:

```
Traceback (most recent call last):
  File "/tmp/err.py", line 13, in <module>
    start()
  File "/tmp/err.py", line 5, in start
    return middle()
  File "/tmp/err.py", line 8, in middle
    return more()
  File "/tmp/err.py", line 11, in more
    err()
  File "/tmp/err.py", line 2, in err
    1/0
ZeroDivisionError: division by zero
```

Трассировки проще всего читаются в обратном направлении: начните снизу, найдите ошибку и посмотрите, где она произошла. При перемещении вверх по трассировке вы фактически поднимаетесь по цепочке вызовов. Это поможет вам выяснить, что происходит в ваших программах.

23.1. «Посмотри, прежде чем прыгнуть»

Предположим, у вас имеется программа, которая выполняет деление. В зависимости от того, как написан код, в какой-то момент она может попытаться выполнить деление на ноль. Программисты обычно применяют

два стиля обработки исключений. Первый стиль называется *LBYL* (Look Before You Leap, то есть «Посмотри, прежде чем прыгнуть»). Его суть в том, чтобы проверить исключительную ситуацию перед выполнением действия. В нашем случае программа проверяет делитель на ноль. Если делитель отличен от нуля, программа может выполнить деление; если нет — операцию следует пропустить.

В Python стиль LBYL можно реализовать с помощью команд `if`:

```
>>> numerator = 10
>>> divisor = 0
>>> if divisor != 0:
...     result = numerator / divisor
... else:
...     result = None
```

ПРИМЕЧАНИЕ

Принцип LBYL не дает гарантии успеха. Даже если вы проверите, что файл существует, прежде чем открывать его, это не означает, что он будет существовать потом. В многопоточных средах такая ситуация называется условием *гонки* (race condition).

ПРИМЕЧАНИЕ

Значение `None` используется для представления неопределенного состояния. Это одна из распространенных идиом мира Python. Будьте внимательны и не пытайтесь вызывать методы для переменной, которой присвоено значение `None`, — это приведет к выдаче исключения.

23.2. «Проще просить прощения, чем разрешения»

Другой стиль обработки исключений обычно обозначается сокращением *EAFP* (Easier to Ask for Forgiveness than Permission, то есть «Проще попросить прощения, чем разрешения»). Если операция завершается неудачей, исключение будет *перехвачено* в блоке *исключения*.

Конструкция `try...except` предоставляет механизм для перехвата исключительных ситуаций в Python:

```
>>> numerator = 10
>>> divisor = 0
>>> try:
...     result = numerator / divisor
... except ZeroDivisionError as e:
...     result = None
```

Конструкция `try` создает блок после ключевого слова `try` (на что указывает двоеточие и отступы). Внутри блока `try` находятся команды, которые могут выдавать исключения. Если при выполнении команд действительно произойдет исключение, Python ищет блок `except`, который *перехватывает* это исключение (или исключение его родительского класса).

В приведенном коде блок `except` перехватывает исключения, являющиеся экземплярами класса `ZeroDivisionError` (или его подклассов). Когда в блоке `try` происходит указанное исключение, выполняется блок `except`, а результату присваивается `None`.

Обратите внимание: в строке

```
except ZeroDivisionError as e:
```

в последней позиции стоит двоеточие. Эта часть не обязательна: если она присутствует, то `e` (или другое имя переменной, которое вы выбрали) будет указывать на экземпляр исключения `ZeroDivisionError`. Вы можете проанализировать объект исключения, нередко в нем содержится более подробная информация. Переменная `e` указывает на *активное исключение*. Если вы не включите секцию `as e` в конце команды `except`, активное исключение в программе все равно будет, но вы не сможете обратиться к его экземпляру.

СОВЕТ

Постарайтесь ограничить область действия блока `try`. Вместо того чтобы включать весь код функции в блок `try`, включите только ту строку, в которой может произойти ошибка.

Так как стиль `LBYL` не гарантирует успешного предотвращения ошибок, обычно разработчики Python предпочитают стиль `EAFP`. Несколько практических правил обработки исключений:

- Обработывайте те ошибки, которые вы можете обработать и которые можно ожидать в программе.
- Не подавляйте те исключения, которые вы не можете обработать, и те, которые вряд ли возникнут в вашей программе.
- Используйте глобальный обработчик исключений для корректной обработки непредвиденных ошибок.

СОВЕТ

Если вы пишете серверное приложение, которое должно работать непрерывно, один из возможных способов показан ниже (функции `process_input` и `log_error` не существуют и приведены исключительно в демонстрационных целях):

```
while 1:
    try:
        result = process_input()
    except Exception as e:
        log_error(e)
```

23.3. Несколько возможных исключений

Если есть сразу несколько исключений, которые должны обрабатываться вашим кодом, перечислите их в нескольких командах `except`, следующих друг за другом:

```
try:
    some_function()
except ZeroDivisionError as e:
    # Обработка конкретного исключения
except Exception as e:
    # Обработка других исключений
```

В этом примере, когда `some_function` выдает исключение, интерпретатор сначала проверяет, соответствует ли оно ошибке класса `ZeroDivisionError` или его подкласса. Если условие не выполняется, код проверяет, относится ли исключение к подклассу `Exception`. После входа в блок `except` Python уже не проверяет последующие блоки.

Если исключение не обработано цепочкой, оно должно быть обработано кодом где-то в стеке вызовов. Если исключение так и остается необработанным, Python прекращает выполнение и выводит трассировку стека. Пример обработки нескольких исключений встречается в стандартной библиотеке. Модуль `argparse` из стандартной библиотеки предоставляет простой механизм разбора параметров командной строки. Он позволяет указать тип некоторых параметров — например, целых чисел или файлов (все параметры поступают в строковой форме). В методе `._get_value` встречаются примеры использования нескольких секций `except`. В зависимости от типа инициированного исключения выдаются разные сообщения об ошибках:

```
def _get_value(self, action, arg_string):

    type_func = self._registry_get('type', action.type, action.type)
    if not callable(type_func):
        msg = _('%r is not callable')
        raise ArgumentError(action, msg % type_func)

    # преобразование значения к соответствующему типу
    try:
        result = type_func(arg_string)

    # ArgumentError - признак ошибки
    except ArgumentError:
        name = getattr(action.type, '__name__', repr(action.type))
        msg = str(_sys.exc_info()[1])
        raise ArgumentError(action, msg)

    # TypeError и ValueErrors тоже являются признаками ошибок
    except (TypeError, ValueError):
        name = getattr(action.type, '__name__', repr(action.type))
        args = {'type': name, 'value': arg_string}
        msg = _('invalid %(type)s value: %(value)r')
        raise ArgumentError(action, msg % args)

    # возвращается преобразованное значение
    return result
```

ПРИМЕЧАНИЕ

Этот пример показывает, что одна команда `except` может перехватывать сразу несколько типов исключений, для чего следует передать кортеж классов исключений:

```
except (TypeError, ValueError):
```

ПРИМЕЧАНИЕ

Этот пример также демонстрирует старый стиль форматирования строк с использованием оператора `%`. Строки

```
msg = _('invalid %(type)s value: %(value)r')
raise ArgumentError(action, msg % args)
```

в современном стиле записываются так:

```
msg = _('invalid {type!s} value: {value!r}')
raise ArgumentError(action, msg.format(**args))
```

23.4. finally

Еще одна конструкция обработки ошибок — секция `finally`. Эта команда используется для определения кода, который будет выполняться всегда — независимо от того, возникло исключение или нет.

Блок `finally` выполняется всегда. Если исключение было обработано, то блок `finally` будет выполнен после обработки. Если исключение не было обработано, то блок `finally` выполняется, а исключение иницируется заново:

```
try:
    some_function()
except Exception as e:
    # Обработка ошибок
finally:
    # Завершающие действия
```

Обычно секция `finally` используется для освобождения внешних ресурсов: файлов, сетевых подключений, баз данных и т. д. Эти ресурсы должны освобождаться независимо от того, была операция выполнена успешно или нет.

