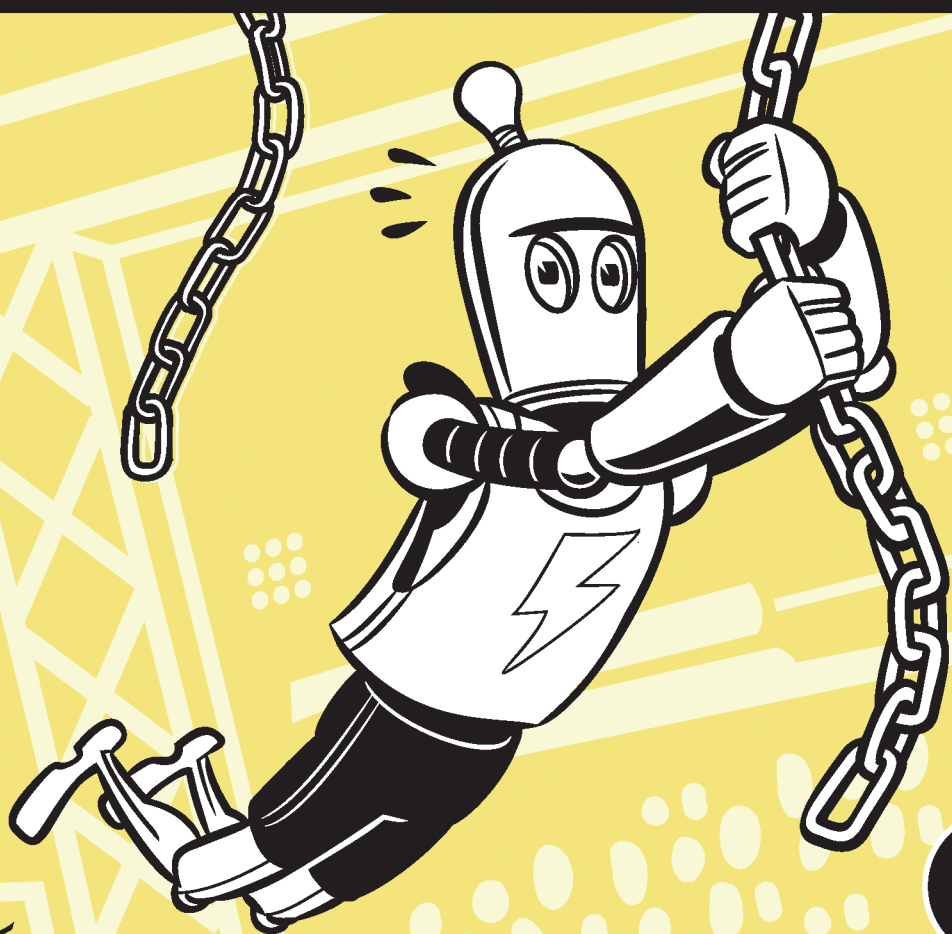


PYTHON

ЧИСТЫЙ КОД ДЛЯ ПРОДОЛЖАЮЩИХ

ЭЛ СВЕЙГАРТ



BEYOND THE BASIC STUFF WITH PYTHON

**Best Practices for
Writing Clean Code**

Al Sweigart



**no starch
press**

San Francisco

PYTHON

ЧИСТЫЙ КОД
ДЛЯ ПРОДОЛЖАЮЩИХ

ЭЛ СВЕЙГАРТ



Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018
УДК 004.41
С24

Свейгарт Эл

С24 Python. Чистый код для продолжающих. — СПб.: Питер, 2022. — 384 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1852-6

Вы прошли обучающий курс программирования на Python или прочли несколько книг для начинающих. Что дальше? Как подняться над базовым уровнем, превратиться в крутого разработчика?

«Python. Чистый код для продолжающих» — это не набор полезных советов и подсказок по написанию чистого кода. Вы узнаете о командной строке и других инструментах профессионального разработчика: средствах форматирования кода, статических анализаторах и контроле версий. Вы научитесь настраивать среду разработки, давать имена переменным и функциям, делающие код удобочитаемым, грамотно комментировать и документировать ПО, оценивать быстродействие программ и сложность алгоритмов, познакомитесь с ООП.

Такие навыки поднимут вашу ценность как программиста не только в Python, но и в любом другом языке.

Ни одна книга не заменит реального опыта работы и не превратит вас из новичка в профессионала. Но «Чистый код для продолжающих» проведет вас чуть дальше по этому пути: вы научитесь создавать чистый, грамотный, читабельный, легко отлаживаемый код, который можно будет назвать истинно питоническим.

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

ББК 32.973.2-018
УДК 004.41

Права на издание получены по соглашению с No Starch Press. 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 978-1593279660 англ.

© 2021 by Al Sweigart. Beyond the Basic Stuff with Python: Best Practices for Writing Clean Code, ISBN 9781593279660, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103.

Russian edition published under license by No Starch Press Inc.

ISBN 978-5-4461-1852-6

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2022

© Серия «Библиотека программиста», 2022

Краткое содержание

Об авторе	18
О техническом редакторе	18
Благодарности	18
Введение	19

ЧАСТЬ I ПЕРВЫЕ ШАГИ

Глава 1. Обработка ошибок и обращение за помощью	26
Глава 2. Подготовка среды и командная строка	40

ЧАСТЬ II ПЕРЕДОВЫЕ ПРАКТИКИ, ИНСТРУМЕНТЫ И МЕТОДЫ

Глава 3. Форматирование кода при помощи Black	70
Глава 4. Выбор понятных имен	85
Глава 5. Поиск запахов в коде	95
Глава 6. Написание питонического кода	113
Глава 7. Жаргон программистов	135
Глава 8. Часто встречающиеся ловушки Python	164
Глава 9. Экзотические странности Python	185
Глава 10. Написание эффективных функций	194
Глава 11. Комментарии, doc-строки и аннотации типов	216

Глава 12. Git и организация программных проектов	235
Глава 13. Измерение быстродействия и анализ сложности алгоритмов	264
Глава 14. Проекты для тренировки	288

ЧАСТЬ III

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ PYTHON

Глава 15. Объектно-ориентированное программирование и классы	316
Глава 16. Объектно-ориентированное программирование и наследование	335
Глава 17. ООП в Python: свойства и dunder-методы	358

Оглавление

Об авторе	18
О техническом редакторе	18
Благодарности	18
Введение	19
Для кого написана эта книга и почему	20
О книге	20
Часть I. Первые шаги	21
Часть II. Передовые практики, инструменты и методы	21
Часть III. Объектно-ориентированный Python	22
Путешествие в мир программирования	23

ЧАСТЬ I ПЕРВЫЕ ШАГИ

Глава 1. Обработка ошибок и обращение за помощью	26
Как понять сообщения об ошибках Python	26
Анализ трассировки	27
Поиск сообщений об ошибках	30
Предотвращение ошибок при помощи статического анализатора	31
Как обратиться за помощью по программированию	33
Избегайте лишних разговоров, предоставляйте информацию заранее ..	34
Формулируйте свой вопрос как вопрос	34
Задавайте вопросы на подходящем веб-сайте	34
Включите краткое описание вопроса в заголовок	34
Объясните, что должен делать ваш код	35
Включите полное сообщение об ошибке	36
Приведите полный код	36
Правильно отформатируйте свой код	36

Сообщите, что вы уже пытались сделать	37
Опишите свою рабочую конфигурацию	37
Примеры вопросов	38
Итоги	39
Глава 2. Подготовка среды и командная строка	40
Файловая система	40
Пути в Python	41
Домашний каталог	42
Текущий рабочий каталог	43
Абсолютные и относительные пути	43
Программы и процессы	44
Командная строка	45
Открытие окна терминала	46
Запуск программ из командной строки	47
Аргументы командной строки	48
Выполнение кода Python из командной строки с ключом -c	50
Выполнение программ Python из командной строки	50
Запуск программы ru.exe	50
Выполнение команд из программы Python	51
Сокращение объема вводимого текста при помощи автозавершения ...	51
Просмотр истории команд	52
Часто используемые команды	53
Переменные среды и PATH	60
Просмотр переменных среды	61
Переменная среды PATH	62
Изменение переменной среды PATH в командной строке	63
Постоянное включение папок в PATH в Windows	63
Постоянное включение папок в PATH в macOS и Linux	65
Запуск программ Python без командной строки	65
Запуск программ Python в Windows	66
Запуск программ Python в macOS	67
Запуск программ Python в Ubuntu Linux	67
Итоги	68

ЧАСТЬ II

ПЕРЕДОВЫЕ ПРАКТИКИ, ИНСТРУМЕНТЫ И МЕТОДЫ

Глава 3. Форматирование кода при помощи Black	70
Как потерять друзей и настроить против себя коллег	70
Руководства по стилю и PEP 8	71
Горизонтальные отступы	72
Использование пробелов для создания отступов	72
Отступы в середине строки	73
Вертикальные отступы	76
Пример использования вертикальных отступов	77
Рекомендации по использованию вертикальных отступов	78
Black: бескомпромиссная система форматирования кода	79
Установка Black	80
Запуск Black из командной строки	80
Отключение Black для отдельных частей кода	83
Итоги	84
Глава 4. Выбор понятных имен	85
Схемы регистра имен	86
Соглашения об именах PEP 8	86
Длина имен	87
Выбирайте имена, пригодные для поиска	90
Избегайте шуток, каламбуров и культурных отсылок	91
Не заменяйте встроенные имена	92
Худшие из возможных имен	93
Итоги	93
Глава 5. Поиск запахов в коде	95
Дублирование кода	95
«Магические» числа	97
Закомментированный и мертвый код	100
Отладочный вывод	101
Переменные с числовыми суффиксами	102
Классы, которые должны быть функциями или модулями	103

Списковые включения внутри списковых включений	104
Пустые блоки except и плохие сообщения об ошибках	106
Мифы о запахах кода	107
Миф: функции должны содержать только одну команду return в самом конце	108
Миф: функции должны содержать не более одной команды try	108
Миф: аргументы-флаги нежелательны	109
Миф: глобальные переменные нежелательны	110
Миф: комментарии излишни	111
Итоги	112
Глава 6. Написание питонического кода	113
«Дзен Python»	113
Как полюбить значимые отступы	117
Использование модуля timeit для оценки быстродействия	118
Неправильное использование синтаксиса	120
Использование enumerate() вместо range()	120
Использование команды with вместо open() и close()	121
Использование is для сравнения с None вместо ==	122
Форматирование строк	123
Использование необработанных строк, если строка содержит много символов \ (обратный слэш)	123
Форматирование с использованием F-строк	124
Поверхностное копирование списков	125
Питонические способы использования словарей	126
Использование get() и setdefault() со словарями	127
Использование collections.defaultdict для значений по умолчанию	128
Использование словарей вместо команды switch	129
Условные выражения: «некрасивый» тернарный оператор Python	130
Работа со значениями переменных	132
Сцепление операторов присваивания и сравнения	132
Проверка того, что переменная содержит одно из нескольких значений	133
Итоги	133

Глава 7. Жаргон программистов	135
Определения	135
Язык Python и интерпретатор Python	136
Сборка мусора	137
Литералы	137
Ключевые слова	138
Объекты, значения, экземпляры и идентичность	139
Элементы	143
Изменяемость и неизменяемость	143
Индексы, ключи и хеш-коды	146
Контейнеры, последовательности, отображения и разновидности множеств	149
Dunder-методы, или магические методы	150
Модули и пакеты	150
Вызываемые объекты и первоклассные объекты	151
Частые ошибки при использовании терминов	152
Команды и выражения	152
Блок, секция и тело	153
Переменные и атрибуты	154
Функции и методы	155
Итерируемые объекты и итераторы	155
Синтаксические ошибки, ошибки времени выполнения и семантические ошибки	157
Параметры и аргументы	159
Явные и неявные преобразования типов	159
Свойства и атрибуты	159
Байт-код и машинный код	160
Сценарии и программы, языки сценариев и языки программирования	161
Библиотеки, фреймворки, SDK, ядра и API	161
Итоги	162
Дополнительные ресурсы	163

Глава 8. Часто встречающиеся ловушки Python	164
Не добавляйте и не удаляйте элементы из списка в процессе перебора ...	164
Не копируйте изменяемые значения без <code>copy.copy()</code> и <code>copy.deepcopy()</code> ...	171
Не используйте изменяемые значения для аргументов по умолчанию	174
Не создавайте строки посредством конкатенации	176
Не рассчитывайте, что <code>sort()</code> выполнит сортировку по алфавиту	178
Не рассчитывайте на идеальную точность чисел с плавающей точкой	179
Не объединяйте операторы <code>!=</code> в цепочку	182
Не забудьте запятую в кортежах из одного элемента	183
Итоги	183
Глава 9. Экзотические странности Python	185
Почему 256 — это 256, а 257 — не 257	185
Интернирование строк	187
Фиктивные операторы инкремента и декремента в языке Python	188
Все или ничего	189
Логические значения как целые числа	190
Сцепление разных видов операторов	192
Антигравитация в Python	193
Итоги	193
Глава 10. Написание эффективных функций	194
Имена функций	194
Плюсы и минусы размера функций	195
Параметры и аргументы функций	198
Аргументы по умолчанию	198
Использование <code>*</code> и <code>**</code> для передачи аргументов функции	199
Использование <code>*</code> при создании вариadicеских функций	201
Использование <code>**</code> при создании вариadicеских функций	203
Использование <code>*</code> и <code>**</code> для создания функций-оберток	205
Функциональное программирование	206
Побочные эффекты	206
Функции высшего порядка	209
Лямбда-функции	209

Отображение и фильтрация со списковыми включениями	210
Возвращаемые значения всегда должны иметь один тип данных	212
Выдача исключений и возвращение кодов ошибок	214
Итоги	215
Глава 11. Комментарии, дос-строки и аннотации типов	216
Комментарии	217
Стиль комментариев	218
Встроенные комментарии	218
Пояснительные комментарии	219
Сводные комментарии	220
Комментарии «полученный опыт»	220
Комментарии об авторских правах и интеллектуальной собственности	221
Профессиональные комментарии	221
Кодовые метки и комментарии TODO	222
Магические комментарии и кодировка исходных файлов	223
Дос-строки	223
Аннотации типов	226
Статические анализаторы	228
Аннотации типов для набора типов	230
Аннотации типов для списков, словарей и т. д.	231
Обратное портирование аннотаций типов	232
Итоги	234
Глава 12. Git и организация программных проектов	235
Коммиты и репозитории	236
Создание новых проектов Python с использованием Cookiecutter	236
Установка Git	239
Настройка имени пользователя и адреса электронной почты	239
Установка графических средств Git	240
Работа с Git	241
Как Git отслеживает статус файлов	241
Для чего нужно индексирование?	243

Создание репозитория Git на вашем компьютере	243
Добавление файлов для отслеживания	245
Игнорирование файлов в репозитории	247
Сохранение изменений	248
Удаление файлов из репозитория	252
Переименование и перемещение файлов из репозитория	253
Просмотр журнала коммитов	255
Восстановление старых изменений	256
Отмена несохраненных локальных изменений	256
Деиндексирование проиндексированного файла	257
Отмена последних коммитов	257
Возврат к конкретному коммиту для отдельного файла	258
Перезапись истории коммитов	259
GitHub и команда git push	260
Отправка существующего репозитория на GitHub	261
Клонирование существующего репозитория GitHub	262
Итоги	262
Глава 13. Измерение быстродействия и анализ сложности алгоритмов	264
Модуль timeit	265
Профилировщик cProfile	267
Анализ алгоритмов с использованием нотации «О-большое»	269
Порядки нотации «О-большое»	270
Книжная полка как метафора порядков «О-большое»	271
«О-большое» как оценка худшего сценария	275
Определение порядка сложности нотации «О-большое» вашего кода	277
Почему низкие порядки и коэффициенты не важны	279
Примеры анализа «О-большое»	280
Порядок «О-большое» для часто используемых функций	283
Моментальный анализ сложности	285
«О-большое» не имеет значения при малых n... а значения n обычно малы	286
Итоги	286

Глава 14. Проекты для тренировки	288
Головоломка «Ханойская башня»	289
Вывод результатов	289
Исходный код	291
Написание кода	293
Игра «Четыре в ряд»	301
Вывод результатов	301
Исходный код	302
Написание кода	305
Итоги	314

ЧАСТЬ III

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ PYTHON

Глава 15. Объектно-ориентированное программирование и классы	316
Аналогия из реального мира: заполнение форм	317
Создание объектов на базе классов	319
Создание простого класса: WizCoin	320
Методы, <code>__init__()</code> и <code>self</code>	322
Атрибуты	323
Приватные атрибуты и приватные методы	324
Функция <code>type()</code> и атрибут <code>__qualname__</code>	326
Примеры программирования с применением ООП и без него: «Крестики-нолики»	327
Трудности проектирования классов для проектов реального мира	333
Итоги	334

Глава 16. Объектно-ориентированное программирование и наследование	335
Как работает наследование	335
Переопределение методов	338
Функция <code>super()</code>	340
Предпочитайте композицию наследованию	342
Обратная сторона наследования	343

Функции <code>isinstance()</code> и <code>issubclass()</code>	346
Методы классов	347
Атрибуты классов	349
Статические методы	350
Когда использовать объектно-ориентированные статические средства и средства уровня классов	350
Термины объектно-ориентированного программирования	351
Инкапсуляция	351
Полиморфизм	351
Когда наследование не используется	352
Множественное наследование	353
Порядок разрешения методов	354
Итоги	356
Глава 17. ООП в Python: свойства и dunder-методы	358
Свойства	358
Преобразование атрибута в свойство	359
Использование методов <code>setter</code> для проверки данных	362
Свойства, доступные только для чтения	364
Когда использовать свойства	365
Dunder-методы Python	366
Dunder-методы строкового представления	366
Числовые dunder-методы	369
Отраженные числовые dunder-методы	373
Dunder-методы присваивания на месте (<code>in-place</code>)	375
Dunder-методы сравнения	377
Итоги	382

Моему племяннику Джеку

Об авторе

Эл Свейгарт — разработчик и автор технической литературы, живет в Сиэтле. Python — его любимый язык программирования, он разработал для Python несколько модулей с открытым кодом. Его другие книги доступны бесплатно на условиях лицензии Creative Commons на его веб-сайте <https://www.inventwithpython.com/>.

Кошка автора книги — Зофи — весит 5 килограммов.

О техническом редакторе

Кеннет Лав (Kenneth Love) — программист, преподаватель и организатор конференций. Он — соавтор Django и член сообщества PSF (Python Software Foundation). В настоящее время работает техническим руководителем и инженером-программистом в O'Reilly Media.

Благодарности

Несправедливо, что на обложке этой книги стоит только мое имя. Эта книга никогда не появилась бы без поддержки многих людей. Хочу поблагодарить своего издателя, Билла Поллока (Bill Pollock), а также всех моих редакторов: Фрэнсис Со (Frances Saux), Энни Чой (Annie Choi), Мег Снеерингер (Meg Sneeringer) и Яна Кэша (Jan Cash). Также хочу сказать спасибо редактору Морин Форис (Maureen Forys), выпускающему редактору Энн Мэри Уокер (Anne Marie Walker) и исполнительному редактору No Starch Press Барбаре Йен (Barbara Yien). Я очень признателен Джошу Эллингсону (Josh Ellingson) за еще одну замечательную иллюстрацию для обложки. Моя благодарность техническому редактору Кеннету Лаву (Kenneth Love) и всем остальным замечательным друзьям, с которыми я познакомился в сообществе Python.

Введение



Hello, world! В конце 1990-х я только начинал программировать, мечтал стать настоящим хакером и листал последние выпуски журнала «2600: The Hacker Quarterly». Однажды я набрался смелости и посетил ежемесячное собрание программистов в моем городе; меня поразило, насколько знающими казались все окружающие. (Позднее я понял, что у многих самоуверенности было больше, чем реальных знаний.) Весь вечер я поддакивал тому, что говорили другие, стараясь не отставать от них. Покидая встречу, я твердо вознамерился посвятить все свободное время изучению компьютеров, программирования и сетевой безопасности, чтобы достойно поучаствовать во встрече в следующем месяце.

Но на следующем собрании я все так же кивал и чувствовал себя болваном по сравнению со всеми остальными. Я снова решил: надо учиться, чтобы стать «достаточно умным» и не отставать от окружающих. Месяц за месяцем я корпел над книгами, но всегда чувствовал себя отстающим. Я начал осознавать, насколько необъятна эта область, и беспокоился, что никогда не буду знать достаточно.

Я знал о программировании больше, чем мои друзья, но безусловно недостаточно для того, чтобы получить должность разработчика. В 1990-е годы Google, YouTube и «Википедии» еще не было. Но даже если бы эти ресурсы были доступны, мне было бы сложно пользоваться ими: я просто не знал, что изучать дальше. Но все это время я учился писать программу «Hello, world!» на разных языках программирования и по-прежнему осознавал, что толком не двигаюсь вперед. Я не понимал, как подняться выше базового уровня.

Разработка вовсе не ограничивается циклами и функциями. Но после того, как вы ознакомитесь со вводным курсом или прочитаете книгу по программированию

начального уровня, поиск новой информации приведет вас к очередному учебнику, посвященному написанию «Hello, world!». Программисты часто называют этот период «пустыней отчаяния»: время, когда вы бесцельно блуждаете по разным учебным материалам, ощущая, что топчетесь на месте. Вы превзошли начальный уровень, но вам еще не хватает опыта для более сложных тем.

Блуждающие в пустыне испытывают сильный «синдром самозванца». Вы не чувствуете себя «настоящим» программистом и не умеете создавать такой код, как «настоящие» программисты. Я писал эту книгу именно для такой аудитории. Если вы изучили основы Python, моя книга поможет вам стать более эффективным разработчиком и преодолеть отчаяние.

Для кого написана эта книга и почему

Книга предназначена для читателей, которые изучили базовый курс языка Python и хотят узнать больше. Базовые знания могут дать моя предыдущая книга «Automate the Boring Stuff with Python» (No Starch Press, 2019)¹, книга Эрика Мэтиза (Eric Matthes) «Python Crash Course» (No Starch Press, 2019)² или какой-нибудь сетевой курс.

Возможно, эти учебники вызвали у вас интерес к программированию, но вам все равно не хватает знаний. Если вы чувствуете, что еще не достигли профессионального уровня, и не знаете, как его достичь, то эта книга написана для вас. А может быть, вы знакомы с другим языком программирования, а теперь хотите переключиться на Python и его экосистему без изучения основ уровня «Hello, world!». В таком случае вам наверняка не захочется читать сотни страниц с объяснением базового синтаксиса; вместо этого достаточно просмотреть статью «Learn Python in Y Minutes» (<https://learnxinyminutes.com/docs/python/>) или страницу Эрика Мэтиза (Eric Matthes) «Python Crash Course — Cheat Sheet» (<https://ehmatthes.github.io/pcc/cheatsheets/README.html>).

О книге

Книга более глубоко знакомит вас с синтаксисом Python, но не ограничивается этим. Я рассказываю о командной строке и средствах командной строки, используемых профессиональными разработчиками: системах форматирования кода, статических анализаторах и системах контроля версий. Я объясняю, какие аспекты делают ваш код более удобочитаемым и как писать более чистый и понятный

¹ Свейгарт Э. Автоматизация рутинных задач с помощью Python.

² Мэтиз Э. Изучаем Python: программирование игр, визуализация данных, веб-приложения. 3-е изд. — СПб.: Питер, 2017.

код. В книге представлен ряд программ, чтобы вы увидели применение этих принципов в реальном коде. И хотя это не учебник компьютерной теории, здесь также рассматривается анализ алгоритмов в нотации «О-большое» и объектно-ориентированная парадигма.

Никакая книга не превратит вас в профессионального разработчика, но я надеюсь, что мой труд расширит ваши познания в этой области. Я представлю некоторые темы, которые обычно познаются лишь на опыте, полученном тяжелой практикой. Прочитав эту книгу, вы приобретете прочную основу, которая позволит вам с большей уверенностью браться за новые задачи.

И хотя я рекомендую читать главы этой книги последовательно, ничто не мешает вам переходить к тем главам, которые вас интересуют больше.

Часть I. Первые шаги

- **Глава 1. Обработка ошибок и обращение за помощью.** Вы узнаете, как эффективно задавать вопросы и самостоятельно находить ответы. Также вы научитесь читать сообщения об ошибках и освоите этикет обращения за помощью в интернете.
- **Глава 2. Подготовка среды и командная строка.** О том, как работать в режиме командной строки и как настроить среду разработки и переменную PATH.

Часть II. Передовые практики, инструменты и методы

- **Глава 3. Форматирование кода при помощи Black.** В этой главе рассматривается руководство по стилю PEP 8 и форматирование кода для улучшения его удобочитаемости. Вы узнаете, как автоматизировать этот процесс при помощи инструмента форматирования кода Black.
- **Глава 4. Выбор понятных имен.** Правила выбора имен переменных и функций для улучшения удобочитаемости кода.
- **Глава 5. Поиск запахов в коде.** Некоторые потенциальные тревожные признаки, которые могут указывать на наличие ошибок в вашем коде.
- **Глава 6. Написание питонического кода.** Некоторые способы создания идиоматического кода Python и признаки, присущие питоническому коду.
- **Глава 7. Жаргон программистов.** Технические термины, используемые в области программирования, и термины, которые нередко вызывают путаницу.
- **Глава 8. Часто встречающиеся ловушки Python.** Типичные источники недоразумений и ошибок в языке Python, способы их исправления и стратегии программирования, которых лучше избегать.

- **Глава 9. Экзотические странности Python.** Некоторые экзотические особенности языка Python, включая интернирование строк и пасхалки, которые могут остаться незамеченными. Разобравшись с тем, почему некоторые типы данных и операторы ведут себя неожиданным образом, вы начнете лучше понимать, как работает Python.
- **Глава 10. Написание эффективных функций.** Глава рассказывает, как структурировать ваши функции для достижения наибольшей практичности и удобочитаемости. Вы узнаете о синтаксисе аргументов * и **, о достоинствах и недостатках больших и малых функций и средствах функционального программирования — таких как лямбда-функции.
- **Глава 11. Комментарии, doc-строки и аннотации типов.** Вы узнаете, насколько важны части вашей программы, не содержащие программного кода, и каково их влияние на сопровождение, с какой частотой следует писать комментарии и doc-строки и как сделать их более содержательными. Также в главе обсуждаются аннотации типов и использование статических анализаторов (таких как MuPy) для выявления ошибок.
- **Глава 12. Git и организация программных проектов.** Система контроля версий Git предназначена для записи истории изменений, вносимых в исходный код, и восстановления предыдущих версий работы или определения точки, в которой ошибка появилась впервые. Также я немного расскажу о структурировании файлов с кодом ваших проектов с использованием программы Cookiecutter.
- **Глава 13. Измерение быстродействия и анализ сложности алгоритмов.** В этой главе рассказано, как объективно измерить скорость выполнения вашего кода при помощи модулей `timeit` и `cProfile`. Кроме того, мы рассмотрим нотацию «О-большое» и способы прогнозирования снижения быстродействия кода с ростом объема обрабатываемых данных.
- **Глава 14. Проекты для тренировок.** Я расскажу о нескольких полезных методах, а затем с их помощью мы напишем пару игр командной строки: головоломку с перемещением дисков «Ханойская башня» и классическую игру «Четыре в ряд» для двух игроков.

Часть III. Объектно-ориентированный Python

- **Глава 15. Объектно-ориентированное программирование и классы.** Здесь мы определим роль объектно-ориентированного программирования (ООП), которое часто понимают неправильно. Многие разработчики злоупотребляют средствами ООП в своем коде, потому что считают, что так поступают все, но это лишь усложняет исходный код. Вы научитесь использовать классы, но что еще важнее, вы узнаете, когда следует и когда не следует их применять.

- **Глава 16. Объектно-ориентированное программирование и наследование.** Наследование классов и его полезность для повторного использования кода.
- **Глава 17. ООП в Python: свойства и dunder-методы.** Средства объектно-ориентированного проектирования, специфические для Python: свойства, dunder-методы и перегрузка операторов.

Путешествие в мир программирования

Попытки новичка стать профессиональным программистом часто выглядят чем-то вроде попытки напиться из пожарного шланга. При таком изобилии ресурсов вы начинаете беспокоиться, что тратите время на неэффективные учебники.

После того как вы прочтете эту книгу (или даже во время чтения), я рекомендую ознакомиться со следующими вводными материалами.

- «Python Crash Course» (No Starch Press, 2019) Эрика Мэттиса (Eric Matthes)¹ — книга для начинающих, но благодаря ее ориентации на конкретные проекты даже опытный программист почувствует вкус использования библиотек Python Pygame, matplotlib и Django.
- В своей книге «Impractical Python Projects» (No Starch Press, 2018) Ли Воган (Lee Vaughan)² проектным методом совершенствует ваши навыки Python. Под его руководством вы построите несколько занимательных программ, которые станут для вас отличной тренировкой.
- «Serious Python» (No Starch Press, 2018) Джульена Данжу (Julien Danjou)³ описывает, что надо предпринять любителю-энтузиасту для превращения в квалифицированного разработчика, который применяет передовые практики и пишет хорошо масштабируемый код.

Впрочем, технические аспекты Python — всего лишь одна из его сильных сторон. Язык программирования сформировал разностороннее сообщество, усилиями которого была создана удобная, доступная документация и система поддержки, превосходящая любую другую экосистему. На ежегодной конференции PyCon наряду со многими региональными конференциями предлагаются многочисленные лекции для программистов разных уровней квалификации. Организаторы

¹ Мэттис Э. Изучаем Python: программирование игр, визуализация данных, веб-приложения. 3-е изд. — СПб.: Питер.

² Воган Л. «Непрактичный» Python. Занимательные проекты для тех, кто хочет поумнеть.

³ Данжу Дж. Путь Python. Черный пояс по разработке, масштабированию, тестированию и развертыванию. — СПб.: Питер.

PyCon предоставляют бесплатный доступ к этим материалам на <https://pyvideo.org/>. На странице **Tags** можно легко найти доклады по темам, интересующим вас.

Если вы захотите глубже изучить расширенные возможности синтаксиса и стандартной библиотеки Python, я рекомендую следующие книги:

- «Effective Python» (Addison-Wesley Professional, 2019) Бретта Слаткина (Brett Slatkin)¹ — впечатляющая подборка передовых практик и языковых средств Python;
- «Python Cookbook» (O'Reilly Media, 2013) Дэвида Бизли (David Beazley) и Брайана К. Джонса (Brian K. Jones)² — обширный список фрагментов кода, которые помогут обновить репертуар любого начинающего разработчика Python;
- «Fluent Python» (O'Reilly Media, 2021) Лучано Рамальо (Luciano Ramalho)³ — капитальный труд для исследования тонкостей языка Python. И хотя почти 800-страничный том выглядит устрашающе, вы не пожалеете о потраченном времени.

Желаю удачи в вашем путешествии в мир программирования. Итак, за дело!

¹ Слаткин Б. Секреты Python. 59 рекомендаций по написанию эффективного кода.

² Бизли Д., Джонс Б. Python. Книга рецептов.

³ Рамальо Л. Python. К вершинам мастерства.

ЧАСТЬ I

ПЕРВЫЕ ШАГИ

1

Обработка ошибок и обращение за помощью



Пожалуйста, не наделяйте компьютеры человеческими качествами; их это очень сильно раздражает. Когда компьютер выдает сообщение об ошибке, это происходит вовсе не потому, что он на вас обиделся. Компьютеры — самые сложные инструменты, с которыми большинству из нас придется работать в жизни, и все же это всего лишь инструменты.

Тем не менее легко обвинить в своих ошибках инструменты. Так как учиться программировать многим из нас в основном приходится самостоятельно, часто мы чувствуем себя неудачниками — приходится неоднократно обращаться к интернету за информацией, хотя мы изучаем Python уже несколько месяцев. Но даже профессиональные разработчики обращаются к интернету или документации, чтобы найти ответы на вопросы по поводу программного обеспечения.

Если вы не располагаете финансовыми или какими-то другими ресурсами, чтобы нанять преподавателя, который ответит на ваши вопросы, выбора не остается: компьютер, поисковые системы в интернете и сила духа. К счастью, на ваши вопросы почти наверняка уже кто-нибудь отвечал. Умение самостоятельно искать ответы для программиста гораздо важнее знания любых алгоритмов или структур данных. В этой главе мы расскажем, как развить этот важнейший навык.

Как понять сообщения об ошибках Python

Когда программисты сталкиваются с лавиной технической информации в сообщениях об ошибках, им прежде всего хочется полностью ее проигнорировать. Но эти

сообщения содержат ответ, что же не так с вашей программой. Процесс поиска информации состоит из двух этапов: анализа трассировки и поиска текста сообщения об ошибке в интернете.

Анализ трассировки

Программы Python аварийно завершаются, когда в коде возникает исключение, не обработанное командой `except`. В таком случае Python выдает сообщение об исключении и *трассировку* (также называемую *трассировкой стека*). Трассировка показывает, в какой точке вашей программы произошло исключение и последовательность вызовов функций, которые к этой точке привели.

Чтобы потренироваться в чтении трассировки, введите следующую программу (она содержит ошибку) и сохраните ее под именем `abcTraceback.py`. Номера строк приведены только для удобства, они не являются частью программы.

```
1. def a():
2.     print('Start of a()')
3.     b() # Вызов b(). ❶
4.
5. def b():
6.     print('Start of b()')
7.     c() # Вызов c(). ❷
8.
9. def c():
10.    print('Start of c()')
11.    42 / 0 # Порождает ошибку деления на ноль. ❸
12.
13. a() # Вызов a().
```

В этой программе функция `a()` вызывает `b()` ❶, которая вызывает функцию `c()` ❷. Внутри `c()` выражение `42 / 0` ❸ вызывает ошибку деления на ноль. Если вы запустите эту программу, результат должен выглядеть так:

```
Start of a()
Start of b()
Start of c()
Traceback (most recent call last):
  File "abcTraceback.py", line 13, in <module>
    a() # Call a().
  File "abcTraceback.py", line 3, in a
    b() # Call b().
  File "abcTraceback.py", line 7, in b
    c() # Call c().
  File "abcTraceback.py", line 11, in c
    42 / 0 # Порождает ошибку деления на ноль.
ZeroDivisionError: division by zero
```

Проанализируем трассировку строку за строкой, начиная с этой:

```
Traceback (most recent call last):
```

Сообщение указывает, что далее идет трассировка. Текст `most recent call last` означает, что вызовы функций перечисляются по порядку, начиная с первой и заканчивая самой последней.

В следующей строке приводится вызов первой функции в трассировке:

```
File "abcTraceback.py", line 13, in <module>
    a() # Вызов a().
```

Следующие две строки содержат *сводку кадра*; в них выводится информация, хранящаяся внутри объекта кадра. При вызове функции данные локальных переменных, а также точка в коде, в которую должно быть возвращено управление после вызова функции, сохраняются в объекте кадра. В объектах кадра хранятся локальные переменные и другие данные, связанные с вызовами функций. Объекты кадров создаются при вызове функции и уничтожаются при возвращении управления. В трассировке выводится сводка для каждого кадра на пути к фатальному сбою. Мы видим, что вызов функции в строке 13 находится в строке `abcTraceback.py`, а текст `<module>` сообщает, что строка находится в глобальной области видимости. Далее строка 13 выводится с отступом из двух пробелов.

В следующих четырех строках выводятся сводки следующих двух кадров:

```
File "abcTraceback.py", line 3, in a
    b() # Call b().
File "abcTraceback.py", line 7, in b
    c() # Call c().
```

Из текста `line 3, in a` мы видим, что функция `b()` была вызвана в строке 3 внутри функции `a()`, что привело к вызову функции `c()` в строке 7 внутри функции `b()`. Обратите внимание: вызовы `print()` в строках 2, 6 и 10 в трассировке не выводятся, несмотря на то что они были выполнены до вызовов функций. В трассировку включаются только строки с вызовами функций, приведших к исключению.

В последней сводке кадра выводится строка, которая стала причиной необработанного исключения. За ней следует имя и сообщение исключения:

```
File "abcTraceback.py", line 11, in c
    42 / 0 # Порождает ошибку деления на ноль.
ZeroDivisionError: division by zero
```

Обратите внимание: номер строки в трассировке указывает, где была обнаружена ошибка. Источник ошибки находится где-то до этой строки.

Сообщения об ошибках часто оказываются слишком короткими и невразумительными; три слова «деление на ноль» ничего не скажут тому, кто не знает, что деление на ноль невозможно с точки зрения математики, и это распространенная программная ошибка. В этой программе найти ошибку несложно. Если взглянуть на строку кода в сводке кадра, становится ясно, что ошибка деления на ноль возникла из-за выражения `42 / 0`.

Но рассмотрим более сложный случай. Введите следующий фрагмент в текстовом редакторе и сохраните его под именем `zeroDivideTraceback.py`:

```
def spam(number1, number2):  
    return number1 / (number2 - 42)  
  
spam(101, 42)
```

При запуске этой программы результат должен выглядеть так:

```
Traceback (most recent call last):  
  File "zeroDivideTraceback.py", line 4, in <module>  
    spam(101, 42)  
  File "zeroDivideTraceback.py", line 2, in spam  
    return number1 / (number2 - 42)  
ZeroDivisionError: division by zero
```

Сообщение об ошибке то же, но деление на ноль в `return number1 / (number2 - 42)` не столь очевидно. По оператору `/` можно заключить, что выполняется деление, а выражение `(number2 - 42)` должно быть равно 0. Отсюда можно сделать вывод, что в функции `spam()` происходит сбой, когда параметр `number2` равен 42.

Иногда трассировка может сообщить, что ошибка находится в строке, расположенной после причины ошибки. Например, в следующей программе в первой строке отсутствует закрывающая круглая скобка:

```
print('Hello.'  
print('How are you?')
```

Но из сообщения об ошибке для этой программы следует, что проблема находится во второй строке:

```
File "example.py", line 2  
    print('How are you?')  
    ^  
SyntaxError: invalid syntax
```

Дело в том, что интерпретатор Python не заметил синтаксическую ошибку до того, как была прочитана вторая строка. Трассировка может сообщить, где обнаружена проблема, но это место не всегда то самое, где находится реальная причина ошибки.

Если сводка кадра не дает достаточной информации для обнаружения ошибки или если истинная причина ошибки находится в предыдущей строке, которая не приводится в трассировке, вам придется выполнять программу в пошаговом режиме в отладчике или просматривать сообщения в журнале. Все это потребует значительного времени. Поиск сообщения об ошибке в интернете иногда намного быстрее предоставит необходимую информацию о проблеме.

Поиск сообщений об ошибках

Часто сообщения об ошибках настолько коротки, что они даже не содержат полных предложений. Программисты регулярно сталкиваются с ними, поэтому они должны быть напоминаниями, а не исчерпывающими объяснениями. Если вы встречаете сообщение об ошибке впервые, скопируйте его в поисковую систему. С большой долей вероятности вы получите подробное описание того, что означает ошибка и каковы ее вероятные причины.

На рис. 1.1 показаны результаты поиска по строке **python "ZeroDivisionError: division by zero"**. Заключение сообщения об ошибке в кавычки помогает найти точную фразу, а добавление слова *python* способствует сужению поиска.

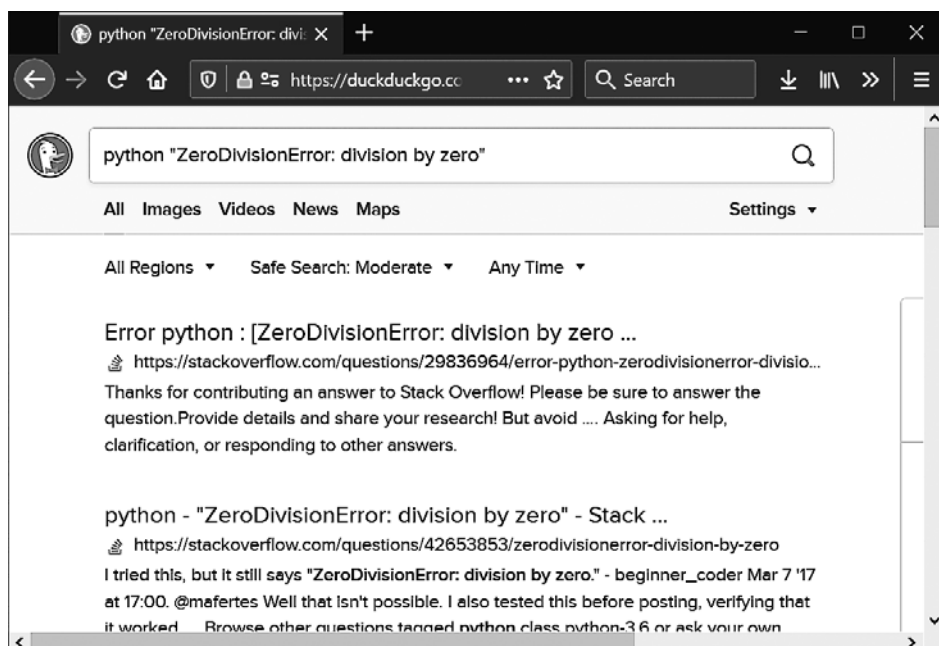


Рис. 1.1. Копирование сообщения об ошибке в поисковую систему ускорит поиск объяснений и решений

Поиск сообщений об ошибках — вполне нормальное дело. Трудно ожидать, чтобы разработчик запомнил все возможные сообщения об ошибках для языка программирования. Для профессионалов поиск ответов в интернете давно стал частью повседневной работы.

Возможно, из поиска стоит исключить фрагменты, относящиеся непосредственно к вашему коду. Для примера рассмотрим следующее сообщение об ошибке:

```
>>> print(employeeRecord)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'employeeRecord' is not defined ❶
>>> 42 - 'hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'int' and 'str' ❷
```

В этом примере допущена опечатка в имени переменной `employeeRecord`, что приводит к ошибке ❶. Так как идентификатор `employeeRecord` в сообщении `NameError: name 'employeeRecord' is not defined` связан с вашим кодом, при поиске стоит использовать строку `python "NameError: name" "is not defined"`. В последней строке часть `'int' and 'str'` в сообщении об ошибке ❷ явно относится к значениям `42` и `'hello'`, так что усечение строки поиска до `python "TypeError: unsupported operand type(s) for" "is not defined"` позволит исключить фрагменты, относящиеся к вашему коду. Если поиск не даст полезных результатов, попробуйте включить полное сообщение об ошибке.

Предотвращение ошибок при помощи статического анализатора

Лучший способ исправления ошибок — не допускать их. *Статические анализаторы* (linters) — приложения, которые анализируют исходный код и предупреждают вас о потенциальных ошибках. Хотя статический анализатор не обнаруживает все ошибки, *статический анализ* (проверка исходного кода без его выполнения) помогает выявить типичные ошибки, возникающие из-за опечаток. (В главе 11 рассказано, как использовать рекомендации типов для статического анализа.) Многие текстовые редакторы и интегрированные среды разработки (IDE) включают статический анализатор, который работает в фоновом режиме и способен выявлять проблемы в реальном времени (рис. 1.2).

Почти моментальные уведомления от статического анализатора сильно повышают эффективность программирования. Без них вам пришлось бы запустить программу, увидеть произошедший сбой, прочитать трассировку, а затем найти строку в исходном коде, чтобы исправить ошибку. А если вы допустите несколько опечаток, то вам придется запускать цикл «запуск — исправление» для каждой из них, поскольку

он находит лишь по одной ошибке за раз. Статический анализ способен выявить сразу несколько ошибок, и это происходит прямо в редакторе — вы видите строки с ошибками.

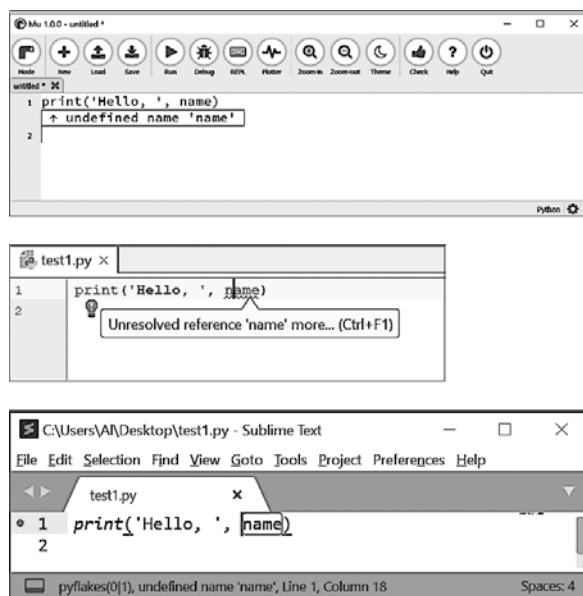


Рис. 1.2. Статический анализатор указывает на не определенную переменную в Mu (наверху), PyCharm (в середине) и Sublime Text (внизу)

Возможно, ваш редактор или IDE не поддерживает статического анализа, но если в нем поддерживаются плагины, статический анализатор почти всегда можно подключить. Обычно такие плагины используют для проведения статического анализа модуль Pylakes или какой-то другой.

Чтобы установить Pylakes, откройте страницу <https://pypi.org/project/pylakes/> или выполните команду `pip install --user pylakes`. Поверьте, дело того стоит.

ПРИМЕЧАНИЕ

В системе Windows можно выполнять команды `python` и `pip`. Но в macOS и Linux эти имена подходят только для Python версии 2, и вместо них следует использовать команды `python3` и `pip3`. Помните об этом, когда вы встречаете `python` или `pip` в книге.

В IDLE (среда, включенная в поставку Python) нет ни статического анализатора, ни возможности его установки.

Как обратиться за помощью по программированию

Если поиск в интернете и статические анализаторы не помогли с решением вашей проблемы, стоит обратиться за помощью к сообществу в интернете. Но существует определенный этикет при обращениях за советом. Если опытные разработчики готовы бесплатно ответить на ваш вопрос, позаботьтесь о том, чтобы эффективно использовать их время.

Обращение за помощью к незнакомым людям всегда должно быть последней мерой. Могут пройти часы и дни, пока кто-нибудь ответит на ваш вопрос — если вы вообще получите ответ. Гораздо быстрее поискать в интернете других людей, которые уже ответили на ваш вопрос, и прочитать их ответы. Электронная документация и поисковые системы создавались как раз для того, чтобы облегчить поиск ответов, за которыми в противном случае пришлось бы обращаться к людям. Но если все иные возможности исчерпаны и вам все-таки приходится обращаться с вопросом к людям, старайтесь избегать некоторых распространенных ошибок.

- Не спрашивайте, можно ли задать вопрос, а просто задайте его.
- Не обозначайте суть вопроса намеками, формулируйте конкретно.
- Не задавайте вопрос на неподходящем форуме или веб-сайте.
- Не используйте неконкретный заголовок или тему сообщения — например, «У меня проблема» или «Помогите, пожалуйста».
- Не пишите «Моя программа не работает», объясните, как она должна работать.
- Не храните в секрете полные сообщения об ошибках.
- Не ленитесь опубликовать ваш код.
- Не приводите плохо отформатированный код.
- Не замалчивайте, что вы уже пытались сделать.
- Не скрывайте информацию об операционной системе или версии.
- Не просите написать программу за вас.

Этот список приведен не просто для красоты; неправильная коммуникация мешает людям помочь вам. Первое, о чем следует просить, — запустить код и попытаться воспроизвести вашу проблему. Для этого тому, кто решит вам помочь, понадобится как можно больше информации о вашем коде, компьютере и намерениях. Гораздо чаще информации оказывается слишком мало, а не слишком много. В нескольких следующих разделах я расскажу, что можно сделать для предотвращения распространенных ошибок. Мои рекомендации относятся к случаю, когда вы публикуете свои вопросы на интернет-форуме, однако они актуальны и тогда, когда вы отправляете вопросы адресно одному или нескольким людям.

Избегайте лишних разговоров, предоставляйте информацию заранее

Если вы обращаетесь к кому-то лично, фраза «Можно ли задать вам вопрос?» станет коротким и любезным способом узнать, доступен ли адресат. Но на интернет-форумах эксперт может ответить не сразу, а тогда, когда у него появится время. Так как переписка иногда растягивается на часы, лучше предоставить всю информацию, которая может понадобиться, в исходном сообщении, а не спрашивать каждый раз разрешения задать вопрос. Если вам не ответили, стоит скопировать эту информацию на другой форум.

Формулируйте свой вопрос как вопрос

Когда вы объясняете свою проблему, иногда возникает иллюзия, что эксперты уже знают, о чем идет речь. Но программирование — весьма обширная область; может оказаться, что у них нет опыта конкретно в той области, которая интересует вас. Поэтому так важно сформулировать ваш вопрос в форме вопроса. Хотя предложения, начинающиеся со слов «Мне хотелось бы...» или «Мой код не работает...», могут намекать на суть, обязательно задайте конкретный вопрос — то есть буквально предложение, завершающееся вопросительным знаком. В противном случае может быть непонятно, о чем вы спрашиваете.

Задавайте вопросы на подходящем веб-сайте

Если вы начнете спрашивать о Python на форуме JavaScript или об алгоритмах — в списке рассылки, посвященном сетевой безопасности, вряд ли это принесет пользу. В большинстве случаев в списках рассылки и интернет-форумах имеется перечень часто задаваемых вопросов (FAQ) или описание обсуждаемых тем. Например, список рассылки `python-dev` посвящен проектированию языка Python и не является общим списком рассылки для получения помощи. Веб-страница <https://www.python.org/about/help/> поможет вам найти подходящий ресурс для запроса о языке Python.

Включите краткое описание вопроса в заголовок

Одно из главных преимуществ публикации вопроса на интернет-форуме заключается в том, что тот, у кого тот же вопрос возникнет позже, сможет найти ответ при помощи поиска в интернете. Обязательно включите заголовок с кратким описанием вопроса, чтобы упростить работу поисковых систем. Заголовок типа «Помогите, пожалуйста» или «Почему мой код не работает?» слишком неопределенный. Если вы просите совета по электронной почте, информация в строке — это то, на что эксперт обратит внимание, когда будет просматривать папку входящих сообщений.

Объясните, что должен делать ваш код

В вопросе «Почему моя программа не работает?» отсутствует важнейшая информация: что должна делать ваша программа. А тот, кто прочитает вопрос, не знает ваших намерений. Даже в формулировке «Почему я получаю эту ошибку?» полезно указать конечную цель вашей программы. В некоторых случаях вам могут ответить, что вы выбрали не тот путь и вам проще начать все заново, вместо того чтобы тратить время на попытки что-то исправить.

STACK OVERFLOW И ПОСТРОЕНИЕ АРХИВА ОТВЕТОВ

Stack Overflow — популярный веб-сайт для получения ответов на вопросы по программированию, но многие новички опасаются пользоваться им. Известно, что модераторы Stack Overflow безжалостно закрывают вопросы, которые не соответствуют их строгим рекомендациям. Тем не менее существует веская причина для поддержания столь жесткой дисциплины на Stack Overflow.

Сайт создавался не столько для получения ответов на вопросы, сколько для построения архива вопросов по программированию вместе с ответами на них. Как следствие, вопросы должны быть конкретными, уникальными и не основанными на субъективных мнениях. Кроме того, они должны быть подробными и хорошо сформулированными, чтобы пользователям поисковых систем было проще найти их. (Ситуация для программистов до появления Stack Overflow легла в основу юмористического комикса XKCD «Wisdom of the Ancients» по адресу <https://xkcd.com/979/>.) Тридцать ответов на один и тот же вопрос не только неэффективно расходуют усилия добровольных экспертов с сайта, но сбивают с толку пользователей поисковых систем множественными результатами. Вопросы должны иметь конкретные, объективные ответы. Ответ на вопрос «Какой самый лучший язык программирования?» является делом вкуса, он только порождает лишние споры. (Ведь мы уже знаем, что лучший язык программирования — Python.)

Тем не менее неприятно, когда ваш вопрос игнорируют, не ответив. Я рекомендую внимательно прочитать советы этой главы и руководство «How do I ask a good question?» на сайте Stack Overflow по адресу <https://stackoverflow.com/help/how-to-ask/>. Поэтому не стесняйтесь использовать псевдоним, если боитесь задать «глупый» вопрос. Stack Overflow не требует указывать настоящие имена для своих учетных записей. Если вы предпочитаете более непринужденную обстановку, возможно, вам стоит обратиться на страницу <https://reddit.com/r/learnpython/> — это сообщество не так строго относится к вопросам. Тем не менее обязательно прочитайте рекомендации, прежде чем писать на сайт.

Включите полное сообщение об ошибке

Не забудьте скопировать все сообщение об ошибке, включая трассировку. Простое описание вроде «Получаю ошибку выхода за границы диапазона» не содержит достаточной информации, чтобы эксперт смог понять, что же пошло не так. Также укажите, всегда ли вы получаете ошибку или проблема возникает эпизодически. Если вы выявили конкретные обстоятельства, в которых происходит ошибка, стоит и их включить в описание.

Приведите полный код

Наряду с полным сообщением об ошибке и трассировкой предоставьте исходный код всей программы. В этом случае эксперт, у которого вы просите помощь, сможет запустить программу на своей машине в отладчике и посмотреть, что в ней происходит. Всегда приводите *минимальный, полный и воспроизводимый* (MCR — Minimum, Complete, Reproducible) пример, надежно воспроизводящий полученную ошибку. Термин MCR возник при обсуждениях на сайте Stack Overflow, подробнее о нем — на странице <https://stackoverflow.com/help/mcve/>. «Минимальный» означает, что ваш код должен быть по возможности коротким, но при этом воспроизводить возникшую проблему. «Полный» — что пример кода содержит все необходимое для воспроизведения ошибки. «Воспроизводимый» означает, что ваш пример кода надежно воспроизводит описанную проблему.

Но если вся ваша программа содержится в одном файле, отправить ее помощнику будет несложно. Только проследите за тем, чтобы она была правильно отформатирована — об этом в следующем разделе.

Правильно отформатируйте свой код

Вы публикуете свой код, чтобы эксперт мог прочитать вашу программу и воспроизвести ошибку. Но помните, что код должен быть правильно отформатирован. Убедитесь, что ваш адресат сможет легко скопировать код и запустить его в том виде, в котором вы его публикуете. Если вы копируете свой код в сообщение электронной почты, учтите, что многие почтовые клиенты удаляют отступы, и ваш код будет выглядеть так:

```
def knuts(self, value):
if not isinstance(value, int) or value < 0:
raise WizCoinException('knuts attr must be a positive int')
self._knuts = value
```

Мало того что эксперту придется потратить время на вставку отступов в каждой строке программы; трудно с ходу сказать, каким отступом должна начинаться каждая строка. Чтобы обеспечить правильное форматирование кода, скопируйте его

на веб-сайт pastebin (например, <https://pastebin.com/> или <https://gist.github.com/>), который сохраняет ваш код с коротким общедоступным URL-адресом (например, <https://pastebin.com/XeU3yusC>). Передать URL намного проще, чем использовать вложенный файл.

Если вы публикуете свой код на веб-сайте (например, <https://stackoverflow.com/> или <https://reddit.com/r/learnpython/>), обязательно используйте средства форматирования в его текстовых полях. Часто при создании отступа из четырех пробелов в строке используется моноширинный *программный шрифт*, который проще читать. Также стоит заключить текст в символы ` (обратный апостроф), чтобы оформить код программным шрифтом. Такие сайты часто включают ссылку на информацию форматирования. Если вы не будете соблюдать эти рекомендации, это может привести к нарушению форматирования исходного кода, отчего он будет выводиться в одной строке:

```
def knuts(self, value):if not isinstance(value, int) or value < 0:raise
WizCoinException('knuts attr must be a positive int') self._knuts = value
```

Также не передавайте свой код на снимках экрана; не пытайтесь сфотографировать экран и отправить изображение. Скопировать код из графического изображения невозможно, к тому же обычно он плохо читается.

Сообщите, что вы уже пытались сделать

Сообщите в вопросе, что вы уже пытались сделать и что у вас получилось в результате. Эта информация избавит эксперта от напрасных попыток пройти по ложному следу, а заодно покажет, что вы прилагали усилия, чтобы решить свою проблему.

Кроме того, так вы продемонстрируете, что обращаетесь за помощью, а не просите других написать программу за вас. К сожалению, нередко студенты просят незнакомых людей в интернете сделать за них домашнюю работу, или предприниматели уговаривают кого-то бесплатно написать «простое приложение». Форумы программистов создаются не для этого.

Опишите свою рабочую конфигурацию

Конфигурация вашего компьютера может повлиять на то, как работает ваша программа и какие ошибки она выдает. Чтобы эксперт мог воспроизвести вашу проблему на своем компьютере, предоставьте следующую информацию.

- Операционная система и ее версия (например, Windows 10 Professional Edition или macOS Catalina).
- Версия Python, использованная для запуска программы (например, Python 3.7 или Python 3.6.6).

- Сторонние модули, использованные в программе, и их версии (например, Django 2.1.1).

Чтобы узнать номера версий установленных сторонних модулей, выполните команду `pip list`. Номер версии принято включать в атрибут `__version__`, как в следующем примере:

```
>>> import django
>>> django.__version__
'2.1.1'
```

Скорее всего, эта информация не потребуется. Но чтобы избежать лишних вопросов, все равно включите эту информацию в исходный пост.

Примеры вопросов

Ниже приведен пример правильно заданного вопроса, который отвечает всем рекомендациям из предыдущего раздела:

«Selenium Webdriver: как мне найти ВСЕ атрибуты элемента?»

В модуле Python Selenium, если у меня есть объект `WebElement`, я могу получить значение любого из его атрибутов вызовом `get_attribute()`:

```
foo = elem.get_attribute('href')
```

Если атрибут с именем `'href'` не существует, возвращается `None`.

Вопрос: как получить список всех атрибутов, имеющихся у элемента? Я не нашел ничего похожего на метод `get_attributes()` или `get_attribute_names()`.

Я использую версию 2.44.0 модуля Selenium для Python».

Этот вопрос был взят со страницы <https://stackoverflow.com/q/27307131/1893164/>. Заголовок обобщает вопрос в одном предложении. Проблема сформулирована в виде вопроса и завершается вопросительным знаком. Если в будущем кто-нибудь прочитает этот заголовок в результатах поиска, он немедленно поймет, имеет это отношение к его проблеме или нет. Код выделен моноширинным программным шрифтом, а текст разбит на абзацы. С первого взгляда понятно, где в этом сообщении находится вопрос: перед ним даже стоит уточнение «Вопрос:». В тексте предполагается, что метод `get_attributes()` или `get_attribute_names()` мог бы присутствовать, но его нет. Также сообщается, что была предпринята попытка найти решение, и приводится предположение о том, как бы мог выглядеть правильный ответ на вопрос. Кроме того, указана версия модуля Selenium на случай,

если она вдруг понадобится. Лучше привести слишком много информации, чем слишком мало.

Итоги

Независимый поиск ответов на вопросы из области программирования — самый важный навык, которым должен владеть каждый программист. Интернет был построен программистами, и в нем имеется множество ресурсов с ответами, которые вам нужны.

Но сначала необходимо разобрать (часто невразумительные) сообщения об ошибках, выдаваемые Python. Если вы не понимаете текст сообщения об ошибке — ничего страшного. Вы все равно можете передать этот текст поисковой системе, чтобы найти описание ошибки на нормальном языке и его наиболее вероятную причину. Трассировка ошибки сообщит, где в вашей программе возникла проблема.

Статический анализатор способен в реальном времени находить опечатки и потенциальные ошибки в процессе написания кода. Статические анализаторы абсолютно необходимы для современной разработки ПО. Если в вашем текстовом редакторе нет статического анализатора или возможности подключить его в виде плагина, подумайте о переходе на другой редактор, в котором такая возможность имеется.

Если вам не удастся найти решение проблемы в интернете, попробуйте опубликовать вопрос на форуме или задайте вопрос эксперту по электронной почте. Чтобы повысить эффективность запроса, в этой главе я рассказал, как оформить «хороший» вопрос по программированию. Задавайте конкретный, хорошо сформулированный вопрос, приведите полный исходный код и подробности сообщения об ошибке, объясните, что вы уже пытались сделать, и назовите используемую версию операционной системы и версию Python. Ответы не только решат вашу проблему, но будут полезны и тем, кто столкнется с той же проблемой в будущем.

Не падайте духом, даже если вам приходится постоянно искать ответы и обращаться за помощью. Программирование — необъятная область, и никто не сможет держать в голове все нюансы. Даже опытные разработчики ежедневно обращаются к интернету за документацией и решениями. Приложите усилия, чтобы научиться искать решения — это поможет вам стать эффективным программистом Python.

2

Подготовка среды и командная строка



Подготовкой среды называется процесс настройки вашего компьютера для программирования. Он включает установку всех необходимых инструментов, настройку их конфигурации и решение всех проблем, возникающих в процессе подготовки. Единственно верного процесса не существует, потому что все используют разные компьютеры с разными операционными системами, версиями операционных систем и версиями интерпретатора Python. Тем не менее в этой главе описаны базовые концепции, которые помогут вам администрировать ваш компьютер, используя командную строку, переменные среды и файловую систему.

Изучение этих концепций и средств может показаться досадной помехой. Вы хотите писать программы, а не возиться с параметрами конфигурации или разбираться в невразумительных консольных командах. Но эти навыки сэкономят ваше время в долгосрочной перспективе. Если вы будете игнорировать сообщения об ошибках или наугад изменять конфигурацию, чтобы ваша система делала то, что нужно, вам удастся замаскировать проблему, но не исправить ее. Но если вы не пожалеете времени и как следует научитесь решать такие задачи, вы сможете предотвратить возникновение проблем в будущем.

Файловая система

Файловая система используется операционной системой для хранения и чтения данных. Файл обладает двумя ключевыми свойствами: именем и путем. Путь задает местоположение файла на компьютере. Например, файл на моем ноутбуке с Windows 10 хранится под именем `project.docx` в `C:\Users\AI\Documents` (это и есть путь).

Часть имени файла после последней точки называется *расширением* файла и сообщает его тип. Расширение в имени файла `project.docx` показывает, что это документ Word, а `Users`, `AI` и `Documents` обозначают названия *папок* (также называемых *каталогами*).

Папки могут содержать файлы и другие папки. Например, файл `project.docx` хранится в папке `Documents`, которая находится в папке `AI`, которая в свою очередь помещена в папку `Users`.

На рис. 2.1 изображена структура папок.

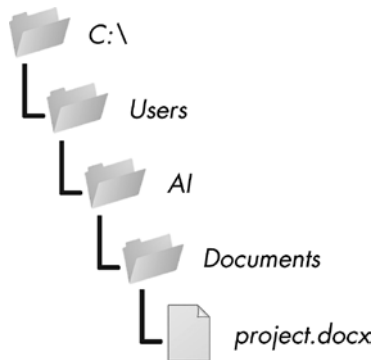


Рис. 2.1. Файл в иерархии папок

Часть пути `C:\` определяет *корневую папку*, которая содержит все остальные папки. В системе Windows корневой папке присваивается имя `C:\`. Также корневую папку называют *диск* `C:`. В macOS и Linux корневая папка обозначается `/`. В этой книге я использую корневую папку для Windows — `C:\`. Если вы будете вводить примеры в интерактивном сеансе macOS или Linux, укажите `/`.

Дополнительные тома — такие, как дисководы DVD или флеш-диски USB, по-разному представлены в разных операционных системах. В Windows — это новые корневые диски, обозначаемые другими буквами, например `D:\` или `E:\`. В macOS — это новые папки внутри папки `/Volumes`. В Linux — новые папки внутри папки `/mnt` (сокращение от `mount`). Учтите, что в Windows и macOS регистр символов в именах папок и файлов игнорируется, но в Linux он значим.

Пути в Python

В Windows для разделения имен папок и файлов используется символ `\` (обратная косая черта), а в macOS и Linux разделителем является символ `/` (косая черта). Чтобы не держать в голове оба варианта для обеспечения межплатформенной совместимости сценариев Python, можно использовать модуль `pathlib` и оператор `/`.

Для импортирования `pathlib` обычно используется команда `from pathlib import Path`. Так как из всех классов `pathlib` чаще всего применяется класс `Path`, эта форма позволяет использовать имя `Path` вместо `pathlib.Path`. Вы можете передать `Path()` строку с именем папки или файла, чтобы создать объект `Path`, представляющий эту папку или файл. Если крайний левый объект в выражении является объектом `Path`, оператор `/` может использоваться для сцепления объектов `Path` или строк. Введите следующие команды в интерактивной оболочке:

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Так как этот код выполняется на компьютере с Windows, `Path()` возвращает объекты `WindowsPath`. В macOS и Linux возвращается объект `PosixPath`. (POSIX — набор стандартов для Unix-подобных операционных систем, эта тема выходит за рамки книги.) Для наших целей различать эти два типа необязательно.

Объект `Path` можно передать любой функции стандартной библиотеки Python, ожидающей получить имя файла. Например, вызов функции `open(Path('C:\\') / 'Users' / 'A1' / 'Desktop' / 'spam.py')` эквивалентен `open(r'C:\Users\A1\Desktop\spam.py')`.

Домашний каталог

Многие пользователи создают папку под названием *домашняя папка* (home folder) или *домашний каталог* (home directory), предназначенную для хранения персональных файлов на компьютере. Чтобы получить объект `Path` для домашней папки, вызовите `Path.home()`:

```
>>> Path.home()
WindowsPath('C:/Users/A1')
```

Домашние каталоги помещаются в определенное место, которое зависит от типа операционной системы.