Пример из модуля `timeit`, входящего в стандартную библиотеку, помогает понять полезность команды `finally`. Модуль `timeit` позволяет разработчику проводить хронометраж кода. В частности, во время проведения хронометража модуль приказывает уборщику мусора прекратить работу. Однако после завершения хронометража уборщик мусора должен быть снова включен независимо от того, был ли хронометраж завершен успешно или произошла ошибка.

Ниже приведен метод `timeit`, выполняющий хронометраж. Он проверяет, включен ли уборщик мусора, при необходимости отключает его, выполняет хронометражный код и, наконец, заново включает уборку мусора, если она была включена до этого:

```
def timeit(self, number=default_number):  
    """Хронометраж 'number' выполнений основной команды.
```

```
    Для повышения точности команда подготовки выполняется однократно,  
    после чего время, необходимое для многократного выполнения основной  
    команды, возвращается как вещественное число (в секундах). Аргумент  
    определяет количество выполнений цикла (по умолчанию один миллион).  
    Основная команда, команда подготовки и используемая функция таймера  
    передаются конструктору.
```

```
    """  
    it = itertools.repeat(None, number)  
    gcold = gc.isenabled()  
    gc.disable()  
    try:  
        timing = self.inner(it, self.timer)  
    finally:  
        if gcold:  
            gc.enable()  
    return timing
```

При вызове `self.inner` может произойти исключение, но, поскольку стандартная библиотека использует `finally`, уборка мусора всегда будет

включаться независимо от исключения (если логическая переменная `gcold` истинна).

ПРИМЕЧАНИЕ

В этой книге менеджеры контекста не рассматриваются, но, чтобы подготовить вас к будущей карьере эксперта Python, мы приведем небольшой совет. Комбинация `try/finally` в Python считается кодом «с душком». Опытные программисты Python в таких случаях применяют *менеджер контекста*. Включите эту тему в список вопросов, которые вам будет нужно изучить после освоения базовых возможностей Python.

23.5. Секция else

Необязательная секция `else` в команде `try` выполняется в том случае, если не было выдано никаких исключений. Она должна следовать за всеми секциями `except` и выполняется перед блоком `finally`. Простой пример:

```
>>> try:
...     print('hi')
... except Exception as e:
...     print('Error')
... else:
...     print('Success')
... finally:
...     print('at last')
hi
Success
at last
```

Ниже приведен пример из модуля `heapq` стандартной библиотеки. Как следует из комментариев, существует ускоренное решение, если число запрашиваемых значений превышает размер кучи. Тем не менее, если при попытке получения размера кучи произойдет ошибка, в коде вызывается `pass`. В результате ошибка игнорируется, а выполнение продолжается по более медленному варианту. Если ошибки не было, можно пойти по пути `else` и выбрать быстрый путь, если `n` превышает размер кучи:

```
def nsmaallest(n, iterable, key=None):
    # ....

    # Если n>=size, быстрее использовать sorted()
    try:
        size = len(iterable)
    except (TypeError, AttributeError) as e:
        pass
    else:
        if n >= size:
            return sorted(iterable, key=key)[:n]
    # Часть кода пропущена .... Использовать более медленный способ
```

23.6. Выдача исключений

Помимо перехвата исключений, Python также дает возможность выдавать исключения в коде (то есть инициировать их). Вспомните, что Дзен Python рекомендует явно выражать свои намерения и бороться с искушением что-либо предполагать. Если функции передается неверный ввод и вы знаете, что не сможете с ним справиться, можно выдать исключение. Исключения являются подклассами класса `BaseException`, а для их выдачи используется команда `raise`:

```
raise BaseException('Program failed')
```

Обычно в программе выдается не обобщенное исключение класса `BaseException`, а исключение одного из его подклассов — уже готовых или определенных разработчиком. В другом распространенном варианте команда `raise` используется сама по себе. Вспомните, что внутри команды `except` существует так называемое активное исключение. В таком случае можно обойтись *минимальной* командой `raise`. Эта команда позволяет обработать исключение, а потом заново выдать исходное исключение. При попытке выполнения кода

```
except (TypeError, AttributeError) as e:
    log('Hit an exception')
    raise e
```

вам удастся успешно инициировать исходное исключение, но трассировка стека будет показывать, что исходное исключение теперь произошло в строке с командой `raise e`, а не там, где оно произошло сначала. У про-

блемы есть два решения. Первое — использование *минимальной* команды `raise`. Второе — использование сцепленных исключений (см. далее).

Рассмотрим пример из модуля `configparser` стандартной библиотеки. Этот модуль обеспечивает чтение и создание INI-файлов. INI-файлы обычно используются для настройки конфигурации; они были популярны до появления форматов JSON и YAML. Метод `.read_dict` пытается прочесть конфигурацию из словаря. Если экземпляр находится в «жестком» режиме, при попытке добавления одной и той же секции более одного раза произойдет исключение. Если «жесткий» режим не включен, метод допускает наличие дубликатов ключей; используется последний ключ. Часть метода, демонстрирующая вариант с минимальной командой `raise`:

```
def read_dict(self, dictionary, source='<dict>'):
    elements_added = set()
    for section, keys in dictionary.items():
        section = str(section)
        try:
            self.add_section(section)
        except (DuplicateSectionError, ValueError):
            if self._strict and section in elements_added:
                raise
            elements_added.add(section)
    # Часть кода пропущена ....
```

Если дубликат добавляется в «жестком» режиме, трассировка стека покажет ошибку в методе `.add_section`, потому что она произошла именно там.

23.7. Упаковка исключений

В Python 3 появилась новая возможность, сходная с минимальной командой `raise`; она описана в «PEP 3134 — Exception Chaining and Embedded Tracebacks». При обработке исключения в коде обработки может произойти другое исключение. В подобных ситуациях бывает полезно знать об обоих исключениях.

Ниже приведен пример кода. У функции `divide_work` могут возникнуть проблемы с делением на 0. Вы можете перехватить эту ошибку и зарегистрировать ее в журнале. Предположим, функция регистрации обращает-

ся к облачному сервису, который в настоящее время недоступен (чтобы смоделировать эту ситуацию, мы заставим `log` выдать исключение):

```
>>> def log(msg):
...     raise SystemError("Logging not up")

>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
```

Если при вызове `divide_work` передать 5 и 0, Python выведет информацию о двух ошибках, `ZeroDivisionError` и `SystemError`. Ошибка `SystemError` будет выведена последней, потому что она произошла последней:

```
>>> divide_work(5, 0)
Traceback (most recent call last):
  File "begpy.py", line 3, in divide_work
    return x/y
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(5, 0)
  File "begpy.py", line 5, in divide_work
    log("System is down")
  File "begpy.py", line 2, in log
    raise SystemError("Logging not up")
SystemError: Logging not up
```

Предположим, облачный сервис журнала заработал (функция `log` уже не выдает ошибку). Если вы хотите изменить тип `ZeroDivisionError` в `divide_work` на `ArithmeticError`, используйте синтаксис, описанный в РЕР 3134. Для этого можно воспользоваться синтаксисом `raise... from`:

```
>>> def log(msg):
...     print(msg)

>>> def divide_work(x, y):
...     try:
...         return x/y
```

```
...     except ZeroDivisionError as ex:
...         log("System is down")
...         raise ArithmeticError() from ex
```

Теперь вы видите два исключения: исходное `ZeroDivisionError` и исключение `ArithmeticError`, которое уже не скрывается `ZeroDivisionError`:

```
>>> divide_work(3, 0)
Traceback (most recent call last):
  File "begpy.py", line 3, in divide_work
    return x/y
ZeroDivisionError: division by zero
```

Во время обработки вышеуказанного исключения произошло другое исключение:

```
Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(3, 0)
  File "begpy.py", line 6, in divide_work
    raise ArithmeticError() from ex
ArithmeticError
```

Если вы хотите подавить исходное исключение `ZeroDivisionError`, используйте следующий код (см. «PEP 0409 — Suppressing exception context»):

```
>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
...         raise ArithmeticError() from None
```

Теперь видна только внешняя ошибка `ArithmeticError`:

```
>>> divide_work(3, 0)
Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(3, 0)
  File "begpy.py", line 6, in divide_work
    raise ArithmeticError() from None
ArithmeticError
```