- В Windows домашние каталоги хранятся в каталоге `C:\Users`.
- На Mac домашние каталоги хранятся в каталоге `/Users`.
- В Linux домашние каталоги часто хранятся в каталоге `/home`.

Ваши сценарии почти наверняка будут иметь разрешение для чтения и записи в файлы в вашем домашнем каталоге, поэтому он идеально подходит для хранения файлов для ваших программ Python.

Текущий рабочий каталог

У каждой программы, выполняемой на компьютере, имеется *текущий рабочий каталог* (CWD, Current Working Directory). Считается, что все имена файлов или пути, имена которых не начинаются с названия корневой папки, хранятся в текущем рабочем каталоге. Хотя термин «папка» считается более современным, стандартным названием является «текущий рабочий каталог» (или просто «рабочий каталог»), а не «текущая рабочая папка».

Текущий рабочий каталог можно получить в виде объекта `Path` при помощи функции `Path.cwd()` и изменить его функцией `os.chdir()`. Введите следующий фрагмент в интерактивной оболочке:

```
>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python38')
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

Здесь в качестве текущего рабочего каталога выбран `C:/Users/Al/AppData/Local/Programs/Python/Python38` ❶, так что имя файла `project.docx` будет относиться к файлу `C:/Users/Al/AppData/Local/Programs/Python/Python38/project.docx`. А если сменить текущий рабочий каталог на `C:/Windows/System32` ❷, имя файла `project.docx` будет относиться к `C:/Windows/System32/project.docx`.

При попытке выбрать несуществующий каталог Python выдает сообщение об ошибке:

```
>>> os.chdir('C:/ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

Ранее для получения текущего рабочего каталога в виде строки использовалась функция `os.getcwd()` из модуля `os`.

Абсолютные и относительные пути

Существуют два способа определения путей к файлам.

- Абсолютный путь — всегда начинается от корневой папки.
- Относительный путь — задается относительно текущего рабочего каталога программы.

Также существуют папки с названием `.` (точка) и `..` (точка, точка). Это не реально существующие папки, а специальные имена, которые могут использоваться в путях. Точка (`.`) вместо имени папки является сокращенным обозначением для текущей папки. Две точки (`..`) обозначают родительскую папку.

На рис. 2.2 представлены примеры папок и файлов. Когда текущим рабочим каталогом назначается `C:\bacon`, относительные пути других файлов и папок задаются так, как показано на иллюстрации.

Префикс `.\` в начале относительного пути не обязателен. Например, `.\spam.txt` и `spam.txt` обозначают один и тот же файл.

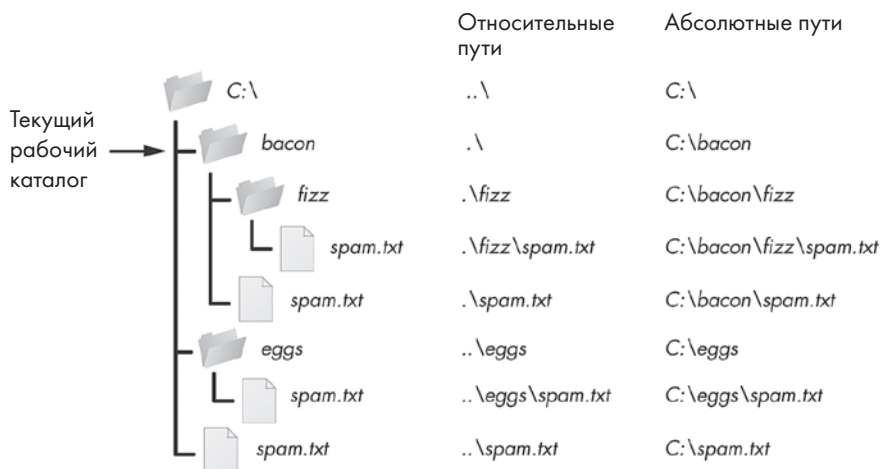


Рис. 2.2. Относительные пути для папок и файлов в рабочем каталоге `C:\bacon`

Программы и процессы

Программой называется любое приложение, которое можно запустить на компьютере: веб-браузер, электронная таблица, текстовый процессор и т. д. *Процесс* представляет собой выполняемый экземпляр программы. Например, на рис. 2.3 показаны пять работающих процессов одной программы-калькулятора.

Процессы существуют независимо друг от друга, даже если они выполняют одну и ту же программу. Например, если вы запустите сразу несколько экземпляров программы Python, каждый процесс имеет собственный набор значений переменных. Каждый процесс (даже если они относятся к одной программе) имеет свой текущий рабочий каталог и значения переменных среды. Как правило, в каждой командной строке выполняется только один процесс (хотя ничто не мешает вам открыть несколько окон командной строки одновременно).



Рис. 2.3. Одна программа-калькулятор запущена в нескольких экземплярах, каждый экземпляр является отдельным процессом

Каждая операционная система использует собственный способ просмотра списка выполняемых процессов. В Windows Диспетчер задач вызывается комбинацией клавиш Ctrl-Shift-Esc. В macOS следует выполнить команду Applications ► Utilities ► Activity Monitor. В Ubuntu Linux комбинация клавиш Ctrl-Alt-Del открывает приложение, также называемое Диспетчером задач (Task Manager). В Диспетчере задач также можно принудительно завершить выполняемый процесс, если он перестает реагировать на действия пользователя.

Командная строка

Командной строкой называется программа текстового режима, в которой можно вводить команды для взаимодействия с операционной системой и запуска программ. Также командную строку иногда называют *интерфейсом командной строки* (CLI, Command Line Interface), *терминалом*, *оболочкой* или *консолью*. Командная строка — альтернатива для графического интерфейса пользователя (GUI, Graphic User Interface), который позволяет вам взаимодействовать с компьютером, не ограничиваясь текстовым интерфейсом. Графический интерфейс представляет информацию визуально, в нем проще выполнять различные операции, чем в режиме командной строки. Многие пользователи считают командную строку слишком сложным инструментом и никогда не обращаются к ней. Эти опасения отчасти

обусловлены нехваткой информации об ее использовании. Если кнопка в графическом интерфейсе хотя бы обозначает место, где следует щелкнуть, то пустое окно терминала не дает подсказки, что именно нужно ввести.

Тем не менее существуют веские причины, для того чтобы освоить командную строку на хорошем уровне. Во-первых, для подготовки среды часто приходится использовать именно ее, а не графические окна. Во-вторых, вводить команды с клавиатуры намного быстрее, чем щелкать мышью в графических окнах. Текстовые команды также более понятны по сравнению с перетаскиванием значка на другой значок. Из-за этого они лучше подходят для автоматизации: вы можете объединить несколько конкретных команд в сценарии для выполнения сложных операций.

Программа командной строки существует в исполняемом файле на вашем компьютере. В этом контексте ее часто называют *программой командной оболочки*. При запуске программы командной оболочки на экране появляется терминальное окно.

- В системе Windows командной оболочкой является файл `C:\Windows\System32\cmd.exe`.
- В macOS — файл `/bin/bash`.
- В Ubuntu Linux — файл `/bin/bash`.

За прошедшие годы программисты написали много разных командных оболочек для операционной системы Unix, включая Bourne Shell (исполняемый файл с именем `sh`) и позднее — Bourne-Again Shell (исполняемый файл с именем `Bash`). Linux использует Bash по умолчанию, тогда как в macOS применяется похожая оболочка Zsh (или Z Shell) в Catalina и более поздних версиях. Из-за другой истории разработки в Windows используется оболочка с именем **Командная строка**. Все эти программы делают одно и то же: открывают окно терминала с текстовым интерфейсом, в котором пользователь вводит команды и запускает программы.

В этом разделе описаны некоторые общие концепции и основные команды командной строки. Вы, конечно, можете изучить массу разнообразных команд и стать настоящим профи, но на самом деле достаточно знать всего около дюжины команд. Точные их имена могут слегка видоизменяться в зависимости от операционной системы, но базовые действия остаются одними и теми же.

Открытие окна терминала

Чтобы открыть окно терминала, выполните следующие действия.

- В Windows щелкните на кнопке Пуск, введите **Командная строка** и нажмите Enter.

- В macOS щелкните на значке Spotlight в правом верхнем углу, введите **Terminal** и нажмите **Enter**.
- В Ubuntu Linux нажмите клавишу **WIN**, чтобы открыть Dash, введите **Terminal** и нажмите **Enter**. Также можно воспользоваться комбинацией клавиш **Ctrl-Alt-T**.

В терминале появляется *приглашение* для ввода команд (по аналогии с интерактивной оболочкой, в которой выводится приглашение `>>>`). В Windows в качестве приглашения используется полный путь к текущей папке:

```
C:\Users\Al>здесь вводятся ваши команды
```

В macOS приглашение состоит из имени компьютера, двоеточия и текущего рабочего каталога, обозначенного тильдой (~); далее идет имя пользователя и знак доллара (\$):

```
Als-MacBook-Pro:~ al$ здесь вводятся ваши команды
```

В Ubuntu Linux приглашение похоже на приглашение macOS, не считая того что оно начинается с имени пользователя и амперсанда (@):

```
al@al-VirtualBox:~$ здесь вводятся ваши команды
```

Во многих книгах и учебниках приглашение командной строки отображается в виде \$ для упрощения примеров. Внешний вид приглашения можно изменять, но эта тема выходит за рамки книги.

Запуск программ из командной строки

Чтобы запустить программу или команду, введите ее имя в командной строке. Давайте запустим программу-калькулятор, входящую в состав стандартных программ операционной системы. Введите в командной строке следующую команду.

- В системе Windows введите **calc.exe**.
- В macOS введите **open -a Calculator**. (Строго говоря, эта команда запускает программу **open**, которая затем запускает программу **Calculator**.)
- В Linux введите **gnome-calculator**.

В Linux в именах программ и команд учитывается регистр символов, но в Windows и macOS регистр игнорируется. Это означает, что в Linux нужно ввести команду **gnome-calculator**, тогда как в Windows допустимо ввести **Calc.exe**, а в macOS — **OPEN -a Calculator**.

Когда вы указываете имена программ в командной строке, это равносильно запуску программы-калькулятора из меню Пуск, Finder или Dash. Имена программ работают как команды, потому что программы `calc.exe`, `open` и `gnome-calculator` находятся в папках, включенных в переменные среды `PATH`. Эта тема более подробно рассматривается в разделе «Переменные среды и `PATH`», с. 60. А пока достаточно сказать, что при вводе имени программы в командной строке оболочка проверяет, существует ли программа с указанным именем в одной из папок, включенных в `PATH`. В Windows оболочка ищет программу в текущем рабочем каталоге (который указан в приглашении), а потом переходит к проверке папок из `PATH`. Чтобы приказать командной строке в macOS и Linux начать с проверки текущего рабочего каталога, необходимо ввести `./` перед именем файла.

Если программа не находится в папке, включенной в `PATH`, у вас есть два варианта.

- Воспользоваться командой `cd` для назначения текущим рабочим каталогом папки, содержащей программу, а затем указать имя программы. Например, можно ввести следующие две команды:

```
cd C:\Windows\System32
calc.exe
```

- Введите полный путь к исполняемому файлу программы. Например, вместо `calc.exe` укажите `C:\Windows\System32\calc.exe`.

В Windows, если имя программы завершается расширением `.exe` или `.bat`, расширение приводить необязательно; если просто ввести имя `calc`, результат будет таким же, как в случае ввода `calc.exe`. Исполняемые программы в macOS и Linux часто не имеют специальных расширений, помечающих их как исполняемые; вместо этого для них устанавливается разрешение исполнения. Дополнительную информацию вы найдете в разделе «Запуск программ Python без командной строки», с. 65.

Аргументы командной строки

Аргументы командной строки представляют собой текстовые фрагменты, которые вводятся после имени команды. Как и аргументы, передаваемые при вызове функций Python, они предоставляют команде дополнительную информацию или указания. Например, при выполнении команды `cd C:\Users` часть `C:\Users` является аргументом команды `cd`, которая сообщает команде, какая папка должна стать текущим рабочим каталогом. Или при запуске сценария Python из окна терминала командой `python yourScript.py` часть `yourScript.py` является аргументом, который сообщает программе `python`, в каком файле следует искать выполняемую последовательность команд.

Ключи командной строки (их также называют флагами или опциями) представляют собой однобуквенные или короткие аргументы командной строки. В системе

Windows ключи командной строки часто начинаются с косой черты (/); в macOS и Linux они начинаются с одного дефиса (-) или двух дефисов (--). Вы уже использовали ключ командной строки `-a` при выполнении команды macOS `open -a Calculator`. В macOS и Linux в ключах командной строки часто различается регистр символов, но в Windows регистр игнорируется. Ключи командной строки разделяются пробелами.

Имена папок и файлов часто используются в качестве аргументов командной строки. Если имя папки или файла содержит пробел, заключите имя в двойные кавычки, чтобы не создавать путаницу в командной строке. Например, если вы хотите сделать текущим каталогом папку с именем **Vacation Photos**, при вводе команды `cd Vacation Photos` командная строка решит, что вы передаете два аргумента, *Vacation* и *Photos*. Поэтому следует ввести команду `cd "Vacation Photos"`:

```
C:\Users\A1>cd "Vacation Photos"
```

```
C:\Users\A1\Vacation Photos>
```

Другой типичный аргумент многих команд — `--help` в macOS и Linux или `/?` в Windows. Они выводят информацию о команде. Например, если выполнить команду `cd /?` в Windows, командная оболочка сообщит, что делает команда `cd`, и выведет сводку аргументов командной строки этой команды:

```
C:\Users\A1>cd /?
```

Вывод имени либо смена текущего каталога.

```
CHDIR [/D] [диск:][путь]
```

```
CHDIR [..]
```

```
CD [/D] [диск:][путь]
```

```
CD [..]
```

.. обозначает переход в родительский каталог.

Команда `CD диск:` отображает имя текущего каталога указанного диска.

Команда `CD` без параметров отображает имена текущих диска и каталога.

Параметр `/D` используется для одновременной смены текущих диска и каталога.

...

Справка сообщает, что команда Windows `cd` также может вводиться по имени `chdir`. (Большинство пользователей не станет вводить `chdir`, когда более короткая команда `cd` делает то же самое.) Квадратные скобки содержат необязательные аргументы. Например, описание `CD [/D] [диск:][путь]` указывает, что с ключом `/D` можно задать диск или путь.

Хотя аргументы `/?` и `--help` для команд выводят информацию, объяснения часто выглядят не совсем понятно и годятся лишь для опытных пользователей. Они вряд ли помогут новичкам. Лучше обратиться к книге или веб-учебнику.

Выполнение кода Python из командной строки с ключом -c

Если вам понадобится небольшой объем временного кода Python, который нужно выполнить один раз, а потом выбросить, передайте ключ `-c` при вызове `python.exe` в Windows или `python3` в macOS и Linux. Выполняемый код следует указать после ключа `-c` в двойных кавычках. Например, введите следующую команду в окне терминала:

```
C:\Users\Al>python -c "print('Hello, world')"  
Hello, world
```

Ключ `-c` хорошо подходит для просмотра результатов одной инструкции Python, когда вы не желаете тратить время на вход в интерактивную оболочку. Например, можно быстро отобразить вывод функции `help()`, а затем вернуться в командную строку:

```
C:\Users\Al>python -c "help(len)"  
Help on built-in function len in module builtins:
```

```
len(obj, /)  
    Return the number of items in a container.
```

```
C:\Users\Al>
```

Выполнение программ Python из командной строки

Программы Python представляют собой текстовые файлы с расширением `.py`. Они не являются исполняемыми файлами; интерпретатор Python читает эти файлы и выполняет содержащиеся в них команды Python. В Windows исполняемый файл интерпретатора называется `python.exe`, а в macOS и Linux — `python3` (файл `python` содержит интерпретатор Python версии 2). При выполнении команды `python yourScript.py` или `python3 yourScript.py` будут выполнены команды Python, хранящиеся в файле с именем `yourScript.py`.

Запуск программы `py.exe`

В системе Windows Python устанавливает программу `py.exe` в папку `C:\Windows`. Эта программа идентична `python.exe`, но она получает дополнительный аргумент командной строки, который позволяет запустить любую версию Python, установленную на вашем компьютере. Команду `py` можно выполнить из любой папки, потому что папка `C:\Windows` включена в переменную среды `PATH`. Если у вас установлено сразу несколько версий Python, команда `py` автоматически запустит новейшую версию из доступных на компьютере. Также можно передать аргумент командной строки `-3` или `-2`, чтобы запустить новейшую из установленных версий: Python 3 или Python 2

соответственно. Еще можно ввести более конкретный номер версии (например, -3.6 или -2.7), чтобы запустить конкретную версию Python. После ключа версии следует передать `py.exe` все те же аргументы командной строки, которые передаются `python.exe`. Выполните следующие команды из командной строки Windows:

```
C:\Users\Al>py -3.6 -c "import sys;print(sys.version)"
3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)]

C:\Users\Al>py -2.7
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Программа `py.exe` удобна в тех случаях, когда на вашем компьютере с Windows установлено несколько версий Python и вы хотите запустить конкретную версию.

Выполнение команд из программы Python

Функция Python `subprocess.run()` из модуля `subprocess` может выполнять команды оболочки из программ Python, а затем выводить результат выполнения команды в виде строки. Например, следующий код выполняет команду `ls -al`:

```
>>> import subprocess, locale
>>> procObj = subprocess.run(['ls', '-al'], stdout=subprocess.PIPE) ❶
>>> outputStr = procObj.stdout.decode(locale.getdefaultlocale()[1]) ❷
>>> print(outputStr)
total 8
drwxr-xr-x  2 al al 4096 Aug  6 21:37 .
drwxr-xr-x 17 al al 4096 Aug  6 21:37 ..
-rw-r--r--  1 al al    0 Aug  5 15:59 spam.py
```

Функции `subprocess.run()` передается список `['ls', '-al']` ❶. Он содержит имя команды `ls`, за которым следуют аргументы в виде отдельных строк. Обратите внимание: с передачей `['ls -al']` функция работать не будет. Вывод команды хранится в виде строки в `outputStr` ❷. Электронная документация функций `subprocess.run()` и `locale.getdefaultlocale()` даст вам лучшее представление о том, как они действуют, причем с ними ваш код будет работать в любой операционной системе, где функционирует Python.

Сокращение объема вводимого текста при помощи автозавершения

Программисты занимаются вводом команд на своих компьютерах по несколько часов в день, поэтому современные средства командной строки предоставляют возможности для сокращения объема вводимого текста. Функция автозавершения

позволяет ввести несколько первых символов имени папки или файла, а затем нажать клавишу **Tab**, чтобы командная оболочка заполнила имя до конца.

Например, если ввести `cd c:\u` и нажать **Tab** в Windows, текущая команда проверит, какие папки или файлы на диске `C:\` начинаются с `u`, и автоматически завершит имя каталога `c:\Users`. Буква `u` нижнего регистра при этом автоматически преобразуется в `U`. (В macOS и Linux автозавершение не исправляет регистр.) Если на диске `C:\` сразу несколько папок, имена которых начинаются на `U`, вы можете перебрать их повторными нажатиями **Tab**. Чтобы сократить количество совпадений, также можно ввести команду `cd c:\us`; тем самым набор возможных вариантов ограничивается папками и файлами, имена которых начинаются с `us`.

Многократное нажатие **Tab** также работает в macOS и Linux. В следующем примере пользователь ввел команду `cd D` и дважды нажал **Tab**:

```
al@al-VirtualBox:~$ cd D
Desktop/  Documents/ Downloads/
al@al-VirtualBox:~$ cd D
```

Двукратное нажатие **Tab** после ввода `D` заставляет оболочку показать все возможные варианты. Оболочка выдает новое приглашение с командой, указанной к настоящему моменту. В этот момент можно ввести, допустим, букву `e` и нажать **Tab**, чтобы оболочка завершила команду `cd Desktop/`.

Автозавершение настолько полезно, что эта функция включена во многие графические интегрированные среды и текстовые редакторы. В отличие от средств командной строки эти GUI-программы обычно отображают небольшое меню под вводимым текстом, чтобы вы могли выбрать нужный вариант для автоматического завершения оставшейся части команды.

Просмотр истории команд

Современные командные оболочки также запоминают введенные команды в *истории команд*. При нажатии клавиши `↑` в терминале командная строка заполняется последней введенной вами командой. Дальнейшие нажатия клавиши `↑` продолжают перебирать более ранние команды, а нажатия клавиши `↓` возвращают к более поздним командам. Если вы хотите отменить команду в приглашении и начать с нового приглашения, нажмите **Ctrl-C**.

В системе Windows можно просмотреть историю команд при помощи команды `doskey/history`. (Странное название программы `doskey` восходит к MS-DOS — операционной системе Microsoft, предшественнику Windows.) В macOS и Linux для истории команд предназначена команда `history`.

Часто используемые команды

В этом разделе приведен короткий список команд, которые чаще других используются в режиме командной строки. Команд и аргументов гораздо больше, но вы можете рассматривать этот список как некий минимум, необходимый для работы с командной строкой.

Аргументы командной строки для команд этого раздела заключены в квадратные скобки. Например, `cd [папка]` означает, что вы вводите команду `cd`, за которой следует имя новой папки.

Определение папок и файлов с использованием универсальных символов

Многим командам в аргументах командной строки должны передаваться имена папок и файлов. Часто такие команды могут получать шаблоны с универсальными символами `*` и `?`, которые определяют набор подходящих файлов. Символ `*` означает любое количество символов, тогда как символ `?` — один произвольный символ. Выражения с универсальными символами `*` и `?` называются *глобальными шаблонами*.

Глобальные шаблоны позволяют определять группы файлов. Например, команда `dir` или `ls` используется для вывода всех файлов и папок в текущем рабочем каталоге. Но если вы хотите просмотреть только файлы Python, команда `dir *.py` или `ls *.py` выведет только файлы с расширением `.py`. Глобальный шаблон `*.py` означает «любая группа символов, за которой следует `.py`»:

```
C:\Users\Al>dir *.py
Volume in drive C is Windows
Volume Serial Number is DFF3-8658

Directory of C:\Users\Al

03/24/2019  10:45 PM                8,399 conwaygameoflife.py
03/24/2019  11:00 PM                7,896 test1.py
10/29/2019  08:18 PM            21,254 wizcoin.py
               3 File(s)             37,549 bytes
               0 Dir(s)  506,300,776,448 bytes free
```

Глобальный шаблон `records201?.txt` означает «символы `records201`, за которыми следует один произвольный символ, а потом `.txt`». Шаблон выберет файлы от `records2010.txt` до `records2019.txt` (а также имена файлов вида `records201X.txt`). Глобальный шаблон `records20???.txt` может включать два любых символа — например, `records2021.txt` или `records20AB.txt`.

Изменение текущего рабочего каталога командой *cd*

Выполнение команды `cd [папка]` заменяет текущий рабочий каталог командной оболочки указанной папкой:

```
C:\Users\Al>cd Desktop
```

```
C:\Users\Al\Desktop>
```

Оболочка выводит текущий рабочий каталог как часть приглашения командной строки, а все папки и файлы, указанные в командах, интерпретируются относительно этого каталога.

Если имя папки содержит пробелы, заключите его в двойные кавычки. Чтобы выбрать в качестве текущего рабочего каталога домашнюю папку пользователя, введите команду `cd ~` в macOS и Linux или команду `cd %USERPROFILE%` в Windows.

В системе Windows, если вы захотите изменить текущий диск, сначала необходимо указать имя диска в отдельной команде:

```
C:\Users\Al>d:
```

```
D:\>cd BackupFiles
```

```
D:\BackupFiles>
```

Для перехода к родительскому каталогу текущего рабочего каталога используйте в качестве имени папки `..`.

```
C:\Users\Al>cd ..
```

```
C:\Users>
```

Вывод содержимого папки командами *dir* и *ls*

В системе Windows команда `dir` выводит список папок и файлов в текущем рабочем каталоге. Команда `ls` делает то же самое в macOS и Linux. Содержимое другой папки выводится командой `dir [другая папка]` или `ls [другая папка]`.

Ключи `-l` и `-a` передают полезные аргументы для команды `ls`. По умолчанию `ls` выводит только имена папок и файлов. Чтобы вывести подробную информацию с размером файла, разрешениями, временными метками последнего изменения и другой информацией, используйте ключ `-l`. По существующим соглашениям операционные системы macOS и Linux считают, что файлы, имена которых начинаются с точки, являются конфигурационными файлами, и скрывают их от нормальных команд. Чтобы команда `ls` выводила все файлы, включая скрытые, используйте ключ `-a`. Чтобы в результатах подробная информация объединялась

с выводом всех файлов, объедините эти ключи: `ls -al`. Вот пример окна терминала для macOS или Linux:

```
al@ubuntu:~$ ls
Desktop    Downloads          mu_code  Pictures  snap      Videos
Documents  examples.desktop  Music    Public    Templates
al@ubuntu:~$ ls -al
total 112
drwxr-xr-x 18 al   al   4096 Aug  4 18:47 .
drwxr-xr-x  3 root root 4096 Jun 17 18:11 ..
-rw-----  1 al   al   5157 Aug  2 20:43 .bash_history
-rw-r--r--  1 al   al    220 Jun 17 18:11 .bash_logout
-rw-r--r--  1 al   al   3771 Jun 17 18:11 .bashrc
drwx----- 17 al   al   4096 Jul 30 10:16 .cache
drwx----- 14 al   al   4096 Jun 19 15:04 .config
drwxr-xr-x  2 al   al   4096 Aug  4 17:33 Desktop
...
```

В системе Windows аналогом `ls -al` является команда `dir`. Пример окна терминала для Windows:

```
C:\Users\Al>dir
Volume in drive C is Windows
Volume Serial Number is DFF3-8658

Directory of C:\Users\Al

06/12/2019  05:18 PM    <DIR>          .
06/12/2019  05:18 PM    <DIR>          ..
12/04/2018  07:16 PM    <DIR>          .android
--snip--
08/31/2018  12:47 AM                14,618 projectz.ipynb
10/29/2014  04:34 PM                121,474 foo.jpg
```

Вывод содержимого вложенных папок командами `dir /s` и `find`

В системе Windows команда `dir /s` выводит содержимое текущего рабочего каталога и его подкаталогов. Например, следующая команда выводит все файлы `.py` в папке `C:\github\ezgmail`, а также во всех ее вложенных папках:

```
C:\github\ezgmail>dir /s *.py
Volume in drive C is Windows
Volume Serial Number is DEE0-8982

Directory of C:\github\ezgmail

06/17/2019  06:58 AM                1,396 setup.py
                        1 File(s)                1,396 bytes

Directory of C:\github\ezgmail\docs
```

```
12/07/2018  09:43 PM                5,504 conf.py
              1 File(s)                5,504 bytes

Directory of C:\github\ezgmail\src\ezgmail

06/23/2019  07:45 PM            23,565 __init__.py
12/07/2018  09:43 PM                56 __main__.py
              2 File(s)            23,621 bytes

Total Files Listed:
              4 File(s)            30,521 bytes
              0 Dir(s)  505,407,283,200 bytes free
```

Команда `find . -name` делает то же самое в macOS и Linux:

```
al@ubuntu:~/Desktop$ find . -name "*.py"
./someSubFolder/eggs.py
./someSubFolder/bacon.py
./spam.py
```

Символ `.` приказывает `find` начать поиск в текущем рабочем каталоге. С ключом `-name` команда `find` ищет папки и файлы по имени. Фрагмент `"*.py"` приказывает `find` искать папки и файлы с именами, соответствующими шаблону `*.py`. Следует заметить, что команда `find` требует, чтобы аргумент после `-name` был заключен в двойные кавычки.

Копирование файлов и папок командами `cp` и `cp`

Чтобы создать дубликат файла или папки в другом каталоге, выполните команду `cp` [*исходный файл или папка*] [*приемная папка*] или `cp` [*исходный файл или папка*] [*приемная папка*]. Пример окна терминала Linux:

```
al@ubuntu:~/someFolder$ ls
hello.py  someSubFolder
al@ubuntu:~/someFolder$ cp hello.py someSubFolder
al@ubuntu:~/someFolder$ cd someSubFolder
al@ubuntu:~/someFolder/someSubFolder$ ls
hello.py
```

Перемещение файлов и папок командами `move` и `mv`

В Windows перемещение исходного файла или папки в другую папку выполняется командой `move` [*исходный файл или папка*] [*приемная папка*]. Команда `mv` [*исходный файл или папка*] [*приемная папка*] делает то же самое в macOS и Linux.

Пример окна терминала Linux:

```
al@ubuntu:~/someFolder$ ls
hello.py  someSubFolder
```



```
al@ubuntu:~/someFolder$ mv hello.py someSubFolder
al@ubuntu:~/someFolder$ ls
someSubFolder
al@ubuntu:~/someFolder$ cd someSubFolder/
al@ubuntu:~/someFolder/someSubFolder$ ls
hello.py
```

Файл `hello.py` переместился из `~/someFolder` в `~/someFolder/someSubFolder`, в исходной папке он отсутствует.

КОРОТКИЕ ИМЕНА КОМАНД

Когда я только начал изучать операционную систему Linux, меня удивило, что известная мне команда `copy` в Linux называлась `cp`. Имя `copy` было намного более понятным, чем `cp`. Стоило ли создавать лаконичное, малопонятное имя ради экономии всего двух вводимых символов?

Со временем у меня появилось больше опыта работы с командной строкой, и я понял, что ответ на этот вопрос — уверенное «да». Мы читаем исходный код чаще, чем пишем его, так что развернутые имена переменных и функций приносят пользу. Но мы вводим команды в командной строке чаще, чем читаем их, поэтому в данном случае справедливо обратное: короткие имена упрощают работу с командной строкой и снижают нагрузку на запястья.

Переименование файлов и папок командами `ren` и `mv`

Команда `ren` [*файл или папка*] [*новое имя*] переименовывает файл или папку в Windows, а команда `mv` [*файл или папка*] [*новое имя*] делает то же самое в macOS и Linux.

Следует заметить, что команда `mv` в macOS и Linux может использоваться для перемещения и переименования файлов. Если указать имя существующей папки во втором аргументе, команда `mv` перемещает туда файл или папку. Если задать имя, которое не соответствует существующему файлу или папке, команда `mv` переименовывает файл или папку. Пример в окне терминала Linux:

```
al@ubuntu:~/someFolder$ ls
hello.py  someSubFolder
al@ubuntu:~/someFolder$ mv hello.py goodbye.py
al@ubuntu:~/someFolder$ ls
goodbye.py  someSubFolder
```

Теперь файл `hello.py` существует под именем `goodbye.py`.

Удаление файлов и папок командами *del* и *rm*

Чтобы удалить файл или папку в Windows, выполните команду `del [файл или папка]`. В macOS и Linux для этого используется команда `rm [файл]` (`rm` — сокращение от `remove`).

Две команды удаления несколько отличаются друг от друга. В Windows выполнение `del` для папки удаляет все содержащиеся в ней файлы, но не вложенные папки. Команда `del` также не удаляет исходную папку; для этого необходимо воспользоваться командой `rd` или `rmdir` (см. «Удаление папок командами `rd` и `rmdir`», с. 59). Кроме того, выполнение команды `del [папка]` не приведет к удалению каких-либо файлов в папках, вложенных в исходную. Чтобы удалить их, выполните команду `del /s /q [папка]`. Ключ `/s` выполняет команду `del` вместе с вложенными папками, а ключ `/q` фактически означает «делай, как я сказал, и не спрашивай подтверждения».

На рис. 2.4 продемонстрированы различия между командами.

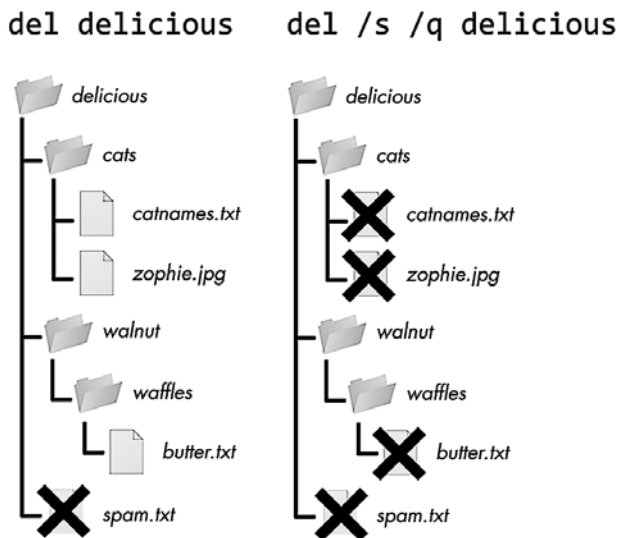


Рис. 2.4. Файлы в этих папках удаляются при выполнении команды `del delicious` (слева) или `del /s /q delicious` (справа)

В macOS и Linux команда `rm` не может использоваться для удаления папок. Впрочем, вы можете выполнить команду `rm -r [папка]` для удаления папки и всего ее содержимого. В системе Windows то же самое делает команда `rd /s /q [папка]` (рис. 2.5).

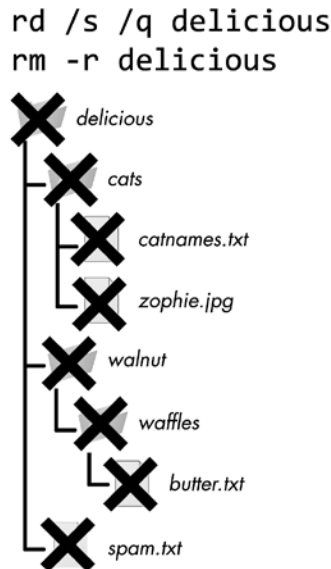


Рис. 2.5. Файлы в этих папках удаляются при выполнении команды `rd /s /q delicious` или `rm -r delicious`

Создание папок командами `md` и `mkdir`

Команда `md [новая папка]` создает новую пустую папку в Windows, а команда `mkdir [новая папка]` делает то же самое в macOS и Linux. Команда `mkdir` также работает в Windows, но `md` вводить быстрее.

Пример в окне терминала Linux:

```
al@ubuntu:~/Desktop$ mkdir yourScripts
al@ubuntu:~/Desktop$ cd yourScripts
al@ubuntu:~/Desktop/yourScripts$ ls ❶
al@ubuntu:~/Desktop/yourScripts$
```

Обратите внимание: только что созданная папка `yourScripts` пуста; когда мы выполняем команду `ls` для вывода ее содержимого, не выводится ничего ❶.

Удаление папок командами `rd` и `rmdir`

Команда `rd [исходная папка]` удаляет исходную папку в Windows, а команда `rmdir [исходная папка]` — в macOS и Linux. Как и в случае с `mkdir`, команда `rmdir` также работает в Windows, но команду `rd` вводить быстрее. Чтобы папку можно было удалить, она должна быть пустой.

Пример в окне терминала Linux:

```
al@ubuntu:~/Desktop$ mkdir yourScripts
al@ubuntu:~/Desktop$ ls
yourScripts
al@ubuntu:~/Desktop$ rmdir yourScripts
al@ubuntu:~/Desktop$ ls
al@ubuntu:~/Desktop$
```

Этот пример сначала создает пустую папку `yourScripts`, а затем удаляет ее.

Чтобы удалить непустую папку (вместе со всеми папками и файлами в ней), выполните команду `rd /s/q [исходная папка]` в Windows или команду `rm -rf [исходная папка]` в macOS и Linux.

Поиск программ командами *where* и *which*

Команда `where [программа]` в Windows или `which [программа]` в macOS и Linux сообщает точное местонахождение программы. Когда вы вводите команду в командной строке, осуществляется поиск программы в папках, входящих в переменную среды `PATH` (правда, Windows начинает с текущего рабочего каталога).

Эти команды сообщают, какая исполняемая программа Python запускается при вводе команды `python` в командной оболочке. Если у вас установлено сразу несколько версий Python, на компьютере могут существовать разные исполняемые программы с одинаковыми именами. Выбор запускаемой программы зависит от порядка следования папок в переменной среды `PATH`, а команды `where` и `which` выведут ее полное имя:

```
C:\Users\Al>where python
C:\Users\Al\AppData\Local\Programs\Python\Python38\python.exe
```

В этом примере имя папки указывает, что версия Python, запускаемая из оболочки, находится в папке `C:\Users\Al\AppData\Local\Programs\Python\Python38\`.

Очистка терминала командами *cls* и *clear*

Команда `cls` в Windows или `clear` в macOS и Linux стирает весь текст в окне терминала. Например, эта возможность пригодится, если вы хотите начать с пустого окна терминала.

Переменные среды и `PATH`

У всех выполняемых процессов программы, на каком бы языке она ни была написана, имеется набор *переменных среды*, где хранятся строки. Переменные среды

часто содержат настройки системного уровня, которые могут оказаться полезными в каждой программе. Например, переменная среды `TEMP` содержит путь к каталогу, где любая программа может хранить временные файлы. Когда операционная система запускает программу (например, командную строку), созданный процесс получает собственную копию переменных среды операционной системы. Переменные среды процесса могут изменяться независимо от набора значений переменных среды операционной системы. Но эти изменения действуют только на этот процесс, но не на операционную систему или другие процессы.

Я рассматриваю переменные среды в этой главе, потому что одна из этих переменных — `PATH` — пригодится вам при запуске программ из командной строки.

Просмотр переменных среды

Чтобы просмотреть список переменных среды для окна терминала, выполните команду `set` (в Windows) или `env` (в macOS или Linux) в командной строке:

```
C:\Users\A1>set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\A1\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
--snip--
USERPROFILE=C:\Users\A1
VBOX_MSI_INSTALL_PATH=C:\Program Files\Oracle\VirtualBox\
windir=C:\WINDOWS
```

Слева от знака равенства (=) указывается имя переменной среды, а справа — строковое значение. Каждый процесс содержит отдельный набор переменных среды, так что разные командные строки могут содержать разные значения своих переменных среды.

Для просмотра значения одной переменной среды также можно воспользоваться командой `echo`. Команда `echo %HOMEPATH%` в Windows или `echo $HOME` в macOS или Linux выводит значение переменных среды `HOMEPATH` или `HOME` соответственно, которые содержат домашнюю папку текущего пользователя. В Windows результат выглядит так:

```
C:\Users\A1>echo %HOMEPATH%
\Users\A1
```

Результат в macOS или Linux:

```
al@a1-VirtualBox:~$ echo $HOME
/home/al
```

Если этот процесс создает другой процесс (например, как командная строка запускает интерпретатор Python), дочерний процесс получает собственную копию

переменных среды родительского процесса. После этого дочерний процесс может изменить значения своих переменных среды, и наоборот.

Набор переменных среды операционной системы можно рассматривать как эталонную копию, с которой процесс копирует свои переменные среды. Переменные среды операционной системы изменяются реже переменных программы Python. Более того, многие пользователи вообще не изменяют значения переменных среды.

Переменная среды PATH

Когда вы вводите команду (например, `python` в Windows или `python3` в macOS и Linux), терминал ищет программу с таким именем в текущей папке. Если программа там не найдена, то проверяются папки, перечисленные в переменной среды PATH.

Например, на моем компьютере с Windows файл программы `python.exe` хранится в папке `C:\Users\Al\AppData\Local\Programs\Python\Python38`. Чтобы его запустить, мне придется ввести полное имя `C:\Users\Al\AppData\Local\Programs\Python\Python38\python.exe` или сначала перейти в эту папку, а затем ввести `python.exe`. Длинное имя приходится набирать долго, поэтому я включил эту папку в переменную среды PATH. Теперь при вводе `python.exe` командная строка ищет программу с заданным именем в папках, перечисленных в PATH, а мне не приходится вводить путь полностью.

Так как переменная среды может содержать только одно строковое значение, для включения нескольких имен папок в переменную среды PATH следует использовать специальный формат. В Windows имена папок разделяются символами `;` (точка с запятой). Для просмотра текущего содержимого PATH введите команду `path`:

```
C:\Users\Al>path
C:\Path;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;
...
C:\Users\Al\AppData\Local\Microsoft\WindowsApps
```

В macOS и Linux имена точек разделяются двоеточиями:

```
al@ubuntu:~$ echo $PATH
/home/al/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/usr/games:/usr/local/games:/snap/bin
```

Порядок имен папок важен. Если в папках `C:\WINDOWS\system32` и `C:\WINDOWS` находятся два разных файла с именем `someProgram.exe`, то при вводе команды `someProgram.exe` будет выполнена программа из `C:\WINDOWS\system32`, потому что эта папка первой встречается в переменной среды PATH.

Если введенная вами программа или команда не существует ни в текущем рабочем каталоге, ни в одном из подкаталогов в PATH, командная строка сообщает об ошибке — например, `command not found or not recognized as an internal or external`

command (Команда не найдена или не опознана как внутренняя или внешняя команда). Если вы не допустили опечатку, проверьте, в какой папке находится программа и включена ли эта папка в переменную среды PATH.

Изменение переменной среды PATH в командной строке

Переменную среды PATH текущего окна терминала можно изменить, включив в нее дополнительные папки. Процессы добавления папок в PATH несколько различаются в Windows и macOS/Linux. В Windows для добавления новой папки в текущее значение PATH можно воспользоваться командой path:

```
C:\Users\Al>path C:\newFolder;%PATH% ❶  
  
C:\Users\Al>path ❷  
C:\newFolder;C:\Path;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;  
...  
C:\Users\Al\AppData\Local\Microsoft\WindowsApps
```

Часть %PATH% ❶ заменяется текущим значением переменной среды PATH, так что фактически вы добавляете новую папку и точку с запятой в начало существующего значения PATH. Снова выполните команду path, чтобы просмотреть новое значение PATH ❷.

В macOS и Linux для назначения переменной среды PATH можно использовать синтаксис, сходный с синтаксисом команды присваивания в Python:

```
al@al-VirtualBox:~$ PATH=/newFolder:$PATH ❶  
al@al-VirtualBox:~$ echo $PATH ❷  
/newFolder:/home/al/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/  
bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Часть \$PATH ❶ заменяется текущим значением переменной среды PATH, так что фактически вы добавляете новую папку и точку с запятой в начало существующего значения PATH. Снова выполните команду echo \$path, чтобы просмотреть новое значение PATH ❷.

Но оба описанных способа добавления папок в PATH применяются только к текущему окну терминала и ко всем программам, которые запускаются из него после добавления. Если вы откроете новое окно терминала, в нем этих изменений уже не будет. Чтобы папка была включена в PATH постоянно, необходимо изменить эталонный набор переменных среды операционной системы.

Постоянное включение папок в PATH в Windows

В Windows существуют два набора переменных среды: *системные переменные среды* (которые распространяются на всех пользователей) и *переменные среды*

пользователя (которые переопределяют значения системных переменных среды, но распространяются только на текущего пользователя). Чтобы отредактировать их, откройте меню Пуск и введите *Edit environment variables for your account* (Отредактируйте переменные среды для своей учетной записи). Открывается окно Переменные среды (рис. 2.6).

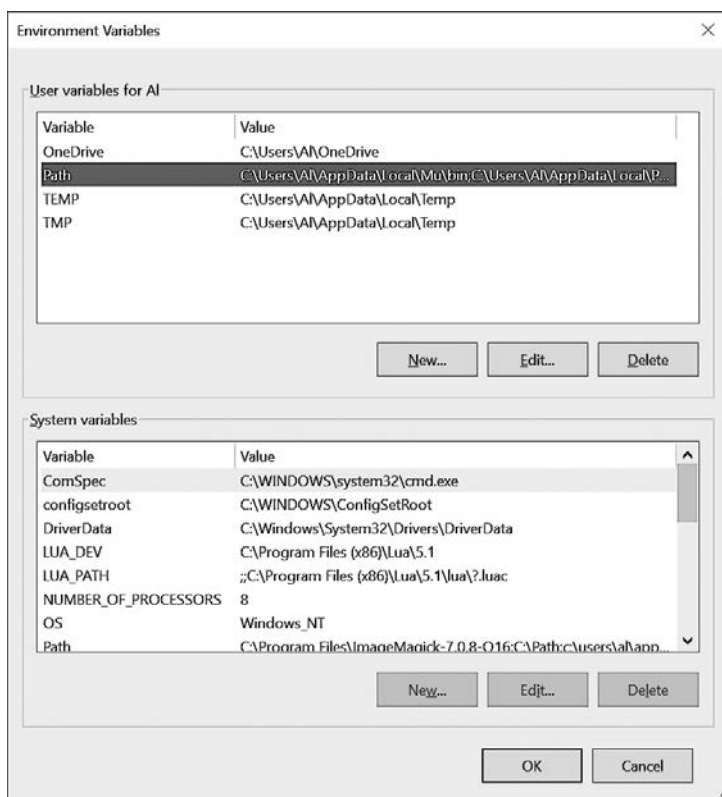


Рис. 2.6. Окно Переменные среды в Windows

Выберите *Path* в списке переменных среды пользователя (не в списке системных переменных!), щелкните на кнопке **Изменить** и введите новое имя папки в открывшемся текстовом поле (не забудьте о разделителе *;*). Щелкните на кнопке **ОК**.

Это не самый удобный интерфейс, и, если вам приходится часто редактировать переменные среды в Windows, я рекомендую установить бесплатную программу *Rapid Environment Editor* (<https://www.rapidee.com/>). Учтите, что после установки необходимо запустить программу с правами администратора, чтобы редактировать системные переменные среды. Щелкните на кнопке **Пуск**, введите *Rapid Environment*

Editor, щелкните правой кнопкой мыши на значке программы и выберите команду **Запуск от имени администратора**.

В командной строке для постоянного изменения системной переменной `PATH` можно воспользоваться командой `setx`:

```
C:\Users\A1>setx /M PATH "C:\newFolder;%PATH%"
```

Для выполнения команды `setx` необходимо запустить командную строку с правами администратора.

Постоянное включение папок в `PATH` в macOS и Linux

Чтобы добавить папки в переменные среды `PATH` для всех окон терминала в macOS и Linux, необходимо изменить текстовый файл `.bashrc` в домашней папке. Добавьте в него следующую строку:

```
export PATH=/newFolder:$PATH
```

Эта строка изменяет `PATH` для всех будущих окон терминала. В macOS Catalina и в более поздних версиях оболочка по умолчанию сменилась с Bash на Z Shell, в этом случае необходимо изменить файл `.zshrc` в домашней папке.

Запуск программ Python без командной строки

Вероятно, вы уже умеете запускать программы из оболочек, предоставляемых вашей операционной системой. В Windows существует меню **Пуск**, в macOS — **Finder** и **Dock**, а в Ubuntu Linux — **Dash**. Программы добавляют себя в эти оболочки при установке. Также программу можно запустить двойным щелчком на значке программы в файловом менеджере (таком, как **Проводник** в Windows, **Finder** в macOS или **Files** в Ubuntu Linux).

Но для программ Python эти способы не годятся. Часто при двойном щелчке на файле `.py` программа Python не запускается, а открывается в редакторе или IDE. А попытавшись запустить Python напрямую, вы просто откроете интерактивную оболочку Python. Чтобы запустить программу Python, пользователи чаще всего открывают ее в IDE и выбирают команду меню **Run** или же выполняют ее в командной строке. Оба способа не слишком удобны, если вам нужно просто запустить программу Python.

Но вы можете настроить программы Python так, чтобы они запускались из оболочки вашей операционной системы (как и другие приложения, установленные в системе). Далее я подробно расскажу, как сделать это в конкретной операционной системе.

Запуск программ Python в Windows

В системе Windows программы Python можно запустить еще несколькими способами. Не стоит открывать окно терминала, можно открыть диалоговое окно запуска программы клавишами WIN-R, а затем ввести команду `py C:\path\to\yourScript.py`, как показано на рис. 2.7. Программа `py.exe` находится в папке `C:\Windows`, которая уже включена в переменную окружения `PATH`, а указывать расширение `.exe` при запуске программ необязательно.

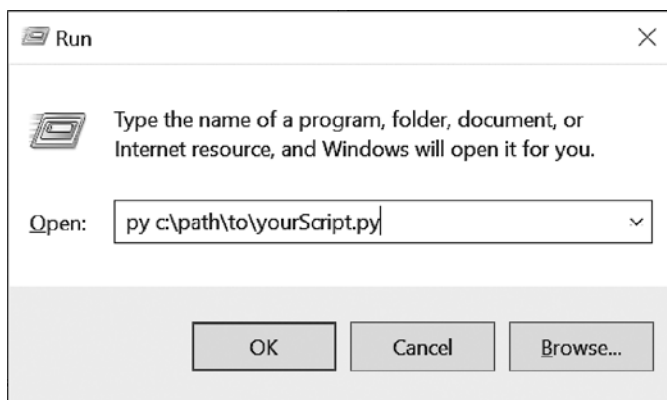


Рис. 2.7. Диалоговое окно запуска программы в Windows

Тем не менее этот способ требует ввода полного пути к вашему сценарию. Кроме того, окно терминала, в котором отображается вывод программы, автоматически закрывается при ее завершении, и вы можете упустить часть вывода.

Проблемы удастся решить созданием *пакетного сценария* — небольшого текстового файла с расширением `.bat`, который может выполнять сразу несколько терминальных команд (по аналогии со сценариями командной оболочки в macOS и Linux). Для создания этих файлов можно воспользоваться текстовым редактором (например, Блокнотом). Создайте новый текстовый файл, содержащий следующие две строки:

```
@py.exe C:\path\to\yourScript.py %*  
@pause
```

Замените этот путь абсолютным путем к вашей программе и сохраните файл с расширением `.bat` (например, `yourScript.bat`). Знак `@` в начале каждой команды предотвращает ее вывод в окне терминала, а `%*` передает все аргументы командной строки, введенные после имени пакетного файла, сценарию Python. В свою очередь, сценарий Python читает аргументы командной строки из списка `sys.argv`. Пакетный файл избавляет вас от необходимости вводить полный абсолютный путь

к программе Python каждый раз, когда вы захотите ее выполнить. Команда `@pause` добавляет сообщение `Press any key to continue...` в конец сценария Python, чтобы окно программы не исчезало с экрана слишком быстро.

Я рекомендую разместить все пакетные файлы и файлы `.py` в одной папке, уже включенной в переменную среды `PATH` — например, в домашнюю папку `C:\Users\<ИМЯ_ПОЛЬЗОВАТЕЛЯ>`. После создания пакетного файла для запуска сценария Python вам достаточно нажать WIN-R, ввести имя пакетного файла (указывать расширение `.bat` не обязательно) и нажать ENTER.

Запуск программ Python в macOS

В macOS можно создать сценарий командной оболочки для запуска сценариев Python; для этого следует создать текстовый файл с расширением `.command`. Создайте такой файл в текстовом редакторе (например, TextEdit) и введите следующие команды:

```
#!/usr/bin/env bash
python3 /path/to/yourScript.py
```

Сохраните этот файл в домашней папке. В окне терминала сделайте этот сценарий исполняемым при помощи команды `chmod u+x вашСценарий.command`. Теперь вы сможете щелкнуть на значке Spotlight (или нажать COMMAND-ПРОБЕЛ) или ввести имя сценария оболочки, чтобы выполнить его. А сценарий оболочки запустит ваш сценарий Python.

Запуск программ Python в Ubuntu Linux

В Ubuntu Linux не существует быстрого способа запуска сценариев Python, похожего на те, которые существуют в Windows и macOS, но все же облегчить себе жизнь можно. Сначала убедитесь в том, что файл `.py` хранится в домашней папке. Затем включите следующую строку, которая должна быть первой строкой файла `.py`:

```
#!/usr/bin/env python3
```

Строка сообщает Ubuntu, что для запуска этого файла должна использоваться программа `python3`. После этого выполните команду `chmod` в терминале, чтобы добавить для этого файла разрешение исполнения:

```
al@al-VirtualBox:~$ chmod u+x yourScript.py
```

Теперь каждый раз, когда вы захотите быстро запустить сценарий Python, откройте новое окно терминала клавишами `Ctrl-Alt-T`. В терминале будет выбрана домашняя

папка, так что вы можете просто ввести `./yourScript.py` для запуска этого сценария. Префикс `./` необходим, потому что он сообщает Ubuntu, что сценарий `yourScript.py` находится в текущем рабочем каталоге (домашней папке в данном случае).

Итоги

Подготовка среды подразумевает действия, необходимые для перевода вашего компьютера в режим, в котором можно легко запускать программы. Для этого необходимо знать ряд низкоуровневых концепций, относящихся к работе вашего компьютера, таких как файловая система, пути к файлам, процессы, командная строка и переменные среды.

Файловая система определяет способ упорядоченного хранения файлов на вашем компьютере. Положение файла обозначается либо полным, абсолютным путем, либо путем, заданным относительно текущего рабочего каталога. Перемещение по файловой системе можно осуществлять из командной строки. Командную строку также называют терминалом, оболочкой или консолью, но все эти слова обозначают одно и то же: текстовую программу, в которой можно вводить команды. И хотя командная строка и имена часто используемых команд несколько различаются в Windows и macOS/Linux, по сути они решают одни и те же задачи.

Когда вы вводите команду или имя программы, командная строка проверяет папки, включенные в переменную среды `PATH`, в поисках имени. Это важно знать, чтобы понять причину ошибок `command not found` (Команда не найдена). Последовательность действий при добавлении новых папок в переменную среды `PATH` также несколько отличается в Windows и macOS/Linux.

На освоение командной строки может потребоваться время, потому что вам придется изучать множество команд и аргументов командной строки. Не огорчайтесь, если вы тратите много времени для поиска информации в интернете; опытные разработчики занимаются этим каждый день.

ЧАСТЬ II

ПЕРЕДОВЫЕ ПРАКТИКИ, ИНСТРУМЕНТЫ И МЕТОДЫ

3

Форматирование кода при помощи Black



Форматированием кода называется его оформление посредством применения к нему определенного набора правил. Хотя форматирование кода не важно для компьютера, разбирающего вашу программу, оно существенно упрощает чтение кода людьми, а это необходимо для его успешного сопровождения. Если ваш код непонятен человеку (это может быть ваш коллега или даже вы сами), в нем трудно исправить ошибки или добавить новую функциональность. Форматирование кода не является чисто косметической операцией. Удобочитаемость — одна из важнейших причин популярности языка Python.

В этой главе рассматривается Black — система автоматического форматирования кода, в результате чего код становится последовательным и удобочитаемым без изменения работы самой программы. Это полезный инструмент, потому что ручное форматирование в текстовом редакторе или IDE слишком утомительно. Сначала я приведу обоснования стилевых решений, которые принимает Black, а потом расскажу, как установить, использовать и настроить эту программу.

Как потерять друзей и настроить против себя коллег

Существует много разных вариантов написания кода, которые приводят к одному и тому же результату. Например, при записи списка можно ставить один пробел после каждой запятой и последовательно применять только одну разновидность кавычек:

```
spam = ['dog', 'cat', 'moose']
```

Но даже если в списке будут встречаться разные отступы и виды кавычек, код все равно останется действительным с точки зрения синтаксиса Python:

```
spam= [ 'dog' , 'cat' , "moose" ]
```

Программистам, предпочитающим первый вариант, может нравиться наглядное разделение элементов и единообразие в выборе кавычек. Но другие программисты иногда выбирают второй вариант, потому что они не хотят беспокоиться о подробностях, которые не влияют на правильность работы программы.

Новички часто не обращают внимания на форматирование, потому что они сосредоточены на концепциях программирования и синтаксисе языка. Тем не менее им следует выработать привычку к качественному форматированию кода. Программирование само по себе достаточно сложно, и написание кода, понятного для других (и для вас в будущем), может весьма упростить его использование.

Даже если вы начинаете писать программу в одиночку, не факт, что впоследствии вы не объединитесь с единомышленниками. Если несколько программистов, работающих над одним файлом с исходным кодом, пишут каждый в своем стиле, результат может выглядеть как невразумительное месиво, даже если код работает без ошибок. Или, что еще хуже, некоторые начнут переформатировать код напарников «под себя», что в итоге приведет к потере времени и вызовет споры. Решение о том, сколько пробелов должно следовать после запятой, один или ни одного, — дело личных предпочтений. Такие стилевые решения можно сравнить с выбором того, по левой или по правой стороне должны двигаться автомобили в стране; неважно, какую сторону считать правильной, — важно, чтобы все участники движения придерживались именно ее.

Руководства по стилю и PEP 8

Чтобы ваш код хорошо читался, проще всего следовать требованиям, прописанным в *руководстве по стилю* — документе, определяющем набор правил форматирования, которые должны соблюдаться в программном проекте. *PEP 8* (Python Enhancement Proposal 8) — одно из таких руководств, написанное командой разработчиков ядра Python. Но некоторые компании разработали собственные руководства по стилю.

Текст PEP 8 можно найти в интернете по адресу <https://www.python.org/dev/peps/pep-0008/>. Многие программисты Python рассматривают PEP 8 как авторитетный набор правил, хотя создатели PEP 8 утверждают иное. Раздел руководства «Тупой единый подход — беда мелких умов» напоминает читателю, что главной причиной для соблюдения руководств по стилю является последовательность и удобочитаемость кода в рамках проекта, а не фанатичное соблюдение отдельных правил форматирования.

В PEP 8 даже включен следующий совет: «Знайте, когда стоит действовать непоследовательно, — иногда рекомендации по стилю попросту неприменимы. Если не

уверены, руководствуйтесь здравым смыслом». Независимо от того, собираетесь вы выполнять рекомендации полностью, частично или вообще никак, документ PEP 8 стоит прочитать.

Так как мы используем систему форматирования Black, наш код будет следовать правилам руководства по стилю Black, которое было разработано согласно рекомендациям PEP 8. Не поленитесь изучить эти рекомендации, потому что Black не всегда имеется под рукой. Рекомендации для форматирования Python, о которых я расскажу в этой главе, также применимы к другим языкам, для которых автоматические системы форматирования не всегда доступны.

Мне не все нравится в том, как Black форматирует код, но это компромиссное решение. Black использует правила форматирования, которые в целом устраивают программистов, чтобы они могли меньше тратить времени на споры и больше — на программирование.

Горизонтальные отступы

Для удобочитаемости вашего кода пустое место не менее важно, чем код, который вы пишете. Отступы помогают отделить обособленные части кода друг от друга, чтобы их было проще распознать. В этом разделе объясняются горизонтальные отступы — то есть пустые места в строке кода, в том числе в начале строки.

Использование пробелов для создания отступов

Для создания отступов в начале строки можно выбрать один из двух символов — пробел или табуляцию. И хотя годится любой вариант, для формирования отступов лучше использовать пробелы.

Дело в том, что эти символы ведут себя по-разному. Пробел всегда выводится на экран как строковое значение, состоящее из одного пробела — ' '. А символ табуляции, который выводится как строковое значение, содержащее служебный символ '\t', не столь однозначен. Символы табуляции часто (хотя и не всегда) выводятся как переменное количество пробелов, чтобы текст начинался со следующей позиции табуляции. Позиции табуляции устанавливаются через каждые восемь пробелов по ширине текстового файла. Этот вариант продемонстрирован в следующем примере, где слова сначала разделяются пробелами, а затем символами табуляции:

```
>>> print('Hello there, friend!\nHow are you?')
Hello there, friend!
How are you?
>>> print('Hello\tthere,\tfriend!\nHow\tare\tyou?')
Hello  there,  friend!
How    are    you?
```


Так как символы табуляции представляют переменное количество пробелов, лучше не использовать их в исходном коде. Многие редакторы кода и IDE при нажатии клавиши TAB автоматически вставляют четыре или восемь пробелов вместо одного символа табуляции.

Также не следует смешивать пробелы и символы табуляции для создания отступов в одном блоке кода. Использование обоих символов настолько часто становилось причиной коварных ошибок в более ранних программах Python, что в Python 3 код с такими отступами даже не выполняется — выдается исключение `TabError: inconsistent use of tabs and spaces in indentation` (Непоследовательное применение табуляций и пробелов в отступах). Black автоматически преобразует все символы табуляции, используемые в отступах, в четыре пробела.

Что касается длины каждого уровня отступов, в Python один уровень обычно обозначен четырьмя пробелами. В следующем примере пробелы для большей наглядности помечены точками:

```
def getCatAmount():
...numCats = input('How many cats do you have?')
...if int(numCats) < 6:
.....print('You should get more cats.')
```

Стандарт с четырьмя пробелами имеет практические преимущества перед другими альтернативами; в коде с восемью пробелами на уровень быстро возникает проблема ограничения длины строки, тогда как при уровне с двумя символами отступы не столь заметны. Программисты обычно не рассматривают другие значения (например, 3 или 6), потому что из-за привычки к двоичным вычислениям они предпочитают работать со степенями двойки: 2, 4, 8, 16 и т. д.

Отступы в середине строки

Пробелы играют важную роль для визуального разделения разных частей кода. Если вы их не используете, ваш код может оказаться слишком плотным и неразборчивым. В следующих подразделах я расскажу о некоторых полезных правилах при создании отступов.

Отделяйте операторы от идентификаторов одним пробелом

Если операторы и идентификаторы не разделены ни одним пробелом, создается впечатление, что ваш код выполняется подряд. Например, в следующей строке операторы и переменные разделены пробелами:

```
ДА: blanks = blanks[:i] + secretWord[i] + blanks[i + 1 :]
```

А здесь все пробелы удалены:

```
НЕТ: blanks=blanks[:i]+secretWord[i]+blanks[i+1:]
```

В обоих случаях оператор `+` используется для сложения трех значений, но без пробелов может показаться, что `+` в `blanks[i+1:]` добавляет четвертое значение. Пробелы более наглядно показывают, что `+` является частью сегмента в значении `blanks`.

Не ставьте пробелы перед разделителями, ставьте один пробел после разделителей

Элементы списков и словарей, а также параметры в определениях функций `def` разделяются запятыми. Не ставьте пробел перед этими запятыми, но поставьте один пробел после них, как в следующем примере:

```
ДА:   def spam(eggs, bacon, ham):
ДА:       weights = [42.0, 3.1415, 2.718]
```

В противном случае у вас получится перенасыщенный код, который плохо читается:

```
НЕТ: def spam(eggs,bacon,ham):
НЕТ:     weights = [42.0,3.1415,2.718]
```

Не добавляйте пробелы перед разделителем, потому что этим вы привлекаете ненужное внимание к символу-разделителю:

```
НЕТ: def spam(eggs , bacon , ham):
НЕТ:     weights = [42.0 , 3.1415 , 2.718]
```

Black автоматически вставляет пробелы после запятых и удаляет пробелы перед ними.

Не ставьте пробелы до и после точек

Python разрешает вставлять пробелы до и после точек, отмечающих начало атрибутов Python, но делать этого не стоит. Таким образом вы подчеркиваете связь между объектом и его атрибутом, как в следующем примере:

```
ДА:   'Hello, world'.upper()
```

Если же поставить пробелы до или после точки, объект и атрибут выглядят так, словно они не связаны друг с другом:

```
НЕТ: 'Hello, world' . upper()
```

Black автоматически удаляет пробелы вокруг точек.

Не ставьте пробелы после имен функций, методов и контейнеров

Имена функций и методов легко узнаваемы, потому что за ними следует пара круглых скобок. Не ставьте пробел между именем и открывающей круглой скобкой. Обычно вызов функции записывается в следующем виде:

ДА: `print('Hello, world!')`

С добавлением пробела единый вызов функции выглядит так, словно в коде идут два отдельных фрагмента:

НЕТ: `print ('Hello, world!')`

Black удаляет все пробелы между именем функции или метода и его открывающей круглой скобкой.

По той же причине не ставьте пробел перед открывающей квадратной скобкой для индекса, сегмента или ключа. Обычно при обращении к элементам типа «контейнер» (список, словарь или кортеж) пробел между именем переменной и открывающей квадратной скобкой не ставится:

ДА: `spam[2]`

ДА: `spam[0:3]`

ДА: `pet['name']`

Как и в предыдущем случае, при добавлении пробела этот код выглядит как два отдельных фрагмента:

НЕТ: `spam [2]`

НЕТ: `spam [0:3]`

НЕТ: `pet ['name']`

Black удаляет все пробелы между именем переменной и открывающей квадратной скобкой.

Не ставьте пробелы после открывающих и перед закрывающими скобками

Не должно быть пробелов, отделяющих круглые, квадратные или фигурные скобки от того, что внутри. Например, параметры в команде `def` или значения в списке должны начинаться и заканчиваться непосредственно после и перед круглыми и квадратными скобками:

ДА: `def spam(eggs, bacon, ham):`

ДА: `weights = [42.0, 3.1415, 2.718]`

Не ставьте пробел после открывающих или перед закрывающими круглыми и квадратными скобками:

```
НЕТ: def spam( eggs, bacon, ham ):
НЕТ:     weights = [ 42.0, 3.1415, 2.718 ]
```

Добавление этих пробелов не делает код более удобочитаемым, поэтому они становятся лишними. Black удаляет эти пробелы, если они присутствуют в вашем коде.

Ставьте два пробела перед комментариями в конце строки

Если вы добавляете комментарии в конце строки кода, поставьте два пробела между последним символом кода и символом #, начинающим комментарий:

```
ДА: print('Hello, world!') # Вывод приветствия.
```

Два пробела помогают отделить код от комментария. С одним пробелом (или еще хуже — без пробелов) заметить комментарий будет трудно:

```
НЕТ: print('Hello, world!') # Вывод приветствия.
НЕТ: print('Hello, world!')# Вывод приветствия.
```

Black вставляет два пробела между последним символом кода и началом комментария.