23.8. Определение собственных исключений

В модуле `exceptions` определено много встроенных исключений. Если ваша ошибка хорошо соответствует какому-то существующему исключению, используйте его. Ниже представлена иерархия классов встроенных исключений:

```
BaseException
  SystemExit
  KeyboardInterrupt
  GeneratorExit
  Exception
    StopIteration
    ArithmeticError
      FloatingPointError
      OverflowError
      ZeroDivisionError
    AssertionError
    AttributeError
    BufferError
    EnvironmentError
      IOError
      OSError
    EOFError
    ImportError
    LookupError
      IndexError
      KeyError
    MemoryError
    NameError
      UnboundLocalError
    ReferenceError
    RuntimeError
      NotImplementedError
    SyntaxError
      IndentationError
      TabError
    SystemError
    TypeError
    ValueError
      UnicodeError
        UnicodeDecodeError
```



```
UnicodeEncodeError
UnicodeTranslateError
Warning
    DeprecationWarning
    PendingDeprecationWarning
    RuntimeWarning
    SyntaxWarning
    UserWarning
    FutureWarning
    ImportWarning
    UnicodeWarning
    BytesWarning
```

Для определения собственных исключений создайте подкласс в классе `Exception` или один из его подклассов. Дело в том, что все остальные подклассы `BaseException` не обязательно являются «исключениями». Например, если вы перехватываете `KeyboardInterrupt`, вам не удастся прервать выполнение процесса клавишами `Ctrl+C`. Если вы перехватите `GeneratorExit`, перестанут работать генераторы.

Вот как выглядит исключение для определения нехватки информации в программе:

```
>>> class DataError(Exception):
...     def __init__(self, missing):
...         self.missing = missing
```

Использовать нестандартное исключение достаточно просто:

```
>>> if 'important_data' not in config:
...     raise DataError('important_data missing')
```

23.9. Итоги

В этой главе были представлены стратегии обработки исключений. В стратегии `LBYL` перед выполнением операции вы проверяете, что в текущей ситуации не будет выдана ошибка. В стратегии `EAFP` весь код, который может выдать ошибку, заключается в блок `try/catch`. В Python предпочтение отдается второму стилю программирования.

Также были рассмотрены различные механизмы перехвата ошибок, их выдачи и повторной выдачи. В завершение главы было показано, как

создавать подклассы, существующие исключения для создания ваших собственных исключений.

23.10. Упражнения

1. Напишите программу, выполняющую функции простейшего калькулятора. Программа получает два числа, а затем запрашивает оператор. Обеспечьте корректную обработку ввода, который не преобразуется в числа. Обработайте ошибки деления на ноль.
2. Напишите программу, которая вставляет нумерацию перед строками файла. Имя файла передается программе в командной строке. Импортируйте модуль `sys` и прочитайте имя файла из списка `sys.argv`. Корректно обработайте возможность передачи несуществующего файла.

24

Импортирование библиотек

В предыдущих главах были рассмотрены основные конструкции языка Python. Эта глава посвящена импортированию кода. Во многих языках существует концепция *библиотек*, или блоков кода, предназначенных для повторного использования. Python поставляется с большой подборкой библиотек, однако, чтобы извлечь пользу из этих библиотек, вы должны уметь пользоваться ими.

Чтобы использовать библиотеку, нужно загрузить код из библиотеки в *пространство имен* вашей программы. Пространство имен содержит функции, классы и переменные, доступные для программы. Если вы хотите вычислить синус угла, вам придется определить функцию, которая выполняет эти вычисления, или же загрузить готовую функцию. Встроенная библиотека `math` содержит функцию `sin` для вычисления синуса угла, выраженного в радианах. Библиотека также содержит переменную, определяющую значение «пи»:

```
>>> from math import sin, pi
>>> sin(pi/2)
1.0
```

Приведенный фрагмент загружает модуль `math`. Тем не менее он не помещает `math` в ваше пространство имен. Вместо этого он создает переменную, которая указывает на функцию `sin` из модуля `math`. Он также создает переменную, указывающую на переменную `pi` из модуля `math`. Если вы

проанализируете текущее пространство имен при помощи функции `dir`, вы сможете убедиться в этом:

```
>>> 'sin' in dir()
True
```

24.1. Способы импортирования

В предыдущем примере мы импортировали одну функцию из библиотеки. Также можно загрузить библиотеку в пространство имен и обращаться ко всем классам, функциям и переменным. Чтобы импортировать модуль `math` в пространство имен, введите следующую команду:

```
>>> import math
```

В этом примере импортируется библиотека `math`. При этом создается новая переменная `math`, указывающая на модуль. Этот модуль содержит разные атрибуты, список которых можно вывести функцией `dir`:

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'acos', 'acosh', 'asin',
 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Большинство этих атрибутов составляют функции. Если вы хотите вызвать функцию `tan`, сделать это не удастся, потому что имя `tan` не входит в ваше пространство имен — только `math`. Однако вы можете провести поиск по переменной `math` с использованием оператора «точка» (`.`). Этот оператор просматривает атрибуты объекта. Так как в Python нет ничего, кроме объектов, вы можете воспользоваться этим оператором для поиска атрибута `tan` объекта `math`:

```
>>> math.tan(0)
0.0
```

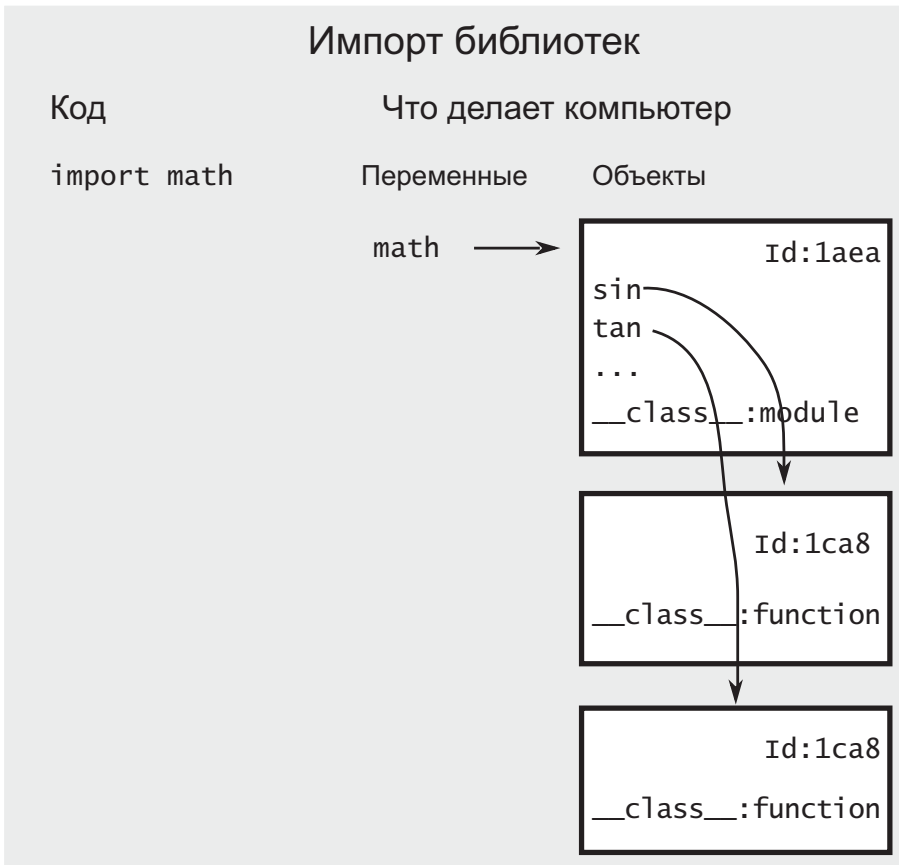


Рис. 24.1. Импорт модуля. Обратите внимание, что этот код создает новую переменную `math`, которая указывает на модуль. Модуль имеет различные атрибуты, к которым вы можете получить доступ, используя период

Если вы захотите прочитать документацию по функции `tan`, воспользуйтесь функцией `help`:

```
>>> help(math.tan)
```

Справка по встроенной функции `tan` в модуле `math`:

```
tan(...)  
tan(x)
```

Возвращает тангенс `x` (в радианах).

СОВЕТ

Когда импортировать функцию с использованием `from`, а когда следует выбрать команду `import`? Если вы используете только пару атрибутов из библиотеки, используйте механизм импортирования в стиле `from`. В конструкции `from` можно перечислить несколько атрибутов, разделенных запятыми:

```
>>> from math import sin, cos, tan
>>> cos(0)
1.0
```

Но если вам нужно обращаться к большей части библиотеки, будет проще импортировать библиотеку командой `import`. Такое решение также подскажет каждому, кто будет читать ваш код (включая вас), откуда взялась та или иная функция (класс, переменная).

ПРИМЕЧАНИЕ

Если вам нужно импортировать несколько атрибутов из библиотеки, команда может занять несколько строк. Если вы попытаетесь продолжить импортирование функций в следующей строке, произойдет ошибка:

```
>>> from math import sin,
...     cos
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: trailing comma not allowed
without surrounding parentheses
```

Чтобы показать, что команда продолжается в следующей строке, поставьте символ `\`:

```
>>> from math import sin,\
...     cos
```

Также возможен другой вариант — перечислить импортируемые имена в круглых скобках. Открывающая круглая скобка (или квадратная, или фигурная) показывает, что команда продолжается в следующей строке:

```
>>> from math import (sin,
...     cos)
```

Последняя форма считается более идиоматичной для Python.

24.2. Конфликты имен при импортировании

Если вы работаете над программой, выполняющей тригонометрические операции, у вас уже может быть определена функция с именем `sin`. А если вы также захотите использовать функцию `sin` из библиотеки `math`? Одно из возможных решений — импортировать `math`; тогда запись `math.sin` будет обозначать библиотечную версию, а `sin` — вашу функцию.

Python также предоставляет другую возможность. Вы можете переопределить импортируемое имя при помощи ключевого слова `as`:

```
>>> from math import sin as other_sin
>>> other_sin(0)
0.0
```

Теперь `other_sin` представляет собой ссылку на версию `sin` из `math`, а вы можете продолжать использовать `sin` без необходимости рефакторинга кода.

Ключевое слово `as` также работает с командами `import`. Если у вас имеется переменная (или функция), имя которой конфликтовало с именем из `math` в вашем пространстве имен, следующий фрагмент демонстрирует одно из возможных решений:

```
>>> import math as other_math
>>> other_math.sin(0)
0.0
```

СОВЕТ

Ключевое слово `as` также поможет сделать код более компактным. Если в вашей любимой библиотеке используются слишком длинные имена, вы можете легко сократить их в своем коде. Пользователи библиотеки `Numpy`¹ приняли в качестве стандарта сокращенные имена из двух букв:

```
>>> import numpy as np
```

Аналогичный стандарт был принят в библиотеке `Pandas`²:

```
>>> import pandas as pd
```

¹ numpy.org

² pandas.pydata.org

24.3. Массовое импортирование

Python также позволяет загромождать пространство имен так называемым *массовым импортированием*:

```
>>> from math import *  
>>> asin(0)  
0.0
```

В этом коде вызывается функция арксинуса, которая не была определена в коде. Та строка, в которой вызывается `asin`, содержит первое упоминание `asin` в коде. Что произошло? Когда вы используете команду `math import *`, эта команда приказывает Python загрузить все содержимое библиотеки `math` (определения классов, функции и переменные) в локальное пространство имен. Хотя такая возможность на первый взгляд кажется удобной, на самом деле она довольно опасна.

Массовое импортирование усложняет отладку, потому что оно не выражает явно, откуда взялся тот или иной код. Еще больше проблем возникает с массовым импортированием из нескольких библиотек. Более поздние команды импортирования могут переопределить что-то такое, что было определено в более ранней библиотеке. По этой причине использовать массовое импортирование не рекомендуется.

СОВЕТ

Не используйте массовое импортирование!

Возможные исключения из этого правила встречаются разве что при самостоятельном написании кода тестирования или при экспериментах в REPL. Авторы библиотек применяют эту конструкцию для импортирования всего содержимого библиотеки, которую они собираются протестировать. Не используйте массовое импортирование только потому, что вы увидели его в чужом коде.

Вспомните Дзен Python:

«Явное лучше, чем неявное».

24.4. Вложенные библиотеки

У некоторых пакетов Python есть вложенное пространство имен. Например, библиотека XML, входящая в поставку Python, включает поддержку `minidom` и `etree`. Обе библиотеки размещаются внутри родительского пакета `xml`:

```
>>> from xml.dom.minidom import \
...     parseString
>>> dom = parseString(
...     '<xml><foo/></xml>')

>>> from xml.etree.ElementTree import \
...     XML
>>> elem = XML('<xml><foo/></xml>')
```

Конструкция `from` позволяет импортировать только нужные функции и классы. Конструкция `import` (без `from`) увеличивает объем кода (но также открывает доступ ко всему содержимому пакета):

```
>>> import xml.dom.minidom
>>> dom = xml.dom.minidom.parseString(
...     '<xml><foo/></xml>')

>>> import xml.etree.ElementTree
>>> elem = xml.etree.ElementTree.XML(
...     '<xml><foo/></xml>')
```

24.5. Организация импортирования

Согласно PEP 8, команды импортирования должны располагаться в начале файла за строкой документации модуля. В каждой строке должна находиться одна команда `import`, причем команды `import` должны быть объединены в группы:

- Импортирование из стандартной библиотеки.
- Импортирование из стороннего кода.
- Импортирование из локальных пакетов.

Например, начало модуля может выглядеть так:

```
#!/usr/bin/env python3
"""
Модуль преобразует записи в JSON
и сохраняет их в базе данных
"""
import json                # Стандартные библиотеки
import sys

import psycopg2            # Сторонние библиотеки

import recordconverter     # Локальные библиотеки
...
```

СОВЕТ

Сгруппированные команды импортирования удобно упорядочить по алфавиту.

СОВЕТ

Иногда команды импортирования бывает полезно отложить по следующим причинам:

- Для предотвращения *циклического импортирования* (ситуации, при которой модули импортируют друг друга). Если вы не можете (или не хотите) провести рефакторинг для предотвращения циклического импортирования, команду `import` можно разместить в функции или методе с тем кодом, из которого вызываются импортируемые элементы.
 - Для предотвращения импортирования модулей, недоступных в некоторых системах.
 - Для предотвращения импортирования больших модулей, не используемых в программе.
-

24.6. Итоги

В этой главе рассматривалось импортирование библиотек в Python. У Python имеется большая стандартная библиотека¹, и вам нужно будет импортировать эти библиотеки для использования в коде. В книге неоднократно подчеркивалось, что в Python нет ничего, кроме объектов, и вы часто создаете переменные, указывающие на эти объекты. При импортировании модуля создается переменная, указывающая на объект модуля. После этого вы можете обратиться к любым элементам пространства имен модуля при помощи операции поиска (`.`).

Также можно избирательно импортировать части пространства имен модуля командой `from`. Если вы хотите переименовать импортируемый элемент, воспользуйтесь конструкцией `as`.

24.7. Упражнения

1. Найдите в стандартной библиотеке Python пакет для работы с JSON. Импортируйте библиотечный модуль и просмотрите атрибуты. Используйте функцию `help` для получения более подробной информации об использовании модуля. Сериализуйте словарь, связывающий ключ `'name'` с вашим именем и ключ `'age'` с вашим возрастом, в строку JSON. Проведите десериализацию JSON в Python.
2. Найдите в стандартной библиотеке Python пакет для вывода содержимого каталогов. Используйте этот пакет и напишите функцию, которая получает имя каталога, получает список всех файлов в этом каталоге и выводит отчет с количеством файлов для каждого расширения.

¹ <https://docs.python.org/3/library/index.html>

25

Библиотеки: пакеты и модули

В главе 24 вы научились импортировать библиотеки. В этой главе мы разберемся в том, что же представляет собой библиотека. При импортировании библиотеки действуют два требования:

1. Библиотека должна представлять собой *модуль* или *пакет*.
2. Библиотека должна находиться в месте, определяемом переменной среды `PYTHONPATH` или переменной Python `sys.path`.