В общем случае я не рекомендую размещать комментарии в конце строки кода, потому что строка становится слишком длинной для чтения на экране.

Вертикальные отступы

Под вертикальными отступами понимается вставка пустых строк между строками кода. Подобно тому как разбивка текста книги на абзацы членит текст и способствует его правильному и быстрому восприятию, вертикальные отступы группируют строки кода и визуальнo отделяют группы друг от друга.

PEP 8 содержит ряд рекомендаций по вставке пустых строк в коде: руководство утверждает, что функции должны разделяться двумя пустыми строками, классы — двумя пустыми строками, а методы внутри класса — одной пустой строкой. Black автоматически следует этим правилам, вставляя или удаляя пустые строки в вашем коде. Например, фрагмент:

```
НЕТ: class ExampleClass:
      def exampleMethod1():
          pass
      def exampleMethod2():
```

```
    pass
def exampleFunction():
    pass
```

преобразуется к следующему виду:

```
ДА: class ExampleClass:
    def exampleMethod1():
        pass

    def exampleMethod2():
        pass

def exampleFunction():
    pass
```

Пример использования вертикальных отступов

Однако Black не может решить, где должны располагаться пустые строки внутри ваших функций, методов или глобальной области видимости. Решение о том, какие из этих строк должны группироваться вместе, принимает программист.

Для примера рассмотрим класс `EmailValidator` из файла `validators.py` фреймворка веб-приложений Django. Понимать, как работает этот код, необязательно. Обратите внимание на то, как пустые строки разделяют код метода `__call__()` на четыре группы:

```
...
def __call__(self, value):
    if not value or '@' not in value:           ❶
        raise ValidationError(self.message, code=self.code)

    user_part, domain_part = value.rsplit('@', 1)  ❷

    if not self.user_regex.match(user_part):      ❸
        raise ValidationError(self.message, code=self.code)

    if (domain_part not in self.domain_whitelist and  ❹
        not self.validate_domain_part(domain_part)):
        # Проверить возможные имена IDN
        try:
            domain_part = punycode(domain_part)
        except UnicodeError:
            pass
        else:
            if self.validate_domain_part(domain_part):
                return
    raise ValidationError(self.message, code=self.code)
...
```

Несмотря на отсутствие комментариев, описывающих эту часть кода, пустые строки показывают, что группы концептуально изолированы друг от друга. Первая группа ❶ проверяет символ @ в параметре `value`. Эта задача отличается от задачи второй группы ❷, которая разбивает строку с адресом электронной почты из `value` на две новые переменные, `user_part` и `domain_part`. Третья ❸ и четвертая ❹ группы используют эти переменные для проверки пользовательской и доменной части адреса электронной почты соответственно.

И хотя четвертая группа состоит из 11 строк (намного больше, чем в других группах), все они решают одну задачу проверки домена из адреса электронной почты. Если вы чувствуете, что в действительности здесь несколько подзадач, разделите их пустыми строками.

Программист, создававший эту часть Django, решил, что все строки проверки домена должны принадлежать одной группе, но другие программисты могут считать иначе. Так как это субъективное мнение, Black не изменяет вертикальные отступы внутри функций или методов.

Рекомендации по использованию вертикальных отступов

Python предоставляет одну возможность, о которой знают не все: точка с запятой (;) может использоваться для разделения нескольких команд в одной строке. Это означает, что следующие две строки:

```
print('What is your name?')
name = input()
```

можно записать в одну, разделив их точкой с запятой:

```
print('What is your name?'); name = input()
```

Как и при использовании запятых, перед точкой с запятой пробел не ставится, а после ее ставится один пробел.

Для команд, завершающихся двоеточием (например, `if`, `while`, `for`, `def` и `class`), однострочный блок, как вызов `print()` в следующем примере:

```
if name == 'Alice':
    print('Hello, Alice!')
```

можно записать в одной строке с командой `if`:

```
if name == 'Alice': print('Hello, Alice!')
```

Но хотя Python и разрешает разместить несколько команд в одной строке, это не считается хорошей практикой. Строки кода получаются слишком длинными

и содержат слишком много информации, что затрудняет их восприятие. Black разбивает такие команды на отдельные строки.

Аналогичным образом можно импортировать несколько модулей одной командой `import`:

```
import math, os, sys
```

Тем не менее PEP 8 рекомендует разбить эту команду на несколько коротких команд, по одной для каждого модуля:

```
import math
import os
import sys
```

Если модули импортируются в отдельных строках, вам будет проще заметить любые добавления или удаления импортированных модулей при сравнении изменений функцией `diff` системы контроля версий (системы контроля версий, такие как Git, рассматриваются в главе 12).

PEP 8 также рекомендует объединять команды `import` в следующие три группы в указанном порядке.

1. Модули стандартной библиотеки Python: `math`, `os`, `sys` и т. д.
2. Сторонние модули: `Selenium`, `Requests`, `Django` и т. д.
3. Локальные модули, являющиеся частью программы.

Эти рекомендации не являются обязательными, и Black не изменяет форматирование команд `import` в вашем коде.

Black: бескомпромиссная система форматирования кода

Black автоматически форматирует код в ваших файлах `.py`. Хотя вы должны понимать правила форматирования, рассмотренные в этой главе, Black выполняет всю работу по стилевому оформлению за вас. Если вы участвуете в совместном проекте, то можете моментально урегулировать многие споры относительно форматирования кода — просто поручите решение Black.

У нас нет возможности изменить многие правила, которым следует Black, поэтому Black описывается как «бескомпромиссная система форматирования кода». Собственно, его название (Black — черный) происходит от знаменитого высказывания Генри Форда относительно цветов выпускаемых автомобилей: «Цвет автомобиля может быть любым, при условии что он черный».

Я описал конкретные стилевые правила, используемые Black; полное руководство по стилю для Black вы найдете по адресу https://black.readthedocs.io/en/stable/the_black_code_style.html.

Установка Black

Установите Black при помощи программы `pip`, входящей в комплект поставки Python. В Windows для этого следует открыть окно командной строки и ввести команду:

```
C:\Users\Al>python -m pip install --user black
```

В macOS и Linux откройте окно терминала и введите команду `python3` вместо `python` (это следует делать во всех фрагментах кода в этой книге, где используется команда `python`):

```
Als-MacBook-Pro:~ al$ python3 -m pip install --user black
```

Ключ `-m` приказывает Python запустить модуль `pip` как приложение (некоторые модули Python имеют соответствующую настройку). Чтобы убедиться в том, что установка прошла успешно, выполните команду `python -m black`. На экране должно появиться сообщение «No paths given. Nothing to do» вместо «No module named black».

Запуск Black из командной строки

Black можно запустить для любого файла Python из командной строки или окна терминала. Кроме того, IDE или редактор кода позволяет запустить Black в фоновом режиме. Инструкции о том, как обеспечить работу Black с Jupyter Notebook, Visual Studio Code, PyCharm и другими редакторами, вы найдете на домашней странице Black по адресу <https://github.com/psf/black/>.

Допустим, вы хотите отформатировать файл с именем `yourScript.py` автоматически. В командной строке Windows выполните следующую команду (в macOS и Linux используйте команду `python3` вместо `python`):

```
C:\Users\Al>python -m black yourScript.py
```

После выполнения этой команды содержимое `yourScript.py` будет отформатировано в соответствии с руководством по стилю Black.

Переменная среды `PATH` может быть уже настроена для прямого запуска Black. В этом случае для форматирования `yourScript.py` достаточно ввести следующую команду:

```
C:\Users\Al>black yourScript.py
```


Чтобы выполнить Black для каждого файла .py в папке, укажите в команде папку вместо отдельного файла. Следующий пример Windows форматирует все файлы в папке C:\yourPythonFiles, включая ее вложенные папки:

```
C:\Users\A1>python -m black C:\yourPythonFiles
```

Передача папки пригодится в том случае, если ваш проект содержит несколько файлов Python и вы не хотите вводить отдельную команду для каждого файла.

Хотя Black довольно строго относится к форматированию кода, в следующих трех подразделах описаны некоторые параметры, которые вы можете настроить. Чтобы просмотреть полный набор ключей Black, выполните команду `python -m black --help`.

Регулировка длины строки Black

Стандартная строка кода Python состоит из 80 символов. История 80-символьных строк уходит корнями в 1920-е годы, когда компания IBM представила перфокарты на 80 столбцов и 12 строк; 80-символьный стандарт стал использоваться для принтеров, мониторов и окон командной строки, появившихся через несколько десятилетий.

Но в XXI веке экраны с высоким разрешением позволяют выводить текст длиной более 80 символов. С большей длиной строки вам не придется выполнять вертикальную прокрутку, чтобы просмотреть файл. С другой стороны, короткие строки не перегружены, а вы можете сравнить два исходных файла, разместив их на экране рядом друг с другом, без горизонтальной прокрутки.

Black по умолчанию использует строку длиной 88 символов. Выбор объясняется не очень убедительно: это на 10% больше стандартной 80-символьной строки. Лично я предпочитаю 120 символов. Чтобы приказать Black форматировать код с 120-символьной (например) строкой, используйте ключ командной строки `-l 120` (буква l в нижнем регистре, не цифра 1). В Windows команда выглядит примерно так:

```
C:\Users\A1>python -m black -l 120 yourScript.py
```

Какую бы длину строки вы ни выбрали для своего проекта, для всех файлов .py в проекте должно использоваться одинаковое значение.

Отключение настройки двойных кавычек для Black

Black автоматически изменяет все строковые литералы в вашем коде так, чтобы вместо одинарных кавычек они заключались в двойные кавычки, если только строка

не содержит внутренние символы двойных кавычек; в этом случае используются одинарные кавычки. Допустим, файл `yourScript.py` содержит следующий фрагмент:

```
a = 'Hello'
b = "Hello"
c = 'Al\'s cat, Zophie.'
d = 'Zophie said, "Meow"'
e = "Zophie said, \"Meow\""
f = '''Hello'''
```

После применения Black файл `yourScript.py` будет отформатирован следующим образом:

```
a = "Hello" ❶
b = "Hello"
c = "Al's cat, Zophie."
d = 'Zophie said, "Meow"' ❷
e = 'Zophie said, "Meow"'
f = """Hello""" ❸
```

В результате применения Black с настройками двойных кавычек код Python будет похож на код, написанный на других языках программирования, где строковые литералы часто заключаются в двойные кавычки. Обратите внимание: строки переменных `a`, `b` и `c` используют двойные кавычки. Строка переменной `d` сохраняет исходные одинарные кавычки, чтобы избежать исчезновения двойных кавычек в строке ❷. Обратите внимание: Black также использует двойные кавычки для многострочных текстов Python, заключенных в тройные кавычки ❸.

Но если вы хотите, чтобы ваши строковые литералы остались в том виде, в котором они были написаны, а тип используемых кавычек не изменялся, передайте Black ключ командной строки `-S` (обратите внимание: буква `S` в верхнем регистре). Например, при применении Black к исходному файлу `yourScript.py` в Windows будет получен следующий результат:

```
C:\Users\Al>python -m black -S yourScript.py
All done!
1 file left unchanged.
```

Также можно использовать ключи длины строки `-l` и `-S` в одной команде:

```
C:\Users\Al>python -m black -l 120 -S yourScript.py
```

Предварительный просмотр изменений, вносимых Black

Хотя Black не переименовывает переменные и не изменяет работу программы, возможно, вам не понравятся стилевые изменения, предложенные Black. Если вы хотите оставить исходное форматирование, то либо используйте механизм контроля

версий для исходного кода, либо создавайте резервные копии самостоятельно. Также можно просмотреть изменения, которые внесет Black, без фактического изменения файлов; для этого следует запустить Black с ключом командной строки `--diff`. В Windows это выглядит так:

```
C:\Users\A1>python -m black --diff yourScript.py
```

Эта команда выводит различия в формате `diff`, который часто используется системами контроля версий, но удобочитаем для людей. Например, если `yourScript.py` содержит строку `weights=[42.0,3.1415,2.718]`, при выполнении с ключом `--diff` будет выведен следующий результат:

```
C:\Users\A1>python -m black --diff yourScript.py
--- yourScript.py      2020-12-07 02:04:23.141417 +0000
+++ yourScript.py      2020-12-07 02:08:13.893578 +0000
@@ -1,2 @@
-weights=[42.0,3.1415,2.718]
+weights = [42.0, 3.1415, 2.718]
```

Знак «минус» означает, что Black удалит строку `weights=[42.0,3.1415,2.718]` и заменит ее строкой, которая выводится с префиксом «плюс»: `weights = [42.0, 3.1415, 2.718]`. Учтите, что после того, как вы запустите Black для изменения файлов с исходным кодом, отменить внесенные изменения уже не удастся. Необходимо либо создать резервные копии вашего исходного кода, либо воспользоваться системой контроля версий (такой как Git) перед запуском Black.

Отключение Black для отдельных частей кода

Каким бы замечательным ни был инструмент Black, возможно, вы не захотите, чтобы он форматировал некоторые части вашего кода. Например, я предпочитаю самостоятельно расставлять отступы в разделах, где я выравниваю несколько взаимосвязанных команд присваивания, как в следующем примере:

```
# Константы для разных интервалов времени:
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY     = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK    = 7  * SECONDS_PER_DAY
```

Black удалит дополнительные пробелы перед оператором присваивания `=`, отчего они, по моему мнению, будут хуже читаться:

```
# Константы для разных интервалов времени:
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY     = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK    = 7  * SECONDS_PER_DAY
```

Добавив комментарии `#fmt: off` и `#fmt: on`, можно запретить Black форматирование строк в этом фрагменте, а затем продолжить его:

```
# Константы для разных промежутков времени:
# fmt: off
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY     = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK    = 7 * SECONDS_PER_DAY
# fmt: on
```

Теперь запуск Black для этого файла не приведет к изменению отступов (и любого другого форматирования) в коде между этими двумя комментариями.

Итоги

Хотя хорошее форматирование люди часто не замечают, плохое форматирование вызывает раздражение. Стил — понятие субъективное, но у разработчиков обычно вырабатываются общие представления о том, что считать хорошим или плохим форматированием; при этом, конечно же, остается место для личных предпочтений.

Синтаксис Python достаточно гибок в том, что касается стиля. Если вы пишете код, который никто никогда не увидит, вы можете писать его так, как вам нравится. Но разработкой ПО чаще всего занимаются в команде. Работаете ли вы с другими программистами над проектом или просто хотите попросить более опытных коллег оценить вашу работу, важно отформатировать код в соответствии с общепринятыми руководствами по стилю.

Форматирование кода в редакторе — монотонная работа, которую можно автоматизировать таким инструментом, как Black. В этой главе рассмотрены некоторые правила, которым следует Black, для того чтобы сделать ваш код более удобочитаемым: вертикальные и горизонтальные отступы, чтобы код не был слишком плотным и хорошо читался, и ограничение длины каждой строки. Black следит за соблюдением этих правил за вас, что предотвращает возможные стилевые разногласия с коллегами.

Однако стиль программирования не сводится к применению отступов и выбору между одинарными и двойными кавычками. Например, использование содержательных имен переменных также является критическим фактором удобочитаемости кода. И хотя такие автоматизированные инструменты, как Black, могут принимать синтаксические решения (например, выбирать число отступов в коде), они не способны принимать семантические решения — скажем, насколько удачно выбрано имя переменной. За это отвечаете вы. Об этом мы поговорим в следующей главе.

4

Выбор понятных имен



«В программировании есть только две сложные проблемы — выбор имен, аннулирование кэша и ошибки смещения на единицу» — эта классическая шутка, приписываемая Леону Бамбрику (Leon Bambrick) и основанная на высказывании Фила Карлтона (Phil Karlton), содержит зерно истины: трудно придумать хорошие имена (формально называемые *идентификаторами*) для переменных, функций, классов и вообще чего угодно в программировании. Компактные, содержательные имена важны для удобочитаемости вашей программы. Тем не менее это проще сказать, чем сделать. Если вы переезжаете в новый дом, можно пометить все коробки надписью «Барахло» — надпись компактная, но не содержательная. Название книги по программированию «Как создавать собственные компьютерные игры на языке Python» — содержательное, но не компактное.

Если вы пишете не «одноразовый» код, который вы не собираетесь сопровождать после однократного запуска программы, вам стоит подумать над выбором хороших имен в вашей программе. Предположим, вы просто присвоили переменным имена `a`, `b` и `c`, значит, вам в будущем придется потратить лишние усилия, чтобы запомнить, для чего предназначались эти переменные.

Выбор имен — субъективный выбор, который делаете именно вы. Автоматизированные средства форматирования (такие как Black из главы 3) не способны решить, как должны называться ваши переменные. В этой главе я расскажу, как выбрать «хорошие» имена и избегать «плохих». Как обычно, мои рекомендации не высечены в камне; руководствуйтесь здравым смыслом, чтобы решить, когда стоит применять их в вашем коде.

МЕТАСИНТАКСИЧЕСКИЕ ПЕРЕМЕННЫЕ

Метасинтаксические переменные обычно используются в учебниках или фрагментах кода, когда необходимо обобщенное имя переменной. В Python в тех переменных, для которых имена несущественны, часто используют имена `spam`, `eggs`, `bacon` и `ham`. По этой причине я применяю их в примерах кода в книге, но не берите их для реальных программ. Имена происходят из скетча «Спам» группы Monty Python ([https://en.wikipedia.org/wiki/Spam_\(Monty_Python\)](https://en.wikipedia.org/wiki/Spam_(Monty_Python))). Метасинтаксическим переменным также часто присваивают имена `foo` и `bar`. Они происходят от сленгового сокращения FUBAR, которое использовали солдаты армии США во времена Второй Мировой войны, когда хотели сообщить, что ситуация «[уделана] до полной неузнаваемости».

Схемы регистра имен

Так как в идентификаторах Python различается регистр символов и они не могут содержать пробельные символы, программисты используют несколько схем в идентификаторах, состоящих из нескольких слов.

- *Змеиный регистр* (`snake_case`) разделяет слова символом подчеркивания, который напоминает ползущую между словами змею. В этом случае все буквы записываются в нижнем регистре, а константы часто записываются в *верхнем змеином регистре* (`UPPER_SNAKE_CASE`).
- *Верблюжий регистр* (`camelCase`) — слова записываются в нижнем регистре, но второе и следующие слова начинаются с заглавной. Эта схема в большинстве случаев подразумевает, что первое слово начинается с буквы нижнего регистра. Буквы верхнего регистра напоминают верблюжьи горбы.
- *Схема Pascal* (`PascalCase`) — названа так, потому что применяется в языке программирования Pascal; аналогична схеме верблюжьего регистра, но первое слово в ней тоже начинается с заглавной.

Выбор регистра относится к области форматирования кода, об этом я рассказывал в главе 3. На практике чаще всего встречаются змеиный и верблюжий регистры. Вы можете использовать любую схему, но в одном проекте — только одну, а не обе сразу.

Соглашения об именах PEP 8

В документе PEP 8 (глава 3) приведены некоторые правила формирования имен в Python.

- Все буквы должны быть буквами ASCII — то есть латинскими буквами верхнего и нижнего регистров без диакритических знаков.
- Имена модулей должны быть короткими и состоять только из букв нижнего регистра.
- Имена классов необходимо записывать в схеме Pascal.
- Имена констант следует записывать в верхнем змеином регистре.
- Имена функций, методов и переменных записывают в нижнем змеином регистре.
- Первый аргумент методов всегда должен называться `self` в нижнем регистре.
- Первый аргумент методов классов всегда должен называться `cls` в нижнем регистре.
- Приватные атрибуты классов всегда начинают с символа подчеркивания (`_`).
- Публичные атрибуты классов никогда не начинают с символа подчеркивания (`_`).

При необходимости вы можете изменять или нарушать эти правила. Например, хотя английский язык доминирует в программировании, в идентификаторах можно использовать символы любых языков: команда `コンピューター = 'laptop'` является синтаксически действительным кодом Python. Как вы видите, мои предпочтения в области имен переменных противоречат PEP 8, потому что я использую верблюжий регистр вместо змеиного. PEP 8 содержит напоминание о том, что программист не обязан неуклонно следовать PEP 8. Важнейший фактор удобочитаемости — не выбор схемы, а последовательность в применении этой схемы.

С разделом «Naming Conventions» документа PEP 8 можно ознакомиться по адресу <https://www.python.org/dev/peps/pep-0008/#naming-conventions>.

Длина имен

Очевидно, имена не должны быть слишком короткими или слишком длинными. Длинные имена переменных утомительно вводить, а короткие могут быть непонятными или дезинформирующими. Но так как код читают чаще, чем пишут, лучше все-таки задавать более длинные имена переменных. Рассмотрим примеры.

Слишком короткие имена

Самая распространенная ошибка при выборе имен — суперкороткие имена. Они зачастую кажутся вам понятными, когда вы впервые их записываете, но вы можете забыть их точный смысл через несколько дней или недель. Рассмотрим несколько разновидностей коротких имен.

- Одно- или двухбуквенное имя (например, `g`), вероятно, обозначает какое-то слово, начинающееся с этой буквы, но таких слов очень много. Сокращения и одно-двухбуквенные имена легко записать, но они плохо читаются. Это замечание относится и к следующему пункту.
- Сокращенные имена вида `mon` — могут означать `monitor`, `month`, `monster` и множество других слов.
- Имя из одного слова — например, `start` (начало) — может трактоваться по-разному: начало чего? При отсутствии уточнения другие люди вас вряд ли поймут.

Одно- и двухбуквенные имена, сокращения или однословные имена могут быть понятны вам, но всегда следует помнить, что другие программисты (или же вы сами через несколько недель) вряд ли поймут их смысл.

В отдельных случаях короткие имена переменных вполне допустимы. Например, имя `i` часто используется с переменными циклов `for`, перебирающих диапазоны чисел или индексов списка, а `j` и `k` (следующие за `i` в алфавитном порядке) используются с вложенными циклами:

```
>>> for i in range(10):  
...     for j in range(3):  
...         print(i, j)  
...  
0 0  
0 1  
0 2  
1 0  
...
```

Еще одно исключение — использование `x` и `y` для декартовых координат. В большинстве других случаев я не рекомендую применять однобуквенные имена переменных. Хотя, допустим, `w` и `h` для ширины (`width`) и высоты (`height`) или `n` для числа (`number`) могут соблазнить вас краткостью, другим это может быть не очевидно.

Слишком длинные имена

Как правило, чем больше область видимости имени, тем более содержательным оно должно быть. Короткое имя (например, `payment`) хорошо подойдет для локальной переменной в одной короткой функции. Однако имя может оказаться недостаточно содержательным, если использовать его для глобальной переменной в 10 000-строчной программе, потому что в такой программе могут обрабатываться различные виды платежных данных. В такой ситуации лучше использовать более содержательное имя `salesClientMonthlyPayment` или `annual_electric_bill_payment`. Дополнительные слова в имени уточняют смысл и устраняют неоднозначность.

Лучше лишние пояснения, чем их нехватка. Однако существуют рекомендации для определения того, когда необходимы более длинные имена.

Н ПРПСКИЙТ БКВ В СВМ КД

Не пропускайте буквы в своем коде. Хотя пропущенные буквы в именах были популярны в языках программирования C до 1990-х годов — например, `memcry` (`memory copy`) или `strcmp` (`string compare`) — они создают плохо читаемый стиль выбора имен, который не стоит использовать сегодня. Если имя нельзя легко произнести, его не удастся легко понять.

Кроме того, старайтесь использовать короткие фразы, с которыми ваш код читается как обычный текст. Например, имя `number_of_trials` читается лучше, чем `number_trials`.

Префиксы в именах

Префиксы в именах иногда избыточны. Например, если у вас есть класс `Cat` с атрибутом `weight`, очевидно, что `weight` (вес) относится к кошке (`cat`). Таким образом, имя `catWeight` будет слишком подробным и длинным.

Аналогичным образом устаревшей считается *венгерская запись* — практика включения сокращения типа данных в имена. Например, имя `strName` указывает, что переменная содержит строковое значение, а `iVacationDays` — что переменная содержит целое число. Современные языки и IDE могут предоставить программисту информацию о типе данных без этих префиксов, поэтому венгерская запись считается излишней в наше время. Если вы все еще включаете типы данных в имена переменных, пора отказаться от этой привычки.

С другой стороны, префиксы `is` и `has` у переменных, содержащих логические значения, или функций и методов, возвращающих логические значения, делают эти имена более понятными. Рассмотрим пример использования переменной с именем `is_vehicle` и метода с именем `has_key()`:

```
if item_under_repair.has_key('tires'):
    is_vehicle = True
```

С методом `has_key()` и переменной `is_vehicle` этот код читается как естественная фраза: «if the item under repair has a key named 'tires,' then it's true that the item is a vehicle» (если у ремонтируемого объекта имеется ключ с именем 'tires', следовательно, это транспортное средство).

Также включение единиц измерения в имена может предоставить полезную информацию. Переменная `weight`, в которой хранится значение с плавающей точкой,

неоднозначна: в чем измеряется вес — в фунтах, килограммах или тоннах? Информация об единицах измерения не является типом данных, поэтому включение префикса или суффикса `kg` или `lbs` нельзя считать венгерской записью. Если вы не используете тип данных, предназначенный специально для хранения веса и содержащий информацию о единицах измерения, имя переменной вида `weight_kg` оказывается вполне разумным. Вспомните, что в 1999 году автоматизированный космический зонд *Mars Climate Orbiter* был потерян из-за того, что программа, разработанная фирмой «Локхид-Мартин», выполняла вычисления в единицах, принятых в Великобритании, тогда как в NASA использовалась метрическая система и это привело к ошибке в расчете траектории. По имеющейся информации создание космического аппарата обошлось в 125 миллионов долларов.

Последовательные числовые суффиксы в именах

Последовательные числовые суффиксы в именах указывают на то, что вам, возможно, стоит изменить тип данных переменной или включить дополнительную информацию в имя. Числа сами по себе, как правило, не предоставляют достаточной информации, чтобы имена можно было отличить друг от друга.

Имена переменных вида `payment1`, `payment2` и `payment3` не сообщают читателю кода, чем они различаются. Возможно, программисту стоит преобразовать эти три переменные в один список или переменную-кортеж с именем `payments`, в которой хранятся три значения.

Функции вида `makePayment1(amount)`, `makePayment2(amount)` и т. д., вероятно, стоит преобразовать в одну функцию, которая получает целочисленный аргумент: `makePayment(1, amount)`, `makePayment(2, amount)` и т. д. Если эти функции обладают разным поведением, оправдывающим определение отдельных функций, смысл чисел должен быть отражен в имени, например: `makeLowPriorityPayment(amount)` и `makeHighPriorityPayment(amount)` или `make1stQuarterPayment(amount)` и `make2ndQuarterPayment(amount)`.

Если у вас имеется веская причина для выбора имен с последовательными числовыми суффиксами, ничто не мешает вам их использовать. Но если вы задаете такие имена, чтобы не задумываться лишний раз, вам стоит пересмотреть свой подход.

Выбирайте имена, пригодные для поиска

Во всех программах, кроме самых коротких, вам, вероятно, придется воспользоваться редактором или функцией поиска (`Ctrl-F`) в IDE, чтобы найти упоминания переменных и функций. Если вы будете выбирать короткие, обобщенные имена переменных (например, `num` или `a`), вы наверняка получите ряд ложных совпадений.

Чтобы имя было найдено немедленно, создавайте уникальные имена с более длинными именами переменных, которые содержат конкретную информацию.

В некоторых IDE реализованы средства рефакторинга, способные идентифицировать имена на основании их использования в вашей программе. Например, часто встречается функция «переименования», способная отличать переменные с именами `num` и `number` или локальную переменную `num` от глобальной переменной `num`. Однако вам следует выбирать имена так, словно эти инструменты недоступны.

Если вы будете помнить об этом правиле, вам будет проще выбирать содержательные имена вместо обобщенных. Имя `email` слишком многозначно, поэтому лучше выбрать более содержательное имя: `emailAddress`, `downloadEmailAttachment`, `emailMessage`, `replyToAddress` и т. д. Такое имя не только более точное, но его проще найти в исходном коде.

Избегайте шуток, каламбуров и культурных отсылок

На одном из моих предыдущих мест работы кодовая база содержала функцию с именем `gooseDownload()`. Я понятия не имел, что это означало, потому что продукт не имел отношения ни к птицам, ни к загрузке птиц (`goose` = «гусь»). Когда я обратился к более опытному коллеге, который когда-то написал эту функцию, он объяснил, что слово `goose` в данном случае было глаголом. Этого я тоже не понял. Ему пришлось дальше объяснять, что выражение `goose the engine` из жаргона водителей означает нарастить обороты двигателя. Таким образом, функция `gooseDownload()` должна была ускорить загрузку. Я кивнул и вернулся к рабочему столу. Годы спустя, когда мой коллега уволился из компании, я переименовал его функцию и присвоил ей имя `increaseDownloadSpeed()`.

При выборе имен в программе у вас может возникнуть соблазн использовать шутки, каламбуры или культурные отсылки, чтобы ваш код выглядел более непринужденно. Не делайте этого. Шутки плохо передаются в текстовом виде, и, возможно, в будущем они уже не покажутся столь забавными. Каламбуры не всегда понятны, а многократные сообщения об ошибках от коллег, принявших каламбур за опечатку, могут быть довольно неприятными.

Культурные отсылки могут помешать ясно передать цель вашего кода. Благодаря интернету исходный код легко распространяется по всему миру, и его читатели не всегда хорошо владеют английским или понимают английские шутки. Как упоминалось ранее в этой главе, имена `spam`, `eggs` и `bacon` в документации Python взяты из комедийных скетчей группы Monty Python, но мы используем их только как метасинтаксические переменные; применять их в реальном коде не рекомендуется.

Лучше всего писать код, понятный тем, для кого английский язык не является родным, то есть прямолинейно, традиционно и без юмора. Возможно, мой бывший коллега решил, что `gooseDownload()` — смешная шутка, но ничто не убивает шутку быстрее, чем необходимость ее объяснять.

Не заменяйте встроенные имена

Никогда не используйте встроенные имена Python для своих переменных. Например, присвоив переменной имя `list` или `set`, вы заместите функции Python `list()` и `set()`, что позднее может привести к появлению ошибок. Функция `list()` создает объекты списков, но ее замена может вызвать ошибку:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list = ['cat', 'dog', 'moose']    ❶
>>> list(range(5))                    ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not callable
```

Присвоив списковое значение переменной `list` ❶, мы потеряем исходную функцию `list()`. Попытка вызвать `list()` ❷ приведет к ошибке `TypeError`. Чтобы узнать, используется ли имя в Python, введите его в интерактивной оболочке или попробуйте импортировать. Если имя не используется, вы получите ошибку `NameError` или `ModuleNotFoundError`. Например, в Python используются имена `open` и `test`, но не используются `spam` и `eggs`:

```
>>> open
<built-in function open >
>>> import test
>>> spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> import eggs
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'eggs'
```

Наиболее часто заменяемые имена Python — `all`, `any`, `date`, `email`, `file`, `format`, `hash`, `id`, `input`, `list`, `min`, `max`, `object`, `open`, `random`, `set`, `str`, `sum`, `test` и `type`. Не берите их для своих идентификаторов.

Другая распространенная проблема — присваивание файлам `.py` имен, совпадающих с именами сторонних модулей. Например, если вы установили сторонний модуль `Rpyperclip`, но также создали файл `rpyperclip.py`, команда `import rpyperclip`

импортирует `pyperclip.py` вместо модуля `Pyperclip`. При попытке вызвать функцию `copy()` или `paste()` модуля `Pyperclip` вы получите ошибку, в которой говорится, что функции не существует:

```
>>> # Запустите этот код с файлом pyperclip.py в текущем каталоге.
>>> import pyperclip # Импортирует ваш файл pyperclip.py вместо настоящего.
>>> pyperclip.copy('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'pyperclip' has no attribute 'copy'
```

Помните об опасности замены существующих имен в коде Python, особенно если вы неожиданно получаете такие сообщения об отсутствии атрибута.

Худшие из возможных имен

Имя `data` — ужасное и абсолютно бессодержательное, потому что буквально любая переменная содержит данные (`data`). То же можно сказать об имени `var` — все равно что выбрать для собаки кличку Собака. Имя `temp` часто используется для переменных, содержащих временные данные, но и этот выбор плох: в конце концов, с точки зрения дзен-буддизма все переменные временны. К сожалению, несмотря на неоднозначность, эти имена встречаются часто; избегайте их использования в своем коде.

Если вам нужна переменная для хранения статистического отклонения температурных данных, используйте имя `temperatureVariance`. Не стоит и говорить, что выбор имени `tempVarData` будет неудачным.

Итоги

Выбор имен не имеет никакого отношения к алгоритмам или компьютерной теории, и все же это важнейший фактор написания удобочитаемого кода. В конечном счете выбор имен, используемых в вашем коде, остается на ваше усмотрение, но вы должны учитывать существующие рекомендации. В РЕР 8 вы найдете несколько соглашений о выборе имен — например, имена в нижнем регистре для модулей и имена в схеме Pascal для классов. Имена не должны быть слишком короткими или слишком длинными. Однако часто лучше сделать имя избыточным, чем недостаточно содержательным.

Имя должно быть лаконичным, но информативным. Оно должно легко находиться функцией поиска `Ctrl-F`. То, насколько просто можно найти выбранное имя, свидетельствует о его уникальности. Также задумайтесь, будет ли понятно это имя программисту, который недостаточно хорошо владеет английским языком; избегайте

шуток, каламбуров и культурных отсылок в своих именах; вместо этого выбирайте имена прямолинейные, традиционные и без отсылок к хохмам.

Избегайте имен, уже используемых стандартной библиотекой Python, — таких как `all`, `any`, `date`, `email`, `file`, `format`, `hash`, `id`, `input`, `list`, `min`, `max`, `object`, `open`, `random`, `set`, `str`, `sum`, `test` и `type`. Их применение может создать трудноуловимые ошибки в вашем коде.

Для компьютера неважно, какие имена вы выбрали, содержательные или общие. Имена упрощают чтение кода людьми, а не выполнение его компьютерами. Если ваш код хорошо читается, он будет понятным. Если он понятен, его легко изменить. А если его легко изменить, вам будет проще исправить ошибки или добавить новые возможности. Использование понятных имен — основополагающий фактор разработки качественного программного обеспечения.

5

Поиск запахов в коде



Если выполнение кода приводит к аварийному завершению программы, с кодом очевидно что-то не так, но сбой — не единственный признак проблем в программах. Другие признаки могут свидетельствовать о наличии более коварных ошибок или неудобочитаемого кода. Подобно тому как запах газа может указывать на утечку, а запах дыма — на возможный пожар, запах кода представляет собой паттерн исходного кода, сигнализирующий о возможных ошибках. Наличие запаха кода не всегда означает, что проблема существует, но вам стоит по крайней мере проанализировать свою программу.

В этой главе я расскажу о некоторых распространенных признаках, указывающих на проблемы в коде. На предотвращение ошибок требуется намного меньше времени и усилий, чем на их выявление, анализ и исправление. У каждого программиста найдется история о том, как он долгие часы отлаживал программу, а потом оказалось, что достаточно было исправить всего одну строку. Из-за этого даже при мимолетном подозрении на потенциальную ошибку вы должны остановиться и лишний раз удостовериться, что вы не создаете себе проблемы в будущем.

Конечно, запахи кода не всегда свидетельствуют о наличии ошибки. В конечном итоге именно вам решать, разбираться со своими подозрениями или не обращать на них внимания.

Дублирование кода

Самый распространенный источник ошибок — *дублирование кода*. Так называют любой исходный код, который создают копированием и вставкой фрагмента вашей программы. Например, следующая короткая программа содержит дублирующийся

код. Обратите внимание: она трижды задает пользователю вопрос о том, как тот себя чувствует (“How are you feeling?”):

```
print('Good morning!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good afternoon!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good evening!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
```

Дублирование кода создает проблемы, потому что оно усложняет редактирование кода: изменение в одной копии должно быть внесено во все его дубли в программе. Если вы забудете где-то внести изменение или вы внесете разные изменения в разных копиях, в вашей программе, скорее всего, возникнет ошибка.

Проблема дублирования кода решается *дедупликацией* — код преобразуют так, чтобы он встречался только в одном месте программы — обычно в функции или цикле. В следующем примере я переместил дубликат в функцию, а затем повторно вызывал эту функцию:

```
def askFeeling():
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')

print('Good morning!')
askFeeling()
print('Good afternoon!')
askFeeling()
print('Good evening!')
askFeeling()
```

В следующем примере дублирующийся код был перемещен в цикл:

```
for timeOfDay in ['morning', 'afternoon', 'evening']:
    print('Good ' + timeOfDay + '!')
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')
```

Также можно объединить два приема и совместить функцию с циклом:

```
def askFeeling(timeOfDay):
    print('Good ' + timeOfDay + '!')
```



```
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')

for timeOfDay in ['morning', 'afternoon', 'evening']:
    askFeeling(timeOfDay)
```

Обратите внимание: код, который выдает сообщения «Good morning/afternoon/evening!», похож, но не идентичен. В третьей версии программы я параметризовал код, чтобы исключить дублирование идентичных частей.

Параметр `timeOfDay` и переменная цикла `timeOfDay` заменяют различающиеся части. Теперь, когда из кода были устранены дубликаты (лишние копии), необходимые изменения достаточно внести только в одном месте.

Как и со всеми запахами кода, исключение дублирования не является неукоснительным правилом, которое следует соблюдать всегда. Как правило, чем длиннее фрагмент дубля или чем больше копий присутствует в вашей программе, тем серьезнее стоит задуматься об их устранении. Я не против того, чтобы скопировать код один и даже два раза. Но если в программе присутствуют три или четыре дубля, я обычно серьезно задумываюсь об их устранении.

Иногда дедупликация не стоит затраченных усилий. Сравните первый пример кода в этом разделе с последним. Хотя код с дубликатами длиннее, он проще и прямей. Пример без дубликатов делает то же самое, но с добавлением цикла, новой переменной цикла `timeOfDay` и новой функции с параметром, которому также присвоено имя `timeOfDay`.

Дублирование кода способно вызывать ошибки, потому что оно усложняет целостное изменение кода. Если программа содержит несколько одинаковых фрагментов, стоит поместить код в функцию или цикл, чтобы он вызывался только один раз.

«Магические» числа

Не приходится удивляться тому, что в программировании используются числа. Но некоторые числа, встречающиеся в исходном коде, могут сбить с толку других программистов (или вас через пару недель после написания программы). Для примера возьмем число **604800** в следующей строке:

```
expiration = time.time() + 604800
```

Функция `time.time()` возвращает целое число, представляющее текущее время. Можно предположить, что переменная `expiration` представляет некий будущий момент, который наступит через 604 800 секунд. Но число **604800** выглядит загадочно: что оно означает? Комментарий поможет прояснить ситуацию:

```
expiration = time.time() + 604800 # Срок действия истекает через неделю.
```

Это хорошее решение, но еще лучше заменить такие «магические» числа константами. *Константы* представляют собой переменные, имена которых записаны в верхнем регистре, это означает, что они не должны изменяться после исходного присваивания. Обычно константы определяются как глобальные переменные в начале файла с исходным кодом:

```
# Константы для разных промежутков времени:
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY     = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK    = 7 * SECONDS_PER_DAY
...
expiration = time.time() + SECONDS_PER_WEEK # Срок действия истекает
                                              # через неделю.
```

Используйте отдельные константы для «магических» чисел, предназначенных для разных целей, даже если их числовые значения совпадают. Например, в колоде 52 карты, а в году 52 недели. Но если в вашей программе используются обе величины, нужно поступить примерно так:

```
NUM_CARDS_IN_DECK = 52
NUM_WEEKS_IN_YEAR = 52

print('This deck contains', NUM_CARDS_IN_DECK, 'cards.')
print('The 2-year contract lasts for', 2 * NUM_WEEKS_IN_YEAR, 'weeks.')
```

При выполнении этого кода результат будет выглядеть так:

```
This deck contains 52 cards.
The 2-year contract lasts for 104 weeks.
```

(Колода содержит 52 карты.
Двухлетний контракт длится 104 недели.)

Использование разных констант позволяет независимо изменять их в будущем. Конечно, значения констант не должны изменяться во время выполнения кода. Но это не означает, что программист никогда не обновит их в исходном коде. Например, если в будущей версии кода появится дополнительная карта-джокер, константу `cards` можно изменить независимо от `weeks`:

```
NUM_CARDS_IN_DECK = 53
NUM_WEEKS_IN_YEAR = 52
```

«Магическими» *числами* также иногда называют нечисловые значения. Например, строковые значения могут использоваться как константы. Возьмем следующую программу, которая предлагает пользователю указать направление и выводит

предупреждение, если пользователь выбрал 'north'. Из-за опечатки 'nrth' возникает ошибка, вследствие чего предупреждение не выводится:

```
while True:
    print('Set solar panel direction:')
    direction = input().lower()
    if direction in ('north', 'south', 'east', 'west'):
        break
print('Solar panel heading set to:', direction)
if direction == 'nrth': ❶
    print('Warning: Facing north is inefficient for this panel.')
```

Найти такую ошибку нелегко: хотя в строке 'nrth' совершена опечатка, она остается синтаксически правильным кодом Python. Программа не завершается аварийно, а предупреждение легко упустить из виду. Но если вы допустите ту же опечатку при использовании констант, ошибка будет обнаружена, потому что Python заметит, что константа NRTN не существует:

```
# Константы для разных промежутков времени:
NORTH = 'north'
SOUTH = 'south'
EAST = 'east'
WEST = 'west'

while True:
    print('Set solar panel direction:')
    direction = input().lower()
    if direction in (NORTH, SOUTH, EAST, WEST):
        break

print('Solar panel heading set to:', direction)
if direction == NRTN: ❶
    print('Warning: Facing north is inefficient for this panel.')
```

Из-за исключения `NameError`, выдаваемого в строке кода с опечаткой `NRTN` ❶, ошибка становится очевидной при запуске программы:

```
Set solar panel direction:
west
Solar panel heading set to: west
Traceback (most recent call last):
  File "panelset.py", line 14, in <module>
    if direction == NRTN:
NameError: name 'NRTN' is not defined
```

«Магические» числа свидетельствуют о наличии запаха кода, потому что они не выполняют свое предназначение, затрудняют чтение и обновление кода и повышают риск опечаток, которые так трудно обнаружить. Проблема решается использованием констант.

Закомментированный и мертвый код

Поместить код в комментарий, чтобы он не выполнялся, — временная мера. Возможно, вы хотите пропустить часть строк, чтобы протестировать другую функциональность; закомментированные строки вы легко вернете позднее. Но если закомментированный код так и останется на месте, для читателя останется абсолютной тайной, почему он был удален и при каких условиях он может опять стать частью программы. Рассмотрим следующий пример:

```
doSomething()  
#doAnotherThing()  
doSomeImportantTask()  
doAnotherThing()
```

Возникает целый ряд вопросов: почему вызов `doAnotherThing()` был закомментирован? Будет ли он снова включен в программу? Почему не был закомментирован второй вызов `doAnotherThing()`? Изначально в коде было два вызова `doAnotherThing()` или только один вызов, который переместился в точку после вызова `doSomeImportantTask()`? Почему закомментированный код не был удален, для этого есть какая-то причина? На все эти вопросы нет очевидных ответов.

Мертвым называется код, который недоступен или никогда не может быть выполнен на логическом уровне. Код внутри функции после команды `return`, команды `if`, условие которой всегда равно `False`, или код функции, которая никогда не вызывается в программе, — все это примеры мертвого кода. Чтобы увидеть пример на практике, введите следующую команду в интерактивной оболочке:

```
>>> import random  
>>> def coinFlip():  
...     if random.randint(0, 1):  
...         return 'Heads!'  
...     else:  
...         return 'Tails!'  
...     return 'The coin landed on its edge!'  
...  
>>> print(coinFlip())  
Tails!1
```

Строка `'The coin landed on its edge!'` является мертвым кодом, потому что код в блоках `if` и `else` возвращает управление до того, как программа сможет достичь этой строки. Мертвый код дезинформирует, поскольку читающие его программисты предполагают, что он составляет активную часть программы, тогда как по сути это закомментированный код.

¹ Heads — орел; Tails — решка; The coin landed on its edge — Монета встала ребром (англ.). — Примеч. пер.

Заглушки (stubs) являются исключением из этих правил. Они представляют собой заменители для будущего кода (например, функции и классы, которые еще не были реализованы). Вместо реального кода заглушка содержит команду `pass`, которая ничего не делает. (Она также называется *пустой операцией*.) Команда `pass` существует как раз для того, чтобы вы могли создавать заглушки в тех местах, где с точки зрения синтаксиса языка должен присутствовать какой-то код:

```
>>> def exampleFunction():
...     pass
... 
```

Если вызвать эту функцию, она не сделает ничего. Вместо этого она лишь показывает, что когда-нибудь в нее будет добавлен код.

Также возможен другой вариант: чтобы предотвратить случайный вызов не-реализованной функции, можно использовать заглушку из команды `raise NotImplementedError`. Тем самым вы покажете, что функция еще не готова к вызову:

```
>>> def exampleFunction():
...     raise NotImplementedError
... 
```

```
>>> exampleFunction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in exampleFunction
NotImplementedError
```

Исключение `NotImplementedError` предупредит вас о том, что в программе была случайно вызвана заглушка (функция или метод).

Закомментированный код и мертвый код считаются запахами кода, потому что они могут создать у программиста ошибочное впечатление, будто код является исполняемой частью программы. Вместо этого следует удалить их и использовать систему контроля версий (например, `Git` или `Subversion`) для отслеживания изменений. Контролю версий я посвятил главу 12. С системой контроля версий вы можете удалить код из своей программы, а при необходимости позднее вернуть его обратно.

Отладочный вывод

Отладочный вывод — это практика включения в программу временных вызовов `print()` для вывода значений переменных и повторного запуска программы.

Последовательность процесса, как правило, такова.

1. Обнаружение ошибки в программе.
2. Включение вызовов `print()` для некоторых переменных, чтобы узнать их текущие значения.

3. Перезапуск программы.
4. Включение новых вызовов `print()`, потому что предыдущие вызовы не предоставили достаточной информации.
5. Перезапуск программы.
6. Предыдущие этапы следует повторять несколько раз, пока вы наконец не найдете причину ошибки.
7. Перезапуск программы.
8. Вы понимаете, что забыли удалить какие-то вызовы `print()`, и удаляете их.

Отладочный вывод обманчиво прост и быстр. Но часто он требует многих циклов перезапуска программы, пока не будет выведена информация, необходимая для исправления ошибки. Проблема решается при помощи отладки или организации журнального вывода в программе. При использовании отладчика вы можете выполнять свои программы по одной строке и проверять значения любых переменных. Иногда кажется, что работа с отладчиком занимает больше времени, чем простая вставка вызова `print()`, но в долгосрочной перспективе она экономит время.

Журнальные файлы сохраняют большие объемы информации из вашей программы, чтобы вы могли сравнить результаты одного запуска с другим. В Python встроенный модуль `logging` предоставляет функциональность, необходимую для простого сохранения журнальной информации, всего в трех строках кода:

```
import logging
logging.basicConfig(filename='log_filename.txt', level=logging.DEBUG,
format='%asctime)s - %(levelname)s - %(message)s')
logging.debug('This is a log message.')
```

После импортирования модуля `logging` и настройки его базовой конфигурации можно вызвать метод `logging.debug()` для записи информации в текстовый файл — в отличие от использования `print()` для вывода ее на экран. В отличие от отладочного вывода вызов `logging.debug()` очевидно показывает, какой вывод содержит отладочную информацию, а какой является результатом нормального выполнения программы. Подробнее об отладке вы можете прочитать в главе 11 книги «Automate the Boring Stuff with Python», 2nd edition (No Starch, 2019), доступной по адресу <https://autbor.com/2e/c11/>.¹

Переменные с числовыми суффиксами

При написании программ вам может понадобиться набор переменных для хранения однотипных данных. И здесь возникает искушение повторно использовать

¹ Свейгарт Э. Автоматизация рутинных задач с помощью Python. 2-е изд.

имя переменной, добавив к нему числовой суффикс. Например, если вы обрабатываете форму ввода регистрационных данных, на которой пользователю предлагается дважды ввести пароль для предотвращения опечаток, две введенные строки можно сохранить в переменных с именами `password1` и `password2`. Эти числовые суффиксы не описывают, что содержат переменные и чем они отличаются. Также они не показывают, сколько всего таких переменных: существуют ли также переменные `password3` или `password4`? Попробуйте создавать разные имена, вместо того чтобы бездумно добавлять числовые суффиксы. В примере с паролями лучше использовать имена `password` и `confirm_password` ("пароль" и "подтвердить пароль").

Другой пример: если у вас есть функция, которая получает две пары координат на плоскости, она может иметь параметры `x1`, `y1`, `x2` и `y2`. Но имена с числовыми суффиксами не передают такой информации, как имена `start_x`, `start_y`, `end_x` и `end_y`. Также очевидно, что переменные `start_x` и `start_y` связаны друг с другом, чего не скажешь о `x1` и `y1`.

Если количество числовых суффиксов больше двух, стоит подумать об использовании структур: списка или множества для хранения данных в виде коллекции. Например, значения `pet1Name`, `pet2Name`, `pet3Name` и т. д. можно хранить в списке с именем `petNames`.

Эти замечания относятся не ко всем переменным, которые заканчиваются цифрой. Например, вполне нормально иметь переменную с именем `enableIPv6`, потому что 6 является частью имени собственного IPv6, а не числовым суффиксом. Но если вы используете числовые суффиксы для серии переменных, подумайте о том, чтобы заменить их структурой данных — например, списком или словарем.

Классы, которые должны быть функциями или модулями

Программисты, работающие на таких языках, как Java, привыкли создавать классы для организации кода их программ. Например, возьмем класс `Dice` с методом `roll()`:

```
>>> import random
>>> class Dice:
...     def __init__(self, sides=6):
...         self.sides = sides
...     def roll(self):
...         return random.randint(1, self.sides)
...
>>> d = Dice()
>>> print('You rolled a', d.roll())
You rolled a 1
```

Может показаться, что перед вами хорошо организованный код, но подумайте, что нам здесь действительно нужно: случайное число от 1 до 6. Стоит заменить весь класс простым вызовом функции:

```
>>> print('You rolled a', random.randint(1, 6))
You rolled a 6
```

По сравнению с другими языками Python использует свободный подход к организации кода, потому что код не обязан существовать в классе или другой шаблонной структуре. Если вы обнаруживаете, что создаете объекты только для того, чтобы вызвать одну функцию, или пишете классы, содержащие только статические методы, возможно, стоит лучше написать функции.

В Python для группировки функций используются модули вместо классов. Так как классы все равно должны находиться в модуле, размещение этого кода в классах только добавляет в ваш код лишний организационный уровень. В главах 15–17 я рассмотрю принципы объектно-ориентированного проектирования более подробно. Джек Дидерих (Jack Diederich) в своем докладе «Перестаньте писать классы» на конференции PyCon 2012 рассказывает и о других возможностях избыточного усложнения кода Python.

Списковые включения внутри списковых включений

Списковые включения (list comprehensions) предоставляют компактный механизм создания сложных списковых значений. Например, чтобы создать список цифр в числах от 0 до 100, из которого исключены все числа, кратные 5, обычно используют цикл `for`:

```
>>> spam = []
>>> for number in range(100):
...     if number % 5 != 0:
...         spam.append(str(number))
...
>>> spam
['1', '2', '3', '4', '6', '7', '8', '9', '11', '12', '13', '14', '16', '17',
...
'86', '87', '88', '89', '91', '92', '93', '94', '96', '97', '98', '99']
```

Однако тот же список можно создать всего одной строкой кода с использованием синтаксиса спискового включения:

```
>>> spam = [str(number) for number in range(100) if number % 5 != 0]
>>> spam
```



```
['1', '2', '3', '4', '6', '7', '8', '9', '11', '12', '13', '14', '16', '17',  
...  
'86', '87', '88', '89', '91', '92', '93', '94', '96', '97', '98', '99']
```

Также в Python существует синтаксис включений множеств и словарных включений:

```
>>> spam = {str(number) for number in range(100) if number % 5 != 0} ❶  
>>> spam  
{'39', '31', '96', '76', '91', '11', '71', '24', '2', '1', '22', '14', '62',  
...  
'4', '57', '49', '51', '9', '63', '78', '93', '6', '86', '92', '64', '37'}  
>>> spam = {str(number): number for number in range(100) if number % 5 != 0} ❷  
>>> spam  
{'1': 1, '2': 2, '3': 3, '4': 4, '6': 6, '7': 7, '8': 8, '9': 9, '11': 11,  
...  
'92': 92, '93': 93, '94': 94, '96': 96, '97': 97, '98': 98, '99': 99}
```

Включение множества ❶ использует фигурные скобки вместо квадратных, а генерируемое им значение представляет собой множество. Словарное включение ❷ создает значение-словарь и использует двоеточие для разделения ключа и значения во включении. Включения компактны, и они могут сделать код более удобочитаемым. Но обратите внимание на то, что включения создают список, множество или словарь на основании итерируемого объекта (в данном примере — объекта диапазона, возвращаемого вызовом `range(100)`). Списки, множества и словари являются итерируемыми объектами, это означает, что включения могут вкладываться во включения, как в следующем примере:

```
>>> nestedIntList = [[0, 1, 2, 3], [4], [5, 6], [7, 8, 9]]  
>>> nestedStrList = [[str(i) for i in sublist] for sublist in nestedIntList]  
>>> nestedStrList  
[['0', '1', '2', '3'], ['4'], ['5', '6'], ['7', '8', '9']]
```

Но вложенные списковые включения (или вложенные включения множеств/словарные включения) упаковывают значительную сложность в небольшой объем кода, что усложняет его чтение. Лучше развернуть списковое включение в один или несколько циклов `for`:

```
>>> nestedIntList = [[0, 1, 2, 3], [4], [5, 6], [7, 8, 9]]  
>>> nestedStrList = []  
>>> for sublist in nestedIntList:  
...     nestedStrList.append([str(i) for i in sublist])  
...  
>>> nestedStrList  
[['0', '1', '2', '3'], ['4'], ['5', '6'], ['7', '8', '9']]
```

Включения также могут содержать множественные выражения `for`, хотя в таких ситуациях также часто появляется нечитаемый код. Например, следующее

списковое включение создает неструктурированный список на базе вложенного списка:

```
>>> nestedList = [[0, 1, 2, 3], [4], [5, 6], [7, 8, 9]]
>>> flatList = [num for sublist in nestedList for num in sublist]
>>> flatList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Это списковое включение содержит два выражения `for`, но даже опытному разработчику Python будет непросто понять его. Развернутая форма с двумя циклами `for` создает тот же деструктурированный список, но читается намного проще:

```
>>> nestedList = [[0, 1, 2, 3], [4], [5, 6], [7, 8, 9]]
>>> flatList = []
>>> for sublist in nestedList:
...     for num in sublist:
...         flatList.append(num)
...
>>> flatList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Включения представляют собой синтаксические сокращения, которые позволяют создавать компактный код. Тем не менее не увлекайтесь и не вкладывайте их друг в друга.

Пустые блоки `except` и плохие сообщения об ошибках

Перехват исключений — один из основных способов восстановить работоспособность программы даже при возникновении проблем. Если в программе возникает исключение, но нет блока `except` для его обработки, программа Python аварийно завершается. Это может привести к потере несохраненной работы или к сохранению полузавершенного кода, который в дальнейшем может стать причиной еще более серьезных ошибок.

Чтобы предотвратить фатальные ошибки, можно добавить блок `except` с кодом обработки ошибки. Но иногда бывает трудно решить, как должна обрабатываться ошибка, и у программиста возникает искушение оставить блок `except` пустым, с одной командой `pass`. Например, в следующем коде команда `pass` используется для создания блока `except`, который ничего не делает:

```
>>> try:
...     num = input('Enter a number: ')
...     num = int(num)
... except ValueError:
...     pass
```

```
...
Enter a number: forty two
>>> num
'forty two'
```

В этом коде не происходит сбой, когда функции `int()` передается строка `'forty two'`, потому что исключение `ValueError`, выдаваемое `int()`, обрабатывается командой `except`. Однако если ошибка попросту игнорируется, это может быть хуже, чем аварийное завершение. Программы аварийно завершаются, чтобы они не продолжали работать с некорректными данными или в неполном состоянии, что может породить более серьезные ошибки в будущем. При вводе нецифровых символов в нашем примере аварийное завершение не происходит. Но теперь переменная `num` содержит строку вместо целого числа, что вызовет проблемы при использовании переменной `num`. Наша команда `except` не столько обрабатывает ошибки, сколько скрывает их.

Обработка исключений с плохими сообщениями об ошибках также относится к категории запахов кода. Взгляните на следующий пример:

```
>>> try:
...     num = input('Enter a number: ')
...     num = int(num)
... except ValueError:
...     print('An incorrect value was passed to int()')
...
Enter a number: forty two
An incorrect value was passed to int()
```

В этом коде не происходит фатального сбоя, и это хорошо, но он не предоставляет пользователю достаточной информации о том, как решить проблему. Сообщения об ошибках предназначены для пользователей, а не для программистов. В данном случае сообщение не только содержит технические подробности, непонятные пользователю (например, ссылку на функцию `int()`), но и не сообщает пользователю, как можно решить проблему. Сообщения об ошибках должны объяснять, что случилось и что пользователь должен сделать.

Программисту проще быстро описать произошедшее (что бесполезно), вместо того чтобы подробно пояснить, что стоит сделать пользователю для решения проблемы. Но помните, что, если ваша программа не обрабатывает все возможные исключения, эта программа еще не завершена.

Мифы о запахах кода

Некоторые запахи кода вообще не являются таковыми. В программировании полно полузабытых плохих советов, которые были вырваны из контекста или существовали так долго, что пережили свою полезность. Я виню в этом авторов

технических книг, которые пытаются выдать свои субъективные мнения за передовые практики. Возможно, вы слышали, что некоторые из них являются причинами ошибок в коде, но в основном в них нет ничего плохого. Я называю их мифами о запахах кода: это всего лишь предупреждения, которые можно и нужно игнорировать. Рассмотрим несколько примеров.

Миф: функции должны содержать только одну команду `return` в самом конце

Идея «один вход, один выход» происходит из неправильно интерпретированного совета из эпохи программирования на языке ассемблера и FORTRAN. Эти языки позволяли войти в подпрограмму (структуру, сходную с функцией) в любой точке, в том числе и в середине, из-за чего в ходе отладки было труднее определить, какие части подпрограммы уже были выполнены. У функций такой проблемы нет (выполнение всегда начинается с начала функции). Но совет продолжал существовать и в конце концов трансформировался в «функции и методы должны содержать только одну команду `return`, которая должна находиться в конце функции или метода».

Попытки добиться того, чтобы в функции или методе была только одна команда `return`, часто приводят к появлению запутанных последовательностей команд `if-else`, которые создают гораздо больше проблем, чем несколько команд `return`. Функция или метод может содержать несколько команд `return`, ничего страшного в этом нет.

Миф: функции должны содержать не более одной команды `try`

«Функции и методы должны делать что-то одно» — в большинстве случаев это хороший совет. Но требовать, чтобы обработка исключений выполнялась в отдельной функции, значит заходить слишком далеко. Для примера рассмотрим функцию, которая проверяет, существует ли удаляемый файл:

```
>>> import os
>>> def deleteWithConfirmation(filename):
...     try:
...         if (input('Delete ' + filename + ', are you sure? Y/N') == 'Y'):
...             os.unlink(filename)
...     except FileNotFoundError:
...         print('That file already did not exist.')
... 
```

Сторонники этого мифа возражают, что функции должны всегда иметь только одну обязанность. Обработка исключений — это обязанность, поэтому функцию нужно разбить на две. Они считают, что, если вы используете команду `try-except`, она должна быть первой командой и охватывать весь код функции:

```
>>> import os
>>> def handleErrorForDeleteWithConfirmation(filename):
...     try:
...         _deleteWithConfirmation(filename)
...     except FileNotFoundError:
...         print('That file already did not exist.')
...
>>> def _deleteWithConfirmation(filename):
...     if (input('Delete ' + filename + ', are you sure? Y/N') == 'Y'):
...         os.unlink(filename)
...
...

```

Этот код излишне усложнен. Функция `_deleteWithConfirmation()` теперь помечена как приватная при помощи префикса `_`, который указывает, что функция никогда не должна вызываться напрямую — только косвенно, через вызов `handleErrorForDeleteWithConfirmation()`. Имя новой функции получилось неудобным, потому что она вызывается для удаления файла, а не для обработки ошибки при удалении.

Ваши функции должны быть простыми и компактными, но это не значит, что они всегда должны делать что-то одно (как бы вы это ни определяли). Вполне нормально, если ваши функции содержат несколько команд `try-except` и эти команды не охватывают весь код функции.

Миф: аргументы-флаги нежелательны

Логические аргументы функций или методов иногда называются аргументами-флагами. В программировании *флагом* называется значение, включающее бинарный выбор «включено — выключено»; для представления флагов часто используются логические значения. Такие настройки можно описать как установленные (`True`) или сброшенные (`False`).

Ложная уверенность в том, что аргументы-флаги функций чем-то плохи, основана на утверждении, что в зависимости от значения флага функция решает две совершенно разные задачи, как в следующем примере:

```
def someFunction(flagArgument):
    if flagArgument:
        # Выполнить код...
    else:
        # Выполнить совершенно другой код...
```

Действительно, если ваша функция выглядит так, лучше создать две разные функции, вместо того чтобы в зависимости от аргумента выбирать, какая половина кода функции должна выполняться. Но большинство функций с аргументами-флагами работает не так. Например, логическое значение может передаваться в ключевом аргументе `reverse` функции `sorted()` для определения порядка сортировки. Разбиение

функции на две функции с именами `sorted()` и `reverseSorted()` не улучшит код (а также удвоит объем необходимой документации). Таким образом, мнение о нежелательности аргументов-флагов является мифом.

Миф: глобальные переменные нежелательны

Функции и методы напоминают мини-программы внутри вашей программы: они содержат код, включая локальные переменные, которые теряются при выходе из функции (подобно тому как переменные программы теряются после ее завершения). Функции существуют изолированно: либо их код выполняется правильно, либо содержит ошибку в зависимости от аргументов, переданных при вызове.

Но функции и методы, использующие глобальные переменные, отчасти утрачивают эту полезную изоляцию. Каждая глобальная переменная, используемая в функции, фактически становится дополнительным входным значением функции наряду с аргументами. Больше аргументов — больше сложности, что в свою очередь означает более высокую вероятность ошибок. Если ошибка проявляется в функции из-за неправильного значения глобальной переменной, это значение может быть задано в любой точке программы. Чтобы найти вероятную причину ошибочного значения, недостаточно проанализировать код функции или строку кода с вызовом функции; придется рассмотреть всю программу. Поэтому следует ограничить использование глобальных переменных.

Для примера возьмем функцию `calculateSlicesPerGuest()` в воображаемой программе `partyPlanner.py`, содержащей тысячи строк. Я включил номера строк, чтобы дать представление о размере программы:

```
1504. def calculateSlicesPerGuest(numberOfCakeSlices):
1505.     global numberOfPartyGuests
1506.     return numberOfCakeSlices / numberOfPartyGuests
```

Допустим, при выполнении этой программы возникает следующее исключение:

```
Traceback (most recent call last):
  File "partyPlanner.py", line 1898, in <module>
    print(calculateSlicesPerGuest(42))
  File "partyPlanner.py", line 1506, in calculateSlicesPerGuest
    return numberOfCakeSlices / numberOfPartyGuests
ZeroDivisionError: division by zero
```

В программе возникает ошибка деления на 0, за которую ответственна строка `return numberOfCakeSlices / numberOfPartyGuests`. Чтобы это произошло, переменная `numberOfPartyGuests` должна быть равна 0, но где `numberOfPartyGuests` было присвоено это значение? Так как переменная является глобальной, это могло произойти в любой из тысяч строк программы! Из данных трассировки мы знаем,

что функция `calculateSlicesPerGuest()` вызывалась в строке 1898 нашей вымышленной программы. Взглянув на строку 1898, можно узнать, какой аргумент передавался для параметра `numberOfCakeSlices`. Но значение глобальной переменной `numberOfPartyGuests` могло быть присвоено где угодно до этого вызова функции.

Следует заметить, что применение глобальных констант не считается нежелательной практикой. Так как их значения никогда не изменяются, они не повышают сложность кода так, как это делают другие глобальные переменные. Когда программисты говорят о том, что глобальные переменные нежелательны, они не имеют в виду константы.

Глобальные переменные увеличивают объем работы по отладке — программист должен найти точку, в которой было присвоено значение, вызвавшее исключение. Из-за этого чрезмерное использование глобальных переменных нежелательно. Но сама идея о том, что *все* глобальные переменные плохи, неверна. Глобальные переменные часто используют в небольших программах и для хранения настроек, действующих во всей программе. Если без глобальной переменной можно обойтись, вероятно, лучше это сделать. Но утверждение «все глобальные переменные плохи» — слишком упрощенное и субъективное.

Миф: комментарии излишни

Плохие комментарии хуже, чем отсутствие комментариев. Комментарий с устаревшей или ошибочной информацией не разъясняет программу, а только создает лишнюю работу для программиста. Но такая локальная проблема иногда порождает тезис, что *все* комментарии плохи. Его апологеты считают, что каждый комментарий должен заменяться более понятным кодом вплоть до момента, когда в программе вообще не останется комментариев.

Комментарии пишутся на английском (или другом языке, на котором общается программист), что дает возможность гораздо более полно и подробно передавать информацию, чем с помощью имен переменных, функций или классов. Тем не менее написать лаконичные и эффективные комментарии непросто. Комментарии, как и код, приходится неоднократно редактировать. Наш код нам абсолютно понятен после того, как он написан, поэтому комментарии могут показаться бессмысленной и лишней работой. Тут и возникает мнение: комментарии излишни.

Но чаще на практике в программах слишком мало комментариев (или их нет вообще) или они так запутаны, что могут дезинформировать. Отказываться от комментариев на этом основании все равно что заявлять: «Перелеты через Атлантический океан безопасны только на 99,999991%, поэтому я лучше поплыву на пароходе».

О том, как пишутся эффективные комментарии, более подробно я расскажу в главе 10.

Итоги

Некие признаки, или запахи кода, указывают на то, что, вероятно, этот код можно написать лучше. Они не всегда требуют изменений, но лучше присмотреться к коду еще раз. Самый распространенный запах кода — дублирование — понуждает программиста к тому, чтобы разместить код в функции или в цикле. Это гарантирует, что изменения будет достаточно внести только в одном месте. Еще один признак возможной проблемы — «магические» числа: загадочные значения, которые можно заменить константами с содержательными именами. Также стоит упомянуть о закомментированном и мертвом коде, который никогда не выполняется компьютером. Он может сбить с толку программистов, которые впоследствии будут его читать. Лучше удалить такие фрагменты полностью и воспользоваться системой контроля версий (такой как Git), если позднее вам потребуется снова включить их в свою программу.

Механизм отладочного вывода использует вызовы `print()` для вывода отладочной информации. Несмотря на простоту такого подхода, в долгосрочной перспективе для диагностики ошибок лучше положиться на отладчик и журнальные файлы.

Переменные с числовыми суффиксами (`x1`, `x2`, `x3` и т. д.) стоит заменить одной переменной, содержащей список. В отличие от таких языков, как Java, в Python для группировки функций используются не классы, а модули. Класс, содержащий только один метод или только статические методы, можно также рассматривать как запах кода, предполагающий, что код лучше разместить в модуле, а не в классе. И хотя списковые включения предоставляют компактный механизм создания списковых значений, вложенные списковые включения обычно очень плохо читаются.

Кроме того, любые исключения, обрабатываемые пустыми блоками `except`, указывают на то, что вы просто игнорируете ошибку, вместо того чтобы обработать ее. Короткое, невразумительное сообщение об ошибке так же бесполезно для пользователя, как и отсутствие сообщения.

Наряду с этими признаками возможных проблем стоит упомянуть и мифы о них: советы по программированию, которые перестали быть актуальными или со временем стали нерациональными. Это и размещение только одной команды `return` или блока `try-except` в каждой функции, полный отказ от использования аргументов-флагов и глобальных переменных и вера в то, что комментарии в коде излишни.

Конечно, как это обычно бывает со всеми советами в области программирования, запахи кода, о которых здесь шла речь, могут не пригодиться вам при реализации вашего проекта или не соответствовать вашим предпочтениям. Все весьма субъективно. С появлением практического опыта вы сами сделаете выводы относительно того, какой код хорошо читается или является надежным, но, надеюсь, что вам все-таки пригодятся краткие рекомендации, изложенные в этой главе.

6

Написание питонического кода



Мощный — бессмысленный эпитет для языков программирования. Каждый язык программирования описывает себя как мощный. Официальный учебник Python начинается с фразы «Python — доступный и мощный язык программирования».

Но не существует алгоритма, который можно реализовать на одном языке и нельзя — на других, и нет единиц для оценки «мощи» языка программирования (хотя, конечно, можно выбрать в качестве параметра громкость изложения программистами аргументов в пользу своего любимого языка).

Но у каждого языка имеются свои паттерны проектирования и скрытые ловушки, которые составляют его сильные и слабые стороны. Чтобы писать код Python на профессиональном уровне, недостаточно знать синтаксис и стандартную библиотеку. Необходимо изучить *идиомы*, то есть практики программирования, специфические для Python. Некоторые языковые средства Python позволяют отлично писать код в стиле, который называется *питоническим*.

В этой главе я покажу некоторые популярные способы написания идиоматического кода Python и его непитонических аналогов. Вопрос о том, какой код считать питоническим, как правило, каждый программист решает сам, но о наиболее распространенных приемах и методах я расскажу ниже. Опытные программисты Python часто применяют эти приемы, и, если вы будете знать их, это поможет вам идентифицировать их в реальном коде.

«Дзен Python»

«Дзен Python» — набор из 20 руководящих принципов проектирования языка Python и программ Python, написанный Тимом Петерсом. Вы не обязаны следовать

всем этим принципам, но их стоит держать в голове. Кроме того, «Дзен Python» — это скрытая шутка, которая появляется при выполнении команды `import this`:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
--snip--
```

ПРИМЕЧАНИЕ

Как ни странно, записаны только 19 принципов. Говорят, Гвидо ван Россум, создатель Python, заявил, что отсутствующий 20-й пункт был «странной шуткой Тима Петерса, понятной только посвященным». Тим оставил в списке пустое место и предложил заполнить его Гвидо, но похоже, у того так и не нашлось времени.