25.1. Модули

Модули представляют собой файлы Python с расширением `.py` и именем, пригодным для импортирования. Согласно PEP 8, имена файлов модулей должны быть короткими и записываться в нижнем регистре. Символы подчеркивания могут использоваться для удобства чтения.

25.2. Пакеты

Пакет в Python представляет собой каталог, содержащий файл с именем `__init__.py`. Файл `__init__.py` может содержать любую реализацию (или вообще быть пустым). Кроме того, каталог может содержать произвольное количество модулей и субпакетов.

Что лучше использовать при написании кода — модуль или пакет? Я обычно начинаю с более простого варианта и использую модуль. А если мне нужно выделять отдельные части в собственные модули, я преобразую модули в пакет.

Пример структуры каталога из популярного проекта SQLAlchemy¹ (средство объектно-реляционного отображения для баз данных):

```
sqlalchemy/  
  __init__.py  
  engine/  
    __init__.py  
    base.py  
    schema.py
```

Согласно PEP 8, имена каталогов пакетов должны быть короткими и записываться в нижнем регистре. Символы подчеркивания в них недопустимы.

25.3. Импортирование пакетов

Чтобы импортировать пакет, используйте команду `import` с именем пакета (именем каталога):

```
>>> import sqlalchemy
```

Эта команда импортирует файл `sqlalchemy/__init__.py` в текущее пространство имен, если пакет будет найден в `PYTHONPATH` или `sys.path`.

Если вы захотите использовать классы `Column` и `ForeignKey` из модуля `schema.py`, подойдет любой из следующих фрагментов. Первый включает `sqlalchemy.schema` в ваше пространство имен, а второй помещает в пространство имен только `schema`:

```
>>> import sqlalchemy.schema  
>>> col = sqlalchemy.schema.Column()  
>>> fk = sqlalchemy.schema.ForeignKey()
```

¹ <https://www.sqlalchemy.org/>

или

```
>>> from sqlalchemy import schema
>>> col = schema.Column()
>>> fk = schema.ForeignKey()
```

А если вам нужен только класс `Column`, импортируйте этот класс одним из двух способов:

```
>>> import sqlalchemy.schema.Column
>>> col = sqlalchemy.schema.Column()
```

или

```
>>> from sqlalchemy.schema import Column
>>> col = Column()
```

25.4. PYTHONPATH

Переменная среды `PYTHONPATH` содержит список нестандартных каталогов, в которых Python ищет модули или пакеты. Обычно эта переменная пуста. Изменять `PYTHONPATH` обычно не обязательно, если только вы в процессе разработки не хотите использовать библиотеки, которые еще не были установлены.

СОВЕТ

Оставьте переменную `PYTHONPATH` пустой, если только у вас нет веских причин для ее изменения. В этом разделе показано, что может произойти при ее изменении. Последствия могут сбить с толку других разработчиков, пытающихся отладить ваш код, если они забудут о том, что переменная `PYTHONPATH` была изменена.

Если вы включили код в файл `/home/test/a/plot.py`, но работали из каталога `/home/test/b/`, использование `PYTHONPATH` предоставит доступ к этому коду. В противном случае, если файл `plot.py` не был установлен системными средствами или средствами Python, попытка его импортирования приведет к ошибке `ImportError`:

```
>>> import plot
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
ImportError: No module named plot
```

Если вы запустите Python с присваиванием значения переменной `PYTHONPATH`, она укажет Python, где нужно искать библиотеки:

```
$ PYTHONPATH=/home/test/a python3
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
>>> import plot
>>> plot.histogram()
...
```

СОВЕТ

Для установки пакетов Python могут использоваться менеджеры пакетов, исполняемые файлы Windows или специализированные средства Python, такие как `pip`¹.

25.5. sys.path

У модуля `sys` имеется атрибут `path` со списком каталогов, в которых Python ищет библиотеки. Просмотрев `sys.path`, вы увидите все просмотренные каталоги:

```
>>> import sys
>>> sys.path

['',
 '/usr/lib/python35.zip',
 '/usr/lib/python3.6',
 '/usr/lib/python3.6/plat-darwin',
 '/usr/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6/site-packages']
```

СОВЕТ

Если вы сталкиваетесь с ошибкой вида

```
ImportError: No module named plot,
```

обратитесь к переменной `sys.path` и посмотрите, присутствует ли в ней каталог, в котором находится файл `foo.py` (если это модуль). Если `plot`

¹ <https://pip.pypa.io/>

является пакетом, то каталог `plot/` должен находиться в одном из каталогов из `sys.path`:

```
>>> import plot
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named plot
>>> sys.path.append('/home/test/a')
>>> import plot
>>> plot.histogram()
```

Также можно включить этот каталог в `PYTHONPATH` из командной строки, используемой для запуска Python.

Еще раз подчеркнем, что обычно задавать вручную `sys.path` или `PYTHONPATH` не нужно. В нормальной ситуации вы устанавливаете библиотеки, а программа установки размещает их в правильном каталоге.

СОВЕТ

Если вы хотите узнать местонахождение библиотеки в файловой системе, проверьте атрибут `__file__`:

```
>>> import json
>>> json.__file__
'/usr/lib/python3.6/json/__init__.py'
```

Этот способ работает только с библиотеками, реализованными на Python. Модуль `sys` не реализован на Python, поэтому эта попытка завершается неудачей:

```
>>> import sys
>>> sys.__file__
Traceback (most recent call last):
...
AttributeError: module 'sys' has no attribute '__file__'
```

25.6. Итоги

В этой главе рассматриваются модули и пакеты. Модуль представляет собой файл Python. Пакет представляет собой каталог, в котором при-

существует файл с именем `__init__.py`. Пакет также может содержать другие модули и пакеты.

Существует список путей, который определяет, к каким каталогам обращается Python для импортирования библиотек. Этот список хранится в `sys.path`. Проверив значение этой переменной, вы узнаете, какие каталоги просматривает Python. Вы также можете обновить этот список при помощи переменной `PYTHONPATH` или же внести изменения напрямую. Но обычно вместо изменения этой переменной вы используете специализированные средства установки пакетов, например `pip`.

25.7. Упражнения

1. Создайте модуль `begin.py`, который содержит функцию с именем `prime`. Функция `prime` должна получать число и возвращать логический признак того, является ли число простым (то есть делится ли оно только на 1 и на себя). Перейдите в другой каталог, запустите Python и выполните команду

```
from begin import prime
```

Попытка должна завершиться неудачей. Обновите переменную `sys.path`, чтобы вы могли импортировать функцию из модуля. Затем присвойте значение `PYTHONPATH`.

2. Создайте пакет `utils`. В файле `__init__.py` разместите код `prime` из предыдущего примера. Перейдите в другой каталог в терминале, запустите Python и выполните команду

```
from utils import prime
```

Попытка должна завершиться неудачей. Обновите переменную `sys.path`, чтобы вы могли импортировать функцию из пакета. Затем присвойте значение `PYTHONPATH`.

26

Полноценный пример

В этой главе вы узнаете, как организовать код в сценарии. В ней приводится исходный код упрощенной реализации команды UNIX `cat`. Команда `cat` читает имена файлов из командной строки и выводит их содержимое на экран. Предусмотрены различные параметры для выполнения таких операций, как добавление нумерации. Этот сценарий показывает, как организован код в типичном файле Python.

26.1. `cat.py`

Ниже приведена реализация команды UNIX `cat` на языке Python. Включен параметр для добавления нумерации строк (`--number`), но другие параметры `cat` не поддерживаются. Сохраните код в файле с именем `cat.py`:

```
#!/usr/bin/env python3
```

```
r"""Простая реализация команды unix ``cat``.  
Поддерживается только параметр ``--number``.  
Пример демонстрирует структуру файла  
и полезные приемы программирования на Python.
```

```
Строка документации в тройных кавычках для всего модуля  
(этот файл). Если импортировать этот модуль  
и выполнить команду ``help(cat)`` , вы сможете  
убедиться в этом.
```

```
Строка документации также содержит ``документирующий тест``,
```

который служит примером программного использования кода.
 Модуль ``doctest`` может выполнить эту строку документации
 и проверить ее правильность по выходным данным.

```
>>> import io
>>> fin = io.StringIO(\

...     'hello\nworld\n')
>>> fout = io.StringIO()
>>> cat = Catter([fin],
...     show_numbers=True)
>>> cat.run(fout)
>>> print(fout.getvalue())
    1 hello
    2 world

"""
import argparse
import logging
import sys

__version__ = '0.0.1'

logging.basicConfig(
    level=logging.DEBUG)

class Catter(object):
    """
    Класс для конкатенации файлов
    в стандартном выводе

    Строка документации класса,
    выводится командой ``help(cat.Catter)``
    """

    def __init__(self, files,
                 show_numbers=False):
        self.files = files
        self.show_numbers = show_numbers

    def run(self, fout):
        # Использовать 6 пробелов для чисел
        # с выравниванием по правому краю
        fmt = '{0:>6} {1}'
```

```
for fin in self.files:
    logging.debug('catting {}'.format(fin))
    for count, line in enumerate(fin, 1):
        if self.show_numbers:
            fout.write(fmt.format(
                count, line))
        else:
            fout.write(line)

def main(args):
    """
    Логика выполнения cat с аргументами
    """
    parser = argparse.ArgumentParser(
        description='Concatenate FILE(s), or '
        'standard input, to standard output')
    parser.add_argument('--version',
        action='version', version=__version__)
    parser.add_argument('-n', '--number',
        action='store_true',
        help='number all output lines')
    parser.add_argument('files', nargs='*',
        type=argparse.FileType('r'),
        default=[sys.stdin], metavar='FILE')
    parser.add_argument('--run-tests',
        action='store_true',
        help='run module tests')
    args = parser.parse_args(args)

    if args.run_tests:
        import doctest
        doctest.testmod()
    else:
        cat = Catter(args.files, args.number)
        cat.run(sys.stdout)
        logging.debug('done catting')

if __name__ == '__main__':
    main(sys.argv[1:])
```

Если вам не хочется вводить весь этот код вручную, загрузите его копию из интернета¹.

¹ <https://github.com/mattharrison/IllustratedPy3/>

26.2. Что делает этот код?

Этот код осуществляет эхо-вывод содержимого файла (возможно, с нумерацией строк) на терминал в системах Windows и UNIX:

```
$ python3 cat.py -n README.md
1 # IllustratedPy3
2
3 If you have questions or concerns, click on Issues above.
```

Если вы запустите этот код с ключом `-h`, он выведет всю справочную документацию по аргументам командной строки:

```
$ python3 cat.py -h
usage: cat.py [-h] [--version] [-n] [--run-tests] [FILE [FILE ...]]
```

Concatenate FILE(s), or standard input, to standard output

positional arguments:

FILE

optional arguments:

```
-h, --help show this help message and exit
--version show program's version number and exit
-n, --number number all output lines
--run-tests run module tests
```

Функциональность разбора командной строки реализована в функции `main` и обеспечивается модулем `argparse` из стандартной библиотеки. Модуль `argparse` берет на себя всю работу по разбору аргументов.

Если вы хотите узнать, что делает модуль `argparse`, обратитесь к документации в интернете или воспользуйтесь функцией `help`. Основная идея заключается в том, что вы создаете экземпляр класса `ArgumentParser` и вызываете `.add_argument` для каждого параметра командной строки. Вы указываете параметры командной строки, сообщаете, какое действие необходимо для них выполнить (по умолчанию это сохранение значения, следующего за параметром), и предоставляете справочную документацию. После добавления аргументов вызывается метод `.parse_args` для аргументов командной строки (которые берутся из `sys.argv`). Результат `.parse_args` представляет собой объект, к которому присоединены

атрибуты, имена которых определяются именами параметров. В данном случае это будут атрибуты `.files` и `.number`.

ПРИМЕЧАНИЕ

Вы также можете воспользоваться REPL для получения информации о модуле и просмотра документации, входящей в его поставку. Помните функцию `help`? Передайте ей модуль, и она выведет строку документации уровня модуля.

Если вы хотите просмотреть исходный код модуля, это тоже возможно. Помните функцию `dir`, которая выводит атрибуты объекта? Просмотрев информацию модуля `argparse`, вы увидите, что у него есть атрибут `__file__`. Он указывает на местонахождение файла на компьютере:

```
>>> import argparse
>>> argparse.__file__
'/usr/local/Cellar/python3/3.6.0/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/argparse.py'
```

Так как код написан на Python (часть модулей написана на C), вы можете просмотреть файл с исходным кодом. К настоящему моменту вы уже сможете прочитать модуль и понять, что он должен делать.

После разбора аргументов программа создает экземпляр `Catter`, определенный в коде, и вызывает для него метод `.run`.

Пример выглядит немного устрашающе, но весь использованный синтаксис рассматривался в книге. В оставшейся части этой главы рассматривается структура и другие аспекты этого кода.

26.3. Типичная структура

Основные компоненты модуля Python в порядке их следования:

- `#!/usr/bin/env python3` (если модуль также используется в качестве сценария);
- строка документации модуля;
- `imports`;

- метаданные и глобальные переменные;
- операции с журналом;
- реализация;
- `if __name__ == '__main__':` (если модуль также используется в качестве сценария);
- `argparse`.

ПРИМЕЧАНИЕ

Этот список — не более чем рекомендация. Большинство элементов может следовать в произвольном порядке, и не все элементы должны присутствовать в каждом файле. Например, не каждый файл должен быть исполняемым в качестве сценария командного интерпретатора.

В своих файлах вы можете применять любую структуру, но ответственность за такое решение ложится на вас. Скорее всего, пользователи вашего кода будут недовольны (или будут выпускать исправления). Любой читатель предпочтет код, построенный по общепринятым правилам, так как он сможет быстро разобраться в его логике.

26.4. #!

Первая строка файла, используемого в качестве сценария, содержит последовательность символов `#!/usr/bin/env python3`. В операционных системах семейства UNIX эта строка обрабатывается для определения того, как следует выполнять сценарий. Соответственно, эта строка включается только в те файлы, которые должны выполняться в качестве сценариев.

В ней должна использоваться команда `python 3`, так как команда `python` в большинстве систем относится к Python версии 2.

ПРИМЕЧАНИЕ

На платформе Windows строка `#!` игнорируется, поэтому ее включение безопасно. Вы найдете ее во многих библиотеках, популярных в системах Windows.

ПРИМЕЧАНИЕ

Вместо того чтобы жестко задавать конкретный путь к исполняемому файлу Python, `/usr/bin/env` выбирает первый исполняемый файл `python3` из каталогов PATH пользователя. Такие средства, как `venv`¹, изменяют содержимое PATH для использования альтернативных исполняемых файлов `python3`; они успешно работают в этой схеме.

СОВЕТ

Если в системах UNIX каталог с файлом присутствует в переменной среды PATH текущего пользователя, а файл является исполняемым, то для выполнения из командного интерпретатора достаточно указать только имя файла.

Чтобы назначить файл исполняемым, введите следующую команду:

```
$ chmod +x <путь/к/file.py>
```

26.5. Строка документации

В начале файла модуля может располагаться строка документации уровня модуля. Она должна следовать за строкой с `#!`, но предшествовать любому коду Python. Строка документации содержит обзор модуля, и в ней должна содержаться сводная информация о коде. Кроме того, в ней могут содержаться примеры использования модуля.

СОВЕТ

Python включает библиотеку `doctest` для проверки примеров из интерактивного интерпретатора. Использование строк документации, содержащих фрагменты кода REPL, могут служить как для документации, так и для простой проверки логики своей библиотеки.

В файле `cat.py` код `doctest` содержится в конце строки документации. При запуске `cat.py` с ключом `--run-tests` библиотека `doctest` перебирает все существующие строки документации и проверяет содержащийся в них код. Данная возможность приведена только для демонстрации: обычно возможность выполнения тестов в сценарии не отображается для рядовых пользователей, не являющихся разработчиками, даже если вы включили код `doctest` в строку документации. В данном случае параметр `--run-test` включен как пример использования модуля `doctest`.