В конечном итоге эти тезисы — всего лишь изложение мнения, с которым программисты могут соглашаться или не соглашаться. Как и во всех хороших сводах нравственных норм, они противоречат друг другу, что обеспечивает наибольшую гибкость в их реализации. Моя интерпретация этих тезисов изложена ниже.

Красивое лучше, чем уродливое. Красивым обычно считается код, который легко читается и воспринимается. Нередко программисты пишут код слишком быстро, не заботясь об удобочитаемости. Компьютер выполнит нечитаемый код, но у программистов возникнут сложности с его сопровождением и отладкой. Красота — понятие субъективное, но код, при написании которого программист не позаботился о том, как он будет читаться, другим часто кажется уродливым. Одна из причин популярности Python как раз связана с тем, что его синтаксис не загромождается загадочными знаками препинания, как в других языках, что упрощает работу с ним.

Явное лучше, чем неявное. Если бы в качестве пояснения данного правила я сказал «это очевидно», это было бы ужасно. Так и в коде лучше выражаться развернуто и явно. Не стоит прятать функциональность в невнятном коде, понимание которого требует знания всех тонкостей языка.

Простое лучше, чем сложное. Сложное лучше, чем запутанное. Эти два утверждения напоминают нам, что все можно строить как простыми, так и сложными средствами. Если перед вами стоит простая задача, для решения которой нужна лопата, использовать 50-тонный гидравлический бульдозер неэффективно. Но для выполнения грандиозной задачи сложность управления одним бульдозером ничто по сравнению с координацией команды из 100 землекопов. Отдавайте предпочтение простоте перед сложностью, но знайте границы простоты.

Плоское лучше, чем вложенное. Программисты любят делить свой код на категории, особенно если те содержат подкатегории, которые содержат другие

подкатегории. Эти иерархии часто продуцируют не столько организацию, сколько бюрократизм. Ничто не мешает вам написать код, содержащийся всего в одной структуре данных или модуле верхнего уровня. Если в вашем коде часто встречаются конструкции вида `spam.eggs.bacon.ham()` или `spam['eggs']['bacon']['ham']`, вы его переусложнили.

Разреженное лучше, чем плотное. Программисты любят втискивать максимум функциональности в минимальный объем кода, как в следующей строке: `print('\n'.join("%i bytes = %i bits which has %i possible values." % (j, j*8, 256**j-1) for j in (1 << i for i in range(8))))`. Хотя такой код наверняка произведет впечатление на друзей, он приведет в ярость коллег, которым придется в нем разбираться. Не стремитесь к тому, чтобы ваш код делал много всего и сразу. Код, распределенный по нескольким строкам, часто читается проще, чем плотные однострочные конструкции. Это правило означает приблизительно то же, что и «простое лучше, чем сложное».

Удобочитаемость важна. Хотя имя `strcmp()` наверняка означает «сравнение строк» для тех, кто программировал на C с 1970-х годов, у современных компьютеров хватает памяти для полных имен функций. Не пропускайте буквы в именах и не пишите слишком лаконичный код. Не жалейте времени на поиск содержательных, конкретных имен для ваших переменных и функций. Пустая строка между разделами кода решает ту же задачу, что и разбивка на абзацы в книге: она сообщает читателю, какие части должны читаться вместе. Это правило означает приблизительно то же, что и «красивое лучше, чем уродливое».

Особые случаи не настолько особые, чтобы нарушать правила. При этом практичность важнее безупречности. Эти два афоризма противоречат друг другу. В программировании полно передовых практик, которые программистам следует использовать в своем коде. Желание обойти эти приемы ради быстрого хитроумного трюка соблазнительно, но иногда приводит к беспорядочному клубку непоследовательного и нечитаемого кода. С другой стороны, фанатичное следование правилам может стать причиной абстрактного, неудобочитаемого кода. Например, в Java попытка написать весь код в парадигме объектно-ориентированного программирования часто приводит к использованию большого объема шаблонов даже в самой мелкой программе. С опытом вы научитесь проходить по узкой кромке между этими двумя положениями. А со временем не только узнаете правила, но и поймете, когда их можно нарушать.

Ошибки никогда не должны замалчиваться. Если только они не замалчиваются явно. Хотя программисты часто игнорируют сообщения об ошибках, это не значит, что программа должна перестать их выдавать. Чаще всего ошибки замалчиваются, когда функции возвращают коды ошибок или `None` вместо выдачи исключений. Эти два тезиса утверждают, что лучше программировать с расчетом на быстрый сбой

и аварийное завершение, чем на замалчивание ошибки с продолжением выполнения. Ошибки, которые неизбежно возникнут позднее, создадут больше проблем с отладкой, потому что они проявят себя далеко от причины, их вызвавшей. Хотя вы можете взять за правило явно игнорировать ошибки, возникающие в вашей программе, убедитесь, что это ваше сознательное решение.

Столкнувшись с неоднозначностью, боритесь с искушением угадать. Компьютеры делают людей суеверными: чтобы изгнать демонов из наших компьютеров, мы выполняем священный ритуал, перезагружая их. Предполагается, что это решит любую непонятную проблему. Однако в компьютерах нет ничего волшебного. Если код не работает, тому есть причина, и проблему можно решить, только призвав на помощь критическое мышление. Боритесь с искушением искать причину наугад, пока что-то не заработает; часто вы всего лишь маскируете проблему, а не решаете ее.

Должен быть один — и желательно только один — очевидный способ сделать это. Этот тезис напрямую противоречит девизу языка программирования Perl: «Это можно сделать разными способами!». Как выясняется, возможность написать код для решения одной задачи тремя-четырьмя разными способами становится палкой о двух концах: у вас появляется большая гибкость в написании кода, но зато вам придется изучать все возможные способы его написания, чтобы читать чужой код. Гибкость не оправдывает лишних усилий, которые потребуются для изучения языка программирования.

Хотя это может быть и не очевидно, если только вы не голландец. Это шутка. Ибо Гвидо ван Россум, создатель Python, — голландец.

Сейчас лучше, чем никогда. Хотя никогда зачастую лучше, чем *прямо* сейчас. Это о том, что код, который работает медленно, очевидно хуже, чем тот, который работает быстро. Но лучше дождаться завершения программы, чем завершить ее слишком рано с неправильными результатами.

Если реализацию сложно объяснить, идея плоха. Если реализацию легко объяснить, возможно, идея хороша. Многие вещи усложняются со временем: налоговое законодательство, романтические отношения, книги по программированию на языке Python. То же можно сказать и о программах. Эти два положения напоминают нам: если код настолько сложен, что программист не сможет его понять и отладить, это плохой код. Но если код объясняется легко, это не обязательно означает, что он хорош. К сожалению, понять, как сделать код простым настолько, насколько это возможно, и ни на йоту проще, достаточно сложно.

Пространства имен — отличная штука. Сделаем их побольше! Пространства имен представляют собой изолированные контейнеры для идентификаторов, предотвращающие конфликты имен. Например, встроенная функция `open()` и функция

`.open()` у `webbrowser` имеют одинаковые имена, но это разные функции. Импортирование `webbrowser` не замещает встроенную функцию `open()`, потому что две функции существуют в разных пространствах имен: встроенном пространстве имен и пространстве имен модуля `webbrowser` соответственно. Однако не забывайте, что плоское лучше вложенного: какими бы замечательными ни были пространства имен, создавать их следует только для предотвращения конфликтов имен, а не для добавления лишней систематизации.

Как и в отношении всех мнений в области программирования, вы можете возразить мне, что мои советы могут быть просто неактуальными для вашей ситуации. Споры о том, как следует писать код или какой код считать питоническим, редко бывают такими плодотворными, как это кажется на первый взгляд. (Если только вы не собираете в книгу одни лишь субъективные мнения.)

Как полюбить значимые отступы

Самая распространенная претензия к Python, которую я слышу от программистов с опытом работы на других языках, — необычность и непривычность *значимых отступов* (которые часто по ошибке называют *значимыми пробелами*). Количество отступов в начале строки кода имеет смысл в Python, потому что оно определяет, какие строки кода принадлежат тому же программному блоку.

Группировка блоков кода в Python при помощи отступов может показаться странной, потому что в других языках блоки начинаются и завершаются фигурными скобками: `{` и `}`. Однако программисты с опытом работы на других языках тоже обычно снабжают свои блоки отступами, как и программисты на Python, чтобы их код лучше читался. Например, в Java значимых отступов нет. Программистам Java не нужно использовать отступы в блоках, но они обычно все равно делают это ради удобочитаемости. В следующем примере функция `Java main()` содержит только один вызов функции `println()`:

```
// Пример на Java
public static void main(String[] args) {
    System.out.println("Hello, world!");
}
```

Этот код Java вполне нормально работал бы и без отступов в строке `println()`, потому что начало и конец блоков в Java отмечаются фигурными скобками, а не отступами. Python не разрешает применять отступы по вашему усмотрению, а заставляет программиста последовательно обеспечивать удобочитаемость кода. Однако следует заметить, что в Python нет значимых пробелов, потому что Python не ограничивает использование пробельных символов вне отступов (и `2 + 2`, и `2+2` являются допустимыми выражениями Python).

Некоторые программисты считают, что открывающая фигурная скобка должна быть в одной строке с открывающей командой, а другие полагают, что ее следует размещать в следующей строке. Программисты могут без конца спорить о преимуществах выбранного стиля. Python изящно обходит эту проблему, вообще отказавшись от фигурных скобок, чтобы программисты занимались более продуктивной работой. Я бы предпочел, чтобы все языки программирования последовали примеру Python в области группировки блоков.

Но некоторые люди все равно привыкли к фигурным скобкам и требуют добавить их в будущую версию Python — при всей их непитоничности. Модуль Python `__future__` выполняет обратное импортирование функциональности в более ранние версии Python, но при попытке импортировать фигурные скобки в Python обнаруживается пасхалка — скрытое послание:

```
>>> from __future__ import braces
SyntaxError: not a chance
```

Я бы не рассчитывал на то, что фигурные скобки будут добавлены в Python в обозримом будущем.

Использование модуля `timeit` для оценки быстродействия

«Предварительная оптимизация — корень всех зол», — говорил Дональд Кнут, знаменитый ученый из Стэнфорда. Предварительная оптимизация, или оптимизация быстродействия программы до того, как вы будете располагать реальными данными о ее быстродействии, часто встречается тогда, когда программисты пытаются применять хитроумные программные трюки для экономии памяти или ускорения работы кода. Например, некоторые программисты используют алгоритм XOR (исключающее ИЛИ), для того чтобы поменять местами два целых числа в памяти без использования третьей временной переменной (как предполагается, для экономии памяти):

```
>>> a, b = 42, 101 # Создание двух переменных.
>>> print(a, b)
42 101
>>> a = a ^ b
>>> b = a ^ b
>>> a = a ^ b
>>> print(a, b) # Значения поменялись местами.
101 42
```

Если вы не знакомы с алгоритмом XOR (использующим поразрядный оператор XOR `^`; о нем можно узнать по адресу <https://ru.wikipedia.org/wiki/XOR-обмен>), этот код выглядит загадочно. Проблема хитроумных трюков в том, что они приводят

к созданию запутанного, нечитаемого кода. Как говорится в тезисах «Дзен Python», «удобочитаемость важна».

Что еще хуже, ваш хитроумный трюк может оказаться не таким уж хитрым и умным. Нельзя бездоказательно полагать, что он поможет; единственный способ это узнать — измерить и сравнить время выполнения. Вспомните принцип «Дзена Python» — «столкнувшись с неоднозначностью, боритесь с искушением угадать». *Профилирование* называется систематический анализ скорости, использования памяти и других характеристик программы. Модуль `timeit` стандартной библиотеки Python может профилировать скорость выполнения способом, управляющим основными искажающими факторами, из-за чего данные получаются более точными, чем при простой регистрации времени запуска и завершения программы. Не пишите собственный непитонический код профилирования, который может выглядеть примерно так:

```
import time
startTime = time.time() # Сохранение времени запуска.
for i in range(1000000): # Код выполняется 1 000 000 раз.
    a, b = 42, 101
    a = a ^ b
    b = a ^ b
    a = a ^ b
print(time.time() - startTime, 'seconds') # Вычисление затраченного времени.
```

На моей машине при выполнении этого кода выводится следующий результат:

```
0.28080058097839355 seconds
```

Вместо этого можно передать код Python в строковом аргументе функции `timeit.timeit()`; функция сообщает среднее время, затраченное на выполнение строки кода 1 000 000 раз. Если вы хотите проверить фрагмент из нескольких строк кода, разделите их символом `;`:

```
>>> timeit.timeit('a, b = 42, 101; a = a ^ b; b = a ^ b; a = a ^ b')
0.19474047500000324
```

На моем компьютере выполнение этого кода заняло приблизительно 1/5 секунды. Насколько это быстро? Сравним с кодом перестановки целых чисел, использующим третью временную переменную:

```
>>> timeit.timeit('a, b = 42, 101; temp = a; a = b; b = temp')
0.09632604099999753
```

Сюрприз! Код с третьей временной переменной не только лучше читается, но и выполняется почти вдвое быстрее метода XOR! «Умный» способ экономит несколько байтов памяти, но за счет скорости и удобочитаемости кода. Если только вы не пишете программы для крупного центра обработки данных, жертвовать удобочитаемостью для экономии нескольких байтов памяти или наносекунд процессорного

времени не стоит. В конце концов, память стоит дешево, и только на чтение этого абзаца вы потратили миллиарды наносекунд.

Что еще лучше, значения двух переменных можно поменять местами при помощи трюка *множественного присваивания*, также называемого *итерируемой распаковкой*:

```
>>> timeit.timeit('a, b = 42, 101; a, b = b, a')
0.0846703500001033
```

Этот код получается не только самым удобочитаемым, но и самым быстрым! И мы знаем это не потому, что предположили, но и потому, что провели объективные измерения.

Функция `time.time()` также может получить второй строковый аргумент с кодом настройки. Код настройки выполняется однократно до кода первой строки для выполнения любой необходимой подготовки. Этот код может импортировать какой-нибудь модуль, присвоить начальное значение переменной или выполнить другое необходимое действие. Также можно изменить количество испытаний по умолчанию; для этого следует передать целое число в ключевом аргументе `number`. Например, следующий тест измеряет время, необходимое модулю Python `random` для генерирования 10 000 000 случайных чисел от 1 до 100. (На моей машине для этого требуется около 10 секунд.)

```
>>> timeit.timeit('random.randint(1, 100)', 'import random', number=10000000)
10.020913950999784
```

Хорошее правило для программистов: сначала заставьте свой код работать, а потом переходите к оптимизации. Когда у вас уже имеется работоспособная программа, тогда и стоит браться за повышение ее эффективности.

Неправильное использование синтаксиса

Если Python — не первый ваш язык программирования, то при написании кода Python вы можете использовать стратегии, знакомые вам по другим языкам. А может, вы изобрели необычный способ написания кода Python, потому что не знали, что существуют общепринятые практики. Ваш неуклюжий код работает, но изучение стандартных способов написания питонического кода экономит время и усилия. В этом разделе я объясняю неправильные решения, используемые программистами, и то, как должен выглядеть правильно написанный код.

Использование `enumerate()` вместо `range()`

При переборе списка или другой последовательности некоторые программисты используют функции `range()` и `len()` для генерирования целых индексов от 0 до длины последовательности (не включая последнее значение). В таких циклах `for`

часто используется переменная с именем `i` (сокращение от `index`). Например, введите следующий непитонический пример в интерактивной оболочке:

```
>>> animals = ['cat', 'dog', 'moose']
>>> for i in range(len(animals)):
...     print(i, animals[i])
...
0 cat
1 dog
2 moose
```

Идиома `range(len())` прямолинейна, но далеко не идеальна, потому что она плохо читается. Вместо этого лучше передать список или последовательность встроенной функции `enumerate()`, которая возвращает целочисленный индекс и элемент с этим индексом. Например, можно написать питонический код следующего вида:

```
>>> # Пример питонического кода
>>> animals = ['cat', 'dog', 'moose']
>>> for i, animal in enumerate(animals):
...     print(i, animal)
...
0 cat
1 dog
2 moose
```

Код с `enumerate()` получается чуть более чистым, чем код с `range(len())`. Если вам нужны только элементы, но не индексы, все равно можно напрямую перебрать список питоническим способом:

```
>>> # Пример питонического кода
>>> animals = ['cat', 'dog', 'moose']
>>> for animal in animals:
...     print(animal)
...
cat
dog
moose
```

Вызов `enumerate()` с прямым перебором последовательности предпочтительнее использования устаревшей схемы `range(len())`.

Использование команды `with` вместо `open()` и `close()`

Функция `open()` возвращает объект файла, содержащий методы для чтения и записи в файл. После завершения работы метод `close()` объекта файла делает файл доступным для чтения и записи со стороны других программ. Эти функции можно использовать по отдельности, но такой подход не соответствует питоническому стилю. Например, введите следующий фрагмент в интерактивной оболочке, чтобы вывести текст «Hello, world!» в файл с именем `spam.txt`:

```
>>> # Пример непитонического кода
>>> fileObj = open('spam.txt', 'w')
>>> fileObj.write('Hello, world!')
13
>>> fileObj.close()
```

Такой код может привести к тому, что файл останется открытым — скажем, если в блоке `try` возникнет ошибка и вызов `close()` будет пропущен. Пример:

```
>>> # Пример непитонического кода
>>> try:
...     fileObj = open('spam.txt', 'w')
...     eggs = 42 / 0 # Здесь происходит ошибка деления на ноль.
...     fileObj.close() # Эта строка никогда не выполняется.
... except:
...     print('Some error occurred.')
...
Some error occurred.
```

При возникновении ошибки деления на ноль выполнение передается в блок `except`, вызов `close()` пропускается, а файл останется открытым. Позднее это может привести к повреждению структуры файла, причины которой будет трудно связать с блоком `try`.

Но можно воспользоваться командой `with`, чтобы функция `close()` автоматически вызывалась при выходе управления из блока `with`. Следующий питонический пример делает то же самое, что и первый пример в этом разделе:

```
>>> # Пример питонического кода.
>>> with open('spam.txt', 'w') as fileObj:
...     fileObj.write('Hello, world!')
...
```

Несмотря на отсутствие явного вызова `close()`, команда `with` обязательно вызовет ее при выходе управления за пределы блока.

Использование `is` для сравнения с `None` вместо `==`

Оператор `==` сравнивает значения двух объектов, тогда как оператор `is` сравнивает два объекта на тождественность. Различия между равенством и тождественностью объясняются в главе 7. Два объекта могут хранить одинаковые значения, но если это два разных объекта, они не тождественны. Однако при сравнении значения с `None` практически всегда следует использовать оператор `is` вместо оператора `==`.

В некоторых случаях выражение `spam == None` может дать результат `True`, даже если `spam` всего лишь содержит `None`. Это может произойти из-за перегрузки оператора `==`

(эта тема более подробно рассматривается в главе 17). Но выражение `spam is None` проверит, является ли значение, хранящееся в переменной `spam`, буквальным значением `None`. Так как `None` является единственным значением типа данных `NoneType`, в любой программе Python существует только один объект `None`. Если переменной присвоено значение `None`, сравнение `is None` всегда дает результат `True`. Особенности перегрузки оператора `==` рассматриваются в главе 17, но ниже приведен пример такого кода:

```
>>> class SomeClass:
...     def __eq__(self, other):
...         if other is None:
...             return True
...
>>> spam = SomeClass()
>>> spam == None
True
>>> spam is None
False
```

Вероятность того, что класс перегружает оператор `==` подобным образом, невелика, но в Python стало идиоматичным всегда использовать `is None` вместо `== None` просто на всякий случай.

Наконец, оператор `is` не следует использовать со значениями `True` и `False`. Оператор проверки равенства `==` может использоваться для сравнения значения с `True` или `False` (например, `spam == True` или `spam == False`). Еще чаще оператор и логическое значение полностью опускаются, а код записывается в виде `if spam:` или `if not spam:` вместо `if spam == True:` или `if spam == False:`.

Форматирование строк

Строки встречаются почти во всех компьютерных программах независимо от языка. Это весьма распространенный тип данных, и не приходится удивляться тому, что существует множество подходов к выполнению строковых операций и форматированию строк. В этом разделе я показываю пару приемов.

Использование необработанных строк, если строка содержит много символов \ (обратный слэш)

Escape-символы позволяют вставлять в строковые литералы текст, который в противном случае было бы невозможно включить¹. Например, в строку `'Zophie\'s`

¹ Escape-последовательности, то есть последовательности, которые начинаются с символа «\», за которым следует один или более символов, также называют экранированными последовательностями. — *Примеч. ред.*

`chair'` необходимо включить символ `\`, чтобы Python интерпретировал вторую кавычку как часть строки, а не как символ, завершающий конец строки. Так как символ `\` является *escape*-символом, если вы хотите включить литеральный символ `\` в строку, его необходимо ввести в виде `\\`.

Необработанные (raw) строки представляют собой строковые литералы с префиксом `r`; они не интерпретируют символы `\` как *escape*-символы. Вместо этого в строку просто включается символ `\`. Например, следующая строка с путем к файлу Windows должна содержать множество экранированных символов `\`, что не соответствует питоническому стилю:

```
>>> # Пример непитонического кода
>>> print('The file is in C:\\Users\\Al\\Desktop\\Info\\Archive\\Spam')
The file is in C:\Users\Al\Desktop\Info\Archive\Spam
```

Необработанная строка генерирует то же строковое значение, но читается гораздо лучше:

```
>>> # Пример питонического кода
>>> print(r'The file is in C:\Users\Al\Desktop\Info\Archive\Spam')
The file is in C:\Users\Al\Desktop\Info\Archive\Spam
```

Необработанные строки не определяют другой тип строковых данных; это всего лишь удобный способ записи строковых литералов, содержащих несколько внутренних символов `\`. Необработанные строки часто используются для записи регулярных выражений или путей к файлам Windows с внутренними символами `\`, которые было бы слишком неудобно экранировать по отдельности символами `\\`.

Форматирование с использованием F-строк

Строковой интерполяцией называется процесс создания строк, включающих другие строки; интерполяция имеет долгую историю в Python. Сначала для конкатенации строк можно было использовать оператор `+`, но это приводило к появлению кода с множеством кавычек и плюсов: `'Hello, ' + name + '. Today is ' + day + ' and it is ' + weather + '.'`. Спецификатор преобразования `%` несколько упростил этот синтаксис: `'Hello, %s. Today is %s and it is %s.' % (name, day, weather)`. В обоих случаях строки из переменных `name`, `day` и `weather` вставляются в строковые литералы для генерирования нового строкового значения: `'Hello, Al. Today is Sunday and it is sunny.'`

Строковый метод `format()` добавляет мини-язык форматных спецификаций (<https://docs.python.org/3/library/string.html#formatspec>), в котором пары фигурных скобок `{}` используются способом, напоминающим спецификатор формата `%s`. Тем не менее это довольно запутанный способ, который может привести к созданию нечитаемого кода, поэтому использовать его я не рекомендую.

Но с выходом Python 3.6 *f-строки* (сокращение от format strings) предоставляют более удобный способ создания строк, включающих другие строки. Подобно тому как у необработанных строк перед первой кавычкой ставится префикс `r`, *f-строки* помечаются префиксом `f`. В *f-строки* можно включать имена переменных в фигурных скобках, чтобы вставлять строки, хранящиеся в этих переменных:

```
>>> name, day, weather = 'Al', 'Sunday', 'sunny'
>>> f'Hello, {name}. Today is {day} and it is {weather}.'
'Hello, Al. Today is Sunday and it is sunny.'
```

Фигурные скобки могут содержать целые выражения:

```
>>> width, length = 10, 12
>>> f'A {width} by {length} room has an area of {width * length}.'
'A 10 by 12 room has an area of 120.'
```

Если в *f-строку* нужно включить фигурную скобку как литерал, экранируйте ее дополнительной фигурной скобкой:

```
>>> spam = 42
>>> f'This prints the value in spam: {spam}'
'This prints the value in spam: 42'
>>> f'This prints literal curly braces: {{spam}}'
'This prints literal curly braces: {spam}'
```

Так как имена переменных и выражений можно встраивать прямо в строку, код читается лучше, чем со старыми средствами форматирования строк.

Все эти разные способы форматирования противоречат тезису из свода правил «Дзен Python», согласно которому должно существовать одно — и желательно только одно — очевидное решение. Но *f-строки* являются усовершенствованием языка (по моему мнению), а, как указано в других рекомендациях, практичность важнее безупречности. Если вы пишете код только для Python 3.6 и выше, используйте *f-строки*. Если ваш код может выполняться в более ранних версиях Python, придерживайтесь метода `format()` или спецификаторов преобразования `%s`.

Поверхностное копирование списков

Синтаксис сегментов позволяет легко создавать новые строки или списки на базе уже существующих. Чтобы увидеть, как это делается, введите следующие команды в интерактивной оболочке:

```
>>> 'Hello, world!'[7:12] # Создание строки из большей строки.
'world'
>>> 'Hello, world!':[5] # Создание строки из большей строки.
'Hello'
```

```
>>> ['cat', 'dog', 'rat', 'eel'][2:] # Создание списка из большего списка.  
['rat', 'eel']
```

Двоеточие (:) разделяет начальный и конечный индексы элементов, помещаемые в создаваемый список. Если начальный индекс перед двоеточием не указан (как в 'Hello, world!')[:5]), то начальный индекс по умолчанию равен 0. Если опустить конечный индекс после двоеточия, как в ['cat', 'dog', 'rat', 'eel'][2:], то конечным индексом по умолчанию становится конец списка.

Если опущены оба индекса, то начальный индекс равен 0 (начало списка), а конечный индекс соответствует концу списка. Фактически эта конструкция создает копию списка:

```
>>> spam = ['cat', 'dog', 'rat', 'eel']  
>>> eggs = spam[:]  
>>> eggs  
['cat', 'dog', 'rat', 'eel']  
>>> id(spam) == id(eggs)  
False
```

Обратите внимание: списки `spam` и `eggs` не тождественны. Строка `eggs = spam[:]` создает *поверхностную копию* списка в `spam`, тогда как `eggs = spam` копирует только ссылку на список. Но синтаксис `[ : ]` выглядит немного странно, а создание поверхностной копии списка функцией `copy()` модуля `copy` читается значительно лучше:

```
>>> # Пример питонического кода.  
>>> import copy  
>>> spam = ['cat', 'dog', 'rat', 'eel']  
>>> eggs = copy.copy(spam)  
>>> id(spam) == id(eggs)  
False
```

Вы должны знать об этом странном синтаксисе на случай, если вам попадется код Python, в котором он используется, но я не рекомендую применять его в своем коде. Помните, что как `[ : ]`, так и вызов `copy.copy()` создают поверхностные копии.

Питонические способы использования словарей

Словари играют важную роль во многих программах Python из-за гибкости пар «ключ — значение» (см. главу 7), связывающих один вид данных с другим. А значит, вам пригодятся некоторые словарные идиомы, часто используемые в коде Python.

За дополнительной информацией о словарях обращайтесь к великолепным докладам программиста Python Брэндона Родса (Brandon Rhodes), посвященным словарям и тому, как они работают: «The Mighty Dictionary» на конференции

PyCon 2010 (<https://inropy.com/mightydictionary>) и «The Dictionary Even Mightier» на конференции PyCon 2017 (<https://inropy.com/dictionaryevenmightier>).

Использование `get()` и `setdefault()` со словарями

Попытка обратиться к несуществующему ключу словаря приводит к ошибке `KeyError`, поэтому для предотвращения ошибки программисты часто пишут непитонический код:

```
>>> # Пример непитонического кода
>>> numberOfPets = {'dogs': 2}
>>> if 'cats' in numberOfPets: # Проверить, существует ли ключ 'cats'.
...     print('I have', numberOfPets['cats'], 'cats.')
... else:
...     print('I have 0 cats.')
...
I have 0 cats.
```

Этот код проверяет, существует ли строка `'cats'` как ключ в словаре `numberOfPets`. Если ключ существует, то вызов `print()` обращается к `numberOfPets['cats']` как части сообщения для пользователя. Если ключ не существует, то другой вызов `print()` выводит строку без обращения к `numberOfPets['cats']`, поэтому исключение `KeyError` не выдается.

Данная схема встречается настолько часто, что у словарей имеется метод `get()`, который позволяет задать значение по умолчанию, возвращаемое в случае, если ключ не существует в словаре. Следующий питонический код эквивалентен предыдущему примеру:

```
>>> # Пример питонического кода.
>>> numberOfPets = {'dogs': 2}
>>> print('I have', numberOfPets.get('cats', 0), 'cats.')
I have 0 cats.
```

Вызов `numberOfPets.get('cats', 0)` проверяет, существует ли ключ `'cats'` в словаре `numberOfPets`. Если он существует, то вызов метода возвращает значение для ключа `'cats'`. Если ключ не существует, вместо значения возвращается второй аргумент `0`. Использование метода `get()` с определением значения по умолчанию, которое должно использоваться для несуществующих ключей, короче и лучше читается, чем решение с командами `if-else`.

И наоборот, может потребоваться задать значение по умолчанию, если ключ не существует. Например, если словарь из `numberOfPets` не содержит ключа `'cats'`, команда `numberOfPets['cats'] += 10` приводит к ошибке `KeyError`. Можно добавить код, который проверяет возможное отсутствие ключа и задает значение по умолчанию:

```
>>> # Пример непитонического кода
>>> numberOfPets = {'dogs': 2}
>>> if 'cats' not in numberOfPets:
...     numberOfPets['cats'] = 0
...
>>> numberOfPets['cats'] += 10
>>> numberOfPets['cats']
10
```

Но этот паттерн встречается настолько часто, что у словарей имеется более питонический метод `setdefault()`. Следующий код эквивалентен предыдущему:

```
>>> # Пример питонического кода.
>>> numberOfPets = {'dogs': 2}
>>> numberOfPets.setdefault('cats', 0) # Ничего не делать, если 'cats' существует.
0
>>> workDetails['cats'] += 10
>>> workDetails['cats']
10
```

Если вы пишете команды `if`, которые проверяют, существует ли код в словаре, и задают значение по умолчанию при его отсутствии, используйте метод `setdefault()`.

Использование `collections.defaultdict` для значений по умолчанию

Класс `collections.defaultdict` можно использовать для полного устранения ошибок `KeyError`. Этот класс позволяет создать словарь по умолчанию; для этого импортируйте модуль `collections` и вызовите метод `collections.defaultdict()`, передав ему тип данных, который должен использоваться для значения по умолчанию. Например, передавая `int` методу `collections.defaultdict()`, можно создать объект, похожий на словарь, в котором `0` используется как значение по умолчанию для несуществующих ключей. Введите следующий фрагмент в интерактивной оболочке:

```
>>> import collections
>>> scores = collections.defaultdict(int)
>>> scores
defaultdict(<class 'int'>, {})
>>> scores['Al'] += 1 # Не нужно сначала задавать значение для ключа 'Al'.
>>> scores
defaultdict(<class 'int'>, {'Al': 1})
>>> scores['Zophie'] # Не нужно сначала задавать значение для ключа 'Zophie'.
0
>>> scores['Zophie'] += 40
>>> scores
defaultdict(<class 'int'>, {'Al': 1, 'Zophie': 40})
```


Обратите внимание: вы передаете функцию `int()`, а не вызываете ее, что позволяет опустить круглые скобки после `int` в `collections.defaultdict(int)`. Также можно передать список, который будет использоваться как пустой список, в значении по умолчанию. Введите следующий фрагмент в интерактивной оболочке:

```
>>> import collections
>>> booksReadBy = collections.defaultdict(list)
>>> booksReadBy['Al'].append('Oryx and Crake')
>>> booksReadBy['Al'].append('American Gods')
>>> len(booksReadBy['Al'])
2
>>> len(booksReadBy['Zophie']) # Значение по умолчанию - пустой список.
0
```

Если вам нужно значение по умолчанию для каждого возможного ключа, использовать `collections.defaultdict()` намного проще, чем использовать обычный словарь и многократно вызывать метод `setdefault()`.

Использование словарей вместо команды `switch`

В таких языках, как Java, существует команда `switch` — разновидность команды `if-elif-else`, выполняющей код в зависимости от того, какое из многих значений содержит конкретная переменная. В Python нет команды `switch`, поэтому программисты Python иногда пишут такой код, как в следующем примере. Он выполняет разные команды присваивания в зависимости от того, какое из многих значений содержит переменная `season`:

```
# Все следующие условия if и elif содержат "season == ":
if season == 'Winter':
    holiday = 'New Year\'s Day'
elif season == 'Spring':
    holiday = 'May Day'
elif season == 'Summer':
    holiday = 'Juneteenth'
elif season == 'Fall':
    holiday = 'Halloween'
else:
    holiday = 'Personal day off'
```

Этот код не обязательно является непитоническим, но он получается слишком длинным. По умолчанию команды `switch` в Java работают по принципу «сквозного прохождения», из-за которого каждый блок должен завершаться командой `break`. В противном случае выполнение продолжается в следующем блоке. Забытые команды `break` часто становятся источником ошибок. Но обилие команд `if-elif` в нашем примере выглядит однообразно. Некоторые программисты Python предпочитают

создать словарь, вместо того чтобы использовать команды `if-elif`. Следующий компактный и питонический код эквивалентен следующему примеру:

```
holiday = {'Winter': 'New Year\'s Day',
           'Spring': 'May Day',
           'Summer': 'Juneteenth',
           'Fall': 'Halloween'}.get(season, 'Personal day off')
```

Этот код является одной командой присваивания. В `holiday` сохраняется возвращаемое значение вызова метода `get()`, который возвращает значение для ключа, присвоенного `season`. Если ключ `season` не существует, то `get()` возвращает строку `'Personal day off'`. Использование словаря делает код более компактным, но также усложняет чтение кода. Решайте сами, хотите вы использовать этот паттерн или нет.

Условные выражения: «некрасивый» тернарный оператор Python

Тернарные операторы (в Python их официально называют *условными выражениями*, или тернарными выражениями выбора) вычисляют выражение и выбирают в качестве результата одно из двух значений в зависимости от условия. Обычно это делается следующей питонической командой `if-else`:

```
>>> # Пример питонического кода.
>>> condition = True
>>> if condition:
...     message = 'Access granted'
... else:
...     message = 'Access denied'
...
>>> message
'Access granted'
```

Тернарный означает, что оператор получает три входных значения, но в программировании оно синонимично условному выражению. Условные выражения также предлагают более компактное однострочное решение, соответствующее этому паттерну. В Python они реализуются странным расположением ключевых слов `if` и `else`:

```
>>> valueIfTrue = 'Access granted'
>>> valueIfFalse = 'Access denied'
>>> condition = True
>>> message = valueIfTrue if condition else valueIfFalse ❶
>>> message
'Access granted'
>>> print(valueIfTrue if condition else valueIfFalse)    ❷
'Access granted'
>>> condition = False
```

```
>>> message = valueIfTrue if condition else valueIfFalse
>>> message
'Access denied'
```

Выражение `valueIfTrue if condition else valueIfFalse` ❶ дает результат `valueIfTrue`, если переменная `condition` равна `True`. Если переменная `condition` содержит `False`, то выражение дает результат `valueIfFalse`. Гвидо ван Россум шутливо описывал спроектированный им синтаксис как «намеренно уродливый». Во многих языках с тернарным оператором сначала указывается условие, затем значение для `True` и потом значение для `False`. Условное выражение может использоваться везде, где может применяться выражение или значение, включая аргумент вызова функции ❷.

Почему этот синтаксис был включен в Python 2.5, хотя он и нарушает первую рекомендацию «красивое лучше, чем уродливое»? К сожалению, многие программисты используют тернарные операторы, несмотря на их неудобочитаемость, и хотят, чтобы этот синтаксис поддерживался в Python. Злоупотребление ускоренным вычислением логических операторов позволяет создать некую разновидность тернарного оператора. Выражение `condition and valueIfTrue or valueIfFalse` дает результат `valueIfTrue`, если `condition` содержит `True`, и `valueIfFalse`, если `condition` содержит `False` (кроме одного важного случая). Введите следующий фрагмент в интерактивной оболочке:

```
>>> # Пример непитонического кода.
>>> valueIfTrue = 'Access granted'
>>> valueIfFalse = 'Access denied'
>>> condition = True
>>> condition and valueIfTrue or valueIfFalse
'Access granted'
```

У этого «псевдотернарного» оператора `condition and valueIfTrue or valueIfFalse` есть один неочевидный дефект: если `valueIfTrue` содержит квазиложное значение (например, `0`, `False`, `None` или пустая строка), выражение неожиданно дает результат `valueIfFalse`, хотя `condition` содержит `True`.

Но программисты продолжают использовать этот фиктивный тернарный оператор, а вопрос «почему в Python нет тернарного оператора?» постоянно приходится слышать разработчикам реализации Python. Условные выражения были созданы для того, чтобы программисты перестали требовать тернарный оператор и не использовали псевдотернарный оператор с его скрытыми ошибками. Но условные выражения достаточно уродливы, чтобы отвести программистов от их применения. И хотя «красивое лучше, чем уродливое», «уродливый» тернарный оператор Python можно считать примером того, когда практичность важнее безупречности.

Условные выражения вряд ли можно назвать питоническими, но и однозначно считать их непитоническими тоже нельзя. Если вы пользуетесь ими, избегайте вложения условных выражений в другие условные выражения:

```
>>> # Пример непитонического кода.  
>>> age = 30  
>>> ageRange = 'child' if age < 13 else 'teenager' if age >= 13 and age < 18  
else 'adult'  
>>> ageRange  
'adult'
```

Вложенные условные выражения — хороший пример того, что плотная однострочная конструкция может быть технически правильной, но плохо восприниматься при чтении.

Работа со значениями переменных

При написании кода часто возникает необходимость в проверке и изменении значений, хранящихся в переменных. В Python это можно сделать несколькими способами. Рассмотрим пару примеров.

Сцепление операторов присваивания и сравнения

Когда требуется проверить, принадлежит ли число некоторому диапазону, можно воспользоваться логическим оператором `and` в следующей конструкции:

```
# Пример непитонического кода.  
if 42 < spam and spam < 99:
```

Но Python позволяет формировать цепочки операторов сравнения, чтобы вам не приходилось использовать оператор `and`. Следующий код эквивалентен предыдущему примеру:

```
# Пример непитонического кода.  
if 42 < spam < 99:
```

В цепочки можно объединять и операторы присваивания. Например, одно значение можно присвоить нескольким переменным в одной строке кода:

```
>>> # Пример питонического кода.  
>>> spam = eggs = bacon = 'string'  
>>> print(spam, eggs, bacon)  
string string string
```

Чтобы удостовериться, что все три переменные содержат одинаковые значения, можно воспользоваться оператором `and` или создать цепочку из операторов `==` для проверки равенства:

```
>>> # Пример питонического кода.  
>>> spam = eggs = bacon = 'string'
```

```
>>> spam == eggs == bacon == 'string'
True
```

Сцепление операторов — небольшая, но полезная форма сокращенной записи в Python. Но при неправильном использовании операторы могут создать проблемы. В главе 8 демонстрируются некоторые примеры, когда неправильное применение операторов может создать неожиданные ошибки в вашем коде.

Проверка того, что переменная содержит одно из нескольких значений

Иногда возникает ситуация, обратная описанной в предыдущем разделе: требуется проверить, содержит ли переменная одно из нескольких возможных значений. Для этого можно воспользоваться оператором `or` (как в выражении `spam == 'cat' or spam == 'dog' or spam == 'moose'`). Со всеми избыточными частями `spam ==` выражение становится излишне громоздким.

Вместо этого можно объединить несколько значений в кортеж и проверить, содержится ли значение переменной в кортеже, оператором `in`, как в следующем примере:

```
>>> # Пример питонического кода.
>>> spam = 'cat'
>>> spam in ('cat', 'dog', 'moose')
True
```

Этот код не только более понятен, но и по данным `timeit` работает чуть быстрее.

Итоги

Во всех языках программирования существуют собственные приемы и «передовые» практики. В этой главе я рассказал о конкретных способах написания питонического кода, которые создали программисты для оптимального использования синтаксиса Python.

В основе питонического кода лежат 20 тезисов документа «Дзен Python», которые отражают принципы написания кода на Python. Эти положения следует рассматривать как субъективное мнение; они не являются абсолютно необходимыми для написания программ на Python, но помнить о них безусловно стоит.

Значимые отступы Python (не путайте со значимыми пробелами!) вызывают больше всего протестов со стороны начинающих программистов. И хотя почти во всех языках программирования отступы используются для удобочитаемости кода, Python требует, чтобы отступы заменяли более привычные фигурные скобки, применяемые в других языках.

Модуль Python `timeit` позволяет быстро профилировать время выполнения кода; это всегда лучше, чем просто предположить, что какой-то код работает быстрее. И хотя многие программисты на Python используют конструкцию `range(len())` для циклов `for`, функция `enumerate()` предоставляет более чистый синтаксис для получения индекса и значения при переборе последовательности. Аналогичным образом команда `with` предоставляет более чистый и надежный механизм работы с файлами по сравнению с ручными вызовами `open()` и `close()`. Команда `with` гарантирует, что метод `close()` будет вызван в любой ситуации, когда управление выйдет за пределы блока команды `with`.

В Python предусмотрено несколько способов интерполяции строк. Изначально спецификатор преобразования `%s` помечал позиции, в которых строки должны вставляться в исходную строку. Современный подход для версии Python 3.6 основан на использовании f-строк. У f-строк строковый литерал помечается префиксом `f`, а фигурные скобки отмечают, где в строке могут размещаться строки (или целые выражения).

Синтаксис `[:]` для создания поверхностных копий списков выглядит немного странно и не может однозначно считаться питоническим, но он стал популярен в качестве способа быстрого создания поверхностных копий списков.

У словарей определены методы `get()` и `setdefault()` для выполнения операций с несуществующими ключами. Также словарь `collections.defaultdict` использует значение по умолчанию для несуществующих ключей. Так как в Python нет команды `switch`, использование словаря позволяет компактно записать эквивалент `switch` без нескольких команд `if-elif-else`; при выборе между двумя значениями также можно использовать тернарный оператор.

Цепочка операторов `==` позволяет проверить, что несколько переменных равны друг другу, а оператор `in` — проверить, что переменная имеет одно из нескольких возможных значений.

В этой главе были рассмотрены некоторые идиомы языка Python, а также даны рекомендации относительно того, как писать более питонический код. В следующей главе рассматриваются некоторые проблемы и ловушки, в которые часто попадают начинающие.

7

Жаргон программистов



В своем комиксе XKCD «Up Goer Five» (<https://xkcd.com/1133/>) создатель веб-комиксов Рэндалл Мунро (Randall Munroe) показал техническую схему ракеты Saturn V, используя только тысячу самых распространенных английских слов.

Технические термины он описал предложениями, понятными даже ребенку. Но также он наглядно продемонстрировал, почему нельзя все объяснять простыми выражениями. «Штуковина, которая может людям очень быстро сбежать, если возникла проблема, все горит и они решают не лететь в космос» для непосвященной аудитории более понятна, чем «запуск системы эвакуации». Однако для инженеров NASA в их повседневной работе все это слишком многословно. Они скорее воспользуются сокращением LES (Launch Escape System).

И хотя компьютерный жаргон может показаться неопытным программистам устрашающим и запутанным, сокращения необходимы. Некоторые термины в Python и в области разработки программного обеспечения имеют тонкие смысловые различия, и даже опытные разработчики иногда ошибочно используют их как синонимы. Технические определения таких терминов могут различаться в разных языках программирования, но здесь я рассказываю о терминах в Python. Вы получите широкое, хотя и не глубокое понимание стоящих за ними концепций языка программирования.

Я полагаю, что вы еще не знакомы с классами и объектно-ориентированным программированием (ООП). Поэтому здесь я кратко расскажу о классах и другом ООП-жаргоне, а о самом жаргоне более подробно мы поговорим в главах 15–17.

Определения

Когда в одной комнате оказываются хотя бы два программиста, вероятность споров о семантике достигает 100%. Язык динамичен, а люди распоряжаются

словами, но не наоборот. Некоторые разработчики используют одни и те же термины несколько в разных смыслах, но хорошо знать терминологию все равно полезно. В этой главе я расскажу о терминах и их взаимосвязях. Если вам понадобится список терминов в алфавитном порядке, обращайтесь к официальному глоссарию Python по адресу <https://docs.python.org/3/glossary.html>; вы найдете здесь канонические определения¹.

Несомненно, некоторые программисты, прочитав определения в этой главе, припомнят особые случаи или исключения, но придираться можно бесконечно. Эта глава не была задумана как исчерпывающее описание; я дал доступные определения, пусть даже не идеальные. Но всегда можно узнать что-то новое, как это обычно бывает в программировании.

Язык Python и интерпретатор Python

Язык программирования Python обязан своим названием не змее, а британской комедийной группе Monty Python (хотя в документации и учебниках Python используются отсылки как к Monty Python, так и к змеям). В области программирования название Python тоже имеет два значения.

Когда мы говорим «Python запускает программу» или «Python выдает исключение», речь идет об *интерпретаторе Python* (Python interpreter) — программе, которая читает текст из файла `.py` и выполняет содержащиеся в нем инструкции. Если вы встречаете выражение «интерпретатор Python», то почти всегда подразумевается CPython — интерпретатор Python, сопровождением которого занимается фонд Python Software Foundation, доступный на сайте <https://www.python.org>. CPython является *реализацией* (implementation) языка Python (то есть программой, написанной для выполнения спецификации), однако существуют и другие. Хотя CPython написан на языке программирования C, реализация Jython написана на Java для выполнения сценариев Python, совместимых с программами Java. PyPy — *JIT-компилятор* (just-in-time compiler) для Python, компилирующий программы непосредственно в процессе их выполнения, — написан на Python.

Все эти реализации выполняют исходный код, написанный на языке программирования Python; именно это мы имеем в виду, говоря: «Это программа Python» или «Я изучаю Python». В идеале любой интерпретатор Python способен выполнить любой исходный код, написанный на языке Python; тем не менее в реальном мире всегда будут существовать небольшие несовместимости и различия

¹ В этой главе все термины глоссария даются на русском и в скобках на английском языке. — *Примеч. ред.*

между интерпретаторами. CPython называется *эталонной реализацией* (reference implementation) языка Python, потому что, если существуют различия между тем, как код Python интерпретируется CPython и другим интерпретатором, поведение CPython считается каноническим и правильным.

Сборка мусора

Во многих ранних языках программисту приходилось приказывать программе выделять, а затем освобождать память для структур данных по мере необходимости. Ручное выделение памяти становилось источником многих ошибок, таких как утечка памяти (когда программист забывал освободить выделенную память) или двойное освобождение (когда программисты освобождали одну и ту же область памяти дважды, что приводило к повреждению данных).

Для предотвращения этих ошибок в Python существует *сборка мусора* (garbage collection) — механизм автоматического управления памятью, который отслеживает выделение и освобождение памяти, чтобы программисту не приходилось заниматься этим самому. Сборку мусора можно рассматривать как освобождение памяти, потому что она делает память доступной для новых данных. Например, введите следующую команду в интерактивной оболочке:

```
>>> def someFunction():
...     print('someFunction() called.')
...     spam = ['cat', 'dog', 'moose']
...
>>> someFunction()
someFunction() called.
```

При вызове `someFunction()` Python выделяет память для списка `['cat', 'dog', 'moose']`. Программисту не нужно самостоятельно вычислять, сколько байтов памяти следует запросить, потому что Python делает это автоматически. Сборщик мусора Python освобождает локальные переменные при выходе из функции, чтобы занимаемая ими память стала доступна для других данных. Сборка мусора существенно упрощает программирование и повышает его надежность.

Литералы

Литерал (literal) — текст в исходном коде программы, определяющий фиксированное типизованное значение. В следующем примере:

```
>>> age = 42 + len('Zophie')
```

фрагменты текста `42` и `'Zophie'` определяют два литерала, целый и строковый. Литерал можно рассматривать как значение, буквально определяемое в исходном

коде. Только встроенные типы данных могут иметь литеральные значения в исходном коде Python, так что переменная `age` литеральным значением не является. В табл. 7.1 приведены примеры литералов Python.

Таблица 7.1. Примеры литералов в Python

Литерал	Тип данных
42	Integer
3.14	Float
1.4886191506362924e+36	Float
"""Howdy!"""	String
r'Green\Blue'	String
[]	List
{'name': 'Zophie'}	Dictionary
b'\x41'	Bytes
True	Boolean
None	NoneType

Дотошные читатели заметят, что некоторые из моих примеров не являются литералами в соответствии с официальной документацией языка Python. Формально `-5` не является литералом в Python, потому что язык определяет знак «минус» (`-`) как оператор, применяемый к литералу `5`. Кроме того, `True`, `False` и `None` считаются ключевыми словами Python, а не литералами, тогда как `[ ]` и `{ }` называются *индикаторами* (*displays*) или *атомами* (*atoms*) в зависимости от того, какую часть официальной документации вы просматриваете. Тем не менее литерал — распространенный термин, который будет использоваться профессиональными разработчиками во всех этих примерах.

Ключевые слова

В каждом языке программирования существует собственный набор ключевых слов. Ключевые слова Python образуют набор имен, которые резервируются как часть языка и не могут использоваться как имена переменных (то есть идентификаторы). Например, создать переменную с именем `while` не удастся, потому что ключевое слово `while` зарезервировано для использования в циклах `while`. Ниже приводится список ключевых слов Python для версии Python 3.9.

and	continue	finally	is	raise
as	def	for	lambda	return
assert	del	from	None	True
async	elif	global	nonlocal	try
await	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield

Следует помнить, что ключевые слова Python всегда записываются на английском языке, они недоступны на других языках. Например, следующая функция содержит идентификаторы, записанные на испанском языке, но ключевые слова `def` и `return` остаются записанными на английском.

```
def agregarDosNúmeros(primerNúmero, segundoNúmero):  
    return primerNúmero + segundoNúmero
```

К сожалению, английский язык доминирует в области программирования — и это определенная сложность для 6,5 миллиарда человек, которые на английском не говорят.

Объекты, значения, экземпляры и идентичность

Объект (object) представляет некоторый фрагмент данных: число, текст или более сложную структуру данных (такую как список или словарь). Все объекты могут сохраняться в переменных, передаваться в аргументах при вызове функций и возвращаться из вызовов функций.

Каждый объект характеризуется *значением*, *идентичностью* и *типом данных* (value, identity и data type). *Значение* — данные, представляемые объектом (например, целое число 42 или строка 'hello'). И хотя это создает неоднозначность, некоторые программисты используют термин «значение» как синоним объекта, особенно для простых типов данных (например, целых чисел или строк). Так, переменная, которая содержит данные 42, является переменной, которая содержит целое значение, но также можно сказать, что это переменная, содержащая целочисленный объект со значением 42.

Объект создается с *идентичностью* — уникальным целым числом, которое можно просмотреть вызовом функции `id()`. Например, введите следующий код в интерактивной оболочке:

```
>>> spam = ['cat', 'dog', 'moose']  
>>> id(spam)  
33805656
```

МЕТАФОРЫ ПЕРЕМЕННЫХ: КОРОБКИ И НАКЛЕЙКИ

Во многих учебниках начального уровня переменные сравниваются с коробками, что представляет собой чрезмерное упрощение. Переменную удобно представить как коробку, в которой находится значение (рис. 7.1), но когда речь заходит о ссылках, метафора начинает рассыпаться. В только что рассмотренном примере в переменных `spam` и `eggs` не хранились разные словари; в них хранились ссылки на один словарь, находящийся в памяти компьютера.

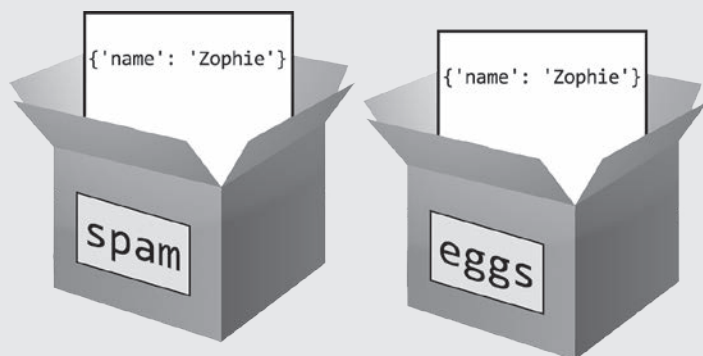


Рис. 7.1. Во многих книгах переменные сравниваются с коробками, в которых хранятся значения

В Python все переменные с технической точки зрения являются ссылками, а не контейнерами значений, независимо от их типа данных. Метафора коробки проста, но не идеальна. Вместо того чтобы рассматривать переменные как коробки, вы также можете рассматривать переменные как наклейки для объектов в памяти. На рис. 7.2 изображены наклейки для переменных `spam` и `eggs` из предыдущего примера.



Рис. 7.2. Переменные также можно представить как наклейки для значений

Так как несколько переменных могут ссылаться на один объект, этот объект будет «храниться» в нескольких переменных. Один объект нельзя положить сразу в несколько коробок, поэтому будет проще использовать метафору наклейки. Дополнительная информация по этой теме содержится в докладе Неда Бэтчелдера (Ned Batchelder) на конференции PyCon 2015, «Facts and Myths about Python Names and Values» (https://youtu.be/_AEJHKGk9ns).

Переменная `spam` содержит объект с типом данных списка и значением `['cat', 'dog', 'moose']`. Ее идентичность равна `33805656`, хотя числовое идентифицирующее значение вычисляется заново при запуске программы, так что с большой вероятностью вы будете получать разные числа на вашем компьютере.

После того как объект будет создан, его идентичность не изменяется на протяжении выполнения программы. Хотя тип данных и идентичность объекта не изменяются во время выполнения, значение объекта может измениться, как мы видим в следующем примере:

```
>>> spam.append('snake')
>>> spam
['cat', 'dog', 'moose', 'snake']
>>> id(spam)
33805656
```

Теперь список также содержит элемент `'snake'`. Но как видно по вызову `id(spam)`, его идентичность не изменилась — список остался тем же. Но давайте посмотрим, что произойдет, если ввести следующий код:

```
>>> spam = [1, 2, 3]
>>> id(spam)
33838544
```

Значение в `spam` было перезаписано новым объектом списка с новой идентичностью: `33838544` вместо `33805656`. *Идентификатор* (identifier) (такой как `spam`) не следует путать с идентичностью (identity), потому что несколько идентификаторов могут ссылаться на объект. Так, в следующем примере двум переменным присваивается один и тот же словарь:

```
>>> spam = {'name': 'Zophie'}
>>> id(spam)
33861824
>>> eggs = spam
>>> id(eggs)
33861824
```

Идентичности обоих идентификаторов, `spam` и `eggs`, равны 33861824, потому что они относятся к одному объекту словаря. Теперь измените значение `spam` в интерактивной оболочке:

```
>>> spam = {'name': 'Zophie'}
>>> eggs = spam
>>> spam['name'] = 'Al' ❶
>>> spam
{'name': 'Al'}
>>> eggs
{'name': 'Al'} ❷
```

Как видите, изменения `spam` ❶, как по волшебству, также появляются в `eggs` ❷. Это объясняется тем, что оба идентификатора относятся к одному и тому же объекту.

Если не понимать, что оператор присваивания `=` всегда копирует ссылку, а не объект, можно внести в программу ошибку: вы думаете, что копируете объект, тогда как в действительности копируется ссылка на исходный объект. К счастью, это не создает проблем с неизменяемыми значениями (целыми числами, строками и кортежами) по причинам, которые объясняются в подразделе «Изменяемость и неизменяемость» на с. 143.

Оператор `is` может использоваться для проверки тождественности, то есть того, что два объекта имеют одинаковую идентичность. С другой стороны, оператор `==` проверяет только равенство значений двух объектов. Можно считать, что `x is y` является сокращенной записью для `id(x) == id(y)`. Введите следующий фрагмент в интерактивной оболочке, чтобы понять суть различий:

```
>>> spam = {'name': 'Zophie'}
>>> eggs = spam ❶
>>> spam is eggs
True
>>> spam == eggs
True
>>> bacon = {'name': 'Zophie'} ❷
>>> spam == bacon
True
>>> spam is bacon
False
```

Переменные `spam` и `eggs` ссылаются на один объект словаря ❶, так что их идентичности и значения одинаковы. Но переменная `bacon` ссылается на другой объект словаря ❷, хотя он и содержит данные, совпадающие с данными `spam` и `eggs`. Совпадение данных означает, что `bacon` содержит то же значение, что и `spam` и `eggs`, но это два разных объекта с двумя разными идентичностями.

Элементы

В Python объект, находящийся в объекте-контейнере (таком как список или словарь), также называется *элементом* (item, element). Например, строки в списке ['dog', 'cat', 'moose'] являются объектами, но они также называются элементами.

Изменяемость и неизменяемость

Как я упоминал ранее, все объекты в Python характеризуются значением, типом данных и идентичностью и из всех этих атрибутов изменяться может только значение. Если значение объекта можно изменить, этот объект называется *изменяемым* (mutable). Если значение объекта изменить нельзя, объект называется *неизменяемым* (immutable). В табл. 7.2 перечислены некоторые изменяемые и неизменяемые типы данных в Python.

Таблица 7.2. Примеры изменяемых и неизменяемых типов данных Python

Изменяемые типы данных	Неизменяемые типы данных
List (Список)	Integer (Целое число)
Dictionaries (Словари)	Floating-point number (Число с плавающей точкой)
Sets (Множества)	Boolean (Логический)
Bytearray (Массив байтов)	String (Строка)
Array (Массив)	Frozen set (Зафиксированное множество)
	Bytes (Байты)
	Tuple (Кортеж)

Когда вы перезаписываете переменную, может показаться, что вы изменяете значение объекта, как в следующем примере:

```
>>> spam = 'hello'
>>> spam
'hello'
>>> spam = 'goodbye'
>>> spam
'goodbye'
```

Но в этом коде вы не заменяете значение объекта 'hello' значением 'goodbye'. Это два разных объекта, а вы только переключаете переменную spam так, чтобы

она содержала ссылку не на объект `'hello'`, а на объект `'goodbye'`. Чтобы убедиться в этом, воспользуемся функцией `id()` для вывода идентичностей двух объектов:

```
>>> spam = 'hello'
>>> id(spam)
40718944
>>> spam = 'goodbye'
>>> id(spam)
40719224
```

Эти два объекта обладают разными идентичностями (40718944 и 40719224), потому что это два разных объекта. Но у переменных, ссылающихся на изменяемые объекты, значения могут изменяться на месте (in-place). Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> spam = ['cat', 'dog']
>>> id(spam)
33805576
>>> spam.append('moose')    ❶
>>> spam[0] = 'snake'       ❷
>>> spam
['snake', 'dog', 'moose']
>>> id(spam)
33805576
```

Метод `append()` ❶ и присваивание элементу по индексу ❷ изменяют значение списка на месте. И хотя значение списка изменилось, его идентичность осталась прежней (33805576). Но при выполнении конкатенации списка с использованием оператора `+` вы создаете новый объект (с новой идентичностью), который заменяет старый список:

```
>>> spam = spam + ['rat']
>>> spam
['snake', 'dog', 'moose', 'rat']
>>> id(spam)
33840064
```

Конкатенация списков создает новый список с новой идентичностью. Когда это происходит, старый список будет со временем удален из памяти сборщиком мусора. За информацией о том, какие методы и операции изменяют объекты на месте, а какие их перезаписывают, обращайтесь к документации Python. Можно запомнить хорошее практическое правило: если вы видите литерал в исходном коде (например, `['rat']` в приведенном примере), то, скорее всего, Python создаст новый объект. Метод, который вызывается для объекта (например, `append()`), часто изменяет объект на месте.

Присваивание проще выполняется для объектов с неизменяемыми типами данных (целые числа, строки, кортежи и т. д.). Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> bacon = 'Goodbye'
>>> id(bacon)
33827584
>>> bacon = 'Hello'           ❶
>>> id(bacon)
33863820
>>> bacon = bacon + ', world!'  ❷
>>> bacon
'Hello, world!'
>>> id(bacon)
33870056
3 >>> bacon[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Строки являются неизменяемыми, так что вы не сможете изменить их значение. Хотя все выглядит так, будто значение строки в переменной `bacon` меняется с `'Goodbye'` на `'Hello'` ❶, в действительности она замещается новым объектом строки с новой идентичностью. Аналогичным образом выражение, использующее конкатенацию строк, создает новый строковый объект ❷ с новой идентичностью. Попытки модифицировать строку «на месте» посредством присваивания элементу запрещены в Python 3.

Значение кортежа определяется как совокупность содержащихся в нем объектов и порядка этих объектов. *Кортежи* представляют собой неизменяемые последовательности объектов, в которых значения заключаются в круглые скобки. Это означает, что перезапись элементов кортежа невозможна:

```
>>> eggs = ('cat', 'dog', [2, 4, 6])
>>> id(eggs)
39560896
>>> id(eggs[2])
40654152
>>> eggs[2] = eggs[2] + [8, 10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Однако изменяемый список в неизменяемом кортеже все равно можно изменить на месте:

```
>>> eggs[2].append(8)
>>> eggs[2].append(10)
>>> eggs
('cat', 'dog', [2, 4, 6, 8, 10])
```

```
>>> id(eggs)
39560896
>>> id(eggs[2])
40654152
```

И хотя это экзотический случай, о нем важно помнить. Кортеж по-прежнему содержит ссылки на те же объекты (рис. 7.3). Но если кортеж содержит изменяемый объект и этот объект изменяет свое значение, то значение кортежа также изменится.