¹ <https://docs.python.org/3/library/venv.html>

26.6. Импортирование

Команды импортирования обычно включаются в начало модулей Python. Строки `import` принято группировать по местонахождению библиотеки. Сначала перечисляются библиотеки, входящие в стандартную библиотеку Python. Далее следуют сторонние библиотеки, и в последнюю очередь перечисляются библиотеки, локальные для текущего кода. Такая структура кода позволит конечным пользователям быстро просмотреть команды импорта, требования и источники происхождения кода.

26.7. Метаданные и глобальные переменные

Если у вас имеются глобальные переменные уровня модуля, определите их после команд импортирования. Это позволит просмотреть модуль и быстро определить, что собой представляют глобальные переменные.

Глобальные переменные определяются на уровне модуля, и они доступны в границах этого модуля. Так как Python позволяет изменить любую переменную, глобальные переменные являются потенциальным источником ошибок. Кроме того, код проще понять, когда переменные определяются и изменяются только в области видимости функции. Тогда вы можете быть твердо уверены в том, с какими данными вы работаете и кто их изменяет. Если глобальная переменная изменяется в нескольких местах (и особенно в разных модулях), вы своими руками готовите себе долгий сеанс отладки.

Один из допустимых вариантов использования глобальных переменных — эмуляция *констант* из других языков программирования. Константа аналогична переменной, но ее значение не может изменяться. В Python не поддерживаются переменные, которые не могут изменяться, но вы можете воспользоваться специальными обозначениями, указывающими, что переменная должна рассматриваться в программе как доступная только для чтения. В PEP 8 указано, что имена глобальных констант должны назначаться по тем же правилам, что и имена переменных, но они должны быть записаны прописными буквами. Например, если вы хотите использовать в программе пропорцию золотого сечения, соответствующее значение может быть определено так:

```
>>> GOLDEN_RATIO = 1.618
```

Если этот код определяется в модуле, регистр символов подсказывает, что программа не должна изменять связывание этой переменной.

ПРИМЕЧАНИЕ

Определяя константы в виде глобальных переменных и используя тщательно продуманные имена, можно избежать проблемы, встречающейся в программировании, — так называемых «волшебных чисел». Под «волшебным числом» понимается число, присутствующее в коде или формуле, которое не хранится в переменной. Это само по себе достаточно плохо, особенно когда кто-то пытается разобраться в вашем коде.

Другая проблема с «волшебными числами» заключается в том, что значения со временем нередко распространяются в коде. Это не создаст проблем, пока вы не захотите изменить это значение. Что делать — провести поиск с заменой? А если «волшебное число» на самом деле представляет два разных значения — например, число сторон треугольника и размерность окружающего мира? В этом случае глобальный поиск с заменой приведет к появлению ошибок.

Обе проблемы (контекст и повторение) решаются сохранением значения в именованной переменной. Таким образом у переменной образуется контекст, а у числа появляется имя. Оно также позволяет легко изменить значение в одном месте.

Кроме глобальных переменных, на этом уровне также существуют переменные *метаданных*. В метаданных хранится информация о модуле: автор, версия и т. д. Обычно метаданные хранятся в «специальных» переменных с двойными подчеркиваниями (`__author__`). Например, PEP 396 рекомендует хранить версию модуля в строковой переменной `__version__` на глобальном уровне модуля.

ПРИМЕЧАНИЕ

Если вы собираетесь опубликовать свою библиотеку, желательно определить ее версию. В PEP 396 указаны некоторые практические приемы объявления строк версий.

К числу других распространенных переменных метаданных относятся имя автора, лицензия, дата и контактная информация. При определении в программном коде они могли бы выглядеть так:

```
__author__ = 'Matt Harrison'
__date__ = 'Jan 1, 2017'
```

```
__contact__ = 'matt_harrison <at> someplace.com'  
__version__ = '0.1.1'
```

26.8. Операции с журналом

Еще одна переменная, часто объявляемая на глобальном уровне, — средство ведения журнала для модуля. Стандартная библиотека Python включает библиотеку ведения журнала, которая позволяет регистрировать информацию на разных уровнях детализации в четко определенных форматах.

Скорее всего, необходимость в записи информации в журнал возникнет у разных классов или функций. Обычно программа выполняет инициализацию средства ведения журнала один раз на глобальном уровне, а затем использует полученный дескриптор в модуле.

26.9. Другие глобальные переменные

Не используйте глобальные переменные там, где хватает локальных переменных. Основные глобальные переменные в коде Python — метаданные, константы и подсистема ведения журнала.

Глобальные переменные нередко встречаются в коде примеров. Не поддавайтесь искушению копировать такой код. Выделите его в функцию или класс. Это принесет пользу в будущем, когда вы займетесь рефакторингом или отладкой вашего кода.

26.10. Реализация

После глобальных переменных и настройки ведения журнала следует непосредственное содержание кода — *реализация*. Существенную часть кода займут функции и классы. Основная логика модуля содержится в классе `Catter`.

26.11. Тестирование

Обычно правильно организованный тестовый код отделяется от кода реализации. Python допускает небольшое исключение из этого правила.

Строки документации Python могут определяться на уровне модулей, функций, классов и методов. В строках документации можно разместить фрагменты REPL, демонстрирующие использование данной функции, класса или модуля. Такие фрагменты, если они будут хорошо продуманы и построены, эффективно документируют стандартные варианты применения модуля.

Другая интересная возможность `doctest` — проверка документации. Если ваши фрагменты когда-то работали, а теперь не работают, значит, либо изменился ваш код, либо сами фрагменты содержат ошибки. Это можно легко выявить до того, как пользователи начнут жаловаться вам.

СОВЕТ

Код `doctest` может размещаться в отдельном текстовом файле. Чтобы выполнить произвольный файл средствами `doctest`, используйте функцию `testfile`:

```
import doctest
doctest.testfile('module_docs.txt')
```

ПРИМЕЧАНИЕ

Кроме `doctest` стандартная библиотека Python включает модуль `unittest`, реализующий типичную методологию xUnit — подготовка/проверка/завершение. У обоих стилей тестирования — `doctest` и `unittest` — есть как достоинства, так и недостатки. Стил `doctest` обычно создает больше трудностей с отладкой, а стил `unittest` содержит шаблонный код, который считается слишком «завязанным на Java». Вы можете сочетать оба стиля, чтобы получить хорошо документированный и хорошо протестированный код.

26.12. if `__name__ == '__main__':`

Если ваш файл должен выполняться в качестве сценария, в конце сценария вы найдете следующий фрагмент:

```
if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]) or 0)
```

Чтобы понять эту команду, необходимо понимать смысл переменной `__name__`.

26.13. `__name__`

Python определяет переменную уровня модуля `__name__` для любого *импортируемого* модуля или любого выполняемого файла. Обычно значением `__name__` является имя модуля:

```
>>> import sys
>>> sys.__name__
'sys'
>>> import xml.sax
>>> xml.sax.__name__
'xml.sax'
```

У этого правила есть исключение. При *выполнении* модуля (то есть `python3 some_module.py`) значением `__name__` является строка `"__main__"`.

По сути, значение `__name__` сообщает, загружается ли файл как библиотека или же выполняется как сценарий.

ПРИМЕЧАНИЕ

Использование `__name__` можно продемонстрировать на простом примере. Создайте файл `some_module.py` со следующим кодом:

```
print("The __name__ is: {}".format(__name__))
```

Теперь запустите REPL и *импортируйте* этот модуль:

```
>>> import some_module
The __name__ is: some_module
```

А теперь *выполните* этот модуль:

```
$ python3 some_module.py
The __name__ is: __main__
```

Одна из распространенных идиом в мире Python — размещение подобных проверок в конце модуля, который также может служить сценарием. Такая проверка определяет, выполняется файл или импортируется:

```
if __name__ == '__main__':  
    # выполнение  
    sys.exit(main(sys.argv[1:]) or 0)
```

Эта простая команда запускает функцию `main`, когда файл выполняется. И наоборот, если файл используется в качестве модуля, функция `main` автоматически выполняться не будет. Функция `sys.exit` вызывается с возвращаемым значением `main` (или 0, если `main` не возвращает значение), как делают все добропорядочные программы в мире UNIX.

Функция `main` получает параметры командной строки в списке `sys.argv`. В самом начале `sys.argv` находится элемент `python3`, поэтому нам приходится создать срез `sys.argv`, чтобы исключить этот элемент перед тем, как передавать параметры `main`.

СОВЕТ

Некоторые разработчики размещают логику выполнения (код, расположенный внутри функции `main`) прямо под проверкой `if __name__ == '__main__':`. Несколько причин для хранения логики в функции:

- Функция `main` может вызываться из других мест.
 - Функцию `main` можно легко тестировать с разными аргументами.
 - Сокращение объема кода, выполняемого на глобальном уровне.
-

26.14. Итоги

В этой главе был проанализирован код Python в сценариях командного интерпретатора. Мы рассмотрели некоторые полезные практические приемы и стандартные соглашения программирования.

Если вы будете структурировать свой код так, как описано в этой главе, то будете следовать лучшим практикам программирования на языке Python. Описанная структура также упростит чтение вашего кода другими разработчиками.

26.15. Упражнения

1. Скопируйте код `cat.py`. Добейтесь того, чтобы он заработал на вашем компьютере. Не думайте, что это напрасный труд — занимаясь программированием, вы очень часто не создаете что-то с нуля, а повторно используете код, написанный другими людьми.
2. Напишите сценарий `convert.py`, который преобразует файл из одной кодировки в другую. Программа должна получать следующие параметры командной строки:
 - Имя входного файла.
 - Входная кодировка (по умолчанию UTF-8).
 - Выходная кодировка.
 - Режим обработки ошибок (игнорировать/выдать исключение).

27

В начале пути

К этому моменту вы уже достаточно хорошо понимаете, как работают программы Python, уверенно владеете REPL и умеете анализировать классы с использованием функций `dir` и `help`.

Что дальше? Это зависит от вас. В принципе, у вас есть все предпосылки для использования Python для сайтов, GUI-программ или вычислительных приложений.

Среди огромных преимуществ Python можно выделить различные сообщества, связанные с разными областями программирования. Существует множество локальных пользовательских групп, конференций, списков рассылки и социальных сетей, посвященных разным аспектам Python. Многие из этих групп охотно принимают новых программистов и делятся своими знаниями. Не бойтесь опробовать что-нибудь новое; в Python это делается просто, и, скорее всего, найдутся и другие разработчики с похожими интересами.

Приложение А

Перемещение по файлам

Если вы еще не знакомы с навигацией в файловой системе с терминала, ниже приведено краткое введение. Сначала необходимо открыть терминал — одно из тех окон с большим объемом текста, которые обычно показывают в фильмах про хакеров. Вообще говоря, программировать можно и без них, но умение переходить между каталогами и выполнять команды из терминала — навык безусловно полезный.

А.1. Mac и UNIX

На компьютерах Mac вызовите Spotlight комбинацией клавиш **Command+Space** и введите `terminal`, чтобы запустить терминальное приложение для Mac.

В системах семейства Linux способ запуска терминала зависит от конфигурации рабочей среды. Например, в системах Ubuntu можно воспользоваться комбинацией клавиш **Ctrl+Alt+T**. Простейший терминал, присутствующий в большинстве систем, называется `xterm`.

Вы должны знать несколько основных команд:

- `cd` — переход в другой каталог. Например, команда

```
$ cd ~/Documents
```

выполняет переход в каталог `Documents`, находящийся в домашнем каталоге (`~` — сокращенное обозначение домашнего каталога, которым является каталог `/Users/<имя_пользователя>` на Mac или `/home/<имя_пользователя>` в Linux);

- `pwd` — вывод текущего каталога, в котором вы находитесь;
- `ls` — вывод содержимого текущего каталога.

Если у вас имеется сценарий Python, хранящийся в файле `~/work/intro-to-py/hello.py`, его можно выполнить следующей последовательностью команд:

```
$ cd ~/work/intro-to-py
$ python3 hello.py
```

A.2. Windows

В системе Windows нажмите **Win+R**, чтобы открыть интерфейс запуска программ. Введите `cmd`, чтобы открыть окно командной строки.

Основные команды:

- `cd` — переход в другой каталог. Например, команда

```
c:> cd C:\Users
```

выполняет переход в каталог `C:\Users`;

- `echo %CD%` — вывод текущего каталога, в котором вы находитесь;
- `dir` — вывод содержимого текущего каталога.

Если у вас имеется сценарий Python, хранящийся в файле `C:\Users\matt\intro-to-py\hello.py`, его можно выполнить следующей последовательностью команд:

```
C:> cd C:\Users\matt\intro-to-py
C:\Users\matt\intro-to-py> python hello.py
```

Приложение Б

Полезные ссылки

Полезные ссылки на тему Python:

<https://python.org/> — домашняя страница Python

<https://github.com/mattharrison/Tiny-Python-3.6-Notebook> — справочник по Python 3.6.

<http://docutils.sourceforge.net/> — reStructuredText — упрощенный язык разметки для документации Python.

<https://pyformat.info> — удобный справочник по форматированию строк.

<https://pypi.python.org/pypi> — Python Package Index — сторонние пакеты.

<https://www.python.org/dev/peps/pep-0008/> — PEP 8 — стандарт оформления кода.

<https://www.anaconda.com/download/> — Anaconda — альтернативная программа установки Python с множеством сторонних пакетов.

<https://www.djangoproject.com/> — Django — популярный веб-фреймворк.

<http://scikit-learn.org/> — машинное обучение на Python.

<https://www.tensorflow.org/> — глубокое обучение на Python.

<https://www.reddit.com/r/Python/> — новости Python.

Об авторе



Мэтт Харрисон использует Python с 2000 года. Он руководит компанией MetaSnake, занимающейся консультационными услугами и проведением корпоративного обучения в области Python и теории анализа данных. В прошлом он работал в областях научных исследований, управления сборкой и тестированием, бизнес-аналитики и хранения данных.

Он выступал с презентациями и учебными лекциями на таких конференциях, как Strata, SciPy, SCALE, PyCON и OSCON, а также на локальных пользовательских конференциях. Структура и материал этой книги основаны на его практическом опыте преподавания Python. Мэтт периодически публикует в «Твиттере» полезную информацию, относящуюся к Python (@__mharrison__).

Научные редакторы

Роджер Э. Дэвидсон (Roger A. Davidson) в настоящее время является деканом факультета математики в колледже Америкэн Ривер (Сакраменто, штат Калифорния). Его докторская диссертация была написана на тему авиационно-космической техники, но он также является обладателем дипломов об образовании в области computer science, электротехники и системотехники, а также недавно получил сертификат в области data science (с чего и началось его увлечение Python). На протяжении своей карьеры Роджер работал в NASA, в компаниях из списка Fortune 50, в стартапах и муниципальных колледжах. Он с энтузиазмом относится к образованию, науке (не только к обработке данных), пирогам с ежевикой и руководству неоднородными коллективами при решении больших задач.

Эндрю Маклафлин (AndrewMcLaughlin) — программист и проектировщик, системный администратор в первую половину дня и семьянин во вторую. Из-за своего внимания к деталям занимается веб-программированием с 1998 года. Обладатель диплома с отличием университета Джорджа Фокса, Эндрю получил степень в области систем управления и информации. В свободное время он ходит в турпоходы со своей женой и двумя детьми, а также иногда работает в столярной мастерской (все пальцы пока на месте). Читайте его публикации в «Твиттере»: @amclaughlin.

Мэтт Харрисон

**Как устроен Python.
Гид для разработчиков, программистов
и интересующихся**

Перевел с английского *Е. Матвеев*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>О. Букатка</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>С. Беляева, И. Тимофеева</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Импортёр в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 31.01.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930. Доп. тираж. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Подробная информация здесь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, гоб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, гоб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, гоб. 6217;
e-mail: kuznetsov@piter.com

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничает с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com
Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com