Я, как и почти все программисты, пишущие на Python, называю кортежи неизменяемыми. Но вопрос о том, можно ли назвать некоторые кортежи изменяемыми, зависит от определения. Эта тема более подробно рассматривается в моем докладе на конференции PyCascades 2019 «The Amazing Mutable, Immutable Tuple» (<https://invyu.com/amazingtuple/>). Или же загляните в главу 2 книги «Fluent Python» (O'Reilly Media, 2015) Лучано Рамальо (Luciano Ramalho).

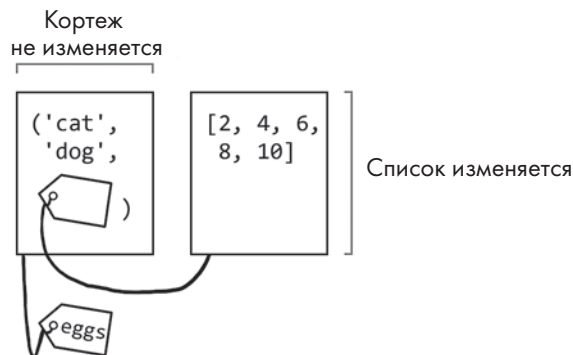


Рис. 7.3. Хотя набор объектов в кортеже неизменяем, сами объекты могут быть изменяемыми

Индексы, ключи и хеш-коды

Списки и словари Python являются значениями, которые могут содержать наборы других значений. Для обращения к этим значениям используется *оператор индексирования*, который состоит из пары квадратных скобок (`[]`) и целого числа, называемого *индексом*; оно указывает, к какому значению вы хотите обратиться. Чтобы понять, как работает индексирование списков, введите следующий фрагмент в интерактивной оболочке:

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam[0]
'cat'
>>> spam[-2]
'dog'
```

В этом примере число 0 — индекс. Первый индекс равен 0, а не 1, потому что в Python (как и в большинстве языков) нумерация индексов начинается с 0. Языки, в которых нумерация индексов начинается с 1, встречаются нечасто: самые заметные примеры — Lua и R. Python также поддерживает отрицательные индексы; -1 обозначает последний элемент списка, -2 — предпоследний элемент и т. д. Отрицательный индекс `spam[-n]` можно рассматривать как эквивалент `spam[len(spam) - n]`.

ПРИМЕЧАНИЕ

Компьютерный теоретик, певец и автор песен Стэн Келли-Бутл (Stan Kelly-Bootle) однажды пошутил: «Должны ли индексы массивов начинаться с 0 или 1? Мое компромиссное предложение 0,5 было отвергнуто без должного рассмотрения».

Оператор индексирования также может использоваться со списковым литералом, хотя в реальном коде все эти квадратные скобки кажутся непонятными и избыточными:

```
>>> ['cat', 'dog', 'moose'][2]
'moose'
```

Индексирование также применяют не только со списками, но и с другими значениями — например, со строками для получения отдельных символов:

```
>>> 'Hello, world'[0]
'H'
```

В словарях Python информация структурирована в виде *пар «ключ — значение»*:

```
>>> spam = {'name': 'Zophie'}
>>> spam['name']
'Zophie'
```

И хотя индексы списков ограничиваются целыми числами, оператор индексирования у словарей Python использует *ключ*, которым может быть любой хешируемый объект. *Хеш-код* представляет собой целое число, своего рода отличительный признак некоторого значения. Хеш-код объекта никогда не изменяется на протяжении его жизненного цикла, и объекты с одинаковыми значениями должны иметь одинаковые хеш-коды. Строка `'name'` в этом экземпляре является ключом для значения `'Zophie'`. Функция `hash()` возвращает хеш-код объекта, если объект является *хешируемым*. Неизменяемые объекты (строки, целые числа, числа с плавающей точкой, кортежи) могут быть хешируемыми. Списки (а также другие изменяемые объекты) хешируемыми не являются. Введите следующий фрагмент в интерактивной оболочке:

```
>>> hash('hello')
-1734230105925061914
>>> hash(42)
```

```

42
>>> hash(3.14)
322818021289917443
>>> hash((1, 2, 3))
2528502973977326415
>>> hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

```

Хотя подробности выходят за рамки книги, хеш-код ключа используется для поиска элементов в структурах данных (словарях и множествах). Это объясняет, почему изменяемый список не может использоваться в ключах словарей:

```

>>> d = {}
>>> d[[1, 2, 3]] = 'some value'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

```

Хеш-код отличается от идентичности. Два разных объекта с одинаковыми значениями имеют разные идентичности, но одинаковые хеш-коды. Например, введите следующий фрагмент в интерактивной оболочке:

```

>>> a = ('cat', 'dog', 'moose')
>>> b = ('cat', 'dog', 'moose')
>>> id(a), id(b)
(37111992, 37112136)
>>> id(a) == id(b)    ❶
False
>>> hash(a), hash(b)
(-3478972040190420094, -3478972040190420094)
>>> hash(a) == hash(b)    ❷
True

```

Кортежи, на которые ссылаются *a* и *b*, имеют разные идентичности ❶, но так как они имеют одинаковые значения, из этого следует, что их хеш-коды идентичны ❷. Обратите внимание: кортеж является хешируемым, если он содержит только хешируемые элементы. Так как ключами словарей могут быть только хешируемые элементы, кортеж, содержащий нехешируемый список, не может использоваться в качестве ключа. Введите следующий фрагмент в интерактивной оболочке:

```

>>> tuple1 = ('cat', 'dog')
>>> tuple2 = ('cat', ['apple', 'orange'])
>>> spam = {}
>>> spam[tuple1] = 'a value'    ❶
>>> spam[tuple2] = 'another value'    ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

```

Обратите внимание: кортеж `tuple1` является хешируемым ❶, но `tuple2` содержит нехешируемый список ❷, а следовательно, хешируемым не является.

Контейнеры, последовательности, отображения и разновидности множеств

Смысл терминов «контейнер», «последовательность» и «отображение» в Python не всегда применим к другим языкам программирования. В Python *контейнером* называется объект любого типа данных, который может содержать несколько других объектов. Из всех видов контейнеров в Python чаще всего используются списки и словари.

Последовательность (sequence) представляет собой объект любого контейнерного типа данных с упорядоченными значениями, к которым можно обращаться по целым индексам. Строки, кортежи, списки и байтовые объекты относятся к типам данных последовательностей. Объекты этих типов могут обращаться к значениям по целым индексам в операторе индексирования (квадратные скобки [и]), а также могут передаваться функции `len()`. Под упорядоченностью мы понимаем, что в последовательности есть первое значение, второе значение и т. д. Например, следующие два списковых значения равными не считаются, потому что их элементы следуют в разном порядке:

```
>>> [1, 2, 3] == [3, 2, 1]
False
```

Отображение (mapping) представляет собой объект любого контейнерного типа данных, использующий ключи вместо индексов. Отображение может быть упорядоченным или неупорядоченным. Словари в Python 3.4 и более ранних версиях не упорядочены, то есть в них не было первой или последней пары «ключ — значение»:

```
>>> spam = {'a': 1, 'b': 2, 'c': 3, 'd': 4} # Запускается из CPython 3.5.
>>> list(spam.keys())
['a', 'c', 'd', 'b']
>>> spam['e'] = 5
>>> list(spam.keys())
['e', 'a', 'c', 'd', 'b']
```

В ранних версиях Python постоянный порядок следования элементов в словарях не гарантировался. В результате неупорядоченной природы словарей два словарных литерала, записанных с разным порядком пар «ключ — значение», все равно считались равными:

```
>>> {'a': 1, 'b': 2, 'c': 3} == {'c': 3, 'a': 1, 'b': 2}
True
```

Но начиная с CPython 3, словари поддерживают порядок вставки своих пар «ключ — значение»:

```
>>> spam = {'a': 1, 'b': 2, 'c': 3, 'd': 4} # Запускается из CPython 3.6.
>>> list(spam)
['a', 'b', 'c', 'd']
>>> spam['e'] = 5
>>> list(spam)
['a', 'b', 'c', 'd', 'e']
```

Это особенность интерпретатора CPython 3.6, отсутствующая в других интерпретаторах Python 3.6. Все интерпретаторы Python 3.7 поддерживают упорядоченные словари, которые стали стандартными для языка Python в версии 3.7. Однако упорядоченность словаря не означает, что к его элементам можно обращаться по целочисленным индексам: `spam[0]` не будет обозначать первый элемент в упорядоченном словаре (если только по совпадению первый элемент не будет иметь ключ 0). Упорядоченные словари также считаются равными, если они содержат одинаковые пары «ключ — значение», даже если пары «ключ — значение» в этих словарях следуют в разном порядке.

Модуль `collections` содержит много других видов отображений, включая `OrderedDict`, `ChainMap`, `Counter` и `UserDict`. Они описаны в электронной документации по адресу <https://docs.python.org/3/library/collections.html>.

Dunder-методы, или магические методы

Dunder-методы, называемые также *магическими*, — это специальные методы в Python, которые используются для перезагрузки операторов. Их имена начинаются и заканчиваются двумя символами подчеркивания (`__`). (Dunder — сокращение от Double UNDERscore.) Самый известный специальный метод `__init__()` инициализирует объекты. В Python определено несколько десятков dunder-методов, они подробно рассматриваются в главе 17.

Модули и пакеты

Модуль (module) представляет собой программу Python, которая может импортироваться другими программами Python, чтобы те могли воспользоваться кодом модуля. Модули, входящие в поставку Python, образуют *стандартную библиотеку* Python, но вы также можете создавать собственные модули. Если сохранить программу Python, например под именем `spam.py`, другие программы смогут выполнить команду `import spam`, чтобы получить доступ к функциям, классам и переменным верхнего уровня программы `spam.py`.

Пакет (package) представляет собой набор модулей. Чтобы создать его, следует разместить в папке файл с именем `__init__.py`. Имя папки становится именем пакета. Пакеты могут содержать несколько модулей (то есть файлов `.py`) или других пакетов (других папок, содержащих файлы `__init__.py`).

За дополнительными объяснениями и подробностями о модулях и пакетах обращайтесь к официальной документации Python по адресу <https://docs.python.org/3/tutorial/modules.html>.

Вызываемые объекты и первоклассные объекты

В Python можно вызывать не только функции и методы. Любой объект, реализующий оператор вызова — круглые скобки `()`, называется *вызываемым объектом* (callable). Например, если у вас имеется команда `def hello():`, код можно рассматривать как переменную с именем `hello`, содержащую объект функции. При применении оператора вызова к этой переменной будет вызвана функция, хранящаяся в переменной: `hello()`.

Классы относятся к концепциям ООП. Класс является примером вызываемого объекта, который не является функцией или методом. Например, класс `date` в модуле `datetime` вызывается с использованием оператора вызова, как в примере `datetime.date(2020, 1, 1)`. При вызове объекта класса выполняется метод `__init__()` этого класса. В главе 15 мы поговорим о классах более подробно.

В Python функции являются *первоклассными объектами* (first-class objects). Это означает, что их можно сохранять в переменных, передавать в аргументах при вызове функций, возвращать при вызове функций и делать все остальное, что можно сделать с объектом. Рассматривайте команду `def` как присваивание объекта функции переменной. Например, можно создать функцию `spam()`, которую затем можно вызвать:

```
>>> def spam():
...     print('Spam! Spam! Spam!')
...
>>> spam()
Spam! Spam! Spam!
```

Также можно присвоить объект функции `spam()` другим переменным. Когда вы вызываете переменную, которой был присвоен объект функции, Python выполняет эту функцию:

```
>>> eggs = spam
>>> eggs()
Spam! Spam! Spam!
```

Таким образом создаются *псевдонимы* (aliases) — другие имена для существующих функций. Они часто используются в тех ситуациях, когда возникает необходимость в переименовании функций. Но старое имя используется в большом объеме существующего кода, и изменение всего этого кода потребовало бы слишком серьезной работы.

Первоклассные функции чаще всего используются для передачи функций другим функциям. Например, можно определить функцию `callTwice()`, которая дважды вызывает переданную ей функцию:

```
>>> def callTwice(func):  
...     func()  
...     func()  
...  
>>> callTwice(spam)  
Spam! Spam! Spam!  
Spam! Spam! Spam!
```

С таким же успехом можно было просто дважды вызвать `spam()` в исходном коде. Но функции `callTwice()` можно передать любую функцию во время выполнения, вместо того чтобы заранее включать повторный вызов функции в исходный код.

Частые ошибки при использовании терминов

Технический жаргон достаточно неоднозначен — особенно для терминов взаимосвязанных, но имеющих разные определения. Ситуация усугубляется тем, что языки, операционные системы и разные области компьютерной теории могут разными терминами обозначать одни и те же понятия или одинаковыми терминами — разные понятия. Чтобы ясно и однозначно общаться с другими программистами, необходимо понимать различия между терминами, о которых речь пойдет далее.

Команды и выражения

Выражениями (expressions) называются инструкции, состоящие из операторов и значений, результатом вычисления которых является одно значение. Таким значением может быть переменная (которая содержит значение) или вызов функции (которая возвращает значение). Таким образом, `2 + 2` является выражением, при вычислении которого будет получено одно значение 4. С другой стороны, `len(myName) > 4` и `myName.isupper()` or `myName == 'Zophie'` также являются выражениями. Значение само по себе также является выражением, результатом вычисления которого является оно само.

Практически все остальные инструкции в Python являются *командами* (statements). К их числу относятся команды `if`, команды `for`, команды `def`, команды `return` и т. д. Команды не вычисляются в конкретное значение. Некоторые команды могут включать выражения — как, например, команда присваивания `spam = 2 + 2` или команда `if myName == 'Zophie':`.

В Python 3 используется функция `print()`, а в Python 2 вместо нее существовала команда `print`. Может показаться, что различия сводятся к добавлению круглых скобок, но важно заметить, что функция `print()` в Python 3 имеет возвращаемое

значение (которое всегда равно `None`), может передаваться в аргументе другим функциям и может присваиваться переменным. С командами все эти действия невозможны. Впрочем, круглые скобки допустимо использовать и в Python 2, как показывает следующий пример в интерактивной оболочке:

```
>>> print 'Hello, world!' # Выполняется в Python 2
Hello, world!
>>> print('Hello, world!') # Выполняется в Python 2 ❶
Hello, world!
```

И хотя все выглядит как вызов функции ❶, на самом деле это команда `print` со строковым значением, заключенным в круглые скобки, — по аналогии с тем, как присваивание `spam = (2 + 2)` эквивалентно `spam = 2 + 2`. В Python 2 и 3 можно передать несколько значений команде `print` или функции `print()` соответственно. В Python 3 это выглядит так:

```
>>> print('Hello', 'world') # Выполняется в Python 3
Hello world
```

Но использование того же кода в Python 2 будет интерпретировано как передача кортежа с двумя строковыми значениями в команде `print`, в результате чего вы получите следующий вывод:

```
>>> print('Hello', 'world') # Выполняется в Python 2
('Hello', 'world')
```

Между командой и выражением, состоящим из вызова функции, существуют неочевидные, но реальные различия.

Блок, секция и тело

Термины «блок», «секция» и «тело» часто используются как синонимы для обозначения группы инструкций Python. *Блок* (block) начинается с отступа и завершается там, где отступ возвращается к предыдущему уровню. Например, код, следующий за командой `if` или `for`, называется блоком команды. Новый блок должен следовать за командами, завершающимися двоеточием (такими как `if`, `else`, `for`, `while`, `def`, `class` и т. д.).

Впрочем, Python допускает однострочные блоки. Это допустимый, хотя и не рекомендуемый синтаксис Python:

```
if name == 'Zophie': print('Hello, kitty!')
```

В блок команды `if` можно включить несколько команд, разделяя их точкой с запятой `;`:

```
if name == 'Zophie': print('Hello, kitty!'); print('Do you want a treat?')
```

Однострочный синтаксис не может использоваться с другими командами, которым требуется новый блок. Следующий фрагмент не является допустимым в Python:

```
if name == 'Zophie': if age < 2: print('Hello, kitten!')
```

Он недопустим, потому что, если в следующей строке будет располагаться команда `else`, мы не сможем однозначно сказать, к какой команде `if` она относится.

В официальной документации Python предпочтение отдается термину *секция* (clause) вместо «блок» (https://docs.python.org/3/reference/compound_stmts.html). Следующий код является секцией:

```
if name == 'Zophie':  
    print('Hello, kitty!')  
    print('Do you want a treat?')
```

Команда `if` является заголовком секции, а два вызова `print()`, вложенных в `if`, являются *телом* (body) секции. В официальной документации Python термином «блок» обозначается фрагмент кода Python, выполняемый как единое целое — например, модуль, функция или определение класса (<https://docs.python.org/3/reference/executionmodel.html>).

Переменные и атрибуты

Переменные (variables) — имена, ссылающиеся на объекты. *Атрибутом* (attribute), согласно официальной документации, называется «любое имя, следующее за точкой» (<https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces>). Атрибуты связываются с объектами (имя перед точкой). Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> import datetime  
>>> spam = datetime.datetime.now()  
>>> spam.year  
2018  
>>> spam.month  
1
```

В этом примере `spam` — переменная, содержащая объект `datetime` (возвращаемый вызовом `datetime.datetime.now()`), а `year` и `month` — атрибуты этого объекта. Даже в случае, скажем, `sys.exit()` функция `exit()` считается атрибутом объекта модуля `sys`.

В других языках атрибуты называются *полями* (fields), *свойствами* (properties) или *компонентными переменными* (member variables).

Функции и методы

Функция (function) — совокупность кода, выполняемого при вызове. *Методом* (method) называется функция (или вызываемый объект, см. следующий раздел), связанная с классом, — по аналогии с тем, как атрибут является переменной, связанной с объектом. К функциям относятся встроенные функции или функции, связанные с модулем. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> len('Hello')
5
>>> 'Hello'.upper()
'HELLO'
>>> import math
>>> math.sqrt(25)
5.0
```

В этом примере `len()` — функция, а `upper()` — метод строк. Методы также считаются атрибутами объектов, с которыми они связываются. Обратите внимание: точка не обязательно означает, что перед вами именно метод, а не функция. Функция `sqrt()` связывается с именем `math`, но это модуль, а не класс.

Итерируемые объекты и итераторы

Циклы `for` языка Python весьма гибки. Команда `for i in range(3):` выполняет блок кода трижды. Не стоит полагать, что вызов `range(3)` — это такой способ приказать циклу `for` «выполнить код три раза», принятый в Python. Вызов `range(3)` возвращает объект диапазона, по аналогии с тем, как вызов `list('cat')` возвращает объект списка. Оба объекта являются примерами *итерируемых объектов* (iterables).

Итерируемые объекты используются в циклах `for`. Введите следующий фрагмент в интерактивной оболочке, чтобы увидеть, как цикл `for` перебирает объект диапазона и объект списка:

```
>>> for i in range(3):
...     print(i) # тело цикла for
...
0
1
2
>>> for i in ['c', 'a', 't']:
...     print(i) # тело цикла for
...
c
a
t
```

К категории итерируемых объектов также относятся все разновидности последовательностей (например, диапазоны, списки, кортежи и строки), а также ряд объектов-контейнеров (словари, множества и объекты файлов).

Тем не менее в этих примерах циклов `for` происходит нечто большее. Во внутренней реализации Python вызывает встроенные функции `iter()` и `next()` для цикла `for`. При использовании цикла `for` итерируемые объекты передаются встроенной функции `iter()`, которая возвращает объекты-итераторы (iterators). И если итерируемый объект содержит элементы, итератор следит за тем, какой элемент будет использован в цикле следующим. При каждой итерации цикла объект-итератор передается встроенной функции `next()` для получения следующего элемента в итерируемом объекте. Вы можете вызвать функции `iter()` и `next()` вручную, чтобы увидеть, как работают циклы `for`. Введите следующий фрагмент в интерактивной оболочке, чтобы выполнить те же инструкции, что и в предыдущем примере цикла:

```
>>> iterableObj = range(3)
>>> iterableObj
range(0, 3)
>>> iteratorObj = iter(iterableObj)
>>> i = next(iteratorObj)
>>> print(i) # тело цикла for
0
>>> i = next(iteratorObj)
>>> print(i) # тело цикла for
1
>>> i = next(iteratorObj)
>>> print(i) # тело цикла for
2
>>> i = next(iteratorObj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration ❶
```

Обратите внимание: если вызвать `next()` после того, как был возвращен последний элемент в итерируемом объекте, Python выдает исключение `StopIteration` ❶. Вместо того чтобы аварийно завершать программы с сообщением об ошибке, циклы `for` в Python перехватывают это исключение, чтобы знать, когда следует остановить перебор.

Итератор способен перебрать все элементы итерируемого объекта только один раз, так же, как можно только один раз вызвать `open()` и `readlines()` для чтения содержимого файла, после чего вам придется открывать файл заново для чтения его содержимого. Если потребуются снова перебрать итерируемый объект, необходимо снова вызвать `iter()` для создания другого итератора. Вы можете создать столько объектов-итераторов, сколько потребуется; каждый объект будет отслеживать

следующий возвращаемый элемент независимо от других. Чтобы понять, как это работает, введите следующий фрагмент в интерактивной оболочке:

```
>>> iterableObj = list('cat')
>>> iterableObj
['c', 'a', 't']
>>> iteratorObj1 = iter(iterableObj)
>>> iteratorObj2 = iter(iterableObj)
>>> next(iteratorObj1)
'c'
>>> next(iteratorObj1)
'a'
>>> next(iteratorObj2)
'c'
```

Помните, что итерируемые объекты передаются в аргументах функции `iter()`, тогда как объект, возвращаемый вызовами `iter()`, является итератором. Объекты-итераторы передаются функции `next()`. Когда вы создаете собственные типы данных командами `class`, вы можете реализовать специальные методы `__iter__()` и `__next__()`, чтобы ваши объекты можно было использовать в циклах `for`.

Синтаксические ошибки, ошибки времени выполнения и семантические ошибки

Известно много способов классификации ошибок. Но на верхнем уровне ошибки программирования можно разделить на три вида: синтаксические, ошибки времени выполнения и семантические.

Синтаксис (syntax) — это набор правил для создания инструкций, действительных в заданном языке программирования. Синтаксическая ошибка (например, пропущенная круглая скобка, точка вместо запятой или другая опечатка) немедленно приводит к ошибке `SyntaxError`. Синтаксические ошибки также называются *ошибками разбора* (parsing errors); они происходят, когда интерпретатор Python не может разобрать текст исходного кода в действительные инструкции. В естественном языке такие ошибки можно сравнить с неправильными грамматическими конструкциями или бессмысленными цепочками слов. Компьютерам нужны предельно точные инструкции; они не могут прочитать мысли программиста, чтобы понять, что должна делать программа. Из-за этого программа с синтаксической ошибкой даже не запустится.

Ошибка времени выполнения (runtime error) возникает, когда работающей программе не удастся выполнить некоторую операцию — скажем, открыть несуществующий файл или разделить число на нуль. В естественном языке ошибку времени выполнения можно сравнить с невыполнимой инструкцией типа «нарисуй квадрат с тремя сторонами». Если ошибка времени выполнения не будет исправлена,

программа аварийно завершается с выдачей трассировки. Но ошибки времени выполнения можно перехватить командами `try-except`, которые выполняют код обработки ошибок. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> slices = 8
>>> eaters = 0
>>> print('Each person eats', slices / eaters, 'slices.')
```

Этот код при запуске выдает трассировку:

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print('Each person eats', slices / eaters, 'slices.')
ZeroDivisionError: division by zero
```

Полезно помнить, что номер строки в трассировке относится только к точке, в которой интерпретатор Python обнаружил ошибку. Истинная причина ошибки может находиться в предыдущей строке кода или даже намного ранее в программе.

Синтаксические ошибки в исходном коде перехватываются интерпретатором до запуска программы, но синтаксические ошибки также происходят во время выполнения. Функция `eval()` может получить строку кода Python и выполнить его, что приводит к ошибке `SyntaxError` во время выполнения. Например, в команде `eval('print("Hello, world")')` пропущена закрывающая двойная кавычка, но программа не обнаружит этот факт, пока не будет вызвана функция `eval()`.

Семантические (semantic) ошибки, также называемые *логическими (logical)*, коварнее предыдущих. Семантические ошибки не порождают сообщений об ошибках или сбоев, но компьютер выполняет инструкции способом, который не предполагался программистом. На естественном языке семантическую ошибку можно сравнить с распоряжением: «Купи в магазине пакет молока, а если есть яйца — купи десяток». Тогда компьютер купит 11 пакетов молока, потому что в магазине есть яйца. Хорошо это или плохо, но компьютеры делают в точности то, что вы им приказываете. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> print('The sum of 4 and 2 is', '4' + '2')
```

Вы получите следующий результат:

```
The sum of 4 and 2 is 42
```

Очевидно, 42 не является правильным ответом. Но стоит заметить, что программа была выполнена без сбоев. Так как оператор Python `+` складывает целые числа и выполняет конкатенацию строковых значений, ошибочное использование строковых значений `'4'` и `'2'` вместо целых чисел привело к такой работе программы.

Параметры и аргументы

Параметры (parameters) — имена переменных, заключенные в круглые скобки в команде `def`. *Аргументами* (arguments) называются значения, передаваемые при вызове функции, которые затем присваиваются параметрам. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> def greeting(name, species): ❶
...     print(name + ' is a ' + description)
...
>>> greeting('Zophie', 'cat') ❷
Zophie is a cat
```

В команде `def` имена `name` и `species` — параметры ❶. В вызове функции `'Zophie'` и `'cat'` — аргументы ❷. Эти два термина часто путают. Помните, что параметры и аргументы — всего лишь другие названия для переменных и значений, когда они используются в этом контексте.

Явные и неявные преобразования типов

Объект одного типа можно преобразовать в объект другого типа. Например, `int('42')` преобразует строку `'42'` в целое число `42`. На самом деле строковый объект `'42'` не преобразуется, так как функция `int()` создает новый объект целого числа на основании исходного объекта. Явные преобразования правильнее называть *приведением типа* (casting), хотя программисты все равно часто называют этот процесс *преобразованием* (converting) *объекта*.

Python часто выполняет *неявные преобразования типов* (coercion) — например, когда вычисление выражения `2 + 3.0` дает результат `5.0`. Значения (такие как `2` и `3.0`) преобразуются к общему типу данных, с которым может работать оператор.

Неявные преобразования часто приводят к непредвиденным результатам. Логические значения `True` и `False` в Python могут быть преобразованы в целые числа `1` и `0` соответственно. Хотя в реальном коде логические значения никогда не записываются в таком виде, это означает, что выражение `True + False + True` эквивалентно `1 + 0 + 1` и дает результат `2`. Зная этот факт, можно подумать, что передача списка логических значений функции `sum()` позволит эффективно подсчитать значения `True` в списке. Но как выясняется, вызов метода `count()` для списка работает быстрее.

Свойства и атрибуты

Во многих языках термины «свойство» (property) и «атрибут» (attribute) используются как синонимы, но в Python эти слова имеют разный смысл. Атрибут, как объяснялось в подразделе «Переменные и атрибуты» на с. 154, представляет

собой имя, связанное с объектом. К числу атрибутов относятся компонентные переменные и методы объекта.

В других языках, таких как Java, классы могут содержать `get`- и `set`-методы. Вместо того чтобы напрямую присваивать атрибуту (возможно, недействительное) значение, программа должна вызвать `set`-метод для этого атрибута.

Код в `set`-методе может гарантировать, что компонентной переменной будут присваиваться только действительные значения. `Get`-метод читает значение атрибута. Если атрибуту, допустим, присвоено имя `accountBalance`, `set`- и `get`-методам обычно присваиваются имена `setAccountBalance()` и `getAccountBalance()` соответственно.

В Python свойства позволяют программистам использовать `get`- и `set`-методы с более чистым синтаксисом. В главе 17 я более подробно расскажу о свойствах Python.

Байт-код и машинный код

Исходный код компилируется в так называемый *машинный код* (machine code) — инструкции, которые могут непосредственно выполняться процессором. Машинный код состоит из инструкций из встроенного *набора команд* (instruction set) процессора. Откомпилированная программа, состоящая из машинного кода, называется *двоичным файлом* (binary). Для таких уважаемых языков, как C, существуют специальные программы — компиляторы; они преобразуют исходный файл с кодом C в двоичные файлы почти для любых существующих процессоров. Но если вы хотите, чтобы Python работал на таком же наборе процессоров, написание компилятора Python для каждого процессора потребует масса времени.

Также существует другой механизм преобразования исходного кода в код, который может выполняться машиной. Вместо машинного кода, который выполняется непосредственно процессором, можно создать *байт-код* (bytecode). Байт-код, также называемый *портируемым* (portable) *кодом*, или *p-кодом*, выполняется не напрямую процессором, а специальной программой — интерпретатором. Байт-код Python состоит из инструкций, хотя эти инструкции не выполняются ни одним реально существующим процессором. Вместо этого интерпретатор выполняет байт-код. Байт-код Python сохраняется в файлах `.pyc`, которые иногда встречаются среди исходных файлов `.py`. Интерпретатор CPython, написанный на C, может компилировать исходный код Python в байт-код Python, а затем выполнять эти инструкции. (Это относится и к виртуальной машине Java [JVM], выполняющей байт-код языка Java.) Так как интерпретатор CPython написан на C, он может быть откомпилирован для любого процессора, для которого уже существует компилятор C.

Если вам захочется больше узнать обо всем этом, есть отличный источник — доклад Скотта Сэндерсона (Scott Sanderson) и Джо Джевника (Joe Jevnik) «Playing with Python Bytecode» на конференции PyCon 2016 (<https://youtu.be/mxjv9KqzwejI>).

Сценарии и программы, языки сценариев и языки программирования

Различия между *сценарием* (script) и программой, и даже языком сценариев и языком программирования, весьма туманны и субъективны. Можно с полным основанием утверждать, что все сценарии являются программами, а все языки сценариев являются языками программирования. Тем не менее языки сценариев иногда считаются более простыми, или «неполноценными», языками программирования.

Один из признаков, отличающих сценарии от программ, — способ выполнения кода. Сценарии, написанные на языках сценариев, интерпретируются непосредственно из исходного кода, тогда как программы, написанные на языках программирования, компилируются в двоичные файлы. Однако Python обычно рассматривается как язык сценариев, хотя при запуске программы Python существует этап компиляции в байт-код. При этом Java обычно языком сценариев не считается, хотя он и генерирует байт-код вместо двоичных файлов с машинным кодом, как и Python. С технической точки зрения языки не компилируются и не интерпретируются; правильнее сказать, что существуют компилируемые и интерпретируемые реализации языка, и компилятор или интерпретатор можно создать для любого языка.

Эти различия вызывают споры, но в конечном итоге они не настолько важны. Языки сценариев вовсе не обязательно обладают меньшей мощностью, а с компилируемыми языками не всегда труднее работать.

Библиотеки, фреймворки, SDK, ядра и API

Использование кода, написанного другими людьми, сильно экономит время. Такой код часто доступен в виде библиотек, фреймворков, SDK, ядер или API. И здесь надо понимать различия между ними.

Библиотека (library) — общий термин для подборки кодов, написанных третьей стороной. Библиотека может содержать функции, классы или другие фрагменты кода, предназначенные для использования разработчиками. Библиотека Python может быть реализована в виде пакета, набора пакетов и даже отдельного модуля. Библиотеки часто предназначены для конкретного языка. Разработчикам не обязательно знать, как работает код библиотеки; им достаточно знать, как вызывать код из библиотеки или взаимодействовать с ним. *Стандартная библиотека* (например, стандартная библиотека Python) представляет собой программную библиотеку, которая должна быть доступна для всех реализаций языка программирования.

Фреймворком (framework) называется подборка кода, работающая по принципу инверсии управления; разработчик пишет функции, которые вызываются фреймворком по мере надобности (вместо вызова функций фреймворка из кода разработчика). Инверсия управления часто описывается фразой «не звоните нам, мы сами вам позвоним». Например, при написании кода для фреймворка веб-приложений

разработчику приходится создавать для веб-страниц функции, которые будут вызываться фреймворком при поступлении веб-запроса.

SDK (Software Development Kit — *комплект разработки ПО*) — это программные библиотеки, документация и программные средства, упрощающие создание приложений для конкретной операционной системы или платформы. Например, Android SDK и iOS SDK используются для создания мобильных приложений для Android и iOS соответственно. JDK (Java Development Kit) — SDK для создания приложений для JVM.

Ядро, или *движок* (engine), — крупная автономная система, которой могут управлять внешние программы разработчика. Разработчики обычно вызывают функции ядра для выполнения больших сложных задач. Примеры — игровые и физические движки, рекомендательные системы, ядра баз данных, ядра для шахматной игры и поисковые системы.

Интерфейс прикладного программирования, или *API* (Application Programming Interface), — интерфейс для работы с библиотекой, SDK, фреймворком или ядром, предназначенный для внешнего использования. API указывает, как вызывать функции или обращаться с запросами к библиотеке для получения доступа к ее ресурсам. Создатели библиотеки (обычно) публикуют документацию API. Многие популярные социальные сети и веб-сайты предоставляют HTTP API, для того чтобы их услугами могли пользоваться программы (а не только люди с веб-браузерами). Используя такие API, можно писать программы, которые, например, автоматически публикуют сообщения в Facebook или читают потоки сообщений Twitter.

Итоги

Даже программист с многолетним опытом не всегда знает все термины из области программирования. Но крупные программные продукты обычно создаются командами разработчиков, а не отдельными людьми. Таким образом, однозначность терминологии играет важную роль при работе в команде.

Теперь вы знаете, что программы Python строятся из идентификаторов, переменных, литералов, ключевых слов и объектов, а каждый объект Python характеризуется значением, типом данных и идентичностью. И хотя каждый объект обладает типом данных, также существуют более широкие категории типов: контейнеры, последовательности, отображения, множества, встроенные типы и типы, определяемые пользователем.

Некоторые понятия (скажем, значения, переменные и функции) могут по-разному называться в особых контекстах — например, элементы, параметры, аргументы и методы. Также программисты нередко путают некоторые термины друг с другом.

В повседневном программировании путаница в таких терминах, как свойства/атрибуты, блок/тело, исключение/ошибка, тонкостях различий между библиотеками, фреймворками, SDK, ядрами и API не создаст особых проблем. Иногда при неправильном выборе термина ваш код будет нормально работать, но вы будете выглядеть непрофессионально; например, новички часто считают синонимами такие понятия, как команда и выражение, функция и метод, параметр и аргумент.

А вот такие термины, как итерируемый объект и итератор, синтаксическая ошибка и семантическая ошибка, байт-код и машинный код, имеют разный смысл. Никогда не путайте их, если только вы не хотите запутать своих коллег.

При этом смысл терминов может изменяться в зависимости от языка и даже от программиста. Со временем и по мере накопления опыта (и количества обращений в интернете) вы начнете более уверенно чувствовать себя при использовании жаргона.

Дополнительные ресурсы

В официальном глоссарии Python (<https://docs.python.org/3/glossary.html>) приведены короткие, но полезные определения, используемые в экосистеме Python. В официальной документации Python (<https://docs.python.org/3/reference/datamodel.html>) объекты Python описываются более подробно.

В докладе Нины Захаренко (Nina Zakharenko) «Memory Management in Python — The Basics» на конференции PyCon 2016 (<https://youtu.be/F6u5rhUQ6dU>) объясняются многие аспекты работы сборщика мусора в Python. Официальная документация Python (<https://docs.python.org/3/library/gc.html>) содержит больше информации о работе сборщика мусора.

Также полезно ознакомиться с обсуждением перехода на упорядоченные словари в Python 3.6 в списке рассылки Python (<https://mail.python.org/pipermail/python-dev/2016-September/146327.html>).

8

Часто встречающиеся ловушки Python



Хотя Python и является моим любимым языком программирования, он не лишен недостатков. У каждого языка есть свои раздражающие особенности (в некоторых языках их больше, чем в других), и Python не исключение. Неопытные программисты, работающие на Python, должны знать некоторые часто встречающиеся ловушки и уметь избегать их.

Как правило, такие знания люди получают на собственном опыте, а я собрал информацию о них в этой главе. Если вы будете знать историю таких ловушек, вам будет легче понять, почему Python иногда ведет себя немного странно.

В этой главе я расскажу, как изменяемые объекты, такие как списки и словари, начинают вести себя неожиданно при изменении их содержимого. Вы узнаете, почему метод `sort()` не сортирует элементы строго по алфавиту, а числа с плавающей точкой могут содержать ошибки округления, почему операторы проверки неравенства `!= has` демонстрируют непредсказуемое поведение при сцеплении, а при записи кортежей, содержащих только один элемент, необходимо включать завершающую запятую. Ну и конечно, из этой главы вы узнаете, как обойти все эти ошибки.

Не добавляйте и не удаляйте элементы из списка в процессе перебора

Добавление или удаление элементов из списка в процессе его перебора в цикле `for` или `while` с большой вероятностью приведет к ошибке. Рассмотрим следующий сценарий: вы хотите перебрать список строк с описаниями предметов одежды

и позаботиться о том, чтобы список содержал четное количество носков; для этого каждый раз, когда в списке обнаруживается один носок, в список добавляется пара для него. Задача кажется тривиальной; нужно перебрать строки из списка и при обнаружении в строке текста 'sock' (например, 'red sock') в список будет добавляться еще один элемент 'red sock'.

Но этот код работать не будет. Программа входит в бесконечный цикл, который приходится прерывать нажатием клавиш Ctrl-C:

```
>>> clothes = ['skirt', 'red sock']
>>> for clothing in clothes: # Перебрать список.
...     if 'sock' in clothing: # Найти строки, содержащие 'sock'.
...         clothes.append(clothing) # Добавить парный элемент.
...         print('Added a sock:', clothing) # Сообщить пользователю.
...
Added a sock: red sock
Added a sock: red sock
Added a sock: red sock
...
Added a sock: red sock
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
KeyboardInterrupt
```

Процесс выполнения этого кода наглядно представлен по адресу <https://author.com/addingloop/>.

Проблема в том, что при добавлении в список `clothes` элемента 'red sock' в списке появляется новый, третий элемент, который тоже должен быть включен в перебор: ['skirt', 'red sock', 'red sock']. Цикл `for` достигает второго элемента 'red sock' при следующей итерации, и он присоединяет вторую строку 'red sock'. Список преобразуется к виду ['skirt', 'red sock', 'red sock', 'red sock'], и в нем появляется еще одна строка, которая должна быть включена в перебор. Все это продолжается, как показано на рис. 8.1, и в результате вы получаете бесконечный поток сообщений 'Added a sock.'. Цикл останавливается только тогда, когда на компьютере будет исчерпана вся свободная память и программа Python аварийно завершится или вы прервете программу клавишами Ctrl-C.

Вывод: не добавляйте элементы в список во время перебора. Вместо этого создайте отдельный список для содержимого измененного списка, такой как `newClothes` в этом примере:

```
>>> clothes = ['skirt', 'red sock', 'blue sock']
>>> newClothes = []
>>> for clothing in clothes:
...     if 'sock' in clothing:
...         print('Appending:', clothing)
```

```

...         newClothes.append(clothing) # Изменяется список newClothes,
                                         # а не clothes.
...
Appending: red sock
Appending: blue sock
>>> print(newClothes)
['red sock', 'blue sock']
>>> clothes.extend(newClothes) # Присоединяем элементы newClothes к clothes.
>>> print(clothes)
['skirt', 'red sock', 'blue sock', 'red sock', 'blue sock']

```



Рис. 8.1. При каждой итерации цикла `for` к списку присоединяется новый объект `'red sock'`, к которому `clothing` переходит при следующей итерации. Цикл повторяется бесконечно

Процесс выполнения этого кода наглядно представлен на <https://auihor.com/addingloopfixed/>.

Цикл `for` перебирает элементы списка `clothes`, но `clothes` не изменяется в цикле. Вместо этого изменяется отдельный список `newClothes`. Затем после завершения цикла список `clothes` дополняется содержимым `newClothes`. В итоге вы получаете список `clothes` с парными носками.

По тем же причинам элементы не должны удаляться из списка в процессе перебора. Рассмотрим пример, в котором из списка нужно удалить любую строку, отличную от 'hello'. Наивное решение перебирает список, удаляя из него элементы, отличные от 'hello':

```
>>> greetings = ['hello', 'hello', 'mello', 'yello', 'hello']
>>> for i, word in enumerate(greetings):
...     if word != 'hello': # Удалить все элементы, отличные от 'hello'.
...         del greetings[i]
...
>>> print(greetings)
['hello', 'hello', 'yello', 'hello']
```

Процесс выполнения этого кода наглядно показан на <https://autbor.com/deletingloop/>.

Похоже, в списке остался лишний элемент 'yello'. Дело в том, что, когда цикл `for` проверял индекс 2, он удалил 'mello' из списка. Но затем все оставшиеся элементы сдвинулись на один индекс вниз и элемент 'yello' перешел с индекса 3 на индекс 2. Следующая итерация цикла проверяет индекс 3, который теперь соответствует последнему элементу 'hello' (рис. 8.2). Строка 'yello' осталась непроверенной! Не удаляйте элементы из списка в процессе перебора этого списка.

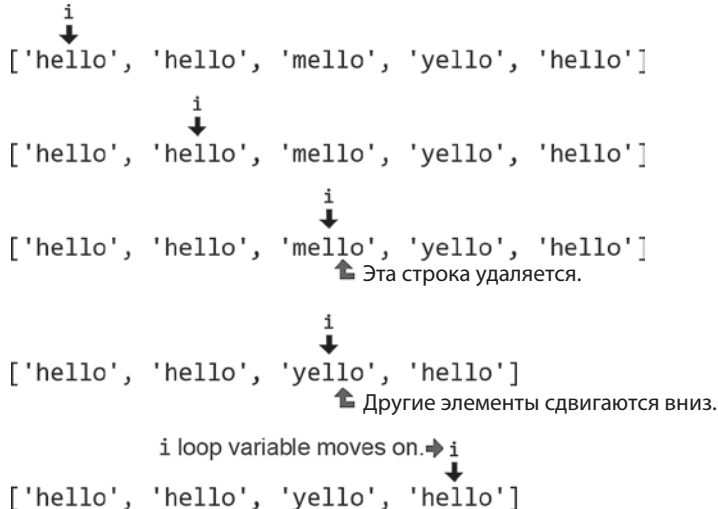


Рис. 8.2. Когда цикл удаляет 'mello', элементы списка сдвигаются на одну позицию вниз, в результате чего `i` пропускает элемент 'yello'

Вместо этого создайте новый список и скопируйте в него все элементы, кроме тех, которые должны быть удалены, после чего замените исходный список. Чтобы

увидеть исправленную версию предыдущего примера, введите следующий код в интерактивной оболочке:

```
>>> greetings = ['hello', 'hello', 'mello', 'yello', 'hello']
>>> newGreetings = []
>>> for word in greetings:
...     if word == 'hello': # Копирование всех элементов, равных 'hello'.
...         newGreetings.append(word)
...
>>> greetings = newGreetings # Заменить исходный список.
>>> print(greetings)
['hello', 'hello', 'hello']
```

Процесс выполнения этого кода наглядно представлен на <https://autbor.com/deletingloopfixed/>.

Так как этот код представляет собой простой цикл, который генерирует список, его можно заменить списковым включением. Списковое включение не выполняется быстрее и не расходует меньше памяти, но оно быстрее вводится без ущерба для удобочитаемости. Введите в интерактивной оболочке следующий фрагмент, эквивалентный коду из предыдущего примера:

```
>>> greetings = ['hello', 'hello', 'mello', 'yello', 'hello']
>>> greetings = [word for word in greetings if word == 'hello']
>>> print(greetings)
['hello', 'hello', 'hello']
```

Списковое включение не только записывается более компактно, но и избегает ошибок, возникающих из-за изменения списка в процессе его перебора.

И хотя элементы не должны добавляться или удаляться из списка (или любого итерируемого объекта) в процессе перебора, ничто не мешает вам изменять содержимое списка. Допустим, у вас есть список чисел в виде строк: ['1', '2', '3', '4', '5']. Его можно преобразовать в список целых чисел [1, 2, 3, 4, 5] в процессе перебора:

```
>>> numbers = ['1', '2', '3', '4', '5']
>>> for i, number in enumerate(numbers):
...     numbers[i] = int(number)
...
>>> numbers
[1, 2, 3, 4, 5]
```

Процесс выполнения этого кода наглядно показан на <https://autbor.com/covertstringnumbers/>. Изменять элементы в списке можно; ошибки возникают при изменении количества элементов в списке.

Также существует другой безопасный способ добавления или удаления элементов из списка: перебор от конца списка к началу. При таком подходе можно удалять элементы из списка в процессе перебора или добавлять элементы — при условии,

что новый элемент добавляется в конец списка. Например, введите следующий код, который удаляет четные целые числа из списка `someInts`:

```
>>> someInts = [1, 7, 4, 5]
>>> for i in range(len(someInts)):
...     if someInts[i] % 2 == 0:
...         del someInts[i]
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
>>> someInts = [1, 7, 4, 5]
>>> for i in range(len(someInts) - 1, -1, -1):
...     if someInts[i] % 2 == 0:
...         del someInts[i]
...
>>> someInts
[1, 7, 5]
```

Этот код работает, потому что ни у одного из элементов, которые будут перебираться в будущем, индекс не изменяется. Однако из-за многократного сдвига значений после удаляемого значения такое решение становится неэффективным для длинных списков. Процесс выполнения этого кода наглядно представлен на <https://author.com/iteratebackwards1/>. Различия между прямым и обратным перебором показаны на рис. 8.3.

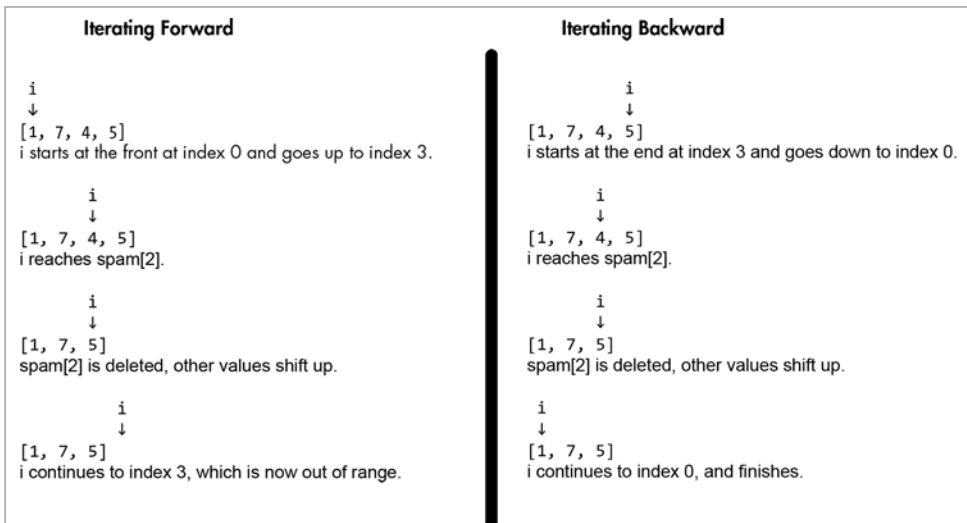


Рис. 8.3. Удаление четных чисел из списка при прямом (слева) и обратном (справа) переборе

ССЫЛКИ, ИСПОЛЬЗОВАНИЕ ПАМЯТИ И `SYS.GETSIZEOF()`

Может показаться, что создание нового списка вместо изменения исходного приводит к лишним затратам памяти. Но вспомните, что с технической точки зрения все переменные содержат ссылки на значения вместо самих значений; так же и списки содержат ссылки на значения. Приведенная выше строка `newGreetings.append(word)` не создает копию строки из переменной `word`, а только копирует ссылку на строку, которая занимает куда меньше памяти.

Чтобы убедиться в этом, можно воспользоваться функцией `sys.getsizeof()`, которая возвращает размер памяти в байтах, занимаемой переданным функцией объектом. В этом примере интерактивной оболочки мы видим, что короткая строка `'cat'` занимает 52 байта, тогда как более длинная строка занимает 85 байт:

```
>>> import sys
>>> sys.getsizeof('cat')
52
>>> sys.getsizeof('a much longer string than just "cat"')
85
```

(В моей версии Python служебная информация объекта строки занимает 49 байт, тогда как каждый символ в строке занимает 1 байт.) Однако список, содержащий любую из этих строк, занимает 72 байта независимо от длины строки:

```
>>> sys.getsizeof(['cat'])
72
>>> sys.getsizeof(['a much longer string than just "cat"'])
72
```

Дело в том, что с технической точки зрения список содержит не строки, а ссылки на строки, а ссылки всегда имеют одинаковые размеры независимо от размера данных, на которые они ссылаются. Вызов `newGreetings.append(word)` копирует не строку, а ссылку на строку. Если вы хотите узнать, сколько памяти занимает объект и все объекты, на которые он содержит ссылки, разработчик ядра Python Рэймонд Хеттингер (Raymond Hettinger) написал специальную функцию, доступную по адресу <https://code.activestate.com/recipes/577504-compute-memory-footprint-of-an-object-and-its-cont/>.

Итак, не стоит думать, что создание нового списка вместо изменения исходного списка в процессе перебора приводит к лишним затратам памяти. Даже если ваш код с изменением списка вроде бы работает, он может стать источником коварных ошибок, а их поиск и исправление займут много времени. Время программиста стоит намного дороже, чем затраты памяти компьютера.

Аналогичным образом можно добавлять элементы в конец списка при обратном переборе. Введите в интерактивной оболочке следующий фрагмент, который присоединяет копию всех четных чисел из списка `someInts` в конец списка:

```
>>> someInts = [1, 7, 4, 5]
>>> for i in range(len(someInts) - 1, -1, -1):
...     if someInts[i] % 2 == 0:
...         someInts.append(someInts[i])
...
>>> someInts
[1, 7, 4, 5, 4]
```

Процесс выполнения этого кода наглядно показан на <https://autbor.com/iteratebackwards2/>. Перебирая список в обратном направлении, можно как присоединять к нему элементы, так и удалять элементы из списка. Тем не менее сделать это иногда непросто, потому что незначительные изменения в этом решении приведут к возникновению ошибок. Гораздо проще создать новый список, вместо того чтобы изменять исходный. Разработчик ядра Python Рэймонд Хеттингер (Raymond Hettinger) сформулировал это так:

«В. Какие практики следует применять при изменении списка во время его перебора?

О. Не делайте этого».

Не копируйте изменяемые значения без `copy.copy()` и `copy.deepcopy()`

Переменные лучше представлять как наклейки или ярлыки, которые прикрепляются к объектам, а не как коробки, в которых лежат объекты. Эта модель особенно полезна в ситуациях, связанных с модификацией изменяемых объектов (списков, словарей и множеств, значение которых может изменяться). Одна из распространенных ошибок: разработчик копирует одну переменную, ссылающуюся на изменяемый объект, в другую переменную и думает, что копируется реальный объект. В Python команды копирования никогда не копируют объекты; они копируют только ссылки на объект. (Разработчик Python Нед Бэтчелдер (Ned Batchelder) выступил с превосходным докладом на эту тему «Facts and Myths about Python Names and Values» на конференции PyCon 2015 (https://youtu.be/_AEJHGK9ns).)

Например, введите следующий фрагмент в интерактивной оболочке; обратите внимание: хотя мы изменяем только переменную `spam`, переменная `cheese` тоже изменяется:

```
>>> spam = ['cat', 'dog', 'eel']
>>> cheese = spam
```

```
>>> spam
['cat', 'dog', 'eel']
>>> cheese
['cat', 'dog', 'eel']
>>> spam[2] = 'MOOSE'
>>> spam
['cat', 'dog', 'MOOSE']
>>> cheese
['cat', 'dog', 'MOOSE']
>>> id(cheese), id(spam)
2356896337288, 2356896337288
```

Процесс выполнения этого кода наглядно показан на <https://autbor.com/listcopygotcha1/>. Если вы думаете, что команда `cheese = spam` копирует объект списка, то вас может удивить, что переменная `cheese` изменилась, хотя в программе изменялась только переменная `spam`. Но команды присваивания *никогда не копируют объекты*, а только ссылки на них. Команда присваивания `cheese = spam` заставляет `cheese` ссылаться на тот же объект списка в памяти, на который ссылается переменная `spam`. Она не создает *дополнительной копии* объекта списка. Вот почему изменение `spam` также изменяет `cheese`: обе переменные ссылаются на один и тот же объект списка.

Тот же принцип действует для изменяемых объектов, передаваемых при вызове функции. Введите в интерактивной оболочке следующий фрагмент; обратите внимание на то, что глобальная переменная `spam` и локальный параметр (напомню: параметрами называются переменные, определяемые в команде `def` функции) `theList` ссылаются на один объект:

```
>>> def printIdOfParam(theList):
...     print(id(theList))
...
>>> eggs = ['cat', 'dog', 'eel']
>>> print(id(eggs))
2356893256136
>>> printIdOfParam(eggs)
2356893256136
```

Процесс выполнения этого кода демонстрируется на <https://autbor.com/listcopygotcha2/>. Обратите внимание: вызовы `id()` для `eggs` и `theList` возвращают одинаковые идентичности; это означает, что переменные ссылаются на один и тот же объект списка.

Объект списка из переменной `eggs` не был скопирован в `theList`; вместо этого была скопирована ссылка, поэтому обе переменные ссылаются на один список. Размер ссылки составляет лишь несколько байтов, но представьте, что Python вместо ссылки скопировал бы весь список. Если бы в списке `eggs` были миллиарды элементов вместо всего трех, при передаче его функции `printIdOfParam()` пришлось

бы скопировать этот огромный список. Простой вызов функции потребовал бы многих гигабайтов памяти! Вот почему команда присваивания в Python копирует только ссылки и никогда не копирует объекты. Одно из возможных решений этой проблемы заключается в копировании объекта списка (а не только ссылки) функцией `copy.copy()`. Введите следующий фрагмент в интерактивной оболочке:

```
>>> import copy
>>> bacon = [2, 4, 8, 16]
>>> ham = copy.copy(bacon)
>>> id(bacon), id(ham)
(2356896337352, 2356896337480)
>>> bacon[0] = 'CHANGED'
>>> bacon
['CHANGED', 4, 8, 16]
>>> ham
[2, 4, 8, 16]
>>> id(bacon), id(ham)
(2356896337352, 2356896337480)
```

Процесс выполнения этого кода показан на <https://autbor.com/copycopy1/>. Переменная `ham` ссылается на скопированный объект списка вместо исходного объекта списка, на который ссылалась переменная `bacon`, поэтому она не страдает от этой проблемы. Но подобно тому, как переменные правильнее сравнивать с наклейками, а не с коробками, в которых хранятся объекты, в списках также хранятся наклейки (ссылки на объекты) вместо самих объектов. Если ваш список содержит другие списки, `copy.copy()` копирует только ссылки на эти внутренние списки. Чтобы увидеть эту проблему, введите следующий фрагмент в интерактивной оболочке:

```
>>> import copy
>>> bacon = [[1, 2], [3, 4]]
>>> ham = copy.copy(bacon)
>>> id(bacon), id(ham)
(2356896466248, 2356896375368)
>>> bacon.append('APPENDED')
>>> bacon
[[1, 2], [3, 4], 'APPENDED']
>>> ham
[[1, 2], [3, 4]]
>>> bacon[0][0] = 'CHANGED'
>>> bacon
[['CHANGED', 2], [3, 4], 'APPENDED']
>>> ham
[['CHANGED', 2], [3, 4]]
>>> id(bacon[0]), id(ham[0])
(2356896337480, 2356896337480)
```

Процесс выполнения этого кода показан на <https://autbor.com/copycopy2/>. Хотя `bacon` и `ham` — два разных объекта списков, они ссылаются на одни и те же внутренние

списки [1, 2] и [3, 4], так что изменения в этих внутренних списках будут отражены в обоих переменных, несмотря на использование `copy.copy()`. Проблема решается использованием функции `copy.deepcopy()`, которая копирует все объекты списков внутри копируемых объектов списков (а также все объекты списков в этих объектах списков, и т. д.). Введите следующий фрагмент в интерактивной оболочке:

```
>>> import copy
>>> bacon = [[1, 2], [3, 4]]
>>> ham = copy.deepcopy(bacon)
>>> id(bacon[0]), id(ham[0])
(2356896337352, 2356896466184)
>>> bacon[0][0] = 'CHANGED'
>>> bacon
[['CHANGED', 2], [3, 4]]
>>> ham
[[1, 2], [3, 4]]
```

Процесс выполнения этого кода показан на <https://autbor.com/copydeepcopy/>. И хотя функция `copy.deepcopy()` работает чуть медленнее `copy.copy()`, она безопаснее, если вы не знаете, содержит ли копируемый список другие списки (или другие изменяемые объекты, такие как словари или множества). В общем случае я рекомендую всегда использовать функцию `copy.deepcopy()`: она способна предотвратить коварные ошибки, а замедление кода вряд ли будет сколько-нибудь заметным.

Не используйте изменяемые значения для аргументов по умолчанию

Python позволяет назначить *аргументы по умолчанию* для параметров функций, которые вы определяете. Если пользователь не задает значение параметра явно, то функция выполняется с аргументом по умолчанию. Это может быть удобно, если большинство вызовов функции использует одно и то же значение аргумента, потому что с аргументами по умолчанию параметр становится необязательным. Например, при передаче `None` методу `split()` разбиение производится по пробельным символам, но `None` также является аргументом по умолчанию: вызов `'cat dog'.split()` делает то же самое, что `'cat dog'.split(None)`. Функция использует аргумент по умолчанию для соответствующего параметра, если вызывающая сторона не передаст значение явно.

Однако в качестве аргумента по умолчанию никогда не следует назначать *изменяемый* объект, такой как список или словарь. Чтобы понять, к каким ошибкам это может привести, рассмотрим следующий пример. В нем определяется функция `addIngredient()`, которая добавляет строку с ингредиентом в список, представляющий собой рецепт сэндвича. Так как первым и последним элементами рецепта

обычно является хлеб (bread), изменяемый список ['bread', 'bread'] используется в качестве аргумента по умолчанию:

```
>>> def addIngredient(ingredient, sandwich=['bread', 'bread']):
...     sandwich.insert(1, ingredient)
...     return sandwich
...
>>> mySandwich = addIngredient('avocado')
>>> mySandwich
['bread', 'avocado', 'bread']
```

Но при использовании в качестве аргумента по умолчанию изменяемого объекта, такого как список ['bread', 'bread'], — возникает неочевидная проблема: список создается при выполнении команды `def` этой функции, а не при каждом вызове функции. Это означает, что создается только один объект списка ['bread', 'bread'], потому что функция `addIngredient()` определяется только один раз. При каждом вызове функции `addIngredient()` этот список будет использоваться повторно. Это приводит к неожиданному поведению:

```
>>> mySandwich = addIngredient('avocado')
>>> mySandwich
['bread', 'avocado', 'bread']
>>> anotherSandwich = addIngredient('lettuce')
>>> anotherSandwich
['bread', 'lettuce', 'avocado', 'bread']
```

Так как `addIngredient('lettuce')` использует тот же список аргументов по умолчанию, что и предыдущие вызовы, в который уже добавлен элемент 'avocado', вместо ['bread', 'lettuce', 'bread'] функция возвращает ['bread', 'lettuce', 'avocado', 'bread']. Строка 'avocado' появляется снова, потому что список для параметра `sandwich` будет тот же, что и при последнем вызове функции. Создается только один список ['bread', 'bread'], потому что команда `def` функции выполняется только один раз, а не при каждом вызове функции. Процесс выполнения этого кода наглядно показан на <https://autilbor.com/sandwich/>.

Если вам когда-либо потребуется использовать список или словарь в качестве аргумента по умолчанию, то питоническим решением будет назначить аргумент по умолчанию `None`. Затем в функцию включается код, который проверяет эту ситуацию и создает новый список или словарь при каждом вызове функции. Тем самым гарантируется, что функция будет создавать новый изменяемый объект при каждом вызове функции, вместо того чтобы делать это только один раз при определении функции, как в следующем примере:

```
>>> def addIngredient(ingredient, sandwich=None):
...     if sandwich is None:
...         sandwich = ['bread', 'bread']
...     sandwich.insert(1, ingredient)
```

```
...     return sandwich
...
>>> firstSandwich = addIngredient('cranberries')
>>> firstSandwich
['bread', 'cranberries', 'bread']
>>> secondSandwich = addIngredient('lettuce')
>>> secondSandwich
['bread', 'lettuce', 'bread']
>>> id(firstSandwich) == id(secondSandwich)
False ❶
```

Обратите внимание: `firstSandwich` и `secondSandwich` не используют одну и ту же ссылку на список ❶, потому что `sandwich = ['bread', 'bread']` создает новый объект списка при каждом вызове `addIngredient()`, а не однократно при определении `addIngredient()`.

К числу изменяемых типов данных относятся списки, словари, множества и объекты, создаваемые командой `class`. *Не задавайте* объекты этих типов как аргументы по умолчанию в командах `def`.

Не создавайте строки посредством конкатенации

В Python строки являются *неизменяемыми* (immutable) объектами. Это означает, что строковые значения не могут изменяться, и любой код, который на первый взгляд изменяет строку, в действительности создает новый объект строки. Например, каждая из следующих операций изменяет содержимое переменной `spam` — не изменением строкового значения, а заменой его новым строковым значением с новой идентичностью:

```
>>> spam = 'Hello'
>>> id(spam), spam
(38330864, 'Hello')
>>> spam = spam + ' world!'
>>> id(spam), spam
(38329712, 'Hello world!')
>>> spam = spam.upper()
>>> id(spam), spam
(38329648, 'HELLO WORLD!')
>>> spam = 'Hi'
>>> id(spam), spam
(38395568, 'Hi')
>>> spam = f'{spam} world!'
>>> id(spam), spam
(38330864, 'Hi world!')
```

Обратите внимание: при каждом вызове `id(spam)` возвращается новая идентичность, потому что объект строки в `spam` не изменяется: он заменяется совершенно

новым объектом строки с новой идентичностью. При создании новых строк с использованием f-строк, строкового метода `format()` или спецификаторов формата `%s` также создаются новые объекты строк, как и при конкатенации строк. Обычно эта техническая подробность ни на что не влияет. Python — высокоуровневый язык, который берет на себя многие технические решения, чтобы вы могли сосредоточиться на создании программы. Каждая итерация цикла создает новый объект строки, а старый объект строки при этом пропадает; в коде происходящее выглядит как выполнение конкатенаций в цикле `for` или `while`, как в следующем примере:

```
>>> finalString = ''
>>> for i in range(100000):
...     finalString += 'spam '
...
>>> finalString
spam spam spam spam spam spam spam spam spam spam --snip--
```

Так как операция `finalString += 'spam '` выполняется 100 000 раз внутри цикла, Python выполняет 100 000 конкатенаций. Процессору приходится создавать все промежуточные строковые значения, объединяя текущее значение `finalString` со строкой `'spam '`, помещать их в память, а затем почти немедленно уничтожать результат при следующей итерации. Все это крайне неэффективно, потому что интересует нас только конечная строка.

Питонический способ построения строк основан на присоединении меньших строк к списку и на последующем слиянии элементов списка в одну строку. Этот способ также создает 100 000 строковых объектов, но с выполнением только одной конкатенации строк при вызове `join()`. Например, следующий код создает эквивалентный объект `finalString`, но без промежуточных конкатенаций:

```
>>> finalString = []
>>> for i in range(100000):
...     finalString.append('spam ')
...
>>> finalString = ''.join(finalString)
>>> finalString
spam spam spam spam spam spam spam spam spam spam --snip--
```

Когда я замерил время выполнения двух фрагментов кода на моей машине, версия с присоединением к списку работала в 10 раз быстрее версии с конкатенацией. (В главе 13 описано, как измерить скорость выполнения ваших программ.) Чем больше итераций содержит цикл, тем заметнее различия. Но если заменить `range(100000)` на `range(100)`, хотя конкатенация все равно работает медленнее присоединения, различия в скорости становятся пренебрежимо малыми. Не обязательно фанатично избегать конкатенации строк, f-строк, метода строк `format()` или спецификатора формата `%s`. Скорость значительно повышается только при большом количестве конкатенаций.

Python избавляет вас от необходимости думать о многих технических подробностях. Это позволяет программисту быстро создавать программы, а, как я упоминал ранее, время программиста ценнее процессорного времени. Но в некоторых случаях желательно понимать суть происходящего (например, различия между неизменяемыми строками и изменяемыми списками), чтобы не попасть в скрытую ловушку, как при построении строк конкатенацией.

Не рассчитывайте, что `sort()` выполнит сортировку по алфавиту

Работа алгоритмов сортировки — алгоритмов, которые систематически упорядочивают значения в некотором установленном порядке, — стала одной из важных тем компьютерной теории. Но эта книга не посвящена компьютерной теории; знать такие алгоритмы не обязательно, потому что вы всегда можете вызвать метод Python `sort()`. Однако неожиданно вы замечаете, что `sort()` начинает странно себя вести при сортировке: буква *Z* в верхнем регистре предшествует букве *a* в нижнем регистре:

```
>>> letters = ['z', 'A', 'a', 'Z']
>>> letters.sort()
>>> letters
['A', 'Z', 'a', 'z']
```

Стандарт ASCII (American Standard Code for Information Interchange) определяет соответствие между числовыми кодами (которые называются *кодowymi пунктами*, или *кодами*, — code points, или ordinals) и символами текста. Метод `sort()` использует ASCII-алфавитную сортировку (обобщенный термин, означающий сортировку по кодовым пунктам) вместо алфавитной сортировки. В системе ASCII буква *A* представляется кодовым пунктом 65, *B* — кодовым пунктом 66 и так далее до буквы *Z* с кодовым пунктом 90. Буква *a* в нижнем регистре представляется кодовым пунктом 97, *b* — кодовым пунктом 98 и так далее до буквы *z* с кодовым пунктом 122. При сортировке в порядке ASCII буква *Z* в верхнем регистре (кодovый пункт 90) предшествует букве *a* в нижнем регистре (кодovый пункт 97).

И хотя кодировка ASCII почти повсеместно применялась в компьютерной области в 1990-е годы и ранее, это чисто американский стандарт: в ASCII существует кодovый пункт для знака доллара \$ (кодovый пункт 36), но нет кодovого пункта для знака британского фунта £. В наши дни кодировку ASCII в основном заменил Юникод, потому что он содержит все кодovые пункты ASCII, а также более 100 000 других кодovых пунктов.

Чтобы получить кодovый пункт символа, передайте этот символ функции `ord()`. Также можно выполнить обратную операцию: передать кодovый пункт функции

`chr()`, которая возвращает строковое представление символа. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> ord('a')
97
>>> chr(97)
'a'
```

Если вы хотите выполнить сортировку по алфавиту, передайте метод `str.lower` в параметре `key`. Список сортируется так, как если бы для значений вызывался метод строк `lower()`:

```
>>> letters = ['z', 'A', 'a', 'Z']
>>> letters.sort(key=str.lower)
>>> letters
['A', 'a', 'z', 'Z']
```

Обратите внимание: реальные строки в списке не преобразуются к нижнему регистру; они только сортируются так, как если бы они к нему были преобразованы. Нед Бэтчелдер (Ned Batchelder) предоставляет дополнительную информацию о Юникоде и кодовых пунктах в своем докладе «Pragmatic Unicode, or, How Do I Stop the Pain?» (<https://nedbatchelder.com/text/unipain.html>).

Кстати, метод Python `sort()` использует алгоритм сортировки Timsort, который был спроектирован разработчиком ядра Python и автором тезисов «Дзен Python» Тимом Петерсом. Алгоритм представляет собой гибрид алгоритмов сортировки методом слияния и сортировки методом вставки, его описание доступно на <https://ru.wikipedia.org/wiki/Timsort>.

Не рассчитывайте на идеальную точность чисел с плавающей точкой

Компьютеры позволяют хранить только цифры двоичной системы счисления, то есть 1 и 0. Для представления знакомых десятичных чисел необходимо преобразовать такое число, как 3,14, в серию двоичных единиц и нулей. Компьютеры делают это в соответствии со стандартом IEEE 754, опубликованным Институтом инженеров по электротехнике и радиоэлектронике (IEEE, Institute of Electrical and Electronics Engineers). Для простоты эти подробности остаются скрытыми от программиста, чтобы он мог просто вводить числа с точкой, не отвлекаясь на процесс преобразования десятичных значений в двоичные:

```
>>> 0.3
0.3
```

Хотя подробности конкретных случаев выходят за рамки книги, представление чисел с плавающей точкой в стандарте IEEE 754 не всегда точно соответствует исходному десятичному числу. Классический пример — число 0.1:

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.3 == (0.1 + 0.1 + 0.1)
False
```

Эта странная сумма с небольшой погрешностью является результатом ошибки округления, возникающей из-за особенностей компьютерного представления и обработки чисел с плавающей точкой. Происходящее не является особенностью *Python*; стандарт IEEE 754 является *аппаратным* стандартом, реализованным на уровне электронных компонентов процессора, обеспечивающих вычисления с плавающей точкой. Те же результаты будут получены при выполнении программы на C++, JavaScript и любом другом языке — на процессоре, использующем стандарт IEEE 754 (то есть практически на любом процессоре в мире).

Стандарт IEEE 754 (по техническим причинам объяснение которых выходит за рамки книги) также не позволяет представлять числовые значения, превышающие 2^{53} . Например, числа 2^{53} и $2^{53}+1$ в представлении с плавающей точкой округляются до 9007199254740992.0:

```
>>> float(2**53) == float(2**53) + 1
True
```

Если вы используете тип данных с плавающей точкой, у этой проблемы нет обходного решения. Не беспокойтесь — если только вы не пишете программы для банков или пункта управления ядерным реактором (или управления ядерным реактором в банке), ошибки округления достаточно незначительны, чтобы не создавать существенных проблем для вашей программы.

Часто проблему можно решить использованием целых чисел с меньшими единицами: например, 133 цента вместо 1.33 доллара, или 200 миллисекунд вместо 0.2 секунды. Таким образом, $10 + 10 + 10$ в сумме дают 30 центов или миллисекунд, тогда как суммирование $0.1 + 0.1 + 0.1$ дает 0.30000000000000004 доллара или секунды.

Но если вам требуется абсолютная точность (допустим, для научных или финансовых вычислений), используйте встроенный модуль Python `decimal`, опубликованный по адресу <https://docs.python.org/3/library/decimal.html>. И хотя объекты `Decimal` работают медленнее, они обеспечивают более точную замену для значений с плавающей точкой. Например, вызов `decimal.Decimal('0.1')` создает объект, представляющий точное число 0.1 без погрешности, которая бы неизбежно присутствовала в значении с плавающей точкой 0.1.

Если передать значение с плавающей точкой `0.1` при вызове `decimal.Decimal()`, будет создан объект `Decimal` с такой же погрешностью, как у значения с плавающей точкой; вот почему полученный объект `Decimal` не будет в точности равен `Decimal('0.1')`. Вместо этого `decimal.Decimal()` следует передавать строковое представление числа с плавающей точкой. Чтобы убедиться в этом, введите следующий фрагмент в интерактивной оболочке:

```
>>> import decimal
>>> d = decimal.Decimal(0.1)
>>> d
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> d = decimal.Decimal('0.1')
>>> d
Decimal('0.1')
>>> d + d + d
Decimal('0.3')
```

У целых чисел ошибки округления отсутствуют, поэтому передавать их `decimal.Decimal()` всегда безопасно. Введите следующий фрагмент в интерактивной оболочке:

```
>>> 10 + d
Decimal('10.1')
>>> d * 3
Decimal('0.3')
>>> 1 - d
Decimal('0.9')
>>> d + 0.1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'decimal.Decimal' and 'float'
```

Однако объекты `Decimal` не обладают неограниченной точностью; просто они имеют прогнозируемый, четко установленный уровень точности. Для примера рассмотрим следующие операции:

```
>>> import decimal
>>> d = decimal.Decimal(1) / 3
>>> d
Decimal('0.333333333333333333333333333333')
>>> d * 3
Decimal('0.999999999999999999999999999999')
>>> (d * 3) == 1 # d is not exactly 1/3
False
```

Выражение `decimal.Decimal(1) / 3` в результате дает значение, которое не равно точно $1/3$. Но по умолчанию оно будет точным до 28 значащих цифр. Чтобы узнать, сколько значащих цифр использует модуль `decimal`, обратитесь к атрибуту

`decimal.getcontext().prec`. (С технической точки зрения `prec` является атрибутом объекта `Context`, возвращаемого вызовом `getcontext()`, но его удобно разместить в той же строке.) Этот атрибут можно изменить, чтобы все объекты `Decimal`, создаваемые в дальнейшем, имели новый уровень точности. Следующий пример, введенный в интерактивной оболочке, понижает точность с исходных 28 значащих цифр до 2:

```
>>> import decimal
>>> decimal.getcontext().prec
28
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal(1) / 3
Decimal('0.33')
```

Модуль `decimal` позволяет точно управлять взаимодействием между числами. Полная документация этого модуля доступна по адресу <https://docs.python.org/3/library/decimal.html>.

Не объединяйте операторы `!=` в цепочку

Сцепленные операторы сравнения (например, `18 < age < 35`) или сцепленные присваивания (например, `six = halfDozen = 6`) являются удобными сокращениями для `(18 < age) and (age < 35)` и `six = 6; halfDozen = 6` соответственно.

Однако оператор проверки неравенства `!=` в цепочках использовать не стоит. На первый взгляд, следующий код проверяет, что все три переменные имеют разные значения, потому что следующее выражение дает результат `True`:

```
>>> a = 'cat'
>>> b = 'dog'
>>> c = 'moose'
>>> a != b != c
True
```

Но на самом деле эта цепочка эквивалентна `(a != b) and (b != c)`. А это означает, что `a` и `c` могут быть равны и выражение `a != b != c` все равно будет равно `True`:

```
>>> a = 'cat'
>>> b = 'dog'
>>> c = 'cat'
>>> a != b != c
True
```

Это весьма коварная ошибка, а код на первый взгляд выглядит нормально, поэтому лучше полностью избегать сцепленных операторов `!=`.

Не забудьте запятую в кортежах из одного элемента

Когда вы записываете значения кортежей в своем коде, помните о том, что завершающая запятая необходима даже в том случае, если кортеж содержит только один элемент. Хотя значение `(42, )` представляет собой кортеж, содержащий целое число 42, запись `(42)` обозначает всего лишь целое число 42. Скобки в `(42)` сходны с используемыми в выражении `(20 + 1) * 2`, результатом которого является целое число 42. Если вы забудете поставить запятую, это может иметь непредвиденные последствия:

```
>>> spam = ('cat', 'dog', 'moose')
>>> spam[0]
'cat'
>>> spam = ('cat')
>>> spam[0]      ❶
'c'
>>> spam = ('cat', ) ❷
>>> spam[0]
'cat'
```

Без запятой при вычислении `('cat')` будет получен строковый результат, поэтому `spam[0]` дает первый символ строки, `'c'` ❶. Завершающая запятая необходима для того, чтобы выражение в круглых скобках распознавалось как значение-кортеж ❷. В Python кортеж определяется запятыми в большей степени, чем круглыми скобками.

Итоги

Несогласованность встречается в любом языке, даже в языках программирования. Python содержит несколько потенциальных ловушек, которые подстерегают невнимательных. И хотя они встречаются достаточно редко, лучше знать о них заранее. Это позволит вам быстро опознать и устранить проблемы, которые могут из-за них возникнуть.

Хотя теоретически элементы можно добавлять и удалять из списка в процессе перебора этого списка, это потенциальный источник ошибок. Намного безопаснее перебрать копию списка, а затем внести изменения в оригинал. Создавая копии списка (или любого другого изменяемого объекта), помните, что команды присваивания копируют только ссылку на объект, а не сам объект. Для создания копий объекта (и копий всех объектов, ссылки на которые в нем хранятся) можно воспользоваться функцией `copy.deepcopy()`.

Изменяемые объекты не должны использоваться в командах `def` для аргументов по умолчанию, потому что они создаются однократно при выполнении команды `def`, а не при каждом вызове функции. Лучше использовать в качестве аргумента по умолчанию `None` и добавить код, который проверяет `None` и создает изменяемый объект при вызове функции.

Еще одна коварная ловушка возникает при конкатенации нескольких строк оператором `+` в цикле. При небольшом количестве итераций этот синтаксис работает нормально. Но «под капотом» Python постоянно создает и уничтожает объекты строк при каждой итерации. Более эффективное решение — присоединять меньшие строки к списку, а затем вызвать оператор `join()` для создания итоговой строки.

Метод `sort()` сортирует элементы по числовым кодовым пунктам, и этот порядок не совпадает с алфавитным: буква `Z` в верхнем регистре предшествует букве `a` в нижнем регистре. Проблему можно решить вызовом `sort(key=str.lower)`.

У чисел с плавающей точкой возникают небольшие погрешности — это побочный эффект способа представления чисел в памяти компьютера. Для большинства программ эти ошибки не важны. Но если для ваших задач это важно, вы можете воспользоваться модулем Python `decimal`.

Никогда не объединяйте операторы `!=` в цепочки, потому что, как ни странно, выражения вида `'cat' != 'dog' != 'cat'` дают результат `True`.

Хотя в этой главе описаны ловушки Python, которые вы наверняка встретите в работе, они не попадают ежедневно в реальный код. Python всеми силами старается свести к минимуму неприятные неожиданности в ваших программах. В следующей главе я расскажу о ловушках еще более редких и экзотических. Вероятность того, что они когда-нибудь попадутся на вашем пути, близка к нулю, и все же вам наверняка будет любопытно исследовать причины их существования.

9

Экзотические странности Python



Системы правил, определяющие язык программирования, достаточно сложны. Порой они приводят к появлению кода хотя и не ошибочного, но достаточно странного и неожиданного. В этой главе мы займемся некоторыми малоизвестными странностями языка Python. Вряд ли вы когда-нибудь столкнетесь с ними в повседневном программировании, но их можно рассматривать как неочевидное применение синтаксиса Python (или злоупотребление им — зависит от точки зрения).

Изучая примеры этой главы, вы начнете лучше представлять, как работают внутренние механизмы Python. Давайте исследуем некоторые экзотические случаи — из интереса и для расширения кругозора.

Почему 256 — это 256, а 257 — не 257

Оператор `==` проверяет, равны ли два объекта, а оператор `is` проверяет их на тождественность, то есть сравнивает их идентичности. И хотя целое число 42 и число с плавающей точкой 42.0 имеют одинаковое значение, это два разных объекта, хранящихся в разных местах компьютерной памяти. Чтобы убедиться в этом, проверьте их идентичности при помощи функции `id()`:

```
>>> a = 42
>>> b = 42.0
>>> a == b
True
>>> a is b
False
>>> id(a), id(b)
(140718571382896, 2526629638888)
```

Когда Python создает новый объект, представляющий целое число, и сохраняет его в памяти, создание объекта занимает очень мало времени. CPython (интерпретатор Python, доступный для загрузки по адресу <https://python.org>) применяет крошечную оптимизацию, создавая объекты целых чисел от -5 до 256 при запуске каждой программы. Эти числа называются *предварительно определенными* (preallocated), и CPython автоматически создает для них объекты, потому что они довольно часто применяются: число 0 или 2 будет использовано в программе с большей вероятностью, чем, скажем, 1729 . При создании объекта нового целого числа в памяти CPython сначала проверяет, принадлежит ли оно диапазону от -5 до 256 . В таком случае CPython экономит время, просто возвращая существующий объект целого числа, вместо того чтобы создавать его заново. Такое поведение также экономит память, так как она не расходуется на хранение малых целых чисел (рис. 9.1).

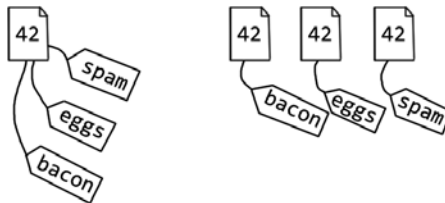


Рис. 9.1. Для экономии памяти Python использует множественные ссылки на один объект целого числа (слева) вместо дублирующихся объектов целых чисел для каждой ссылки (справа)

Из-за этой оптимизации некоторые нетипичные ситуации могут приводить к странным результатам. Чтобы увидеть пример такого рода, введите следующий фрагмент в интерактивной оболочке:

```
>>> a = 256
>>> b = 256
>>> a is b ❶
True
>>> c = 257
>>> d = 257
>>> c is d ❷
False
```

Все объекты, представляющие целое число 256 , в действительности являются одним объектом, поэтому для a и b оператор возвращает `True` ❶. Но для c и d Python создает разные объекты 257 , поэтому оператор `is` возвращает `False` ❷.

Выражение `257 is 257` дает результат `True`, потому что CPython повторно использует объект целого числа, созданного для идентичного литерала в той же команде:

```
>>> 257 is 257
True
```

Конечно, в реальных программах обычно используется только значение целого числа, а не его идентичность. Программы никогда не используют оператор `is` для сравнения целых чисел, чисел с плавающей точкой, строк, логических значений или значений других простых типов данных. Одно из исключений такого рода встречается при использовании `is None` вместо `== None`, и я объясняю это в подразделе «Использование `is` для сравнения с `None` вместо `==`», с. 122. В остальных ситуациях это исключение почти не встречается.

Интернирование строк

Аналогичным образом Python повторно использует строки для представления идентичных строковых литералов в вашем коде вместо создания нескольких копий одной строки. Чтобы убедиться в этом, введите следующий фрагмент в интерактивной оболочке:

```
>>> spam = 'cat'
>>> eggs = 'cat'
>>> spam is eggs
True
>>> id(spam), id(eggs)
(1285806577904, 1285806577904)
```

Python замечает, что строковый литерал `'cat'`, присвоенный `eggs`, совпадает со строковым литералом `'cat'`, присвоенным `spam`; таким образом, вместо второго, избыточного объекта `string` переменной `eggs` просто присваивается ссылка на тот же объект строки, который используется `spam`. Этим объясняется совпадение идентичностей строк.

Такая оптимизация называется *интернированием* (string interning) строк. Как и предварительное определение целых чисел, это всего лишь подробность реализации CPython. Никогда не пишите код, который зависит от нее. Кроме того, эта оптимизация не обнаруживает все возможные идентичные строки. Попытки идентифицировать все экземпляры, к которым можно применить оптимизацию, часто занимают больше времени, чем сэкономят оптимизация. Например, попробуйте создать строку `'cat'` из `'c'` и `'at'` в интерактивной оболочке; вы заметите, что CPython создает итоговую строку `'cat'` как новый объект строки, вместо того чтобы повторно использовать объект строки, созданный для `spam`:

```
>>> bacon = 'c'
>>> bacon += 'at'
>>> spam is bacon
False
```

```
>>> id(spam), id(bacon)
(1285806577904, 1285808207384)
```

Интернирование строк — метод оптимизации, применяемый интерпретаторами и компиляторами для многих языков программирования. За дополнительной информацией обращайтесь на https://en.wikipedia.org/wiki/String_interning.

Фиктивные операторы инкремента и декремента в языке Python

В Python можно увеличить значение переменной на 1 или уменьшить его на 1 при помощи расширенных операторов присваивания. Выражения `spam += 1` и `spam -= 1` увеличивают и уменьшают числовые значения в `spam` на 1 соответственно.

В других языках, таких как C++ и JavaScript, существуют операторы `++` и `--` для выполнения инкремента и декремента. (Этот факт отражен в самом названии языка C++; это иронический намек на то, что язык является улучшенной формой языка C.) В коде на языках C++ и JavaScript могут встречаться такие операции, как `++spam` или `spam++`. Создатели Python благоразумно не стали включать в язык эти операторы, получившие печальную известность из-за коварных ошибок (см. <https://softwareengineering.stackexchange.com/q/59880>).

Тем не менее следующий код Python вполне допустим:

```
>>> spam = --spam
>>> spam
42
```

Первое, на что следует обратить внимание: операторы `++` и `--` в Python не увеличивают и не уменьшают значение, хранимое в переменной `spam`. Начальный знак `-` является унарным оператором отрицания Python. Он позволяет писать код следующего вида:

```
>>> spam = 42
>>> -spam
-42
```

Ничто не запрещает поставить перед значением несколько унарных операторов отрицания. С двумя операторами выполняется отрицание отрицания значения, что для целых чисел просто дает исходное значение:

```
>>> spam = 42
>>> -(-spam)
42
```

Это очень глупая операция, и, скорее всего, вы никогда не увидите двукратное использование унарного оператора отрицания в реальном коде. (А если увидите, то, наверное, программист учился программировать на другом языке и просто написал ошибочный код на Python!)

Также существует унарный оператор `+`. Он создает целое значение с таким же знаком, как у исходного значения, то есть по сути не делает ничего:

```
>>> spam = 42
>>> +spam
42
>>> spam = -42
>>> +spam
-42
```

Запись `+42` (или `++42`) выглядит так же глупо, как и `--42`, так почему же в Python был включен этот унарный оператор? Он существует только как парный по отношению к оператору `-`, если вам потребуется перегрузить эти операторы в ваших классах. (Здесь много терминов, которые могут быть вам незнакомы. Перегрузка операторов более подробно рассматривается в главе 17.)

Унарные операторы `+` и `-` могут располагаться только перед значением Python, но не после него. И хотя выражения `spam++` и `spam--` могут быть действительным кодом в C++ или JavaScript, в Python они вызывают синтаксические ошибки:

```
>>> spam++
  File "<stdin>", line 1
    spam++
      ^
SyntaxError: invalid syntax
```

В Python нет операторов инкремента и декремента. Эта странность языкового синтаксиса только создает иллюзию того, что они существуют.

Все или ничего

Встроенная функция `all()` получает значение-последовательность (например, список) и возвращает `True`, если все значения в этой последовательности являются истинными. Если хотя бы одно или несколько значений являются ложными, возвращается `False`. Вызов функции `all([False, True, True])` можно рассматривать как эквивалент выражения `False and True and True`.

Функция `all()` может использоваться в сочетании со списковыми включениями, для того чтобы сначала создать список логических значений на основании другого

списка, а затем вычислить их сводное значение. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> spam = [67, 39, 20, 55, 13, 45, 44]
>>> [i > 42 for i in spam]
[True, False, False, True, False, True, True]
>>> all([i > 42 for i in spam])
False
>>> eggs = [43, 44, 45, 46]
>>> all([i > 42 for i in eggs])
True
```

Функция `all()` возвращает `True`, если все числа в `spam` или `eggs` больше 42.

Но если передать `all()` пустую последовательность, функция всегда возвращает `True`. Введите следующий фрагмент в интерактивной оболочке:

```
>>> all([])
True
```

Правильнее считать, что `all([])` проверяет утверждение «ни один из элементов списка не является ложным» (вместо «все элементы списка являются истинными»). В противном случае вы можете получить неожиданные результаты. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> spam = []
>>> all([i > 42 for i in spam])
True
>>> all([i < 42 for i in spam])
True
>>> all([i == 42 for i in spam])
True
```

Пример вроде бы показывает, что все значения в `spam` (пустой список!) не только больше 42, но они одновременно меньше 42 и равны 42! Это кажется невозможным с точки зрения логики. Но помните, что каждое из трех списковых включений при вычислении дает пустой список. В нем нет ни одного ложного элемента, а все функции `all()` возвращают `True`.

Логические значения как целые числа

Подобно тому как Python считает, что значение с плавающей точкой `42.0` равно целому значению `42`, логические значения `True` и `False` считаются эквивалентными `1` и `0` соответственно. В Python тип данных `bool` является субклассом типа данных `int`. (Классы и субклассы рассматриваются в главе 16.) Вы можете воспользоваться `int()` для преобразования логических значений в целые числа:

```
>>> int(False)
0
>>> int(True)
1
>>> True == 1
True
>>> False == 0
True
```

Вы также можете использовать `isinstance()`, чтобы подтвердить, что логическое значение относится к типу `integer`:

```
>>> isinstance(True, bool)
True
>>> isinstance(True, int)
True
```

Значение `True` относится к типу данных `bool`. Но так как `bool` является субклассом `int`, `True` также является частным случаем `int`. Это означает, что `True` и `False` могут использоваться практически в любом месте, где допустимо применение целых чисел. Это порождает странный код:

```
>>> True + False + True + True # То же, что 1 + 0 + 1 + 1
3
>>> -True # То же, что -1.
-1
>>> 42 * True # То же, что математическое умножение 42 * 1.
42
>>> 'hello' * False # То же, что репликация строк 'hello' * 0.
''
>>> 'hello'[False] # То же, что 'hello'[0]
'h'
>>> 'hello'[True] # То же, что 'hello'[1]
'e'
>>> 'hello'[-True] # То же, что 'hello'[-1]
'o'
```

Конечно, из того, что значения `bool` могут использоваться как числа, не следует, что это стоит делать. Все приведенные примеры очень плохо читаются и никогда не должны использоваться в реальном коде. Изначально в Python не было типа данных `bool`. Логический тип появился только в Python 2.3, когда тип `bool` был определен как субкласс `int` для упрощения реализации. С историей типа данных `bool` можно ознакомиться в PEP 285 на <https://www.python.org/dev/peps/pep-0285/>.

Кстати говоря, `True` и `False` стали ключевыми словами только в Python 3. А следовательно, в Python 2 `True` и `False` можно было использовать как имена переменных, что приводит к появлению парадоксального кода следующего вида:

```
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> True is False
False
>>> True = False
>>> True is False
True
```

К счастью, подобный запутанный код невозможен в Python 3, и при попытке использовать ключевые слова `True` и `False` в качестве имен переменных происходит синтаксическая ошибка.

Сцепление разных видов операторов

Цепочки разных операторов в одном выражении могут привести к неожиданным ошибкам. Например, следующий (откровенно говоря, нереалистичный) пример использует операторы `==` и `in` в одном выражении:

```
>>> False == False in [False]
True
```

Результат `True` выглядит удивительно, потому что можно ожидать, что выражение будет вычисляться по одной из следующих схем.

- `(False == False) in [False]`, дает результат `False`.
- `False == (False in [False])`, также дает результат `False`.

Но выражение `False == False in [False]` не эквивалентно ни одному из этих выражений. Вместо этого оно эквивалентно `(False == False) and (False in [False])`, подобно тому как `42 < spam < 99` эквивалентно `(42 < spam) and (spam < 99)`. Это выражение вычисляется по следующей диаграмме:

```
(False == False) and (False in [False])
      ↓
  (True) and (False in [False])
      ↓
  (True) and (True)
      ↓
      True
```

Выражение `False == False in [False]` скорее является любопытной головоломкой на языке Python, но вряд ли вы когда-нибудь встретите его в реальном коде.

Антигравитация в Python

Введите следующую команду в интерактивной оболочке:

```
>>> import antigravity
```

Эта строка — забавная пасхалка, которая открывает в браузере классический комикс XKCD о Python (<https://xkcd.com/353/>). Вас может удивить, что Python открывает ваш браузер, но это встроенная возможность, предоставляемая модулем `webbrowser`. Модуль Python `webbrowser` содержит функцию `open()`, которая находит браузер по умолчанию вашей операционной системы и открывает его окно с заданным URL-адресом. Введите следующую команду в интерактивной оболочке:

```
>>> import webbrowser
>>> webbrowser.open('https://xkcd.com/353/')
```

Возможности модуля `webbrowser` ограничены, но он позволяет легко перенаправить пользователя на дополнительную информацию в интернете.

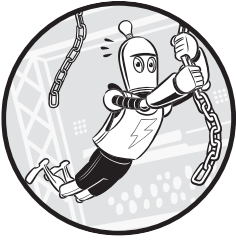
Итоги

Мы часто забываем, что компьютеры и языки программирования создавались людьми и у них есть свои ограничения. Очень многие программы зависят от предпочтений и вкусов проектировщиков языков и разработчиков оборудования. Эти люди прикладывают невероятные усилия к тому, чтобы все ошибки в ваших программах возникали из-за дефектов самой программы, а не интерпретатора или процессора, на котором программа выполняется. В конечном итоге мы начинаем воспринимать эти инструменты как нечто данное свыше. Но именно в этом случае становится ясной ценность изучения потаенных «уголков» и «закоулков» компьютеров и программного обеспечения. Когда в вашем коде возникают ошибки или фатальные сбои (или он просто странно ведет себя, и вы начинаете думать: «Здесь что-то не так»), необходимо знать стандартные ловушки, чтобы устранить эти проблемы.

Вы почти наверняка не столкнетесь ни с одной из странностей работы кода, упомянутых в этой главе, но именно их знание сделает вас настоящим программистом на языке Python.

10

Написание эффективных функций



Функции — своего рода мини-программы, которые позволяют разбить код на меньшие части. Они избавляют нас от необходимости писать повторяющийся код, который может стать причиной ошибок. Но для написания эффективных функций приходится принимать много решений — относительно имени, размера, параметров и сложности.

В этой главе я покажу различные способы написания функций, а также достоинства и недостатки различных компромиссных решений. Мы разберемся, когда стоит отдавать предпочтение большим или малым функциям, как количество параметров влияет на сложность функции и как писать функции с переменным количеством аргументов при помощи операторов `*` и `**`. Также в этой главе я расскажу о парадигме функционального программирования и преимуществах написания функций в соответствии с этой парадигмой.

Имена функций

Имена функций подчиняются тем же правилам, что и имена идентификаторов вообще (см. главу 4). Но обычно имена функций должны включать глагол, потому что функции выполняют некоторое действие. Также можно включить существительное для описания объекта, с которым это действие выполняется. Например, имена `refreshConnection()`, `setPassword()` и `extract_version()` хорошо поясняют, что делает функция и к чему применяется действие.

Существительное может оказаться излишним для методов, которые являются частью класса или модуля. Метод `reset()`, принадлежащий классу `SatelliteConnection`, или функция `open()` модуля `webbrowser` уже предоставляют необходимый контекст.

Из названий и так ясно, что `satellite connection` — это то, что перезагружается (`reset`), а веб-браузер — это то, что открывается (`open`).

Лучше использовать длинные, содержательные имена вместо сокращений или слишком коротких имен. Возможно, математик сразу поймет, что функция с именем `gcd()` возвращает наибольший общий делитель двух чисел, но для всех остальных имя `getGreatestCommonDenominator()` станет более понятным.

Не используйте для своих переменных или функций имена встроенных функций или модулей Python — такие как `all`, `any`, `date`, `email`, `file`, `format`, `hash`, `id`, `input`, `list`, `min`, `max`, `object`, `open`, `random`, `set`, `str`, `sum`, `test` и `type`.

Плюсы и минусы размера функций

Некоторые программисты считают, что функции должны быть по возможности короткими, а весь код — помещаться на одном экране. Функцию, которая занимает всего десяток строк, легко понять — по крайней мере в отличие от той, что состоит из сотни строк. Однако разбиение кода функции на несколько меньших функций также имеет свои недостатки. Рассмотрим некоторые преимущества малых функций.

- Код функции проще понять.
- Функции, скорее всего, требуются меньше параметров.
- Снижается вероятность побочных эффектов (см. «Функциональное программирование», с. 206).
- Упрощается тестирование и отладка функции.
- Вероятно, функция будет выдавать меньше разных видов исключений.

Однако у коротких функций имеются свои недостатки.

- Написание коротких функций часто означает увеличение их числа в программе.
- Чем больше функций, тем сложнее становится программа.
- Увеличение количества функций также означает, что вам придется придумывать больше содержательных и точных имен, а это не так просто.
- Вам придется писать больше документации.
- Чем больше функций, тем сложнее связи между ними.

Некоторые программисты доводят принцип «чем короче, тем лучше» до крайности и заявляют, что все функции должны содержать не более трех-четырех строк кода. Это настоящее безумие. Для примера ниже приведена функция `getPlayerMove()`

из игры «Ханойские башни» (см. главу 14). Конкретные подробности работы этого кода сейчас не важны. Просто взгляните на общую структуру функции:

```
def getPlayerMove(towers):
    """Запрашивает ход у пользователя. Возвращает (fromTower, toTower)."""

    while True: # Пока пользователь не введет допустимый ход.
        print('Enter the letters of "from" and "to" towers, or QUIT.')
        print("(e.g. AB to moves a disk from tower A to tower B.)")
        print()
        response = input("> ").upper().strip()

        if response == "QUIT":
            print("Thanks for playing!")
            sys.exit()

        # Проверить, что пользователь ввел допустимые буквы:
        if response not in ("AB", "AC", "BA", "BC", "CA", "CB"):
            print("Enter one of AB, AC, BA, BC, CA, or CB.")
            continue # Снова запросить ход у пользователя.

        # Использовать более содержательные имена переменных:
        fromTower, toTower = response[0], response[1]

        if len(towers[fromTower]) == 0:
            # Башня "from" не может быть пустой:
            print("You selected a tower with no disks.")
            continue # Снова запросить ход у пользователя.
        elif len(towers[toTower]) == 0:
            # Любой диск можно переместить на пустую башню "to":
            return fromTower, toTower
        elif towers[toTower][-1] < towers[fromTower][-1]:
            print("Can't put larger disks on top of smaller ones.")
            continue # Снова запросить ход у пользователя.
        else:
            # Ход проверен, вернуть выбранные башни:
            return fromTower, toTower
```

Функция состоит из 34 строк. И хотя она решает несколько задач — ввод хода пользователем, проверка допустимости этого хода, повторный запрос хода в том случае, если ввод недопустим, — эти задачи относятся к категории получения хода пользователя. С другой стороны, если бы мы фанатично стремились к написанию коротких функций, код `getPlayerMove()` можно было бы разбить на меньшие функции:

```
def getPlayerMove(towers):
    """Запрашивает ход у пользователя. Возвращает (fromTower, toTower)."""
    while True: # Пока пользователь не введет допустимый ход.

        response = askForPlayerMove()
```

```
    terminateIfResponseIsQuit(response)
    if not isValidTowerLetters(response):
        continue # Снова запросить ход у пользователя.

    # Использовать более содержательные имена переменных:
    fromTower, toTower = response[0], response[1]

    if towerWithNoDisksSelected(towers, fromTower):
        continue # Снова запросить ход у пользователя.
    elif len(towers[toTower]) == 0:
        # Любой диск можно переместить на пустую башню "to":
        return fromTower, toTower
    elif largerDiskIsOnSmallerDisk(towers, fromTower, toTower):
        continue # Снова запросить ход у пользователя.
    else:
        # Ход проверен, вернуть выбранные башни:
        return fromTower, toTower

def askForPlayerMove():
    """Выдает запрос и возвращает выбранные башни."""
    print('Enter the letters of "from" and "to" towers, or QUIT.')
    print("(e.g. AB to moves a disk from tower A to tower B.)")
    print()
    return input("> ").upper().strip()

def terminateIfResponseIsQuit(response):
    """Завершить программу, если введен ответ 'QUIT'"""
    if response == "QUIT":
        print("Thanks for playing!")
        sys.exit()

def isValidTowerLetters(towerLetters):
    """Возвращает True, если значение `towerLetters` допустимо."""
    if towerLetters not in ("AB", "AC", "BA", "BC", "CA", "CB"):
        print("Enter one of AB, AC, BA, BC, CA, or CB.")
        return False
    return True

def towerWithNoDisksSelected(towers, selectedTower):
    """Возвращает True, если `selectedTower` не содержит дисков."""
    if len(towers[selectedTower]) == 0:
        print("You selected a tower with no disks.")
        return True
    return False

def largerDiskIsOnSmallerDisk(towers, fromTower, toTower):
    """Возвращает True, если больший диск перекладывается на меньший."""
    if towers[toTower][-1] < towers[fromTower][-1]:
        print("Can't put larger disks on top of smaller ones.")
        return True
    return False
```

Эти шесть функций занимают 56 строк, почти вдвое больше, чем у исходной версии, но делают они то же самое. Хотя каждую функцию проще понять, чем исходную функцию `getPlayerMove()`, группировка этих функций приводит к возрастанию сложности. Читателям вашего кода трудно понять, как взаимодействуют все эти функции. Функция `getPlayerMove()` — единственная, которая вызывается из других частей программы; остальные пять функций вызываются только один раз из `getPlayerMove()`. Однако большое количество функций не поясняет этот факт.

Мне также пришлось изобретать новые имена и `doc`-строки (строки в тройных кавычках под каждой командой `def`, более подробно они описаны в главе 11) для каждой новой функции. Это приводит к появлению функций с похожими именами — например, `getPlayerMove()` и `askForPlayerMove()`. Кроме того, функция `getPlayerMove()` все равно длиннее трех-четырёх строк, и если бы я следовал принципу «чем короче, тем лучше», ее пришлось бы еще «измельчить»!

В данном случае политика использования самых коротких функций привела к определению более простых функций, но общая сложность программы заметно возросла. На мой взгляд, в идеале функции должны быть короче 30 строк и ни в коем случае не длиннее 200 строк. Делайте свои функции короткими в пределах разумного.

Параметры и аргументы функций

Параметры функции представляют собой имена переменных, заключенные в круглые скобки в команде `def` этой функции, тогда как аргументами называются значения, указанные в круглых скобках при вызове функции. Чем больше параметров у функции, тем больше возможностей для настройки и обобщения ее кода. Но увеличение количества параметров также означает повышение сложности.

Предложу хорошее правило: от нуля до трех параметров — это нормально, но больше пяти или шести, пожалуй, слишком много. Если функции становятся очень сложными, лучше рассмотреть возможность их разбиения на меньшие функции с меньшим количеством параметров.

Аргументы по умолчанию

Один из способов сокращения сложности параметров функций заключается в определении аргументов по умолчанию для параметров. *Аргумент по умолчанию* представляет собой значение, которое будет использоваться как аргумент, если он не указан явно при вызове функции. Если при большинстве вызовов функций используется конкретное значение параметра, то это значение можно сделать аргументом по умолчанию, чтобы его не приходилось многократно указывать при вызове. Аргумент по умолчанию задается в команде `def`, за ним следует имя параметра и знак

равенства. Например, в функции `introduction()` параметру с именем `greeting` присваивается значение `'Hello'`, если оно не задается при вызове функции:

```
>>> def introduction(name, greeting='Hello'):
...     print(greeting + ', ' + name)
...
>>> introduction('Alice')
Hello, Alice
>>> introduction('Hiro', 'Ohiyo gozaimasu')
Ohiyo gozaimasu, Hiro
```

Когда функция `introduction()` вызывается без второго аргумента, по умолчанию используется строка `'Hello'`. Учтите, что параметры с аргументами по умолчанию всегда должны следовать после параметров без аргументов по умолчанию.

Вспомните, о чем мы говорили в главе 8: изменяемые объекты (такие как пустой список `[]` или пустой словарь `{}`) никогда не следует использовать в качестве значения по умолчанию. В подразделе «Не используйте изменяемые значения для аргументов по умолчанию», с. 174, объясняется проблема, которая может при этом возникнуть, и способы ее решения.

Использование `*` и `**` для передачи аргументов функции

Вы можете использовать синтаксисы `*` и `**` для отдельной передачи групп аргументов функциям. Синтаксис `*` позволяет передать элементы итерируемого объекта (такого как список или кортеж). Синтаксис `**` позволяет передавать пары «ключ — значение» из объекта отображения (например, словаря) как отдельные аргументы.

Например, функция `print()` может получать несколько аргументов. По умолчанию при выводе аргументы разделяются пробелами:

```
>>> print('cat', 'dog', 'moose')
cat dog moose
```

Эти аргументы называются *позиционными* (positional arguments), потому что их позиция в вызове функции определяет, какой аргумент должен быть присвоен тому или иному параметру. Но если сохранить эти строки в списке и попытаться передать список, функция `print()` решит, что вы хотите передать список как одно значение:

```
>>> args = ['cat', 'dog', 'moose']
>>> print(args)
['cat', 'dog', 'moose']
```

При передаче списка функция `print()` выводит список, включая квадратные скобки, кавычки и запятые.

В одном из способов вывода отдельных элементов при вызове указывается индекс каждого элемента. Такой код хуже читается:

```
>>> # Пример плохо читаемого кода:
>>> args = ['cat', 'dog', 'moose']
>>> print(args[0], args[1], args[2])
cat dog moose
```

Но существует более простой способ передачи значений `print()`. Синтаксис `*` можно использовать для интерпретации элементов списка (или любого другого итерируемого типа данных) как отдельных позиционных аргументов. Введите следующий пример в интерактивной оболочке:

```
>>> args = ['cat', 'dog', 'moose']
>>> print(*args)
cat dog moose
```

Синтаксис `*` позволяет передать функции элементы списка по отдельности, сколько бы элементов ни содержал список.

Синтаксис `**` используется для передачи типов данных отображений (например, словарей) как набора ключевых аргументов. При передаче ключевого аргумента указывается имя параметра, за которым следует знак равенства. Например, у функции `print()` имеется ключевой аргумент `sep`, который определяет строку, разделяющую аргументы при выводе. По умолчанию используется строка из одного пробела ' '. Ключевому аргументу можно присвоить другое значение командой присваивания или синтаксисом `**`. Чтобы понять, как работает этот синтаксис, введите следующий фрагмент в интерактивной оболочке:

```
>>> print('cat', 'dog', 'moose', sep='- ')
cat-dog-moose
>>> kwargsForPrint = {'sep': '- '}
>>> print('cat', 'dog', 'moose', **kwargsForPrint)
cat-dog-moose
```

Обратите внимание: эти инструкции генерируют одинаковый вывод. В примере для создания словаря `kwargsForPrint` используется только одна строка кода. Но в более сложных сценариях создание словаря ключевых аргументов потребует большего объема кода. Синтаксис `**` позволяет создать специализированный словарь с настройками конфигурации, которые должны передаваться при вызове функции. Данная возможность особенно полезна для функций и методов с большим количеством ключевых аргументов.

Изменяя список или словарь во время выполнения, можно передавать при вызове функции разное количество аргументов с использованием синтаксисов `*` и `**`.

Использование * при создании вариадических функций

Синтаксис `*` также может использоваться в командах `def` для создания *вариадических* (variadic, или varargs) функций, получающих переменное количество позиционных аргументов. Например, функция `print()` является вариадической, потому что ей можно передать любое количество строк, например `print('Hello!')` или `print('My name is', name)`. Обратите внимание: если в предыдущем разделе синтаксис `*` использовался при вызове функций, здесь мы его используем в определениях функций.

Рассмотрим пример: функция `product()` получает произвольное количество аргументов и перемножает их:

```
>>> def product(*args):
...     result = 1
...     for num in args:
...         result *= num
...     return result
...
>>> product(3, 3)
9
>>> product(2, 1, 2, 3)
12
```

Внутри функции `args` — обычный кортеж Python, содержащий все позиционные аргументы. С технической точки зрения можно присвоить параметру любое имя, при условии, что оно начинается со звездочки (`*`), но среди программистов принято использовать имя `args`.

Вопрос о том, в каких случаях стоит использовать `*`, не так прост. Ведь альтернативой вариадической функции становится создание одного параметра, в который передается список (или другой итерируемый тип данных) с переменным количеством элементов. Этот способ используется встроенной функцией `sum()`:

```
>>> sum([2, 1, 2, 3])
8
```

Функция `sum()` ожидает получить один итерируемый аргумент, и при попытке передать несколько аргументов выдается исключение:

```
>>> sum(2, 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() takes at most 2 arguments (4 given)
```

С другой стороны, встроенные функции `min()` и `max()`, определяющие минимум и максимум среди нескольких значений, получают один итерируемый аргумент или несколько разных аргументов:

```
>>> min([2, 1, 3, 5, 8])
1
>>> min(2, 1, 3, 5, 8)
1
>>> max([2, 1, 3, 5, 8])
8
>>> max(2, 1, 3, 5, 8)
8
```

Все эти функции получают переменное количество аргументов, так почему же их параметры устроены по-разному? И когда функция должна получать один итерируемый аргумент, а когда — несколько разных аргументов с использованием синтаксиса `*`?

Вопрос о структуре параметров зависит от ваших намерений относительно того, как будет использоваться ваш код. Функция `print()` получает несколько аргументов, потому что программисты часто передают несколько строк или переменных, содержащих строки, — например, `print('My name is', name)`. Вариант с объединением этих строк в список и последующей передачей списка `print()` встречается реже. Кроме того, если передать `print()` список, функция выведет его полностью; она не может использоваться для вывода отдельных значений из списка.

Вызов `sum()` с разными аргументами не имеет смысла, потому что в Python для этого уже есть оператор `+`. Так как вы можете написать код вида `2 + 4 + 8`, вам не понадобится код `sum(2, 4, 8)`. Логично, что переменное число аргументов в `sum()` может передаваться только в виде списка.

Функции `min()` и `max()` поддерживают оба стиля. Если программист передает один аргумент, то функция предполагает, что это список или кортеж с проверяемыми значениями. Эти две функции обычно обрабатывают списки значений во время выполнения программы, как при вызове функции `min(allExpenses)`. Им также приходится иметь дело с наборами отдельных аргументов, выбранных программистом во время написания кода — например, `max(0, someNumber)`. Из-за этого функции написаны так, чтобы они поддерживали обе разновидности аргументов. Следующая функция `myMinFunction()` — моя собственная реализация функции `min()` — демонстрирует обе возможности:

```
def myMinFunction(*args):
    if len(args) == 1:
        values = args[0] ❶
    else:
        values = args ❷

    if len(values) == 0:
        raise ValueError('myMinFunction() args is an empty sequence') ❸

    for i, value in enumerate(values): ❹
```

```

if i == 0 or value < smallestValue:
    smallestValue = value
return smallestValue

```

`myMinFunction()` использует синтаксис `*` для получения переменного количества аргументов в виде кортежа. Если кортеж содержит только одно значение, предполагается, что это последовательность проверяемых значений ❶. В противном случае предполагается, что `args` содержит кортеж проверяемых значений ❷. В любом случае переменная `values` будет содержать последовательность значений, которые проверяются в остальной части кода. Как и реальная функция `min()`, функция выдает исключение `ValueError`, если вызывающая сторона не передала никаких аргументов или передала пустую последовательность ❸. Остальная часть кода перебирает значения и возвращает наименьшее из найденных значений ❹. Чтобы не усложнять пример, `myMinFunction()` принимает только такие последовательности, как списки или кортежи (вместо произвольного итерируемого объекта).

Почему же мы не всегда пишем функции так, чтобы они поддерживали оба способа передачи переменного количества аргументов? Потому что лучше делать ваши функции как можно проще. Если только оба способа вызова не используются одинаково часто, выберите один из них. Когда функция имеет дело со структурой данных, создаваемой во время работы программы, то лучше, чтобы она принимала один параметр. А когда — с аргументами, которые задаются программистом во время написания кода, стоит использовать синтаксис `*` для получения переменного количества аргументов.

Использование ****** при создании вариадических функций

Вариадические функции могут также использовать синтаксис `**`. И если синтаксис `*` в командах `def` представляет переменное количество позиционных аргументов, синтаксис `**` представляет переменное количество необязательных ключевых аргументов.

Когда вы определите функцию, которая получает числовые необязательные ключевые аргументы без синтаксиса `**`, ваша команда `def` быстро усложнится. Возьмем гипотетическую функцию `formMolecule()`, которая получает параметры для всех 118 известных химических элементов:

```
>>> def formMolecule(hydrogen, helium, lithium, beryllium, boron, --snip--
```

Передавать значение 2 для параметра `hydrogen` и 1 для параметра `oxygen` (молекула воды состоит из 2 атомов водорода и 1 атома кислорода) было бы громоздко и неудобно, потому что для всех остальных элементов пришлось бы передавать нули:

```
>>> formMolecule(2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 --snip--
'water'
```

Чтобы функция была более удобной, можно воспользоваться именованными ключевыми параметрами, каждому из которых назначен аргумент по умолчанию, что избавляет вас от необходимости передавать аргумент для этого параметра при вызове функции.

ПРИМЕЧАНИЕ

Хотя термины «аргумент» и «параметр» четко определены, программисты часто используют термины «ключевой аргумент» и «ключевой параметр» как синонимы.

Например, следующая команда `def` назначает аргументы по умолчанию 0 для каждого из ключевых параметров:

```
>>> def formMolecule(hydrogen=0, helium=0, lithium=0, beryllium=0, --snip--
```

Это упрощает вызов `formMolecule()`, потому что вам достаточно лишь задать аргументы для параметров со значениями, отличными от аргументов по умолчанию. Кроме того, ключевые аргументы можно задавать в любом порядке:

```
>>> formMolecule(hydrogen=2, oxygen=1)
'water'
>>> formMolecule(oxygen=1, hydrogen=2)
'water'
>>> formMolecule(carbon=8, hydrogen=10, nitrogen=4, oxygen=2)
'caffeine'
```

Однако остается громоздкая команда `def` с 118 именами параметров. А если вдруг будут открыты новые химические элементы? Придется обновлять команду `def` функции со всей документацией, относящейся к параметрам функции.

Вместо этого можно объединить все параметры с их аргументами в виде пар «ключ — значение» в словаре с использованием синтаксиса `**` для ключевых аргументов. С технической точки зрения можно присвоить параметру `**` любое имя, но у программистов принято использовать имя `kwargs`:

```
>>> def formMolecules(**kwargs):
...     if len(kwargs) == 2 and kwargs['hydrogen'] == 2 and
...                               kwargs['oxygen'] == 1:
...         return 'water'
...     # (... остальной код функции ...)
...
>>> formMolecules(hydrogen=2, oxygen=1)
'water'
```

Синтаксис `**` показывает, что параметр `kwargs` может использоваться для всех аргументов «ключ — значение», переданных при вызове функции. Они будут храниться в виде пар «ключ — значение» в словаре, присвоенном параметру `kwargs`.

При обнаружении новых химических элементов достаточно обновить код функции, но не ее команду `def`, потому что все ключевые аргументы помещены в `kwargs`:

```
>>> def formMolecules(**kwargs):           ❶
...     if len(kwargs) == 1 and kwargs.get('unobtainium') == 12:  ❷
...         return 'aether'
...     # (... остальной код функции ...)
...
>>> formMolecules(unobtainium=12)
'aether'
```

Как видите, команда `def` ❶ осталась неизменной и в обновлении нуждается только код функции ❷. При использовании синтаксиса `**` команда `def` и вызовы функций значительно упрощаются, а код все еще нормально читается.

Использование `*` и `**` для создания функций-оберток

Синтаксисы `*` и `**` в командах `def` часто используются для создания функций-оберток, которые передают аргументы другой функции и возвращают возвращаемое значение этой функции. Вы можете использовать синтаксисы `*` и `**` для передачи любых аргументов функции, скрытой за оберткой. Например, можно создать функцию `printLowercase()`, которая является оберткой для встроенной функции `print()`. Вся реальная работа выполняется функцией `print()`, а обертка сначала переводит строковые аргументы в нижний регистр:

```
>>> def printLower(*args, **kwargs):       ❶
...     args = list(args)                  ❷
...     for i, value in enumerate(args):
...         args[i] = str(value).lower()
...     return print(*args, **kwargs)      ❸
...
>>> name = 'Albert'
>>> printLower('Hello, ', name)
hello, albert
>>> printLower('DOG', 'CAT', 'MOOSE', sep=', ')
dog, cat, moose
```

Функция `printLower()` ❶ использует синтаксис `*` для получения переменного количества позиционных аргументов в кортеже, присвоенном параметру `args`, тогда как синтаксис `**` присваивает все ключевые аргументы словарю из параметра `kwargs`. Если функция использует `*args` вместе с `**kwargs`, параметр `*args` должен предшествовать параметру `**kwargs`. Они передаются функции `print()`, заключенной в обертку, но сначала наша функция изменяет некоторые аргументы, поэтому мы создаем списковую форму кортежа `args` ❷.

После преобразования строк в `args` к нижнему регистру элементы `args` и пары «ключ — значение» в `kwargs` передаются как отдельные аргументы функции `print()`

с использованием синтаксисов `*` и `**` ❸. Возвращаемое значение `print()` также возвращается как возвращаемое значение `printLower()`. Таким образом создается обертка для функции `print()`.

Функциональное программирование

Функциональное программирование — парадигма программирования, уделяющая особое внимание написанию функций, которые выполняют вычисления без изменения глобальных переменных или какого-либо внешнего состояния (файлов на жестком диске, подключений к интернету или баз данных). Некоторые языки программирования — такие как Erlang, Lisp и Haskell — в значительной мере проектировались на основе концепций функционального программирования. И хотя язык Python не прикован намертво к этой парадигме, в нем реализованы некоторые средства функционального программирования. Главные из них, которые могут использоваться в программах Python, — функции, свободные от побочных эффектов, функции высшего порядка и лямбда-функции.

Побочные эффекты

К побочным эффектам относятся любые изменения, вносимые функцией в части программы, существующие за рамками ее собственного кода и локальных переменных. Для демонстрации создадим функцию `subtract()`, которая реализует оператор вычитания Python (`-`):

```
>>> def subtract(number1, number2):  
...     return number1 - number2  
...  
>>> subtract(123, 987)  
-864
```

Эта функция `subtract()` не имеет побочных эффектов. Другими словами, она не влияет в программе ни на что за пределами ее кода. По состоянию программы или компьютера невозможно определить, была ли функция `subtract()` ранее вызвана один раз, два раза или миллион раз. Функция может изменять локальные переменные внутри функции, но эти изменения остаются изолированными от остального кода программы.

Теперь рассмотрим функцию `addToTotal()`, которая прибавляет числовой аргумент к глобальной переменной с именем `TOTAL`:

```
>>> TOTAL = 0  
>>> def addToTotal(amount):  
...     global TOTAL
```

```
...     TOTAL += amount
...     return TOTAL
...
>>> addToTotal(10)
10
>>> addToTotal(10)
20
>>> addToTotal(9999)
10019
>>> TOTAL
10019
```

Функция `addToTotal()` имеет побочный эффект, потому что она изменяет элемент, существующий за пределами функции, — глобальную переменную `TOTAL`. Побочные эффекты не ограничиваются изменением глобальных переменных. К ним относится обновление или удаление файлов, вывод текста на экран, подключение к базе данных, аутентификация на сервере или внесение любых других изменений за пределами функции. Любой след, оставленный вызовом функции после возврата управления, является побочным эффектом.

К побочным эффектам также можно отнести модификацию на месте изменяемых объектов, ссылки на которые существуют за пределами функции. Например, следующая функция `removeLastCatFromList()` изменяет последний аргумент на месте:

```
>>> def removeLastCatFromList(petSpecies):
...     if len(petSpecies) > 0 and petSpecies[-1] == 'cat':
...         petSpecies.pop()
...
>>> myPets = ['dog', 'cat', 'bird', 'cat']
>>> removeLastCatFromList(myPets)
>>> myPets
['dog', 'cat', 'bird']
```

В этом примере переменная `myPets` и параметр `petSpecies` содержат ссылки на один и тот же список. Любые изменения на месте, вносимые в объект списка внутри функции, также будут существовать за пределами функции, вследствие чего изменение становится побочным эффектом.

С концепцией побочных эффектов связана концепция *детерминированной функции* (deterministic function), которая для одного набора аргументов всегда возвращает одно и то же значение. Вызов функции `subtract(123, 987)` всегда возвращает `-864`. Встроенная функция Python `round()` всегда возвращает 3, если передать ей аргумент 3.14. *Недетерминированная функция* (nondeterministic function) не всегда возвращает одно и то же значение при передаче одного набора аргументов. Например, вызов `random.randint(1, 10)` возвращает случайное целое число от 1 до 10. Функция `time.time()` не получает аргументов, но возвращает разные значения

в зависимости от показаний компьютерных часов на момент вызова. В случае `time.time()` часы являются внешним ресурсом, который фактически предоставляет входные данные функции, так же как это делает аргумент. Функции, зависящие от ресурсов, внешних по отношению к функции (включая глобальные переменные, файлы на жестком диске, базы данных и подключения к интернету), не считаются детерминированными.

Одно из преимуществ детерминированных функций — возможность кэширования их значений. Нет необходимости вызывать `subtract()` для вычисления разности 123 и 987 более одного раза, если функция может запомнить возвращаемое значение при первом вызове с этими аргументами. Таким образом, детерминированные функции позволяют выбрать компромисс между затратами памяти и затратами времени, ускоряя выполнение функции за счет использования памяти для кэширования предыдущих результатов.

Детерминированная функция, свободная от побочных эффектов, называется *чистой функцией*. Функциональные программисты стараются создавать в своих программах только чистые функции. В дополнение к уже упоминавшимся чистые функции имеют следующие преимущества.

- Они хорошо подходят для модульного тестирования, потому что не требуют подготовки внешних ресурсов.
- В чистых функциях проще воспроизводятся ошибки, для чего достаточно вызвать функцию с теми же аргументами.
- Чистые функции могут вызывать другие чистые функции, оставаясь чистыми.
- В многопоточных программах чистые функции являются потоково-безопасными и могут выполняться параллельно. (Тема многопоточности выходит за рамки книги.)
- Множественные вызовы чистых функций способны выполняться на параллельных ядрах процессора или в многопоточной программе, потому что они не зависят от внешних ресурсов, требующих их выполнения в определенной последовательности.

На языке Python можно и нужно писать чистые функции там, где это возможно. Функции Python делаются чистыми только по соглашению; нет никаких настроек, которые бы заставляли интерпретатор Python обеспечивать чистоту функций. Самый распространенный способ сделать ваши функции чистыми — избегать использования глобальных переменных и следить за тем, чтобы они не взаимодействовали с файлами, интернетом, системными часами, генератором случайных чисел или другими внешними ресурсами.

Функции высшего порядка

Функции высшего порядка (higher-order functions) могут получать другие функции в аргументах или использовать функции как возвращаемые значения. Например, определим функцию с именем `callItTwice()`, которая вызывает заданную функцию дважды:

```
>>> def callItTwice(func, *args, **kwargs):
...     func(*args, **kwargs)
...     func(*args, **kwargs)
...
>>> callItTwice(print, 'Hello, world!')
Hello, world!
Hello, world!
```

Функция `callItTwice()` работает с любой передаваемой функцией. В Python функции являются *первоклассными объектами* (first-class objects); это означает, что они ничем не отличаются от других объектов: функции можно сохранять в переменных, передавать в аргументах или использовать как возвращаемые значения.

Лямбда-функции

Лямбда-функции (lambda functions), также называемые *анонимными* (anonymous) или *безымянными* (nameless) функциями, представляют собой упрощенные функции, у которых нет имен, а код состоит из одной команды `return`. Лямбда-функции часто используются для передачи функций как аргументов других функций.

Например, можно создать обычную функцию, которая получает список с шириной и высотой прямоугольника 4×10 :

```
>>> def rectanglePerimeter(rect):
...     return (rect[0] * 2) + (rect[1] * 2)
...
>>> myRectangle = [4, 10]
>>> rectanglePerimeter(myRectangle)
28
```

Эквивалентная лямбда-функция выглядит так:

```
lambda rect: (rect[0] * 2) + (rect[1] * 2)
```

Чтобы определить лямбда-функцию на Python, укажите ключевое слово `lambda`, за которым следуют: список параметров (если они есть), разделенных запятыми, двоеточие и выражение, которое действует как возвращаемое значение. Так как функции являются первоклассными объектами, лямбда-функцию можно присвоить переменной, фактически повторяя то, что делает команда `def`:

```
>>> rectanglePerimeter = lambda rect: (rect[0] * 2) + (rect[1] * 2)
>>> rectanglePerimeter([4, 10])
28
```

Лямбда-функция присваивается переменной с именем `rectanglePerimeter`, тем самым фактически создается функция `rectanglePerimeter()`. Как видите, функции, созданные командами `lambda`, ничем не отличаются от функций, созданных командами `def`.

ПРИМЕЧАНИЕ

В реальном коде лучше использовать команды `def`, вместо того чтобы присваивать лямбда-функции неизменяемым переменным. Лямбда-функции специально создавались для ситуаций, в которых функции не нуждаются в имени.

Синтаксис лямбда-функций хорошо подходит для определения небольших функций, которые служат аргументами для вызова других функций. Например, у функции `sorted()` есть ключевой аргумент `key`, который позволяет задать функцию. Вместо того чтобы сортировать элементы списка на основании значения элементов, она сортирует в зависимости от возвращаемого значения функции. В следующем примере `sorted()` передается лямбда-функция, которая возвращает периметр заданного прямоугольника. В результате функция `sorted()` сортирует по вычисленному периметру из списка `[width, height]`, а не непосредственно по списку `[width, height]`:

```
>>> rects = [[10, 2], [3, 6], [2, 4], [3, 9], [10, 7], [9, 9]]
>>> sorted(rects, key=lambda rect: (rect[0] * 2) + (rect[1] * 2))
[[2, 4], [3, 6], [10, 2], [3, 9], [10, 7], [9, 9]]
```

Вместо того чтобы сортировать, например, значения `[10, 2]` или `[3, 6]`, функция теперь выполняет сортировку на основании возвращаемых значений периметров 24 и 18. Лямбда-функции являются удобным синтаксическим сокращением: вы можете задать одну маленькую лямбда-функцию вместо определения новой именованной функции командой `def`.

Отображение и фильтрация со спискавыми включениями

В более ранних версиях Python функции `map()` и `filter()` были обычными функциями высшего порядка, которые могли преобразовывать и фильтровать списки, часто при помощи лямбда-функций. Отображение способно строить списки значений на основании значений из другого списка. Фильтрация позволяет создать список, который содержит только те значения из другого списка, которые соответствуют некоторому критерию.

Например, если вы хотите создать новый список, содержащий строки вместо целых чисел [8, 16, 18, 19, 12, 1, 6, 7], можно передать функции `map()` этот список и лямбда-функцию `lambda n: str(n)`:

```
>>> mapObj = map(lambda n: str(n), [8, 16, 18, 19, 12, 1, 6, 7])
>>> list(mapObj)
['8', '16', '18', '19', '12', '1', '6', '7']
```

Функция `map()` возвращает объект `map`, который можно получить в форме списка, для чего он передается функции `list()`. Отображенный список теперь содержит строковые значения на основании целых значений из исходного списка. Функция `filter()` работает аналогично, но в этом случае аргумент лямбда-функции определяет, какие элементы должны остаться в списке (если лямбда-функция возвращает `True`) или быть отфильтрованными (если она возвращает `False`). Например, передача функции `lambda n: n % 2 == 0` позволяет отфильтровать все нечетные числа:

```
>>> filterObj = filter(lambda n: n % 2 == 0, [8, 16, 18, 19, 12, 1, 6, 7])
>>> list(filterObj)
[8, 16, 18, 12, 6]
```

Функция `filter()` возвращает объект-фильтр, который можно снова передать функции `list()`. В отфильтрованном списке остаются только четные числа.

Однако создание отображенных или отфильтрованных списков функциями `map()` и `filter()` в Python считается устаревшим. Вместо этого рекомендуется создавать их при помощи списковых включений. Списковые включения не только освобождают вас от необходимости писать лямбда-функции, но и работают быстрее функций `map()` и `filter()`.

Следующий код воспроизводит пример с функцией `map()` с использованием спискового включения:

```
>>> [str(n) for n in [8, 16, 18, 19, 12, 1, 6, 7]]
['8', '16', '18', '19', '12', '1', '6', '7']
```

Обратите внимание: часть спискового включения `str(n)` похожа на `lambda n: str(n)`.

А следующий фрагмент воспроизводит пример с функцией `filter()` с использованием спискового включения:

```
>>> [n for n in [8, 16, 18, 19, 12, 1, 6, 7] if n % 2 == 0]
[8, 16, 18, 12, 6]
```

Обратите внимание: часть спискового включения `if n % 2 == 0` похожа на `lambda n: n % 2 == 0`.

Во многих языках существует концепция функций как первоклассных объектов, что делает возможным существование функций высшего порядка, включая функции отображения и фильтрации.

Возвращаемые значения всегда должны иметь один тип данных

Python является языком с динамической типизацией; это означает, что функции и методы Python способны возвращать значения любого типа данных. Но чтобы ваши функции были более предсказуемыми, вы должны стремиться к тому, чтобы они возвращали значения только одного типа данных.

Например, следующая функция в зависимости от случайного числа возвращает целое число или строковое значение:

```
>>> import random
>>> def returnsTwoTypes():
...     if random.randint(1, 2) == 1:
...         return 42
...     else:
...         return 'forty two'
```

Когда вы пишете код с вызовом этой функции, легко забыть, что вам нужно обрабатывать разные типы данных. Продолжим этот пример: допустим, что вы вызвали `returnsTwoTypes()` и хотите преобразовать возвращенное число в шестнадцатеричную форму:

```
>>> hexNum = hex(returnsTwoTypes())
>>> hexNum
'0x2a'
```

Встроенная функция Python `hex()` возвращает строку с шестнадцатеричным представлением переданного ей числа. Этот код работает при условии, что `returnsTwoTypes()` вернет целое число; возникает впечатление, что этот код свободен от ошибок. Но когда `returnsTwoTypes()` возвращает строку, выдается исключение:

```
>>> hexNum = hex(returnsTwoTypes())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer
(Объект 'str' не может быть интерпретирован как integer.)
```

Конечно, вы постоянно должны помнить о необходимости обрабатывать все возможные типы данных возвращаемого значения. Но в реальности об этом легко

забыть. Чтобы предотвратить такие ошибки, всегда стремитесь к тому, чтобы ваши функции возвращали значения только одного типа данных. Это не является жестким требованием, и иногда просто невозможно предотвратить возвращение функцией значений разных типов данных. Но чем ближе вы подходите к возвращению только одного типа, тем проще и надежнее будут ваши функции.

Есть один случай, на который необходимо обратить особое внимание: не возвращайте `None` из функций (единственное исключение — если ваша функция всегда возвращает `None`). Значение `None` — единственное значение типа данных `NoneType`. Появляется искушение написать функцию, которая возвращает `None`, сообщая тем самым о возникшей ошибке (эта практика рассматривается в следующем разделе «Выдача исключений и возвращение кодов ошибок»), но возвращение `None` лучше зарезервировать для функций, у которых возвращаемое значение не имеет смысла.

Дело в том, что возвращение `None` как признака ошибки становится распространенным источником перепрежваченных исключений `'NoneType' object has no attribute` (Объект `'NoneType'` не имеет атрибута...):

```
>>> import random
>>> def sometimesReturnsNone():
...     if random.randint(1, 2) == 1:
...         return 'Hello!'
...     else:
...         return None
...
>>> returnVal = sometimesReturnsNone()
>>> returnVal.upper()
'HELLO!'
>>> returnVal = sometimesReturnsNone()
>>> returnVal.upper()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'upper'
```

Сообщение об ошибке выглядит довольно туманно. Вероятно, вы не сразу отследите его происхождение до функции, которая обычно возвращает ожидаемый результат, но также может вернуть `None` при возникновении ошибки. Проблема возникла из-за того, что `sometimesReturnsNone()` возвращает значение `None`, которое затем присваивается переменной `returnVal`. Но сообщение об ошибке заставляет думать, что проблема возникла при вызове метода `upper()`.

В своем докладе на конференции в 2009 году компьютерный теоретик Тони Хоар (Tony Hoare) извинился за то, что он изобрел `null`-ссылку (общий аналог значения `None` в Python) в 1965 году. Он сказал: «Я называю это своей ошибкой на миллиард долларов. <...> Я не устоял перед искушением включить `null`-ссылки просто

потому, что их было так легко реализовать. Это привело к неисчислимым ошибкам, уязвимостям и системным сбоям, которые за последние 40 лет, вероятно, создали проблемы и неприятности на миллиард долларов». Полное выступление можно посмотреть на <https://austbor.com/billiondollarmistake>.

Выдача исключений и возвращение кодов ошибок

В Python термины «исключение» (exception) и «ошибка» (error) имеют приблизительно одинаковый смысл: аномальная ситуация в программе, которая обычно указывает на возникшую проблему. Исключения стали популярным языковым механизмом в 1980-е и 1990-е годы в C++ и Java. Ими заменили *коды ошибок* (error codes) — значения, возвращаемые функциями для обозначения проблемы. Преимущество исключений состоит в том, что возвращаемые значения связаны только с предназначением функции и не указывают на присутствие ошибки.

Коды ошибок также способны создавать проблемы в ваших программах. Например, метод строк Python `find()` обычно возвращает индекс, по которому была найдена подстрока, а если найти подстроку не удалось, возвращается код ошибки `-1`. Но поскольку индекс `-1` также может использоваться для отсчета индекса от конца строки, случайное использование `-1` в качестве кода ошибки создаст ошибку. Чтобы понять, как это происходит, введите следующий фрагмент в интерактивной оболочке:

```
>>> print('Letters after b in "Albert":', 'Albert'['Albert'.find('b') + 1:])
Letters after b in "Albert": ert
>>> print('Letters after x in "Albert":', 'Albert'['Albert'.find('x') + 1:])
Letters after x in "Albert": Albert
```

Часть кода `'Albert'.find('x')` при вычислении возвращает код ошибки `-1`. В результате выражение `'Albert'['Albert'.find('x') + 1:]` преобразуется в `'Albert'[-1 + 1:]`, что далее дает результат `'Albert'[0:]` и, наконец, `'Albert'`.

Очевидно, это не то поведение, которое ожидалось от кода. При вызове `index()` вместо `find()`, как в `'Albert'['Albert'.index('x') + 1:]`, возникло бы исключение. Проблема становится очевидной, и проигнорировать ее не удастся.

С другой стороны, метод строк `index()` выдает исключение `ValueError`, если он не может найти подстроку. Если исключение не будет обработано, оно приведет к аварийному завершению программы — обычно это лучше, чем ошибка, которая осталась незамеченной.

Имена классов исключений часто завершаются словом `Error`, когда исключение указывает на фактическую ошибку — такую как `ValueError`, `NameError` или `SyntaxError`.

К категории классов исключений, представляющих аномальные ситуации, которые не обязательно являются ошибками, относятся `StopIteration`, `KeyboardInterrupt` и `SystemExit`.

Итоги

Функции предоставляют популярный механизм группировки кода наших программ; при их написании необходимо принимать ряд решений: какое имя им присвоить, какой размер они должны иметь, сколько у них должно быть параметров и сколько аргументов должно передаваться для этих параметров. Синтаксисы `*` и `**` в командах `def` позволяют функциям получать переменное количество параметров; такие функции называются вариадическими.

Хотя Python не является языком функционального программирования, в нем реализованы многие возможности, используемые в языках функционального программирования. Функции являются первоклассными объектами; это означает, что их можно сохранять в переменных и передавать как аргументы других функций (которые в этом случае называются функциями высшего порядка). Лямбда-функции предоставляют короткий синтаксис для определения анонимных функций как аргументов функций высшего порядка. Самые распространенные функции высшего порядка в Python — `map()` и `filter()`, хотя предоставляемая ими функциональность быстрее реализуется списковыми включениями.

Возвращаемые значения функций всегда должны иметь постоянный тип данных. Возвращаемые значения не должны использоваться как коды ошибок; для этой цели следует использовать исключения. В частности, значение `None` часто определяется как код ошибки.

11

Комментарии, дос-строки и аннотации типов



Комментарии и документация в исходном коде не менее важны, чем сам код. Причина в том, что программный продукт никогда не бывает полностью готовым; всегда приходится вносить в него изменения — как при добавлении новых возможностей, так и при исправлении ошибок. Как однажды заметили специалисты по теории вычислений Гарольд Абельсон (Harold Abelson), Джеральд Джей Зюссман (Gerald Jay Sussman) и Джулия Зюссман (Julie Sussman), «программы пишутся для того, чтобы их читали люди, и лишь изредка для того, чтобы они выполнялись машинами».

Комментарии, дос-строки и аннотации типов помогают поддерживать код в работоспособном состоянии. *Комментарии* представляют собой короткие объяснения на естественном языке, которые записываются прямо в исходном коде; компьютер их игнорирует. Комментарии содержат полезные заметки, предупреждения и напоминания для сторонних читателей кода, а иногда и для самого автора кода в будущем. Почти каждому программисту доводилось задавать себе вопрос: «Кто написал это нечитаемое месиво?», только чтобы вспомнить ответ: «Это я».

Дос-строки представляют собой форму документирования функций, методов и модулей, специфическую для Python. Когда вы задаете комментарии в формате дос-строки, автоматизированные средства (такие как генераторы документации или встроенный модуль Python `help()`) позволяют разработчикам легко найти информацию о вашем коде.

Аннотации типов (type hints) представляют собой директивы, которые можно добавить в исходный код Python для указания типов данных переменных, параметров и возвращаемых значений. Это позволяет средствам статического анализа кода

убедиться в том, что ваш код не сгенерирует исключений, обусловленных неправильными типами значений. Аннотации типов впервые появились в Python 3.5, но так как они создаются на основе комментариев, их можно использовать в любой версии Python.

Итак, в этой главе я расскажу о трех упомянутых способах встраивания документации в код с целью улучшения его читабельности. Внешняя документация — руководства пользователя, электронные учебники и справочные материалы — важна, но в этой книге она не рассматривается. Если вы захотите узнать больше о внешней документации, поищите информацию о генераторе документации Sphinx (<https://www.sphinx-doc.org/>).

Комментарии

Как и большинство языков программирования, Python поддерживает однострочные и многострочные комментарии. Любой текст, заключенный между знаком # и концом строки, является однострочным комментарием. Хотя в Python нет специального синтаксиса для многострочных комментариев, в этом качестве можно использовать многострочный текст в тройных кавычках. В конце концов, строковое значение само по себе еще не заставляет интерпретатор Python что-либо сделать. Рассмотрим пример:

```
# Это однострочный комментарий.  
"""А это  
многострочный текст, который  
также работает как многострочный комментарий. """
```

Если комментарий занимает несколько строк, лучше использовать один многострочный блок, чем несколько последовательных однострочных комментариев. Второй вариант хуже читается, как видно из следующего примера:

```
"""Хороший способ  
записи комментариев,  
занимающих несколько строк. """  
# А это плохой способ  
# записи комментариев,  
# занимающих несколько строк.
```

Комментарии и документацию программисты зачастую включают в код задним числом, а некоторые даже считают, что от них больше вреда, чем пользы. Но как я уже объяснял в подразделе «Миф: комментарии излишни» на с. 111, комментарии абсолютно обязательны, если вы хотите писать профессиональный, удобочитаемый код. В этом разделе вы научитесь писать полезные комментарии, которые предоставляют дельную информацию читателю без ущерба для удобочитаемости программы.

Стиль комментариев

Примеры комментариев, отвечающих правилам хорошего стиля:

```
# Комментарий, относящийся к следующему коду: ❶
someCode()
# Более длинный комментарий, который занимает несколько строк ❷
# из нескольких последовательных однострочных комментариев.
# ❸
# Такие комментарии называются блоковыми.

if someCondition:
    # Комментарий о другом коде: ❹
    someOtherCode() # Встроенный комментарий. ❺
```

Как правило, комментарии лучше размещать в отдельной строке, а не в конце строки с кодом. В большинстве случаев лучше использовать полноценные предложения с соответствующим регистром символов и знаками препинания, а не короткие фразы или отдельные слова ❶. При этом комментарии должны подчиняться тем же ограничениям длины строки, что и исходный код. Комментарии, занимающие несколько строк ❷, могут состоять из нескольких последовательных однострочных комментариев (такие комментарии называются блоковыми). Абзацы в комментариях разделяются пустым однострочным комментарием ❸. Уровень отступа комментария должен соответствовать уровню отступов в комментируемом коде ❹. Комментарии, следующие за строкой кода, называются *встроенными* (inline) ❺, и код отделяется от комментария минимум двумя пробелами.

В однострочных комментариях после знака # ставят пробел:

```
#Комментарий не должен начинаться сразу же после знака #.
```

Комментарии могут включать ссылки на URL с сопутствующей информацией, но ссылки не должны заменять комментарии, потому что связанный с ними контент может исчезнуть из интернета в любой момент:

```
# Подробное объяснение некоторого аспекта кода с дополнительной
# информацией по URL. Подробнее см. https://example.com
```

Все эти соглашения являются делом стиля, а не контента, но они вносят свой вклад в удобочитаемость комментариев. Чем лучше читаются комментарии, тем с большей вероятностью программисты обратят на них внимание, — а комментарии приносят пользу только в том случае, если программисты их читают.

Встроенные комментарии

Встроенные комментарии размещаются в конце строки:

```
while True: # Пока игрок не введет допустимый ход.
```

Встроенные комментарии коротки, потому что они должны укладываться в ограничения длины строки, указанные в руководстве по стилю. Это означает, что они могут оказаться слишком короткими, чтобы содержать достаточную информацию. Если вы решите использовать встроенные комментарии, убедитесь в том, что комментарий описывает только непосредственно предшествующую ему строку кода. Если вашему встроенному комментарию требуется больше места или он описывает другие строки кода, разместите его в отдельной строке.

Встроенные комментарии чаще всего применяют — и это весьма уместно — для пояснения смысла переменных или другой информации, относящейся к переменным. Такие встроенные комментарии записываются в команде присваивания, которая создает переменную:

```
TOTAL_DISKS = 5 # Чем больше дисков, тем сложнее головоломка.
```

Другое типичное применение встроенных комментариев — добавление информации о значениях переменных при их создании:

```
month = 2 # Месяцы пронумерованы от 0 (январь) до 11 (декабрь).  
catWeight = 4.9 # Вес в килограммах.  
website = 'inventwithpython.com' # Префикс "https://" не включается.
```

Во встроенных комментариях не следует указывать тип данных переменной, потому что он должен быть очевиден из команды присваивания; исключение составляет такая разновидность комментариев, как аннотации типов (см. подраздел «Обратное портирование аннотаций типов» далее в этой главе, с. 232).

Пояснительные комментарии

Как правило, комментарии должны сообщать, почему код был написан именно так, а не иначе, что он делает или как он это делает. Даже при использовании правильного стиля кода и полезных соглашений об именах, о которых я рассказывал в главах 3 и 4, реальный код не способен объяснить, какие задачи ставил программист. Если вы сами написали код, то через несколько недель можете забыть какие-то подробности. Поэтому лучше сегодня дать содержательные комментарии, чтобы завтра вам не пришлось проклинать вас вчерашнего. Например, ниже приведен бесполезный комментарий, который объясняет, что делает код. Вместо того чтобы пояснить задачу кода, он констатирует очевидное:

```
>>> currentWeekWages *= 1.5 # Заработок за текущую неделю умножается на 1.5
```

Такой комментарий хуже бесполезного. Из кода и так очевидно, что переменная `currentWeekWages` умножается на 1.5, и если полностью убрать комментарий, это только упростит ваш код. Следующий комментарий гораздо лучше:

```
>>> currentWeekWages *= 1.5 # Включить в расчет полуторную ставку.
```

Этот комментарий объясняет смысл строки кода, а не пересказывает, как тот работает. Он предоставляет информацию, которую даже хорошо написанный код передать не сможет.

Сводные комментарии

Практическая польза комментариев не ограничивается пояснением намерений программиста. Краткие комментарии, резюмирующие работу нескольких строк кода, позволяют читателю бегло просмотреть исходник и получить общее представление о том, что он делает. Программисты часто вставляют пустые строки, чтобы отделить «абзацы» кода друг от друга. Сводные комментарии обычно занимают одну строку в начале таких абзацев. В отличие от однострочных комментариев, поясняющих отдельные строки кода, сводные комментарии на более высоком уровне абстракции описывают, что делает код.

Например, из следующих четырех строк кода можно понять, что они присваивают переменной `playerTurn` значение, обозначающее другого игрока. Но короткий однострочный комментарий избавит читателя от необходимости читать и обдумывать код, чтобы понять смысл происходящего:

```
# Ход передается другому игроку:
if playerTurn == PLAYER_X:
    playerTurn = PLAYER_O
elif playerTurn == PLAYER_O:
    playerTurn = PLAYER_X
```

Если вы включите эти сводные комментарии в свою программу, это значительно упростит ее чтение. Далее программист сможет внимательнее проанализировать те места, которые вызывают у него особый интерес. Сводные комментарии также препятствуют формированию ошибочных представлений о том, что делает код. Краткий сводный комментарий позволяет разработчику удостовериться, что он правильно понял, как работает код.

Комментарии «полученный опыт»

Когда я работал в компании, занимающейся разработкой программного обеспечения, однажды мне предложили адаптировать библиотеку построения графиков — требовалось, чтобы она в реальном времени обновляла миллионы точек данных на диаграмме. Используемая библиотека могла либо обновлять графики в реальном времени, либо поддерживать графики с миллионами точек данных, но не то и другое одновременно. Я думал, что справлюсь с этой задачей за несколько дней. Через три недели я все еще был убежден, что до окончания работы рукой подать. Каждый день мне казалось, что решение совсем рядом, и на пятую неделю у меня появился рабочий прототип.

За это время я много узнал о том, как работает библиотека, каковы ее возможности и ограничения. Поэтому я потратил несколько часов на то, чтобы оформить мои знания в комментарий длиной в страницу, который я включил в исходный код. Я был убежден, что каждый, кому потребуется внести изменения в работу программы, столкнется с теми же (простыми на первый взгляд) проблемами, что и я, а написанная мной документация сэкономит им целые недели работы.

Такие комментарии могут занимать несколько абзацев, из-за чего они кажутся неуместными в файлах с исходным кодом. Но содержащаяся в них информация бесценна для любого специалиста, который будет заниматься сопровождением программы. Не бойтесь включать в свой код длинные, подробные комментарии, которые объясняют, как он работает. Другие программисты могут не знать тонкости вашей реализации кода, они могут понять их неправильно или упустить их из виду. Если разработчика комментарии не интересуют, он с легкостью их пропустит, зато те, кому они нужны, будут благодарны.

Как и в предыдущих случаях, этот вид комментариев не должен заменять документацию модуля или функции (для которой создаются дос-строки). Комментарии типа «полученный опыт» — не учебник и не сборник рецептов для пользователей программы. Они предназначены для разработчиков, читающих исходный код. Так как мой комментарий относился к библиотеке с открытым кодом и мог пригодиться другим, я опубликовал его в ответе на общедоступном сайте <https://stackoverflow.org>, где он оказался доступен другим пользователям, оказавшимся в аналогичной ситуации.

Комментарии об авторских правах и интеллектуальной собственности

Некоторые компании-разработчики или проекты с открытым кодом используют политику включения сведений об авторских правах и интеллектуальной собственности, а также текстов лицензий в начало каждого файла с исходным кодом. Такие аннотации должны содержать всего несколько строк и выглядеть примерно так:

```
""Cat Herder 3.0 Copyright (C) 2021 Al Sweigart. All rights reserved.  
See license.txt for the full text.""
```

Если возможно, включите ссылку на внешний документ или веб-сайт с полным текстом лицензии (вместо того, чтобы включать всю длинную лицензию в начало каждого файла с исходным кодом). Прокручивать несколько лишних экранов текста каждый раз, когда вы открываете файл с исходным кодом, утомительно, а публикация текста полной лицензии не обеспечивает дополнительной юридической защиты.

Профессиональные комментарии

На моей первой работе в качестве программиста старший коллега, которого я очень уважал, отвел меня в сторону и объяснил, что поскольку мы иногда

предоставляем исходный код наших продуктов клиентам, очень важно выдерживать профессиональный тон в комментариях. Как выяснилось, я написал «Какого черта?» в одном из комментариев, относящихся к особенно противной части кода. Я устыдился, немедленно извинился и отредактировал текст. С этого момента я пишу в комментариях (даже в персональных проектах) только профессиональные замечания.

Возможно, вам захочется высказать свое раздражение или проявить остроумие в комментариях программы, но делать так не стоит. Вы не знаете, кто будет читать ваш код в будущем, а тональность текста легко интерпретируется неверно. Как я объяснял в разделе «Избегайте шуток, каламбуров и культурных отсылок», с. 91, лучше всего писать комментарии прямолинейно, четко и без юмора.

Кодовые метки и комментарии *TODO*

Программисты иногда оставляют короткие комментарии, напоминающие им о работе, которую еще предстоит сделать. Обычно их оформляют в виде *кодовых меток* — комментариев, которые предваряет метка, записанная в верхнем регистре (например, *TODO*). В идеале для памяток лучше использовать средства управления проектами, а не зарывать их в исходном коде. Но в небольших персональных проектах, в которых такие инструменты не используются, встречающиеся время от времени метки *TODO* могут стать полезным напоминанием. Пример:

```
_chargeIonFluxStream() # TODO: Выяснить, почему каждый вторник происходит сбой.
```

Для таких напоминаний часто используются следующие метки:

- **TODO** — общее напоминание о работе, которую необходимо выполнить;
- **FIXME** — эта часть кода работает не полностью;
- **HACK** — эта часть кода работает (возможно, по минимуму), но ее можно улучшить;
- **XXX** — общее предупреждение, часто весьма серьезное.

За метками, записанными в верхнем регистре, следует давать более конкретные описания задачи или проблемы. Позднее вы сможете провести поиск меток в исходном коде и найти фрагмент, который необходимо доработать.

С другой стороны, у таких напоминаний есть и недостаток: о них легко забыть, если только вы не читаете ту часть кода, где они находятся. Кодовые метки не заменяют формальной системы отслеживания ошибок или программ отправки отчетов об ошибках. Если вы используете кодовые метки в своей программе, я рекомендую предельно упростить их: используйте только *TODO* и откажитесь от остальных.

This module provides a Session object to manage and persist settings across requests (cookies, auth, proxies)

"""

import os

import sys

--snip--

class Session(SessionRedirectMixin):

"""A Requests session. ❸

Provides cookie persistence, connection-pooling, and configuration.

Basic Usage::

>>> import requests

>>> s = requests.Session()

>>> s.get('https://httpbin.org/get')

<Response [200]>

--snip--

def get(self, url, **kwargs):

r"""Sends a GET request. Returns :class:`Response` object. ❹

:param url: URL for the new :class:`Request` object.

:param **kwargs: Optional arguments that ``request`` takes.

:rtype: requests.Response

"""

--snip--

Файл `sessions.py` содержит дос-строки для модуля ❷, класса `Session` ❸ и метода `get()` класса `Session` ❹. Обратите внимание: хотя дос-строка модуля должна быть первой строкой в модуле, она располагается после любых специальных комментариев — в частности, определения кодировки ❶.

Позднее вы можете прочитать дос-строки модуля, класса, функции или метода, проверяя атрибут `__doc__` соответствующего объекта. Так, в следующем примере проверяются дос-строки для получения дополнительной информации о модуле `sessions`, классе `Session` и методе `get()`:

>>> from requests import sessions

>>> sessions.__doc__

'\nrequests.session\n~~~~~\n\nThis module provides a Session object to manage and persist settings across\nrequests (cookies, auth, proxies).\n'

>>> sessions.Session.__doc__

"A Requests session.\n\n Provides cookie persistence, connection-pooling, and configuration.\n\n Basic Usage::\n\n >>> import requests\n

--snip--

>>> sessions.Session.get.__doc__

'Sends a GET request. Returns :class:`Response` object.\n\n :param url: URL for the new :class:`Request` object.\n :param **kwargs:

--snip--

Средства автоматизированного документирования могут пользоваться дос-строками для предоставления информации, соответствующей контексту. Одно из таких средств — встроенная функция Python `help()` — выводит дос-строку переданного ей объекта в более удобочитаемом формате, чем у необработанной строки `__doc__`. Данная возможность может оказаться полезной при экспериментах с интерактивной оболочкой, потому что вы можете немедленно получить информацию о любых модулях, классах и функциях, которые вы пытаетесь использовать:

```
>>> from requests import sessions
>>> help(sessions)
Help on module requests.sessions in requests:

NAME
    requests.sessions

DESCRIPTION
    requests.session
    ~~~~~

    This module provides a Session object to manage and persist settings
-- More --
```

Если дос-строка слишком велика, чтобы поместиться на экране, Python выводит подсказку `-- More --` в нижней части окна. Нажмите ENTER, чтобы прокрутить текст к следующей строке, пробел для вывода следующей страницы или Q для прекращения просмотра дос-строки.

В общем случае дос-строка должна содержать одну строку текста с обобщающим описанием модуля, класса или функции, за которым следует пустая строка и более подробная информация. Для функций и методов она может включать информацию об их параметрах, возвращаемом значении и побочных эффектах. Дос-строки пишутся для других программистов, а не для пользователей программы, поэтому они должны содержать техническую информацию, а не обучающие советы.

У дос-строк также есть второе ключевое преимущество: они интегрируют документацию в исходный код. Когда документация пишется отдельно от кода, о ней часто вообще забывают. С другой стороны, когда дос-строки размещены в начале модулей, классов и функций, информацию легко просматривать и обновлять.

Возможно, вы не сможете сразу писать дос-строки, если работа над кодом, который вы хотите описывать, еще не завершена. В таком случае включите в дос-строку комментарий `TODO` с напоминанием. Например, следующая вымышленная функция `reverseCatPolarity()` содержит плохую дос-строку, которая утверждает очевидное:

```
def reverseCatPolarity(catId, catQuantumPhase, catVoltage):
    """Reverses the polarity of a cat.
```

```
    TODO Finish this docstring.""
--snip--
(    """Меняет полярность у кота.
    TODO: Закончить эту дос-строку)
```

Так как каждый класс, функция и метод должны содержать дос-строку, может появиться искушение написать минимальную документацию и двигаться дальше. Без комментария `TODO` слишком легко забыть о том, что дос-строку следует переписать.

PEP 257 содержит дополнительную документацию о дос-строках по адресу <https://www.python.org/dev/peps/pep-0257/>.

Аннотации типов

Во многих языках программирования используется *статическая типизация*; это означает, что программист должен объявить типы данных всех переменных, параметров и возвращаемых значений в исходном коде. Такая возможность позволяет интерпретатору или компилятору проверить правильность применения всех объектов до запуска программы. В Python используется *динамическая типизация*: переменные, параметры и возвращаемые значения могут иметь любой тип данных и даже менять типы данных во время выполнения программы. На динамических языках обычно проще программировать, потому что они требуют меньше формальных определений, но зато в них отсутствуют средства предотвращения ошибок, присущие статическим языкам. Когда вы пишете строку кода Python — например, `round('forty two')`, вы можете не понять, что строка передается функции, получающей только аргументы `int` и `float`, пока программа не будет запущена и не произойдет ошибка. Языки со статической типизацией выдают ранние предупреждения при присваивании значения или передаче аргумента неправильного типа.

Аннотации типов в Python предоставляют необязательные средства статической типизации. В следующем примере аннотации типов выделены жирным шрифтом:

```
def describeNumber(number: int) -> str:
    if number % 2 == 1:
        return 'An odd number. '
    elif number == 42:
        return 'The answer. '
    else:
        return 'Yes, that is a number. '
myLuckyNumber: int = 42
print(describeNumber(myLuckyNumber))
```

Как видите, для параметров или переменных аннотация типа использует двоеточие для отделения имени от типа, тогда как для возвращаемых значений закрывающая круглая скобка команды `def` отделяется от типа стрелкой (`->`). Аннотации типов

функции `describeNumber()` показывают, что функция получает целое число в параметре `number` и возвращает строковое значение.

Если вы используете аннотации типов, вам не нужно применять их к каждому значению данных в программе. Вместо этого можно воспользоваться методом *постепенной типизации*, сочетающим гибкость динамической типизации с безопасностью статической типизации: аннотации типов включаются только для некоторых переменных, параметров и возвращаемых значений. Но чем больше аннотаций типов появляется в вашей программе, тем больше информации получают средства статического анализа кода для выявления потенциальных ошибок в вашей программе.

Обратите внимание: в этом примере имена заданных типов совпадают с именами функций-конструкторов `int()` и `str()`. В Python термины «класс», «тип» и «тип данных» имеют одинаковый смысл. Для любых экземпляров, созданных на основе классов, имя класса используется как его тип:

```
import datetime
noon: datetime.time = datetime.time(12, 0, 0) ❶

class CatTail:
    def __init__(self, length: int, color: str) -> None:
        self.length = length
        self.color = color

zophieTail: CatTail = CatTail(29, 'grey') ❷
```

Переменная `noon` снабжена аннотацией типа `datetime.time` ❶, потому что это объект `time` (определенный в модуле `datetime`). Аналогичным образом объект `zophieTail` снабжен аннотацией типа `CatTail` ❷, потому что это объект класса `CatTail`, созданного командой `class`. Аннотации типов автоматически распространяются на все subclasses заданного типа. Например, переменной с аннотацией типа `dict` можно присвоить не только любое значение словаря, но и любое из значений `collections.OrderedDict` и `collections.defaultdict`, потому что эти классы являются subclasses `dict`. О subclasses мы более подробно поговорим в главе 16.

Средствам статической проверки типов не обязательно нужны аннотации типов для переменных. Дело в том, что средства статической проверки типов выполняют автоматическое определение типа на основании первой команды присваивания переменной. Например, по строке `spam = 42` система проверки типов может автоматически определить, что переменная `spam` должна иметь аннотацию типа `int`. Тем не менее я все равно рекомендую включать аннотации типов. Будущий переход к типу `float`, как в команде `spam = 42.0`, также изменит автоматически определяемый тип, что может противоречить вашим намерениям. Лучше заставить программиста изменить аннотацию типа при изменении значения, чтобы он подтвердил, что изменение было внесено намеренно, а не случайно.

Статические анализаторы

Хотя Python поддерживает синтаксис аннотаций типов, интерпретатор Python полностью их игнорирует. Если вы запустите программу Python, которая передает функции переменную с неправильным типом, Python будет вести себя так, словно аннотации типа не существует. Иначе говоря, аннотации типов не заставляют интерпретатор Python выполнять какую-либо проверку типов на стадии выполнения. Аннотации типов существуют только для средств статической проверки типов, которые анализируют код до запуска программы, а не во время выполнения.

Эти средства называются средствами статического анализа, потому что они анализируют исходный код до запуска программы, тогда как средства динамического анализа работают с уже запущенными программами. (В данном случае термины «статический» и «динамический» относятся к тому, выполняется ли программа, но под терминами «статическая типизация» и «динамическая типизация» понимается способ объявления типов данных переменных и функций. Python является языком с динамической типизацией, для которого были написаны средства статического анализа — такие как Муру.)

Установка и запуск Муру

Хотя у Python нет официальных средств проверки типов, из сторонних разработок в настоящее время наибольшей популярностью пользуется Муру. Чтобы установить Муру с помощью `pip`, выполните следующую команду:

```
python -m pip install -user mypy
```

В macOS и Linux выполните команду `python3` вместо `python`. Среди других популярных систем проверки типов можно выделить Pyright (Microsoft), Pyre (Facebook) и Pytype (Google).

Чтобы запустить программу проверки типов, откройте окно командной строки или терминала и выполните команду `python -m mypy` (чтобы запустить модуль как приложение) с именем проверяемого файла с кодом Python. В следующем примере проверяется код программы из файла с именем `example.py`:

```
C:\Users\Al\Desktop>python -m mypy example.py
Incompatible types in assignment (expression has type "float", variable has
type "int")
Found 1 error in 1 file (checked 1 source file)
```

Программа не выводит ничего, если ошибки не найдены, и выводит сообщения об ошибках в противном случае. В файле `example.py` проблема обнаруживается в строке 171, потому что переменная с именем `spam` имеет аннотацию типа `int`, но

ей присваивается значение `float`. Это присваивание способно привести к ошибке, и к нему стоит присмотреться. Некоторые сообщения об ошибках трудно понять с первого взгляда. В сообщениях Муру могут упоминаться многочисленные возможные ошибки — слишком многочисленные, чтобы перечислять их здесь. Чтобы понять, что означает та или иная ошибка, проще всего поискать информацию в интернете.

Запускать Муру из командной строки при каждом изменении кода неэффективно. Чтобы пользоваться средствами проверки типов было удобнее, необходимо настроить IDE или текстовый редактор так, чтобы эти средства выполнялись в фоновом режиме. В этом случае редактор будет постоянно запускать Муру при вводе кода, а затем выводить найденные ошибки в редакторе. На рис. 11.1 изображена ошибка из предыдущего примера в текстовом редакторе Sublime Text.

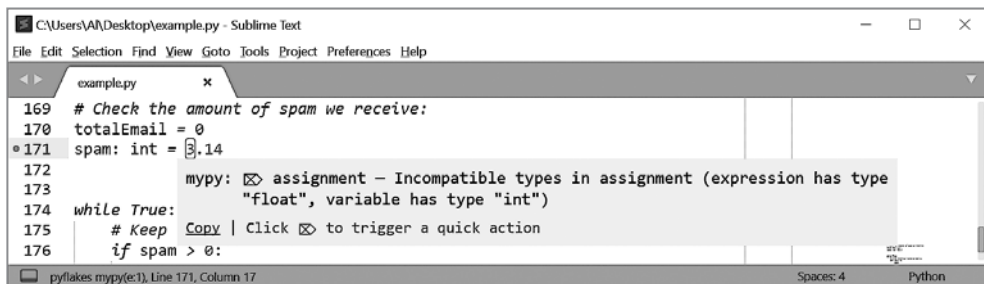


Рис. 11.1. Ошибки Муру в текстовом редакторе Sublime Text

Конкретная последовательность действий по настройке IDE или текстового редактора для работы с Муру зависит от того, в каком редакторе или IDE вы работаете. Инструкции можно найти в интернете — используйте условие поиска «*<ваша IDE>* Муру настройка» («*<your IDE>* Мyру configure»), «*<ваша IDE>* аннотации типов настройка» («*<your IDE>* type hints setup») или что-нибудь в этом роде. Если все попытки оказались безуспешными, вы всегда можете запустить Муру из окна командной строки или терминала.

Подавление обработки кода в Муру

Представьте, что вы написали блок кода, для которого по какой-то причине вы не получаете предупреждения, относящиеся к аннотациям типов. С точки зрения средств статического анализа в строке вроде бы используется неправильный тип, но вы уверены, что во время выполнения эта строка будет работать правильно. Чтобы подавить любые предупреждения о типах, добавьте комментарий `# type: ignore` в конец строки. Пример:

```
def removeThreesAndFives(number: int) -> int:
    number = str(number) # type: ignore
    number = number.replace('3', '').replace('5', '') # type: ignore
    return int(number)
```

Чтобы удалить из целого числа, передаваемого `removeThreesAndFives()`, все цифры 3 и 5, мы временно преобразуем целочисленную переменную в строку. Из-за этого программа проверки типов выдает предупреждения о двух первых строках функции, поэтому в эти строки добавляются аннотации типов `# type: ignore` для подавления предупреждений.

Пользуйтесь директивами `# type: ignore` осмотрительно. Игнорирование предупреждений от средств проверки типов открывает возможности для проникновения ошибок в ваш код. Код почти всегда можно переписать так, чтобы предупреждения не выдавались. Например, если создать новую переменную командой `numberAsStr = str(number)` или заменить все три строки кода одной строкой `return int(str(number.replace('3', '').replace('5', '')))`, можно избежать повторного использования переменной `number` для разных типов. Не следует подавлять предупреждения, изменяя аннотацию типа для этого параметра на `Union[int, str]`, потому что этот параметр предназначен только для целых чисел.

Аннотации типов для набора типов

Переменные, параметры и возвращаемые значения Python могут принимать разные типы данных. Чтобы учесть эту возможность в программе, следует задать аннотации типов с несколькими типами; для этого надо импортировать `Union` из встроенного модуля `typing`. Набор допустимых типов задается в квадратных скобках после имени класса `Union`:

```
from typing import Union
spam: Union[int, str, float] = 42
spam = 'hello'
spam = 3.14
```

В этом примере аннотация `Union[int, str, float]` указывает, что переменной `spam` может быть присвоено целое число, строка или число с плавающей точкой. Обратите внимание: лучше использовать форму команды импортирования `from typing import X` вместо `import typing`, а затем последовательно использовать развернутую форму `typing.X` для аннотаций типов во всей программе.

Можно указать несколько типов данных в ситуациях, когда переменная или возвращаемое значение могут принимать значение `None` в дополнение к другому типу. Чтобы включить `NoneType` (тип значения `None`) в аннотацию типа, укажите в квадратных скобках `None` вместо `NoneType`. (Формально `NoneType` не является встроенным идентификатором, каким является `int` или `str`.)

Еще лучше вместо `Union[str, None]` импортировать `Optional` из модуля `typing` и использовать запись `Optional[str]`. Эта аннотация типа означает, что функция или метод может вернуть `None` вместо значения ожидаемого типа. Пример:

```
from typing import Optional
lastName: Optional[str] = None
lastName = 'Sweigart'
```

Здесь переменной `lastName` может быть присвоено значение `str` или `None`. Тем не менее к использованию `Union` и `Optional` стоит подходить осмотрительно. Чем меньше типов допускают ваши переменные и функции, тем проще будет ваш код, а в простом коде реже возникают ошибки, чем в сложном. Вспомните тезис «Дзен Python»: простое лучше, чем сложное. Если функция возвращает `None` как признак ошибки, рассмотрите возможность замены кода ошибки исключением (см. раздел «Выдача исключений и возвращение кодов ошибок», с. 214).

Чтобы указать, что переменная, параметр или возвращаемое значение может иметь любой тип данных, используйте аннотацию типа `Any` (также из модуля `typing`):

```
from typing import Any
import datetime
spam: Any = 42
spam = datetime.date.today()
spam = True
```

В этом примере аннотация типа `Any` позволяет вам присвоить переменной `spam` значение любого типа данных — например, `int`, `datetime.date` или `bool`. Также в качестве аннотации типа можно использовать `object`, потому что этот класс является базовым для всех типов данных Python. Тем не менее аннотация типа `Any` лучше читается, чем `object`.

`Any`, как и `Union` и `Optional`, следует использовать осмотрительно. Назначив всем переменным, параметрам и возвращаемым значениям аннотацию типа `Any`, вы лишитесь преимуществ проверки типов, которые предоставляет статическая типизация. Разница между аннотацией типа `Any` и отсутствием аннотации состоит в том, что `Any` явно сообщает, что переменная или функция принимает значения любого типа, тогда как отсутствие аннотации означает, что переменная или функция пока еще не аннотированы.

Аннотации типов для списков, словарей и т. д.

Списки, словари, кортежи, множества и другие контейнерные типы данных могут содержать другие значения. Если вы укажете список (`list`) как аннотацию типа для переменной, эта переменная должна содержать список, но тот в свою очередь может содержать значения произвольного типа. Следующий код не вызовет никаких протестов у программы проверки типов:

```
spam: list = [42, 'hello', 3.14, True]
```

Чтобы объявить типы данных значений, хранящихся в списке, используйте аннотацию типа `List` модуля `typing`. Обратите внимание: `List` начинается с буквы `L` в верхнем регистре, что отличает ее от типа данных `list`:

```
from typing import List, Union
catNames: List[str] = ['Zophie', 'Simon', 'Pooka', 'Theodore'] ❶
numbers: List[Union[int, float]] = [42, 3.14, 99.9, 86]        ❷
```

В этом примере переменная `catNames` содержит список строк, поэтому после импортирования `List` из модуля `typing` мы задаем аннотацию типа `List[str]` ❶. Система проверки типов перехватывает все вызовы методов `append()` или `insert()` или любого другого кода, который помещает в список значения, не являющиеся строками. Если список должен содержать данные разных типов, используйте `Union` в аннотации типов. Например, список `numbers` может содержать целые числа и числа с плавающей точкой, поэтому для него задается аннотация `List[Union[int, float]]` ❷.

Модуль `typing` имеет отдельный *псевдоним типа* (type alias) для каждой разновидности контейнеров. Ниже перечислены псевдонимы для всех распространенных контейнерных типов Python:

- `List` для типа данных `list`;
- `Tuple` для типа данных `tuple`;
- `Dict` для типа данных словаря (`dict`);
- `Set` для типа данных `set`;
- `FrozenSet` для типа данных `frozenset`;
- `Sequence` для `list`, `tuple` и любых других типов данных последовательностей;
- `Mapping` для словарей (`dict`), `set`, `frozenset` и любых других типов данных отображений;
- `ByteString` для типов данных `bytes`, `bytearray` и `memoryview`.

Полный перечень этих типов доступен по адресу <https://docs.python.org/3/library/typing.html#classes-functions-and-decorators>.

Обратное портирование аннотаций типов

Обратным портированием (backporting) называется процесс выделения некоторой функциональности из новой версии программного продукта и портирование ее

(то есть адаптация и добавление) в более раннюю версию. Аннотации типов Python появились только в версии 3.5. Но в коде Python, который должен выполняться в версиях интерпретатора ниже 3.5, все равно можно использовать аннотации типов, размещая информацию о типах в комментариях. Для переменных используется встроенный комментарий после команды присваивания. Для функций и методов аннотации типов записываются в строке после команды `def`. Комментарий начинается с `type:`, а затем указывается тип данных. Пример кода с аннотациями типов в комментариях:

```
from typing import List    ❶

spam = 42 # type: int      ❷
def sayHello():
    # type: () -> None      ❸
    """Doc-строка следует за комментарием с аннотацией типа."""
    print('Hello!')

def addTwoNumbers(listOfNumbers, doubleTheSum):
    # type: (List[float], bool) -> float    ❹
    total = listOfNumbers[0] + listOfNumbers[1]
    if doubleTheSum:
        total *= 2
    return total
```

Обратите внимание: даже если вы используете стиль аннотаций типов в комментариях, все равно необходимо импортировать модуль `typing` ❶, а также все псевдонимы типов, которые вы будете использовать в комментариях. В версиях до 3.5 стандартная библиотека не включала модуль `typing`, его следует установить отдельно следующей командой:

```
python -m pip install --user typing
```

В macOS и Linux используйте команду `python3` вместо `python`.

Чтобы связать переменную `spam` с целым типом, мы добавляем `# type: int` в комментарий в конце строки ❷. Для функций комментарий должен включать круглые скобки со списком аннотаций типов, разделенных запятыми, порядок которых соответствует порядку параметров. Функции без параметров должны иметь пустой набор круглых скобок ❸. Если параметров несколько, разделите их запятыми в круглых скобках ❹.

Аннотации типов в комментариях читаются хуже, чем просто аннотации типов, поэтому они используются только для кода, который может выполняться версиями Python до 3.5.

Итоги

Программисты часто забывают о документировании своего кода. Но потратив немного времени на добавление комментариев, doc-строк и аннотаций типов в ваш код, вы избежите временных затрат в будущем. Хорошо документированный код также проще сопровождать.

Соблазнительно принять точку зрения, что комментарии и документация не важны или даже приносят вред при написании программ. (Такая позиция также избавляет программистов от работы по написанию документации — очень удобно.) Не обманывайте себя: хорошо написанная документация всегда экономит гораздо больше времени и усилий, чем требуется на ее создание. Программистам намного чаще приходится иметь дело со страницами невразумительного кода без комментариев, чем с избытком полезной информации.

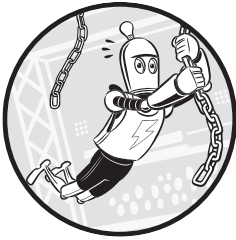
Хорошие комментарии предоставляют полезную, краткую и точную информацию тем, кому придется читать код позднее и разбираться в том, что код делает. Комментарии должны объяснять намерения программиста и обобщать смысл небольших блоков кода, а не пересказывать очевидный смысл одной строки кода. Иногда в комментариях содержится подробное описание информации, полученной и усвоенной программистом во время написания кода. В будущем эти ценные сведения избавят тех, кто занимается сопровождением кода, от горькой участи заново добывать ее, тратя время и силы.

Dos-строки — разновидность комментариев, специфическая для Python, — представляют собой многострочные тексты, следующие непосредственно за командами `class` или `def` или расположенные в начале модуля. Средства документирования (например, встроенная функция Python `help()`) могут извлекать doc-строки из кода, чтобы предоставить конкретные сведения о предназначении класса, функции или модуля.

Аннотации типов, появившиеся в Python 3.5, дают Python механизм постепенной типизации. Постепенная типизация позволяет программисту пользоваться преимуществами статической типизации по определению типов без потери гибкости динамической типизации. Интерпретатор Python игнорирует аннотации типов, потому что в Python отсутствует проверка типов на стадии выполнения. Тем не менее средства статической разработки типов используют аннотации типов для анализа исходного кода до его выполнения. Средства проверки типов — такие как Муру — следят за тем, чтобы переменным, передаваемым функциям, не присваивались недопустимые значения. Это экономит время и усилия за счет предотвращения самых разнообразных ошибок.

12

Git и организация программных проектов



Системы контроля версий представляют собой программные средства, которые регистрируют все изменения в исходном коде и позволяют легко восстановить старые версии. Рассматривайте их как сильно усовершенствованную функцию отмены (undo). Например, если вы заменяете функцию,

а потом решаете, что старая версия нравится вам больше, вы можете восстановить исходный фрагмент кода. Или при обнаружении новой ошибки вы можете вернуться к более ранним версиям, чтобы определить, когда она впервые появилась и какие изменения ее вызвали.

Система контроля версий управляет файлами при внесении в них изменений. Пользоваться ею удобнее, чем, скажем, работать с копией папки `myProject`. Первой копии вы присвоите имя `myProject-copy`. Затем, если вы продолжите вносить изменения, вам потребуется создать новую копию с именем `myProject-copy2`, потом `myProject-copy3`, `myProject-copy3b`, `myProject-copyAsOfWednesday` и т. д. Возможно, копирование папок — простой метод, но он не масштабируется. Умение пользоваться системой контроля версий экономит ваше время и избавит от многих хлопот в долгосрочной перспективе.

Git, Mercurial и Subversion — популярные приложения контроля версий, хотя система Git остается самой популярной. В этой главе вы узнаете, как подготовить файлы для программного проекта и как использовать Git для отслеживания в них изменений.

Коммиты и репозитории

Git позволяет сохранить состояние файлов проекта при внесении в них изменений. Такие сохранения называются *снимками* (snapshots) или *коммитами* (commits). Благодаря этому вы сможете, если потребуется, вернуться к любой предшествующей версии.

Системы контроля версий также позволяют команде разработчиков синхронизировать свою работу при внесении изменений в исходный код проекта. Когда каждый программист закрепляет свои изменения, другие могут извлекать эти обновления на своих компьютерах. Система контроля версий следит за тем, какие изменения были внесены, кто и когда их сделал, а также сохраняет комментарии разработчиков, описывающие эти изменения.

Система контроля версий управляет исходным кодом проекта, который хранится в специальной папке — *репозитории* (repo). Как правило, для каждого проекта, над которым вы работаете, следует создать отдельный репозиторий Git. Предположим, вы работаете в основном самостоятельно над своей частью программы и вам не нужны расширенные возможности Git (такие как *ветвление* и *слияние*), упрощающие работу с остальными участниками. Но даже если вы работаете в одиночку, самый маленький проект все равно выиграет от применения системы контроля версий.

Создание новых проектов Python с использованием Cookiecutter

В терминологии Git папка, содержащая весь исходный код, документацию, тесты и другие файлы, относящиеся к проекту, называется *рабочим каталогом* или *рабочим деревом*, а в более общей терминологии — *папкой проекта*. Файлы в рабочем каталоге в совокупности называются рабочей копией. Прежде чем создавать репозиторий Git, следует создать файлы для проекта Python.

У каждого программиста есть свой способ выполнения этой операции. Тем не менее для проектов Python действуют определенные соглашения по поводу имен папок и иерархий. Более простая программа может содержать один файл `.py`. Но когда проекты усложнятся, в них будут включаться дополнительные файлы `.py`, файлы данных, документация, модульные тесты и т. д. Как правило, корневая папка проекта содержит папку `src` для файлов с исходным кодом `.py`, папку `tests` для модульных тестов и папку `docs` для документации (например, сгенерированной системой документирования Sphinx). Другие файлы содержат информацию о проекте и конфигурации системы: `README.md` для общей информации, `.coveragerc` для конфигурации покрытия кода, `LICENSE.txt` для текста программной лицензии

проекта и т. д. Описание этих средств и файлов выходит за рамки книги, но они заслуживают вашего внимания. С появлением практического опыта программирования необходимость заново создавать одни и те же базовые файлы для всех новых проектов надоедает и утомляет. Чтобы ускорить выполнение рутинных операций, можно использовать модуль Python `cookiecutter` для автоматического создания этих файлов и папок. Полная документация по модулю и программе командной строки Cookiecutter доступна на <https://cookiecutter.readthedocs.io/>.

Чтобы установить Cookiecutter, выполните команду `pip install --user cookiecutter` (в системе Windows) или `pip3 install --user cookiecutter` (в macOS и Linux). В установку включаются программа командной строки Cookiecutter и модуль Python `cookiecutter`. В процессе установки вы можете получить предупреждение о том, что программа командной строки устанавливается в папку, не входящую в переменную окружения `PATH`:

```
Installing collected packages: cookiecutter
WARNING: The script cookiecutter.exe is installed in 'C:\Users\AI\AppData\Roaming\Python\Python38\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
```

Возможно, вам стоит включить папку (`C:\Users\AI\AppData\Roaming\Python\Python38\Scripts` в данном случае) в переменную среды `PATH`; инструкции приведены в разделе «Переменные среды и `PATH`», с. 60. В противном случае вам придется запускать Cookiecutter как модуль Python командой `python -m cookiecutter` (в Windows) или `python3 -m cookiecutter` (в macOS или Linux) вместо простой команды `cookiecutter`.

В этой главе мы создадим репозиторий для модуля `wizcoin`, предназначенного для работы с различными видами волшебной валюты. Модуль `cookiecutter` использует шаблоны для создания начальных файлов для различных видов проектов. Часто шаблон представляет собой простую ссылку на *GitHub.com*. Например, в папке `C:\Users\AI` можно ввести приведенную ниже команду из окна терминала для создания папки `C:\Users\AI\wizcoin` с заготовками файлов для базового проекта Python. Модуль `cookiecutter` загружает шаблон с GitHub и задает серию вопросов о создаваемом проекте:

```
C:\Users\AI>cookiecutter gh:asweigart/cookiecutter-basicpythonproject
project_name [Basic Python Project]: WizCoin
module_name [basicpythonproject]: wizcoin
author_name [Susie Softwaredeveloper]: AI Sweigart
author_email [susie@example.com]: ai@inventwithpython.com
github_username [susieexample]: asweigart
project_version [0.1.0]:
project_short_description [A basic Python project.]: A Python module to
represent the galleon, sickle, and knut coins of wizard currency.
```

Если вы получите ошибку, попробуйте выполнить команду `python -m cookiecutter` вместо `cookiecutter`. Команда загружает созданный мной шаблон на <https://github.com/asweigart/cookiecutter-basicpythonproject>. В разделе <https://github.com/cookiecutter/cookiecutter> вы найдете шаблоны для многих языков программирования. Так как шаблоны Cookiecutter часто размещаются на GitHub, вы также можете ввести `gh`: как сокращение для <https://github.com/> в аргументе командной строки.

Когда Cookiecutter задает свои вопросы, следует либо ввести ответ, либо просто нажать ENTER, чтобы использовать ответ по умолчанию, приведенный в квадратных скобках. Например, запрос `project_name [Basic Python Project]`: предлагает ввести имя проекта. Если не указать ничего, Cookiecutter использует имя проекта Basic Python Project. Значения по умолчанию также подсказывают, какой ответ предполагается. В подсказке `project_name [Basic Python Project]`: в имени проекта используются символы верхнего регистра, тогда как из подсказки `module_name [basicpythonproject]`: видно, что имя модуля записывается в нижнем регистре и не содержит пробелов. Мы не ввели ответ на запрос `project_version [0.1.0]`:, поэтому по умолчанию используется ответ `0.1.0`.

После ответов на вопросы Cookiecutter создает папку `wizcoin` в текущем рабочем каталоге с базовыми файлами, необходимыми для проекта Python (рис. 12.1).

Name	Date modified	Type	Size
docs	8/31/2021 12:37 PM	File folder	
src	8/31/2021 12:37 PM	File folder	
tests	8/31/2021 12:37 PM	File folder	
.coveragerc	8/31/2021 12:37 PM	COVERAGERC File	1 KB
.gitignore	8/31/2021 12:37 PM	Text Document	2 KB
code_of_conduct...	8/31/2021 12:37 PM	MD File	4 KB
LICENSE.txt	8/31/2021 12:37 PM	TXT File	35 KB
pyproject.toml	8/31/2021 12:37 PM	TOML File	0 KB
README.md	8/31/2021 12:37 PM	MD File	1 KB
setup.py	8/31/2021 12:37 PM	PY File	2 KB
tox.ini	8/31/2021 12:37 PM	INI File	1 KB

Рис. 12.1. Файлы в папке `wizcoin`, созданной Cookiecutter

Если вы не знаете, для чего нужны те или иные файлы, не огорчайтесь. Хотя полное описание назначения каждого файла выходит за рамки книги, на странице <https://github.com/asweigart/cookiecutter-basicpythonproject> присутствуют ссылки и нужная информация. Итак, базовые файлы созданы, и мы переходим к отслеживанию изменений в них в системе Git.

Установка Git

Возможно, Git уже установлена на вашем компьютере. Чтобы узнать это, введите команду `git --version` в командной строке. Если вы увидите сообщение вида `git version 2.29.0.windows.1`, значит, у вас уже установлена программная поддержка Git. Если вы увидите сообщение «команда не найдена», Git придется установить. В системе Windows перейдите на страницу <https://git-scm.com/download>, загрузите и запустите программу установки Git. В macOS Mavericks (10.9) и более поздних версиях просто выполните команду `git --version` из окна терминала; вам будет предложено установить Git, как показано на рис. 12.2.

В Ubuntu или Debian Linux выполните команду `sudo apt install git-all` из окна терминала. В Red Hat Linux выполните команду `sudo dnf install git-all` из окна терминала. Инструкции для других дистрибутивов Linux находятся на <https://git-scm.com/download/linux>. Чтобы убедиться в том, что установка прошла успешно, выполните команду `git --version`.

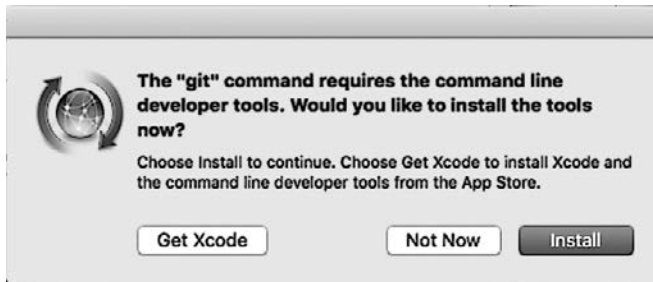


Рис. 12.2. При первом выполнении команды `git --version` в macOS 10.9 и выше вам будет предложено установить Git

Настройка имени пользователя и адреса электронной почты

После установки Git необходимо задать ваше имя и адрес электронной почты, чтобы в ваши коммиты включалась информация об авторе. В терминале выполните следующие команды `git config`, используя собственное имя и адрес электронной почты:

```
C:\Users\AI>git config --global user.name "AI Sweigart"
C:\Users\AI>git config --global user.email al@inventwithpython.com
```

Эта информация хранится в файле `.gitconfig` в вашей домашней папке (например, в `C:\Users\AI` на моем ноутбуке с Windows). Вам никогда не придется редактировать этот текстовый файл напрямую. Для его изменения достаточно выполнить команду `git config`. Для настройки текущей конфигурации Git используйте команду `git config --list`.

Установка графических средств Git

В этой главе основное внимание мы уделим средствам командной строки Git, но установка программы, добавляющей графический интерфейс (GUI) для Git, поможет вам выполнять повседневные задачи. Даже профессиональные разработчики, хорошо знающие командную строку Git, часто используют графические средства Git. На веб-странице <https://git-scm.com/downloads/guis> приведены ссылки на некоторые программы, включая TortoiseGit для Windows, GitHub Desktop для macOS и GitExtensions для Linux.

На рис. 12.3 показано, как TortoiseGit для Windows добавляет к значкам в Проводнике цветные метки в зависимости от их статуса: зеленые для неизмененных файлов в репозитории и красные для измененных файлов (или папок, содержащих измененные файлы); метка отсутствует у файлов, которые не отслеживаются. Конечно, ориентироваться по меткам гораздо проще, чем постоянно вводить команды в терминале для получения нужной информации. TortoiseGit также добавляет контекстное меню для выполнения команд Git (рис. 12.3).

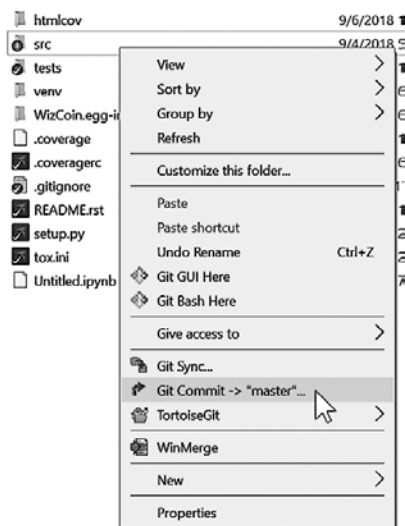


Рис. 12.3. TortoiseGit для Windows добавляет графический интерфейс для выполнения команд Git из Проводника

Графические средства Git удобны, но они не заменяют применения команд, описанных в этой главе. Возможно, вам когда-нибудь придется работать с Git на компьютере, на котором графические средства не установлены.

Работа с Git

Работа с репозиторием Git состоит из нескольких этапов. Сначала вы создаете репозиторий Git командой `git init` или `git clone`. Затем файлы добавляются в репозиторий для отслеживания командой `git add <имя_файла>`. Наконец, после добавления файлов они сохраняются командой `git commit -am "<сообщение, описывающее содержание коммита>"` (часто называют просто «сообщение коммита»). Теперь все готово для внесения изменений в код.

Вы можете просмотреть справку по каждой из этих команд командой `git help <команда>` — например, `git help init` или `git help add`. Справочные страницы удобны, но они предлагают слишком сухую и техническую информацию, чтобы ее можно было легко и просто использовать для обучения. Позднее я расскажу более подробно о каждой из этих команд, но сначала необходимо представить некоторые концепции Git — это поможет вам понять материал главы.

Как Git отслеживает статус файлов

Все файлы в рабочем каталоге либо отслеживаются, либо не отслеживаются Git. *Отслеживаемые файлы* были добавлены и сохранены в репозитории, все остальные файлы не отслеживаются. Для репозитория Git неотслеживаемые файлы в рабочей копии не существуют. С другой стороны, отслеживаемые файлы существуют в одном из трех состояний.

- В *сохраненном* (закрепленном) состоянии файл в рабочей копии идентичен последнему коммиту в репозитории. (Иногда это состояние называется *неизменным*, или *чистым*.)
- В *измененном* состоянии файл в рабочей копии отличается от последнего коммита в репозитории.
- В *индексированном*, или *подготовленном* (staged), состоянии файл был изменен и помечен для включения в следующий коммит. Также говорят, что файл находится в *индексной области* (или *кэше*).

На рис. 12.4 изображена диаграмма перехода файла между четырьмя возможными состояниями. Вы добавляете неотслеживаемый файл в репозиторий Git, после чего он становится отслеживаемым и индексированным. Далее можно сохранить индексированные файлы, чтобы перевести их в индексированное состояние. Для перевода файла в измененное состояние никакие команды Git не нужны; как только вы внесете изменения в сохраненный файл, он автоматически помечается как измененный.

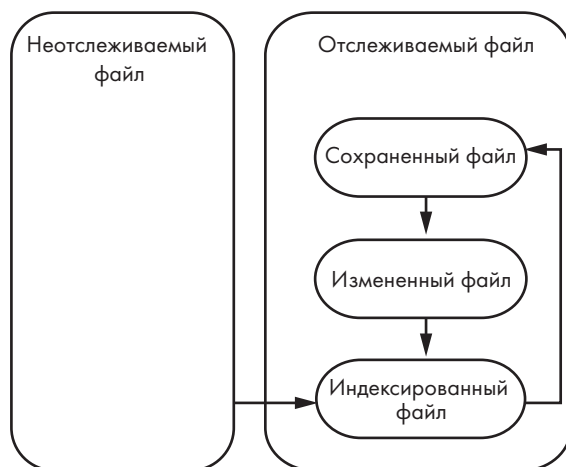


Рис. 12.4. Возможные состояния файла в репозитории Git и переходы между ними

На любом этапе после создания репозитория выполните команду `git status` для просмотра текущего статуса репозитория и состояния его файлов. Эта команда часто выполняется при работе в Git. Для следующего примера я подготовил файлы в разных состояниях. Обратите внимание на то, как эти четыре файла представлены в выходных данных `git status`:

```
C:\Users\Al\ExampleRepo>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   new_file.py      ❶
    modified:   staged_file.py   ❷

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   modified_file.py ❸

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    untracked_file.py            ❹
```

В этой рабочей копии присутствует файл `new_file.py` ❶, который недавно был добавлен в репозиторий, а следовательно, находится в индексированном состоянии. Также присутствуют два отслеживаемых файла `staged_file.py` ❷ и `modified_file.py` ❸, которые находятся в индексированном и измененном состоянии соответственно. Также присутствует неотслеживаемый файл `untracked_file.py` ❹. В выходные

данные `git status` также включены команды Git, переводящие файлы в другие состояния.

Для чего нужно индексирование?

Возникает вопрос — для чего нужно индексированное состояние? Почему нельзя просто переходить между измененным и сохраненным состоянием без индексирования файлов? Область индексирования полна неприятных особых случаев, и часто у новичков в Git вызывает недоумение. Например, файл может быть изменен после того, как он был проиндексирован, в результате чего файл существует как в измененном, так и в индексированном состоянии (см. предыдущий раздел). С технической точки зрения область индексирования содержит не столько файлы, сколько описания изменений, потому что одни части измененного файла могут быть индексированы, а другие — нет. Именно из-за таких случаев Git считается сложной системой, а многие источники о работе Git часто содержат неточную информацию в лучшем случае или дезинформацию — в худшем.

Однако большую часть этих сложностей можно обойти. В этой главе я рекомендую так и поступить, используя команду `git commit -am command` для индексирования и закрепления измененных файлов на одном шаге. В этом случае файлы переходят из измененного состояния сразу же в чистое. Кроме того, я рекомендую всегда немедленно сохранять файлы после их добавления, переименования или удаления из репозитория. Кроме того, использование графических средств Git (о которых я расскажу позднее) вместо командной строки поможет избежать этих нетривиальных случаев.

Создание репозитория Git на вашем компьютере

Git является *распределенной системой контроля версий*; это означает, что все коммиты и метаданные репозитория хранятся локально на вашем компьютере в папке с именем `.git`. В отличие от централизованных систем контроля версий Git не нужно подключаться к серверу по интернету для сохранения данных. Благодаря такому подходу система Git работает быстро и остается доступной при автономной работе.

Введите в окне терминала следующие команды для создания папки `.git` (в macOS и Linux необходимо выполнить команду `mkdir` вместо `md`):

```
C:\Users\Al>md wizcoin
C:\Users\Al>cd wizcoin
C:\Users\Al\wizcoin>git init
Initialized empty Git repository in C:/Users/Al/wizcoin/.git/
```

Когда вы превращаете папку в репозиторий Git командой `git init`, все файлы в ней в исходном состоянии являются неотслеживаемыми. Для папки `wizcoin` команда `git init` создает папку `wizcoin/.git` с метаданными репозитория Git. Наличие папки `.git` превращает папку в репозиторий Git; без нее остается просто набор файлов с исходным кодом в обычной папке. Файлы никогда не следует изменять непосредственно в папке `.git`, поэтому на нее можно не обращать внимания. Собственно, имя `.git` присваивается ей из-за того, что многие операционные системы автоматически скрывают папки и файлы, имена которых начинаются с точки.

Теперь в вашем рабочем каталоге `C:\Users\AI\wizcoin` создан репозиторий. Репозиторий на вашем компьютере называется *локальным*; репозиторий на другом компьютере называется *удаленным*. Эти различия важны, потому что сохраненные данные часто приходится размещать как в локальных, так и в удаленных репозиториях, чтобы вы могли работать с другими разработчиками над одним проектом.

ВЫПОЛНЕНИЕ КОМАНДЫ `git status` КОМАНДОЙ `watch`

При использовании средств командной строки `git` часто приходится выполнять команду `git status` для получения информации о статусе репозитория. Вместо того чтобы вводить эту команду вручную, можно воспользоваться командой `watch`. Команда `watch` выполняет заданную команду через каждые две секунды и обновляет экран новыми результатами.

Для Windows команду `watch` можно загрузить со страницы <https://invent-withpython.com/watch.exe> и поместить ее в папку из PATH (например, `C:\Windows`). Пользователи macOS могут зайти на сайт <https://www.macports.org/>, загрузить и установить MacPorts, а затем выполнить команду `sudo ports install watch`. В системе Linux команда `watch` уже установлена. Когда установка будет завершена, откройте новое окно командной строки или терминала, перейдите в папку проекта репозитория Git командой `cd` и выполните команду `watch "git status"`. Команда `watch` выполняет `git status` каждые две секунды и выводит полученные результаты на экран. Окно можно оставить открытым, пока вы используете командную строку Git в другом окне терминала, чтобы наблюдать за изменениями статуса репозитория в реальном времени. Вы можете открыть другое окно терминала и выполнить команду `watch "git log -online"`, чтобы просмотреть сводку вносимых изменений (также обновляемую в реальном времени). С этой информацией вам не придется гадать, что вводимые вами команды Git делают с репозиторием.

После того как репозиторий будет создан, команда `git` может использоваться для добавления файлов и отслеживания изменений в рабочем каталоге. Выполнив команду `git status` в созданном репозитории, вы получите следующую информацию:

```
C:\Users\Al\wizcoin>git status
On branch master
```

```
No commits yet
```

```
nothing to commit (create/copy files and use "git add" to track)
```

Из выходных данных команды видно, что в репозитории еще ничего не сохранено.

Добавление файлов для отслеживания

Только отслеживаемые файлы можно сохранять, откатывать к предыдущей версии или выполнять иные операции командой `git`. Выполните команду `git status`, чтобы просмотреть статус файлов в папке проекта:

```
C:\Users\Al\wizcoin>git status
On branch master
```

```
No commits yet
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
.coveragerc
.gitignore
LICENSE.txt
README.md
```

```
--snip--
tox.ini
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Ни один файл в папке `wizcoin` в настоящее время не отслеживается ❶. Чтобы отслеживать их, следует выполнить исходное сохранение этих файлов. Процесс сохранения состоит из двух этапов: выполнения команды `git add` для каждого сохраняемого файла и последующего выполнения команды `git commit` для создания коммитов всех этих файлов. После того как файл будет сохранен, Git начинает его отслеживать.

Команда `git add` переводит файлы из неотслеживаемого или измененного состояния в индексируемое состояние. Можно выполнить команду `git add` для каждого файла, который нужно проиндексировать (например, `git add .coveragerc`,

`git add .gitignore`, `git add LICENSE.txt` и т. д.), но это утомительно. Лучше воспользоваться подстановочным символом `*` для добавления сразу нескольких файлов. Например, команда `git add *.py` добавляет все файлы `.py` из текущего рабочего каталога и его подкаталогов. Чтобы добавить все неотслеживаемые файлы, используйте точку (`.`):

```
C:\Users\Al\wizcoin>git add .
```

Выполните команду `git status`, чтобы просмотреть список проиндексированных файлов:

```
C:\Users\Al\wizcoin>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   .coveragerc
    new file:   .gitignore
--snip--
    new file:   tox.ini
```

Вывод команды `git status` сообщает, какие файлы были проиндексированы для коммита при следующем выполнении команды `git commit` ❶. Также из него можно узнать, что это новые файлы, добавленные в репозиторий ❷, а не существующие файлы в репозитории, которые были изменены.

После выполнения команды `git add` для выбора файлов, добавляемых в репозиторий, выполните команду `git commit -m "Adding new files to the repo."` (или с другим похожим сообщением). Потом снова выполните команду `git status`, чтобы просмотреть отчет о статусе репозитория:

```
C:\Users\Al\wizcoin>git commit -m "Adding new files to the repo."
[master (root-commit) 65f3b4d] Adding new files to the repo.
15 files changed, 597 insertions(+)
create mode 100644 .coveragerc
create mode 100644 .gitignore
--snip--
create mode 100644 tox.ini

C:\Users\Al\wizcoin>git status
On branch master
nothing to commit, working tree clean
```

Обратите внимание: файлы, перечисленные в файле `.gitignore`, не включаются в индексирование. Об этом я расскажу в следующем разделе.

Игнорирование файлов в репозитории

Файлы, не отслеживаемые Git, отображаются как неотслеживаемые при выполнении команды `git status`. Однако в процессе написания кода некоторые файлы можно исключить из системы контроля версий, чтобы предотвратить их случайное отслеживание. К этой категории относятся:

- временные файлы в папке проекта;
- файлы `.рус`, `.pyo` и `.pyd`, генерируемые интерпретатором Python при выполнении программ `.py`;
- папки `.tox`, `htmlcov` и другие папки, генерируемые различными средствами разработчика;
- другие откомпилированные или сгенерированные файлы, которые можно сгенерировать заново (потому что репозиторий предназначен для исходных файлов, а не для производных файлов, которые генерируются на их основе);
- файлы с исходным кодом, содержащие пароли баз данных, маркеры аутентификации, номера кредитных карт или другие конфиденциальные данные.

Чтобы предотвратить включение этих файлов, создайте текстовый файл с именем `.gitignore` и перечислите в нем файлы и папки, которые не должны отслеживаться Git. Git автоматически исключает их из команд `git add` или `git commit`, и они не будут отображаться при выполнении команды `git status`.

Файл `.gitignore`, созданный шаблоном `cookiecutter-basicpythonproject`, выглядит примерно так:

```
ates looks like this:
# Байт-компилируемые / оптимизированные / файлы DLL
__pycache__/
*.py[cod]
*$py.class
--snip--
```

В файле `.gitignore` символ `*` используется для шаблонов, а `#` — для комментариев. За дополнительной информацией обращайтесь к электронной документации по адресу <https://git-scm.com/docs/gitignore>.

Файл `.gitignore` также следует добавить в репозиторий Git, чтобы он был у других программистов, клонировавших ваш репозиторий. Если вы хотите видеть, какие файлы в вашем рабочем каталоге игнорируются на основании настроек в `.gitignore`, выполните команду `git ls-files --other --ignored --exclude-standard`.

Сохранение изменений

После добавления новых файлов в репозиторий вы можете продолжить писать код для вашего проекта. Когда потребуется создать очередной коммит, выполните команду `git add .` для индексирования всех измененных файлов и команду `git commit -m <сообщение>` для сохранения всех индексированных файлов. Впрочем, это проще делается одной командой `git commit -am <сообщение>`:

```
C:\Users\Al\wizcoin>git commit -am "Fixed the currency conversion bug."
[master (root-commit) e1ae3a3] Fixed the currency conversion bug.
 1 file changed, 12 insertions(+)
```

Если вы хотите сохранить только некоторые (но не все) измененные файлы, не добавляйте ключ `-a` в `-am` и перечислите файлы после сообщения — например, `git commit -m <сообщение> file1.py file2.py`.

Сообщение **Fixed the currency conversion bug** (Исправлена ошибка конвертации валюты) содержит подсказку на будущее: оно напоминает, какие изменения были внесены в этом коммите. Не поддавайтесь искушению написать короткое обобщенное сообщение вида «Обновленный код», «Исправлены некоторые ошибки» или просто «х» (потому что пустые сообщения запрещены). Через три недели, когда вам захочется вернуться к более ранней версии вашего кода, подробные сообщения в каждом коммите сэкономят вам немало времени, когда вы будете выбирать, к какой именно версии следует вернуться.

Если вы забудете добавить аргумент командной строки `-m "<сообщение>"`, Git откроет текстовый редактор Vim в окне терминала. Описание Vim выходит за рамки книги, поэтому нажмите клавишу `Esc` и введите `qa!`, чтобы безопасно завершить Vim и отменить коммит. Затем снова введите команду `git commit`, на этот раз с аргументом командной строки `-m "<сообщение>"`.

Чтобы посмотреть примеры профессиональных сообщений коммитов, обращайтесь к истории коммитов веб-фреймворка Django (<https://github.com/django/django/commits/master>). Так как Django является большим проектом с открытым кодом, коммиты выполняются часто, а сообщения коммитов формализованы.

Редкие коммиты с невразумительными сообщениями могут неплохо работать в небольших персональных проектах, но над Django работает более 1000 участников. Некачественные сообщения коммитов от любого участника создадут проблемы для всех.

Файлы были безопасно сохранены в репозитории Git. Еще раз выполните команду `git status`, чтобы просмотреть их статус:


```
C:\Users\Al\wizcoin>git status
On branch master
nothing to commit, working tree clean
```

Сохранив индексированные файлы, вы вернули их в сохраненное состояние, и Git говорит, что рабочее дерево чисто; другими словами, в нем нет измененных или индексированных файлов.

Напомню, что при добавлении файлов в репозиторий Git файлы перешли из неотслеживаемого состояния в индексированное, а затем в сохраненное. Теперь файлы готовы к будущим изменениям.

Заметим, что в репозитории Git нельзя сохранять папки. Git автоматически включает папки в репозитории при сохранении хранящихся в них файлов, но сохранить пустую папку не получится.

Если вы допустили ошибку в последнем сообщении коммита, его можно переписать командой `git commit --amend -m "<новое_сообщение>"`.

Просмотр изменений перед коммитом

Прежде чем сохранять код, следует быстро просмотреть изменения, которые будут сохранены при выполнении команды `git commit`. Для просмотра различий между рабочей копией кода и последним сохраненным кодом можно воспользоваться командой `git diff command`.

Рассмотрим пример использования `git diff`. Откройте файл `README.md` в текстовом редакторе или IDE. (Вы должны были создать этот файл при запуске `Cookiecutter`. Если он не существует, создайте пустой текстовый файл и сохраните его под именем `README.md`.) Это файл с разметкой Markdown, но, как сценарий Python, он должен быть записан в формате простого текста. Замените текст `TODO - fill this in later` в приведенном ниже тексте из раздела «Quickstart Guide» следующим фрагментом (пока не исправляйте ошибку `xample`; мы сделаем это позднее):

```
Quickstart Guide
-----
```

Here's some xample code demonstrating how this module is used:

```
>>> import wizcoin
>>> coin = wizcoin.WizCoin(2, 5, 10)
>>> str(coin)
'2g, 5s, 10k'
>>> coin.value()
1141
```

Прежде чем добавить и сохранить файл `README.md`, выполните команду `git diff` для просмотра внесенных изменений:

```
C:\Users\Al\wizcoin>git diff
diff --git a/README.md b/README.md
index 76b5814..3be49c3 100644
--- a/README.md
+++ b/README.md
@@ -13,7 +13,14 @@ To install with pip, run:
 Quickstart Guide
-----

-TODO - fill this in later
+Here's some xample code demonstrating how this module is used:
+
+  >>> import wizcoin
+  >>> coin = wizcoin.WizCoin(2, 5, 10)
+  >>> str(coin)
+  '2g, 5s, 10k'
+  >>> coin.value()
+  1141

Contribute
-----
```

Результат показывает, что файл `README.md` в рабочей копии изменился по сравнению с файлом `README.md`, существовавшим при последнем сохранении репозитория. Строки, начинающиеся со знака `-`, были удалены; строки, начинающиеся со знака `+`, были добавлены.

В процессе просмотра изменений также можно заметить опечатку — «xample» вместо «example». Исправим ее, а потом снова выполним команду `git diff`, чтобы проверить изменения и сохранить их в репозитории:

```
C:\Users\Al\wizcoin>git diff
diff --git a/README.md b/README.md
index 76b5814..3be49c3 100644
--- a/README.md
+++ b/README.md
@@ -13,7 +13,14 @@ To install with pip, run:
 Quickstart Guide
-----

-TODO - fill this in later
+Here's some example code demonstrating how this module is used:
+--snip--
C:\Users\Al\wizcoin>git add README.md

C:\Users\Al\wizcoin>git commit -m "Added example code to README.md"
[master 2a4c5b8] Added example code to README.md
 1 file changed, 8 insertions(+), 1 deletion(-)
```

Исправление сохранено в репозитории.

Просмотр изменений в графическом приложении командой `git difftool`

Изменения проще просматривать в программе с графическим интерфейсом. Для Windows можно загрузить WinMerge (<https://winmerge.org/>) — бесплатную программу с открытым кодом. В Linux можно установить либо Meld командой `sudo apt-get install meld`, либо Kompare командой `sudo apt-get install kompare`. В macOS для установки программы tkdiff начните с команд, которые устанавливают и настраивают Homebrew (менеджер пакетов для установки программ), а затем воспользуйтесь Homebrew для установки tkdiff:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
brew install tkdiff
```

Чтобы настроить Git для использования этих программ, выполните команду `git config diff.tool <название>`, где `<название>` — winmerge, tkdiff, meld или kompare. Затем выполните команду `git difftool <имя_файла>` для просмотра изменений, внесенных в файл, в графическом интерфейсе программы (рис. 12.5).

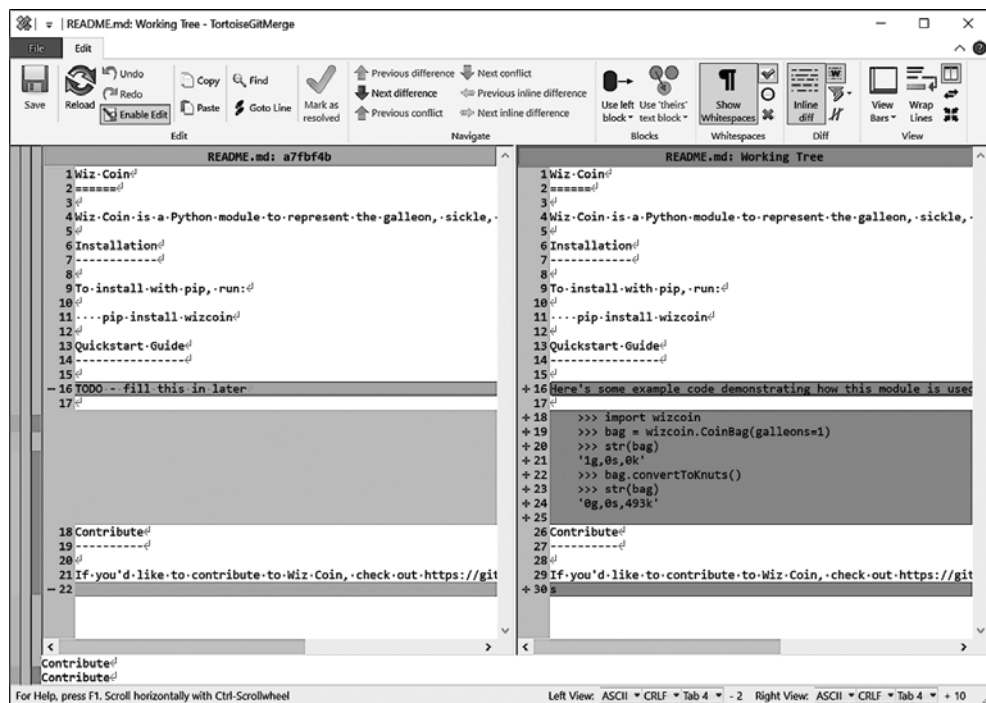


Рис. 12.5. Просматривать результаты в программе с графическим интерфейсом (WinMerge в данном случае) удобнее, чем в текстовом выводе `git diff`

Кроме того, выполните команду `git config --global difftool.prompt false`, чтобы система Git не запрашивала подтверждения каждый раз, когда вы хотите запустить программу просмотра изменений. Если вы установили Git-клиент с графическим интерфейсом, вы также можете настроить его для использования этих инструментов (а может быть, он содержит собственные визуальные средства сравнения).

Частота сохранения изменений

Хотя системы контроля версий позволяют вернуть файлы к более раннему коммиту, может возникнуть вопрос, насколько часто следует сохранять изменения. Если делать это слишком часто, вам придется пробиваться через множество незначительных комментариев, чтобы найти нужную версию кода. Если делать редко, то каждый коммит будет содержать множество изменений и возврат к конкретной версии отменит больше изменений, чем вам хотелось бы. На практике программисты обычно выполняют коммиты реже, чем следовало бы.

Код следует сохранять при завершении блока функциональности, класса или исправления ошибки. Не сохраняйте код, содержащий синтаксические ошибки или очевидно неработоспособный. Коммиты могут состоять из нескольких строк или нескольких сотен строк измененного кода, но в любом случае вы должны иметь возможность вернуться к более ранней копии и все еще иметь в распоряжении рабочую программу. Всегда выполняйте все модульные тесты перед сохранением. В идеале все тесты должны проходить (а если они не проходят, упомяните об этом в сопроводительном сообщении).

Удаление файлов из репозитория

Если вам не нужно, чтобы какой-либо файл отслеживался в Git, вы не можете просто удалить его из файловой системы. Это необходимо сделать через Git командой `git rm`, которая также приказывает Git перестать отслеживать файл. Чтобы потренироваться в выполнении этой операции, выполните команду `echo "Test file" > deleteme.txt`, чтобы создать маленький файл с именем `deleteme.txt` и содержимым "Test file". Затем сохраните файл в репозитории следующими командами:

```
C:\Users\Al\wizcoin>echo "Test file" > deleteme.txt
C:\Users\Al\wizcoin>git add deleteme.txt
C:\Users\Al\wizcoin>git commit -m "Adding a file to test Git deletion."
[master 441556a] Adding a file to test Git deletion.
 1 file changed, 1 insertion(+)
 create mode 100644 deleteme.txt
C:\Users\Al\wizcoin>git status
On branch master
nothing to commit, working tree clean
```

Не удаляйте файлы командой `del` в Windows или командой `rm` в macOS и Linux. (А если вы это сделаете, файл можно восстановить командой `git restore <имя_файла>` или просто перейти к команде `git rm`, чтобы удалить файл из репозитория.) Вместо этого воспользуйтесь командой `git rm`, чтобы удалить и проиндексировать файл `deleteme.txt`, как это делается в следующем примере:

```
C:\Users\Al\wizcoin>git rm deleteme.txt
rm deleteme.txt'
```

Команда `git rm` удаляет файл из рабочей копии, но это еще не все. Как и `git add`, команда `git rm` индексирует файл. Удаление файла необходимо закрепить точно так же, как любые другие изменения:

```
C:\Users\Al\wizcoin>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    deleteme.txt
C:\Users\Al\wizcoin>git commit -m "Deleting deleteme.txt from the repo to
finish the deletion test."
[master 369de78] Deleting deleteme.txt from the repo to finish the deletion
test.
 1 file changed, 1 deletion(-)
 delete mode 100644 deleteme.txt
C:\Users\Al\Desktop\wizcoin>git status
On branch master
nothing to commit, working tree clean
```

Хотя файл `deleteme.txt` был удален из рабочей копии, он все еще существует в истории репозитория. В разделе «Восстановление старых изменений» этой главы рассказано, как восстановить удаленный файл или отменить изменение.

Команда `git rm` работает только с файлами, находящимися в чистом, сохраненном, состоянии без каких-либо изменений. В противном случае Git предложит сохранить изменения или отменить их командой `git reset HEAD <имя_файла>`. (Вывод `git status` напомнит вам об этой команде ❶.) Такая процедура предотвращает случайное удаление несохраненных изменений.

Переименование и перемещение файлов из репозитория

Как и при удалении файлов, вы не должны переименовывать или перемещать файлы в репозитории в обход Git. В противном случае Git решит, что вы просто удалили файл, и создаст новый файл с прежним содержимым. Вместо этого используйте команду `git mv` с последующей командой `git commit`.

Переименуйте README.md в README.txt следующими командами:

```
C:\Users\Al\wizcoin>git mv README.md README.txt
C:\Users\Al\wizcoin>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README.md -> README.txt
```

```
C:\Users\Al\wizcoin>git commit -m "Testing the renaming of files in Git."
[master 3fee6a6] Testing the renaming of files in Git.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README.md => README.txt (100%)
```

В этом случае история изменений README.txt также включает историю README.md.

Также можно воспользоваться командой `git mv` для перемещения файла в новую папку. Введите следующие команды, чтобы создать новую папку с именем `movetest` и переместить в нее файл `README.txt`:

```
C:\Users\Al\wizcoin>mkdir movetest
C:\Users\Al\wizcoin>git mv README.txt movetest/README.txt
C:\Users\Al\wizcoin>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README.txt -> movetest/README.txt
```

```
C:\Users\Al\wizcoin>git commit -m "Testing the moving of files in Git."
[master 3ed22ed] Testing the moving of files in Git.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README.txt => movetest/README.txt (100%)
```

Также можно совместить переименование файла с перемещением — для этого следует передать `git mv` новое имя и местоположение файла. Вернем файл `README.txt` в корневой рабочий каталог с восстановлением исходного имени:

```
C:\Users\Al\wizcoin>git mv movetest/README.txt README.md
C:\Users\Al\wizcoin>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    movetest/README.txt -> README.md
```

```
C:\Users\Al\wizcoin>git commit -m "Moving the README file back to its original
place and name."
[master 962a8ba] Moving the README file back to its original place and name.
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
rename movetest/README.txt => README.md (100%)
```

Хотя файл `README.md` вернулся в исходную папку и имеет исходное имя, репозиторий Git запоминает изменения имени и местоположения. Историю изменений можно вызвать командой `git log`, о которой пойдет речь в следующем разделе.

Просмотр журнала коммитов

Команда `git log` выводит список всех коммитов:

```
C:\Users\Al\wizcoin>git log
commit 962a8baa29e452c74d40075d92b00897b02668fb (HEAD -> master)
Author: Al Sweigart <al@inventwithpython.com>
Date:   Wed Sep 1 10:38:23 2021 -0700
```

Moving the README file back to its original place and name.

```
commit 3ed22ed7ae26220bbd4c4f6bc52f4700dbb7c1f1
Author: Al Sweigart <al@inventwithpython.com>
Date:   Wed Sep 1 10:36:29 2021 -0700
```

Testing the moving of files in Git.

--snip--

Команда способна выводить большой объем текста. Если журнал не помещается в окне терминала, текст можно прокрутить клавишами `↑` и `↓`. Чтобы завершить просмотр, нажмите клавишу `q`.

Если вы хотите вернуть файлы к более раннему коммиту, сначала следует найти *хеш коммита* — строку из 40 шестнадцатеричных цифр (0–9 и буквы A–F), которая служит уникальным идентификатором коммита. Например, последний коммит в нашем репозитории представлен хешем `962a8baa29e452c74d40075d92b00897b02668fb`. На практике обычно используются только первые семь знаков: `962a8ba`.

Со временем журнал может стать очень длинным. Ключ `--oneline` усекает вывод до сокращенных хешей и первой строки каждого сообщения коммита. Введите команду `git log --oneline` в командной строке:

```
C:\Users\Al\wizcoin>git log --oneline
962a8ba (HEAD -> master) Moving the README file back to its original place and
name.
3ed22ed Testing the moving of files in Git.
15734e5 Deleting deleteme.txt from the repo to finish the deletion test.
441556a Adding a file to test Git deletion.
2a4c5b8 Added example code to README.md
e1ae3a3 An initial add of the project files.
```

Если вывод остается слишком длинным, используйте ключ `-n` для ограничения вывода *n* последними коммитами. Введите команду `git log --oneline -n 3`, чтобы просмотреть только три последних коммита:

```
C:\Users\Al\wizcoin>git log --oneline -n 3
962a8ba (HEAD -> master) Moving the README file back to its original place and
name.
3ed22ed Testing the moving of files in Git.
15734e5 Deleting deleteme.txt from the repo to finish the deletion test.
```

Чтобы вывести содержимое файла на момент конкретного коммита, можно задать команду `git show <хеш>:<имя_файла>`. Впрочем, графические средства Git предоставляют более удобный интерфейс для просмотра журнала, чем командная строка Git.

Восстановление старых изменений

Допустим, вы хотите вернуться к более ранней версии своего исходного кода, потому что в программе была допущена ошибка или вы случайно удалили файл. Система контроля версий позволяет вернуть рабочую копию к состоянию более раннего коммита. Конкретная команда зависит от состояния файлов в рабочей копии.

Помните, что системы контроля версий только добавляют информацию. Даже при удалении файла из репозитория Git запомнит его, чтобы его можно было восстановить в будущем. Отмена изменения в действительности добавляет новое изменение, которое возвращает файл к его состоянию при предыдущем коммите. Подробная информация о различных видах отмены доступна на <https://github.blog/2015-06-08-how-to-undo-almost-anything-with-git/>.

Отмена несохраненных локальных изменений

Если вы внесли в файл несохраненные изменения, но хотите вернуть его к версии в последнем коммите, выполните команду `git restore <имя_файла>`. В следующем примере мы изменяем файл `README.md`, но не индексируем и не сохраняем его:

```
C:\Users\Al\wizcoin>git status
On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
C:\Users\Al\wizcoin>git restore README.md
```



```
C:\Users\Al\wizcoin>git status
On branch master
Your branch is up to date with 'origin/master'.
```

```
nothing to commit, working tree clean
```

После выполнения команды `git restore README.md` содержимое файла `README.md` становится таким, как в последнем коммите. Фактически это операция отмены изменений, внесенных в файл (который еще не был проиндексирован или сохранен). Но будьте внимательны: вы уже не сможете отменить эту «отмену», чтобы вернуть последние изменения. Также можно выполнить команду `git checkout .`, чтобы отменить все изменения во всех файлах рабочей копии.

Деиндексирование проиндексированного файла

Если вы проиндексировали измененный файл командой `git add`, а теперь хотите исключить его из индексированного состояния, чтобы он не был включен в следующий коммит, выполните команду `git restore --staged <имя_файла>`:

```
C:\Users\Al>git restore --staged README.md
Unstaged changes after reset:
M      spam.txt
```

Файл `README.md` остается измененным, как это было до его индексирования командой `git add`, но он не находится в индексированном состоянии.

Отмена последних коммитов

Допустим, вы сделали несколько бесполезных коммитов и теперь хотите вернуться к предыдущему коммиту. Чтобы отменить конкретное число последних коммитов (например, 3), используйте команду `git revert -n HEAD~3..HEAD`. Вместо 3 можно указать любое количество коммитов. Предположим, вы отслеживаете изменения в детективном романе, который вы пишете, и у вас имеется следующий журнал Git со всеми коммитами и сообщениями:

```
C:\Users\Al\novel>git log --oneline
de24642 (HEAD -> master) Changed the setting to outer space.
2be4163 Added a whacky sidekick.
97c655e Renamed the detective to 'Snuggles'.
8aa5222 Added an exciting plot twist.
2590860 Finished chapter 1.
2dece36 Started my novel.
```

В какой-то момент вы решаете, что хотите начать заново с сюжетного поворота с хешем `8aa5222`. Это означает, что нужно отменить изменения трех последних коммитов: `de24642`, `2be4163` и `97c655e`. Выполните команду `git revert -n HEAD~3..`

HEAD, чтобы отменить эти изменения, а затем выполните команды `git add .` и `git commit -m "<сообщение>"` для сохранения контента, как и для любого другого изменения:

```
C:\Users\Al\novel>git revert -n HEAD~3..HEAD
```

```
C:\Users\Al\novel>git add .
```

```
C:\Users\Al\novel>git commit -m "Starting over from the plot twist."
[master faec20e] Starting over from the plot twist.
 1 file changed, 34 deletions(-)
```

```
C:\Users\Al\novel>git log --oneline
faec20e (HEAD -> master) Starting over from the plot twist.
de24642 Changed the setting to outer space.
2be4163 Added a whacky sidekick.
97c655e Renamed the detective to 'Snuggles'.
8aa5222 Added an exciting plot twist.
2590860 Finished chapter 1.
2dece36 Started my novel.
```

Репозитории Git обычно только добавляют информацию, поэтому при отмене коммитов они остаются в истории коммитов. Если потребуется «отменить отмену», вы можете снова вернуться к нужному состоянию командой `git revert`.

Возврат к конкретному коммиту для отдельного файла

Так как коммиты отражают состояние всего репозитория, а не отдельных файлов, то вам потребуется другая команда, если вы захотите отменить изменения для отдельного файла. Допустим, я веду репозиторий Git для небольшого программного проекта. Я создал файл `eggs.py`, добавил в него функции `spam()` и `bacon()`, а затем переименовал `bacon()` в `cheese()`. Журнал репозитория будет выглядеть примерно так:

```
C:\Users\Al\myproject>git log --oneline
895d220 (HEAD -> master) Adding email support to cheese().
df617da Renaming bacon() to cheese().
ef1e4bb Refactoring bacon().
ac27c9e Adding bacon() function.
009b7c0 Adding better documentation to spam().
0657588 Creating spam() function.
d811971 Initial add.
```

Но я решил, что я хочу вернуться к файлу до добавления функции `bacon()`, но не изменять любые другие файлы в репозитории. Можно воспользоваться командой `git show <хеш>: <имя_файла>` для вывода этого файла на момент последнего конкретного коммита. Команда выглядит примерно так:

```
C:\Users\A1\myproject>git show 009b7c0:eggs.py
<содержимое eggs.py на момент сохранения 009b7c0>
```

При помощи команды `git checkout <хеш> -- <имя_файла>` можно вернуть содержимое `eggs.py` к этой версии и сохранить измененный файл обычным способом. Команда `git checkout` изменяет только рабочую копию. Вам останется только проиндексировать и сохранить эти коррективы, как и любые другие:

```
C:\Users\A1\myproject>git checkout 009b7c0 -- eggs.py
```

```
C:\Users\A1\myproject>git add eggs.py
```

```
C:\Users\A1\myproject>git commit -m "Rolled back eggs.py to 009b7c0"
[master d41e595] Rolled back eggs.py to 009b7c0
1 file changed, 47 deletions(-)
```

```
C:\Users\A1\myproject>git log --oneline
d41e595 (HEAD -> master) Rolled back eggs.py to 009b7c0
895d220 Adding email support to cheese().
df617da Renaming bacon() to cheese().
ef1e4bb Refactoring bacon().
ac27c9e Adding bacon() function.
009b7c0 Adding better documentation to spam().
0657588 Creating spam() function.
d811971 Initial add.
```

Файл `eggs.py` был возвращен к прежнему состоянию, а оставшаяся часть репозитория осталась неизменной.

Перезапись истории коммитов

Если вы случайно сохранили файл, содержащий конфиденциальную информацию (пароли, ключи API, номера кредитных карт), недостаточно вычеркнуть эту информацию и создать новый коммит. Каждый, кто имеет доступ к репозиторию на вашем компьютере или к удаленному репозиторию, сможет вернуться к версии, содержащей эту информацию.

Удалить информацию из репозитория так, чтобы ее было невозможно восстановить, непросто, но возможно. Подробно рассказывать об этом здесь я не буду, но вы можете воспользоваться либо командой `git filter-branch`, либо программой BFG Repo-Cleaner (этот вариант считается предпочтительным). Оба варианта описаны на <https://help.github.com/en/articles/removing-sensitive-data-from-a-repository>.

Простейшая превентивная мера — разместить конфиденциальную информацию в файле с именем `secrets.txt`, `confidential.py` или что-нибудь в этом роде. Файл включается в `.gitignore`, чтобы он никогда не был сохранен в репозитории. Ваша программа

может прочитать конфиденциальную информацию из файла — это лучше, чем размещать такую информацию непосредственно в исходном коде.

GitHub и команда `git push`

Хотя репозитории Git могут существовать на вашем компьютере, многие бесплатные веб-сайты позволяют размещать клоны ваших репозиторий в интернете, чтобы другие люди могли легко загрузить ваши проекты и участвовать в работе над ними. Самый большой из таких сайтов — GitHub. Если вы сохраните клон своего проекта в интернете, коллеги смогут дополнять ваш код, даже когда компьютер, на котором вы работаете, отключен. Кроме того, клонированная копия фактически выполняет роль резервной копии.

ПРИМЕЧАНИЕ

Чтобы избежать путаницы с терминами: Git — система контроля версий, которая поддерживает репозиторий и включает команду `git`. GitHub — веб-сайт для размещения репозиторий Git в интернете.

Owner: asweigart / Repository name *: wizcoin ✓

Great repository names are short and memorable. Need inspiration? How about [super-octo-couscous](#)?

Description (optional): A Python module to represent the galleon, sickle, and knut coins of wizard currency.

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: None | Add a license: None ⓘ

Create repository

Рис. 12.6. Создание нового репозитория на GitHub

Зайдите на сайт <https://github.com> и зарегистрируйтесь для получения бесплатной учетной записи. На домашней странице GitHub или на вкладке **Repositories** страницы вашего профиля щелкните на кнопке **New**, чтобы создать новый проект. Введите имя репозитория `wizcoin` и описание проекта — об этом я рассказывал в разделе «Создание новых проектов Python с использованием Cookiecutter» на с. 236 (рис. 12.6). Пометьте репозиторий как общедоступный (**Public**) и снимите флажок **Initialize this repository with a README**, потому что мы импортируем существующий репозиторий. Щелкните на кнопке **Create repository**. Все эти действия эквивалентны выполнению команды `git init` на веб-сайте GitHub.

Веб-страница для ваших репозиториях будет располагаться по адресу https://github.com/<имя_пользователя>/<имя_репозитория>. Мой репозиторий `wizcoin` размещается на <https://github.com/asweigart/wizcoin>.

Отправка существующего репозитория на GitHub

Чтобы отправить существующий репозиторий из командной строки, введите следующие команды:

```
C:\Users\Al\wizcoin>git remote add origin https://github.com/<пользователь_github>/wizcoin.git
C:\Users\Al\wizcoin>git push -u origin master
Username for 'https://github.com': <пользователь_github>
Password for 'https://<github_username>@github.com': <пароль_github>
Counting objects: 3, done.
Writing objects: 100% (3/3), 213 bytes | 106.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/<ваш_github>/wizcoin.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Команда `git remote add origin https://github.com/<пользователь_github>/wizcoin.git` добавляет GitHub как удаленный репозиторий, соответствующий вашему локальному репозиторию. После этого вы можете отправить все изменения, внесенные в локальном репозитории, в удаленный командой `git push -u origin master`. Следующие отправки из локального репозитория вы сможете осуществлять простой командой `git push`. Отправка копий на GitHub после каждого коммита — хорошая тактика, гарантирующая синхронизацию удаленного репозитория на GitHub с вашим локальным репозиторием, но она не обязательна.

Загрузив веб-страницу репозитория на GitHub, вы должны получить информацию о файлах и коммитах. Конечно, это далеко не все, что можно узнать о GitHub — включая и то, как принимать действия других людей в ваших репозиториях посредством pull-запросов. Эта тема, наряду с другими расширенными возможностями GitHub, выходит за рамки книги.

Клонирование существующего репозитория GitHub

Также возможно и обратное: создать новый репозиторий на GitHub и клонировать его на ваш компьютер. Создайте новый репозиторий на веб-сайте GitHub, но на этот раз установите флажок **Initialize this repository with a README**.

Чтобы клонировать этот репозиторий на локальный компьютер, перейдите на страницу репозитория на GitHub и щелкните на кнопке **Clone** или **Download**; откроется окно с URL-адресом вида https://github.com/<пользователь_github>/wizcoin.git. Используйте URL-адрес своего репозитория с командой `git clone`, чтобы загрузить его на ваш компьютер:

```
C:\Users\Al>git clone https://github.com/<пользователь_github>/wizcoin.git
Cloning into 'wizcoin'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
```

Теперь вы сможете сохранять и отправлять изменения, используя этот репозиторий Git, точно так же, как если бы он был создан командой `git init`.

Команда `git clone` также пригодится в том случае, если ваш локальный репозиторий оказался в состоянии, когда вы просто не знаете, что с ним делать и как отказаться от последних изменений. И хотя такое решение далеко не идеально, вы всегда можете сохранить копию файлов в вашем рабочем каталоге, удалить локальный репозиторий и воспользоваться командой `git clone` для повторного создания репозитория. В такой ситуации порой оказываются даже опытные разработчики, и это легло в основу шутки <https://xkcd.com/1597/>.

Итоги

Системы контроля версий спасают программистов от многих бед. Сохранение кода упрощает анализ хода работы над проектом и в некоторых случаях позволяет отменять нежелательные изменения. Изучение основ работы с системой контроля версий — такой как Git — безусловно, сэкономит ваше время в долгосрочной перспективе.

Проекты Python обычно состоят из нескольких стандартных файлов и папок, и модуль `cookiecutter` помогает создать заготовки кода для многих таких файлов. Они станут первыми из сохраненных в вашем локальном репозитории Git. Папка, содержащая весь этот контент, называется *рабочим каталогом* или *папкой проекта*.

Git отслеживает файлы в рабочем каталоге. Каждый файл может существовать в одном из трех состояний: сохраненном (или чистом), измененном или индексированном. Командная строка Git поддерживает ряд команд (например, `git status` или `git log`) для просмотра этой информации, но вы также можете воспользоваться сторонними средствами с графическим интерфейсом.

Команда `git init` создает новый пустой репозиторий на вашем локальном компьютере. Команда `git clone` копирует репозиторий с удаленного сервера (например, с популярного веб-сайта GitHub). После создания репозитория вы можете воспользоваться командами `git add` и `git commit` для сохранения изменений в репозитории и командой `git push` для отправки коммитов в удаленный репозиторий GitHub. В этой главе я рассказал и о командах для отмены внесенных изменений. Отмена позволяет вернуться к более ранней версии ваших файлов.

Git — сложный инструмент со множеством возможностей, и эта глава знакомит вас только с основами системы контроля версий. Существует множество ресурсов для изучения расширенной функциональности Git. Я рекомендую две бесплатные книги, которые можно найти в интернете: «Pro Git» Скотта Чаркона (Scott Charcon) (<https://git-scm.com/book/en/v2>)¹ и «Version Control by Example» Эрика Синка (Eric Sink) (<https://ericsink.com/vcbe/index.html>).

¹ Существует версия этой книги на русском языке: <http://git-scm.com/book/ru/v2>. — *Примеч. ред.*

13

Измерение быстродействия и анализ сложности алгоритмов



Для многих небольших программ быстродействие не так уж важно. Можно потратить целый час на написание сценария для автоматизации задачи, который выполняется за несколько секунд. Даже если выполнение потребует больше времени, то программа, вероятно, завершится к тому моменту, когда вы вернетесь к столу с чашкой кофе.

Иногда стоит озадачиться ускорением работы сценария. Но чтобы понять, привели ли изменения к повышению быстродействия, необходимо знать, как измерить скорость программы. На помощь приходят модули Python `timeit` и `cProfile`. Они не только изменяют время выполнения кода, но и строят *профиль*, показывающий, какие части кода уже выполняются быстро, а какие можно улучшить.

Кроме измерения скорости программ, из этой главы вы также узнаете, как оценивать теоретический рост времени выполнения с увеличением объема данных. В компьютерной науке эта зависимость называется *нотацией «О-большое»*¹. Разработчик без подготовки в области традиционной теории обработки данных иногда осознаёт, что в его знаниях есть пробелы. Теория важна, но она не всегда имеет прямое отношение к реальной разработке. Я шучу (но только наполовину), что нотация «О-большое» составляет около 80% полезности моего диплома. В этой главе я кратко познакомлю вас с этой темой, имеющей значительное практическое применение.

¹ В современных статьях и в интернете вы часто встретите термин Big-O-нотация. В классических учебниках по алгоритмам принято использовать обозначение «нотация «О-большое»». — *Примеч. ред.*

Модуль `timeit`

«Преждевременная оптимизация — корень всех зол» — популярное изречение в области разработки. (Его часто приписывают знаменитому ученому Дональду Кнуту, который отдает его авторство Тони Хоару. В свою очередь Тони Хоар описывает ситуацию с точностью до наоборот.) *Преждевременная оптимизация* (то есть оптимизация, выполняемая до того, как вы поймете, что же нужно оптимизировать) часто проявляется, когда программисты пытаются использовать хитроумные трюки для экономии памяти или написания более быстрого кода. Например, один из таких трюков основан на использовании алгоритма XOR, для того чтобы поменять местами два целых значения без использования третьей временной переменной:

```
>>> a, b = 42, 101 # Создание двух переменных.
>>> print(a, b)
42 101
>>> # Серия операций ^ XOR меняет значения местами:
>>> a = a ^ b
>>> b = a ^ b
>>> a = a ^ b
>>> print(a, b) # Значения поменялись местами.
101 42
```

Если вы не знакомы с алгоритмом XOR (использующим поразрядный оператор `^`), этот код выглядит загадочно. Недостаток хитроумных программных трюков заключается в том, что они приводят к появлению сложного нечитаемого кода. Вспомните один из принципов «Дзен Python»: удобочитаемость важна.

Что еще хуже, ваш хитроумный трюк может оказаться не таким уж хитроумным. Нельзя просто решить, что новый прием работает быстрее просто в силу своей затейливости или что старый код, который вы заменяете, изначально работал медленно. Узнать это можно только одним способом — измерить и сравнить время выполнения (то есть время, необходимое для выполнения программы или некоторой части кода). Помните, что увеличение времени выполнения означает замедление работы программы: ей требуется больший срок для выполнения того же объема работы.

Модуль `timeit` стандартной библиотеки Python способен измерить скорость выполнения небольшого фрагмента кода. Для этого его запускают тысячи или миллионы раз, после чего вычисляют среднее время выполнения. Модуль `timeit` также временно отключает автоматический сборщик мусора для получения более стабильных данных времени выполнения. Если вы хотите вычислить время выполнения нескольких строк, передайте многострочный текст или разделите строки кода символами `;`:

```
>>> import timeit
>>> timeit.timeit('a, b = 42, 101; a = a ^ b; b = a ^ b; a = a ^ b')
0.13077666299999998
```

```
>>> timeit.timeit("""a, b = 42, 101
... a = a ^ b
... b = a ^ b
... a = a ^ b""")
0.13515726800000039
```

На моем компьютере выполнение этого кода с алгоритмом XOR занимает приблизительно 1/10 секунды. Быстро это или нет? Сравним с кодом, меняющим местами два числа с использованием третьей временной переменной:

```
>>> import timeit
>>> timeit.timeit('a, b = 42, 101; temp = a; a = b; b = temp')
0.027540389999998638
```

Сюрприз! Код с третьей временной переменной не только лучше читается, но и работает почти вдвое быстрее! Возможно, трюк с XOR экономит несколько байтов памяти, но за счет скорости и удобочитаемости кода. Нет смысла жертвовать удобочитаемостью ради нескольких байтов памяти или наносекунд выполнения.

Еще лучше поменять местами две переменные, используя уловку *множественного присваивания*, которая также требует меньше времени:

```
>>> timeit.timeit('a, b = 42, 101; a, b = b, a')
0.024489236000007963
```

Этот вариант не только читается лучше остальных, но и работает быстрее всего. И мы знаем это не умозрительно, а потому что провели объективное измерение. Функция `timeit.timeit()` также может получить второй строкой аргумент с подготовительным кодом, который выполняется только один раз перед выполнением кода первой строки. Также можно изменить количество испытаний по умолчанию, передав целое число в ключевом аргументе `number`. Например, следующий тест измеряет, с какой скоростью модуль Python `random` генерирует 10 000 000 случайных чисел от 1 до 100. (На моей машине на это потребовалось около 10 секунд.)

```
>>> timeit.timeit('random.randint(1, 100)', 'import random', number=10000000)
10.020913950999784
```

По умолчанию код строки, переданной `timeit.timeit()`, не может обращаться к переменным и функциям в остальном коде программы:

```
>>> import timeit
>>> spam = 'hello' # Определяем переменную spam.
>>> timeit.timeit('print(spam)', number=1) # Измеряем время выполнения вывода spam.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\Al\AppData\Local\Programs\Python\Python37\lib\timeit.py",
line 232, in timeit
    return Timer(stmt, setup, timer, globals).timeit(number)
```

```
File "C:\Users\Al\AppData\Local\Programs\Python\Python37\lib\timeit.py",
line 176, in timeit
    timing = self.inner(it, self.timer)
    File "<timeit-src>", line 6, in inner
NameError: name 'spam' is not defined
```

Чтобы решить эту проблему, передадим функции возвращаемое значение `globals()` в ключевом аргументе `globals`:

```
>>> timeit.timeit('print(spam)', number=1, globals=globals())
hello
0.000994909999462834
```

Хорошее правило: сначала добейтесь того, чтобы ваш код работал, а потом занимайтесь его ускорением. Сначала работоспособность, потом эффективность!

Профилировщик cProfile

Хотя модуль `timeit` полезен для хронометража небольших фрагментов кода, модуль `cProfile` более эффективен для анализа целых функций или программ. Процесс профилирования систематически анализирует скорость вашей программы, затраты памяти и другие аспекты. Модуль `cProfile` — профилировщик Python, то есть программа, измеряющая время выполнения программы, а также строящая профиль времени выполнения отдельных вызовов функций программы. Эта информация предоставляет намного более детализированные результаты хронометража вашего кода.

Чтобы воспользоваться профилировщиком `cProfile`, передайте код, для которого хотите провести измерения, при вызове `cProfile.run()`. Следующий пример показывает, как `cProfiler` измеряет и выводит время выполнения короткой функции, суммирующей все числа от 1 до 1 000 000:

```
import time, cProfile
def addUpNumbers():
    total = 0
    for i in range(1, 1000001):
        total += i

cProfile.run('addUpNumbers()')
```

Результат выполнения этой программы выглядит примерно так:

```
4 function calls in 0.064 seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.000    0.000    0.064    0.064  <string>:1(<module>)
1      0.064    0.064    0.064    0.064  test1.py:2(addUpNumbers)
```

```

1    0.000    0.000    0.064    0.064 {built-in method builtins.exec}
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
                                     Profiler' objects}

```

Каждая строка представляет некоторую функцию и время, требуемое для выполнения этой функции. Столбцы выходных данных `cProfile.run()`:

- `ncalls` — количество вызовов функции;
- `tottime` — общее время, требуемое для выполнения функции, не считая времени в подфункциях;
- `percall` — общее время, разделенное на количество вызовов;
- `cumtime` — накопленное время для выполнения функции и ее подфункций;
- `percall` — общее время, деленное на количество вызовов;
- `filename:lineno(function)` — файл, в котором определяется функция, и номер строки.

Например, загрузите файлы `rsaCipher.py` и `al_sweigart_pubkey.txt` на <https://nostarch.com/crackingcodes/>. Программа RSA Cipher была представлена в книге *Cracking Codes with Python* (издательство No Starch Press, 2018)¹. Введите следующий фрагмент в интерактивной оболочке, чтобы профилировать функцию `encryptAndWriteToFile()`. Эта функция шифрует сообщение из 300 000 символов, созданное выражением `'abc' * 100000`:

```

>>> import cProfile, rsaCipher
>>> cProfile.run("rsaCipher.encryptAndWriteToFile('encrypted_file.txt', 'al_sweigart_pubkey.
txt', 'abc'*100000)")
11749 function calls in 28.900 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.001    0.001   28.900   28.900 <string>:1(<module>)
      2   0.000    0.000    0.000    0.000 _bootlocale.py:11(getpreferredencoding)
--snip--
      1   0.017    0.017   28.900   28.900 rsaCipher.py:104(encryptAndWriteToFile)
      1   0.248    0.248    0.249    0.249 rsaCipher.py:36(getBlocksFromText)
      1   0.006    0.006   28.873   28.873 rsaCipher.py:70(encryptMessage)
      1   0.000    0.000    0.000    0.000 rsaCipher.py:94(readKeyFile)
--snip--
    2347   0.000    0.000    0.000    0.000 {built-in method builtins.len}
    2344   0.000    0.000    0.000    0.000 {built-in method builtins.min}
    2344  28.617    0.012   28.617    0.012 {built-in method builtins.pow}
      2   0.001    0.000    0.001    0.000 {built-in method io.open}

```

¹ Свейгарт Эл. Криптография и взлом шифров на Python.

```
4688      0.001      0.000      0.001      0.000 {method 'append' of 'list' objects}
--snip--
```

Мы видим, что выполнение кода, переданного `cProfile.run()`, заняло 28,9 секунды. Обратите внимание на функции с наибольшим общим временем; в данном случае на работу встроенной функции Python `pow()` потребовалось 28,617 секунды. Это почти все время выполнения кода! Изменить этот фрагмент нельзя (он является частью реализации Python), но можно ли изменить наш код, чтобы он в меньшей степени зависел от этой функции?

В данном случае ответ «нет», потому что программа `rsaCipher.py` уже неплохо оптимизирована. Даже при этом профилирование кода дало нам информацию о том, что главным узким местом является функция `pow()`. А значит, нет смысла пытаться улучшить, скажем, функцию `readKeyFile()` (выполнение которой занимает так мало времени, что `cProfile` сообщает, что ее время выполнения равно 0).

Эта идея отражена в *законе Амдала* — формуле, которая вычисляет, насколько ускорится выполнение программы при улучшении одного из ее фрагментов. Согласно этой формуле ускорение всей операции равно $1 / ((1 - p) + (p / s))$, где s — ускорение компонента, а p — доля этого компонента в общем времени выполнения программы. Таким образом, если увеличить вдвое скорость фрагмента, требующего 90% от общего времени выполнения программы, общее ускорение составит $1 / ((1 - 0,9) + (0,9/2)) = 1,818$, или 82%. Это лучше, чем, скажем, утроение скорости элемента, который занимает всего 25% от общего времени выполнения; в этом случае общее ускорение составит $1 / ((1 - 0,25) + (0,25/2)) = 1,43$, или 43%. Заучивать эту формулу не нужно. Просто запомните, что удвоение скорости самых медленных или длинных частей вашего кода более продуктивно, чем удвоение скорости уже быстрых или коротких частей. Этот вывод подтверждается здравым смыслом: 10-процентная скидка на изделие дорогого модного дома лучше 10-процентной скидки на пару дешевой обуви.

Анализ алгоритмов с использованием нотации «О-большое»

Нотация «О-большое» — метод анализа алгоритмов, описывающий масштабирование времени выполнения кода. Код классифицируется по нескольким порядкам сложности, каждый из которых в общем виде показывает, насколько увеличится время выполнения кода при возрастании объема выполняемой работы. Разработчик Python Нед Бэтчелдер (Ned Batchelder) описывает нотацию «О-большое» как метод анализа, который определяет «насколько замедлится код с ростом объема данных»; этой теме был посвящен его доклад на конференции PyCon 2018, доступный на <https://youtu.be/duwZ-2UK0fc/>.

Представьте следующую ситуацию: имеется объем работы, на выполнение которой уходит час. Если объем работы увеличится вдвое, сколько времени потребуется в этом случае? Кто-то скажет, что вдвое больше, но на самом деле ответ зависит от выполняемой работы. Если на чтение короткой книги уходит час, то на чтение двух коротких книг потребуется около двух часов. Но если вы выстраиваете по алфавиту 500 книг в час, то расстановка по алфавиту 1000 книг займет больше двух часов, потому что вам придется найти правильное место для каждой книги в значительно большем наборе книг. С другой стороны, если вы просто проверяете, пуста ли книжная полка, то совершенно неважно, сколько книг стоит на полке — 0, 10 или 1000. Ответ будет понятен с первого взгляда. Время выполнения остается приблизительно постоянным независимо от количества книг. И хотя некоторые люди могут быстрее или медленнее справляться с чтением или алфавитной расстановкой книг, общая картина остается неизменной.

Нотация «О-большое» описывает эту картину. Алгоритм может выполняться на быстром или медленном компьютере, но нотация «О-большое» все равно может использоваться для описания быстродействия алгоритма в целом независимо от того, на каком оборудовании этот алгоритм выполняется. В нотации «О-большое» не используются конкретные единицы для описания времени выполнения алгоритма (секунды, такты процессора и т. д.), потому что эти показатели будут изменяться на разных компьютерах или при использовании разных языков программирования.

Порядки нотации «О-большое»

Нотация «О-большое» обычно определяет несколько *порядков сложности*. Ниже эти порядки перечислены по возрастанию (сначала указаны *низкие* порядки, при которых код с ростом объема данных замедляется в наименьшей степени, а в конце — *высокие* порядки с наибольшим замедлением).

1. $O(1)$, постоянное время (самый низкий порядок).
2. $O(\log n)$, логарифмическое время.
3. $O(n)$, линейное время.
4. $O(n \log n)$, время $N\text{-Log-}N$.
5. $O(n^2)$, полиномиальное время.
6. $O(2^n)$, экспоненциальное время.
7. $O(n!)$, факториальное время (наивысший порядок).

Обратите внимание на форму записи порядка «О-большое»: буква О в верхнем регистре, за ней следует пара круглых скобок с описанием порядка. Буква n в скобках представляет размер входных данных, с которыми работает код.

Чтобы использовать нотацию «О-большое», не обязательно понимать точный математический смысл таких терминов, как «логарифмическое» или «полиномиальное». Все порядки я более подробно опишу в следующем разделе, а пока ограничусь простым обозначением.

- $O(1)$ и $O(\log n)$ — быстрые алгоритмы.
- $O(n)$ и $O(n \log n)$ — неплохие алгоритмы.
- $O(n^2)$, $O(2^n)$ и $O(n!)$ — медленные алгоритмы.

Конечно, можно найти и контрпримеры, но в общем случае эту классификацию можно считать хорошей. Здесь перечислены не все порядки нотации «О-большое», а только самые распространенные. Рассмотрим примеры задач для каждого из них.

Книжная полка как метафора порядков «О-большое»

В следующих примерах нотации «О-большое» я продолжу использовать метафору с книжной полкой. В описаниях n обозначает количество книг на полке, а порядок «О-большое» описывает, как растет время выполнения различных операций с увеличением количества книг.

$O(1)$, постоянное время

Если вы проверяете, пуста ли книжная полка, это операция с постоянным временем. Неважно, сколько книг на полке; вы с первого взгляда определите, есть ли на ней книги. Их количество может изменяться, но время выполнения останется постоянным, потому что, если вы видите на полке хотя бы одну книгу, дальше можно не проверять. Значение n не влияет на скорость выполнения задачи, поэтому n не входит в обозначение $O(1)$. Также постоянное время иногда записывается в виде $O(c)$.

$O(\log n)$, логарифмическое время

Логарифм является операцией, обратной по отношению к возведению в степень; результат 2^4 , или $2 \times 2 \times 2 \times 2$, равен 16, тогда как логарифм $\log_2(16)$ (читается «логарифм 16 по основанию 2») равен 4. В программировании часто предполагается, что логарифм вычисляется по основанию 2, поэтому мы используем $O(\log n)$ вместо $O(\log_2 n)$.

Поиск на полке книги, если они упорядочены по алфавиту, является операцией с логарифмическим временем. Сначала вы проверяете книгу в середине полки. Если это та книга, которую вы искали, поиск завершен. В противном случае можно

определить, находится ли искомая книга до или после книги в середине. При этом количество книг, среди которых вы ищете нужную, фактически сокращается вдвое. Этот процесс можно повторить и проверить книгу в середине той половины, где она может находиться. Этот алгоритм называется *алгоритмом бинарного поиска*; пример его применения рассмотрен в разделе «Примеры анализа “О-большое”» позднее в этой главе, с. 280.

Количество разбиений набора из n книг надвое равно $\log_2 n$. На полке с 16 книгами для нахождения нужного издания потребуется не более 4 итераций. Так как каждая сокращает количество книг вдвое, удвоение количества книг добавит в поиск всего один дополнительный шаг. Даже если на упорядоченной полке стоят 4,2 миллиарда книг, для нахождения нужного экземпляра потребуется всего 32 итерации.

Алгоритмы $\log n$ обычно включают принцип «разделяй и властвуй», который выбирает половину входных данных размера n , потом половину от этой половины и т. д. Операции $\log n$ хорошо масштабируются: рабочая нагрузка n возрастает вдвое, тогда как время выполнения увеличивается всего на один шаг.

$O(n)$, линейное время

Чтение всех книг на полке является операцией с линейным временем. Если книги приблизительно одинакового объема, то при удвоении количества книг на полке для чтения понадобится приблизительно вдвое больше времени. Время выполнения растет пропорционально количеству книг n .

$O(n \log n)$, время $N\text{-Log-}N$

Сортировка набора книг в алфавитном порядке является операцией со временем $n\text{-log-}n$. Этот порядок является произведением $O(n)$ и $O(\log n)$. Можно рассматривать задачу $O(n \log n)$ как задачу $O(\log n)$, которую необходимо выполнить n раз. Я попробую неформально объяснить почему.

Начните со стопки книг, которую необходимо расставить по алфавиту, и пустой книжной полки. Выполните последовательность действий алгоритма бинарного поиска, описанную в подразделе « $O(\log n)$, логарифмическое время», с. 271, чтобы определить место одной книги на полке. Как вы уже знаете, эта операция имеет порядок $O(\log n)$. Если есть n книг и упорядочение каждой книги требует $\log n$ итераций, упорядочение всего набора книг займет $n \times \log n$, или $n \log n$ итераций. При удвоении количества книг количество шагов увеличится чуть более чем вдвое, так что алгоритмы $n \log n$ неплохо масштабируются.

Как выясняется, все эффективные обобщенные алгоритмы сортировки имеют порядок $O(n \log n)$: сортировка слиянием, быстрая сортировка, пирамидальная

сортировка и Timsort (алгоритм Timsort, изобретенный Тимом Петерсом (Tim Peters)), используется методом `Python sort()`).

$O(n^2)$, полиномиальное время

Проверка наличия одинаковых книг на неупорядоченной книжной полке является операцией с полиномиальным временем. Если на полке 100 книг, можно начать с первой и сравнить ее с 99 остальными книгами, чтобы узнать, найдется ли такая же. Затем вы берете вторую книгу и сравниваете ее с 99 остальными. Поиск дубликата одного издания выполняется на 99 шагов (округлим до 100, то есть n в нашем примере). Это необходимо сделать 100 раз, по одному для каждого экземпляра. Таким образом, количество шагов для выявления всех дубликатов составит приблизительно $n \times n$, или n^2 . (Приближение n^2 верно даже в том случае, если действовать умнее и не повторять уже выполненные сравнения.)

Время выполнения возрастает в квадратичной зависимости от количества книг. Проверка 100 книг на наличие дубликатов выполняется за 100×100 , или 10 000 шагов. Но при удвоении числа книг потребуется 200×200 , или 40 000 итераций: объем работы увеличивается в четыре раза.

По собственному опыту программирования я обнаружил, что анализ нотации «О-большое» чаще всего используется для того, чтобы избежать случайного применения алгоритма $O(n^2)$ там, где действует алгоритм $O(n \log n)$ или $O(n)$. Порядок $O(n^2)$ обычно становится переломной точкой для существенного замедления алгоритмов. Когда вы осознаете, что ваш код выполняется со временем $O(n^2)$ и выше, стоит сделать паузу. Возможно, есть другой алгоритм, который позволит решить проблему быстрее. В таких ситуациях вам безусловно пригодится учебный курс структур данных и алгоритмов — университетский или онлайн-курс.

Время $O(n^2)$ также называют *квадратичным*. Кроме того, известны алгоритмы с временем $O(n^3)$, или *кубическим временем*, которое медленнее $O(n^2)$; алгоритмы с временем $O(n^4)$, или *биквадратным временем*, которое медленнее $O(n^3)$, и другие варианты полиномиальной сложности.

$O(2^n)$, экспоненциальное время

Фотографирование всех возможных комбинаций книг на полке — операция с экспоненциальным временем. Логика такова: каждая книга на полке может либо присутствовать на фотографии, либо отсутствовать. На рис. 13.1 изображены все возможные комбинации для $n = 1, 2$ или 3. Для $n = 1$ возможны всего две фотографии: с книгой и без нее. Если $n = 2$, количество фотографий увеличивается до четырех: обе книги на полке, обе книги не на полке, только первая книга на полке или только

вторая книга на полке. При добавлении третьей книги объем необходимой работы снова удваивается: необходимо обработать каждое подмножество двух книг, которое включает третью книгу (четыре фотографии), и каждое подмножество двух книг, которое не включает третью книгу (еще четыре фотографии — итого 2^3 , или 8 фотографий.)

Каждая новая книга удваивает объем работы. Для n книг количество фотографий (то есть объем работы, которую необходимо выполнить) составляет 2^n .

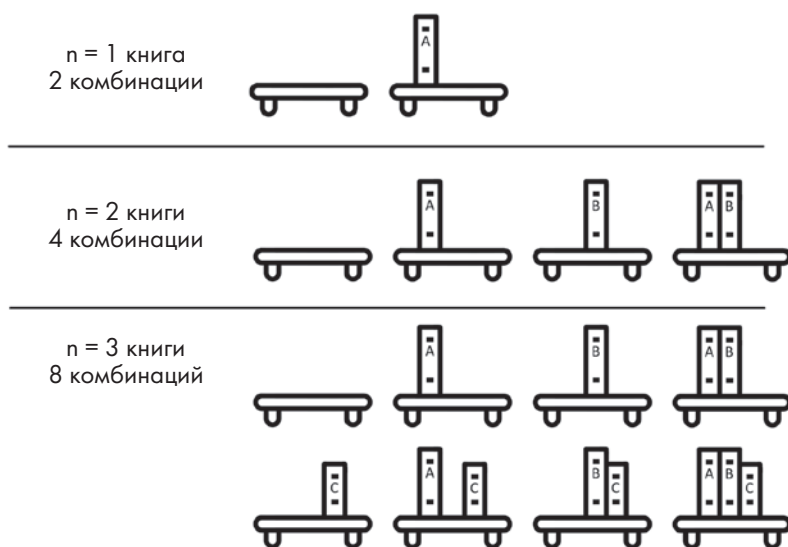


Рис. 13.1. Все комбинации книг на полке для одной, двух и трех книг

Время выполнения экспоненциальных задач растет очень быстро. Для 6 книг потребуется 2^6 , или 32 фотографии, но для 32 книг количество фотографий достигает 2^{32} , то есть более 4,2 миллиарда фотографий. Алгоритмы $O(2^n)$, $O(3^n)$, $O(4^n)$ и т. д. имеют экспоненциальную сложность.

$O(n!)$, факториальное время

Фотографирование книг на полке во всех возможных последовательностях — операция с факториальным временем. Все возможные варианты упорядочения n книг называются *перестановками*. Всего существуют $n!$ (n факториал) перестановок. Факториал числа равен произведению всех положительных целых чисел вплоть до этого числа. Например, факториал 3 равен $3 \times 2 \times 1$, то есть 6. На рис. 13.2 представлены все возможные перестановки трех книг.



Рис. 13.2. Все $3!$ (то есть 6) перестановки трех книг на полке

Чтобы получить этот результат самостоятельно, подумайте, как бы вы сгенерировали все перестановки n книг. Первую книгу можно выбрать n возможными способами; вторую книгу — $n - 1$ разными способами (все книги, кроме выбранной в первую позицию); для третьей книги остаются $n - 2$ возможных варианта, и т. д. Для 6 книг вычисление $6!$ дает $6 \times 5 \times 4 \times 3 \times 2 \times 1$, или 720 фотографий. При добавлении всего одной книги потребуется $7!$, или 5040 фотографий. Даже для небольших значений n часто оказывается, что алгоритмы с факториальным временем невозможно завершить за разумное время. Если вы возьмете 20 книг, будете переставлять их и фотографировать каждую секунду, для перебора всех возможных перестановок потребуется время, превышающее срок существования Вселенной.

Среди задач со временем $O(n!)$ хорошо известна задача о коммивояжере. Коммивояжер должен посетить n городов; он хочет вычислить суммарное расстояние для всех $n!$ возможных вариантов их посещения. По результатам вычислений он рассчитывает определить вариант с кратчайшим расстоянием. При большом количестве городов завершить перебор за разумное время не удастся. К счастью, оптимизированные алгоритмы способны найти короткий (но не обязательно кратчайший!) маршрут намного быстрее $O(n!)$.

«О-большое» как оценка худшего сценария

«О-большое» оценивает *худший сценарий* для любой задачи. Например, чтобы найти конкретную книгу на неупорядоченной книжной полке, вы начинаете с одного конца и просматриваете книги, пока не найдете нужную. Возможно, вам повезет и первая же книга окажется той, которую вы ищете. Но возможна и обратная ситуация: вам не повезет и книга окажется последней или ее вообще не будет на полке. Таким образом, при лучшем сценарии на полке могут быть миллиарды книг — но это неважно, потому что вы немедленно находите искомую книгу. Но при анализе алгоритмов от такого оптимизма нет никакого прока. Нотация «О-большое» описывает, что происходит, если вам *не* повезет: на полке стоят n книг и вам придется проверить их все. В таком случае время выполнения возрастает пропорционально количеству книг.

Некоторые программисты также используют нотацию «Омега-большое», описывающую лучший сценарий для алгоритма. Например, алгоритм $\Omega(n)$ в лучшем случае работает с линейной эффективностью. В худшем случае он может работать

медленное. Некоторые алгоритмы сталкиваются с особенно удачными случаями, в которых никакая работа вообще не требуется — например, поиск маршрута, когда вы уже находитесь в конечной точке.

Нотация «Тэта-большое» описывает алгоритмы с одинаковыми порядками в лучшем и худшем случае. Например, $\Theta(n)$ описывает алгоритм с линейной эффективностью в лучшем и худшем случае — то есть алгоритм $O(n)$ и $\Omega(n)$. Эти обозначения не используются в программировании настолько часто, как нотация «О-большое», но вам следует знать об их существовании.

Иногда приходится слышать о быстродействии «О-большое для среднего случая», когда они имеют в виду «Тэта-большое», или «О-большое для лучшего случая», подразумевая «Омега-большое». Такие высказывания следует считать оксюмороном; «О-большое» относится конкретно к худшему времени выполнения алгоритма. Но несмотря на техническую неточность, вы все равно сможете понять их смысл.

ВСЯ НЕОБХОДИМАЯ МАТЕМАТИКА ДЛЯ НОТАЦИИ «О-БОЛЬШОЕ»

Если вы подзабыли алгебру, я приведу более чем достаточные математические обоснования для анализа «О-большое».

Умножение. Многократное сложение: $2 \times 4 = 8$, так как $2 + 2 + 2 + 2 = 8$. С переменными $n + n + n$ равно $3 \times n$.

Умножение. В алгебраической записи знак $\times$ часто опускается, так что $2 \times n$ записывается в виде $2n$. С числами 2×3 записывается в виде $2(3)$ или просто 6.

Свойство **мультипликативного тождества**. При умножении числа на 1 будет получено то же число: $5 \times 1 = 5$, $42 \times 1 = 42$. В более общем виде $n \times 1 = n$.

Дистрибутивность умножения. $2 \times (3 + 4) = (2 \times 3) + (2 \times 4)$. Обе части записи равны 14. В более общем виде $a(b + c) = ab + ac$.

Возведение в степень. Многократное умножение: $2^4 = 16$ (читается «2 в четвертой степени равно 16»), так как $2 \times 2 \times 2 \times 2 = 16$. Число 2 называется основанием, а 4 — показателем степени. С переменными $n \times n \times n \times n$ равно n^4 . В Python для возведения в степень используется оператор `**`: `2 ** 4` дает результат 16.

При **возведении в степень 1** результатом является основание степени: $2^1 = 2$, а $9999^1 = 9999$. В более общем виде $n^1 = n$.

Результат **возведения в степень 0** всегда равен 1: $2^0 = 1$, и $9999^0 = 1$. В более общем виде $n^0 = 1$.

Коэффициенты. Множители: в выражении $3n^2 + 4n + 5$ коэффициенты равны 3, 4 и 5. 5 тоже является коэффициентом, потому что 5 можно переписать в виде $5(1)$, а затем записать в виде $5n^0$.

Логарифм. Операция, обратная возведению в степень. $2^4 = 16$, а следовательно, $\log_2(16) = 4$ (читается «логарифм 16 по основанию 2 равен 4»). В Python для вычисления логарифма используется функция `math.log()`: `math.log(16, 2)` дает результат 4.0.

Вычисление «О-большое» часто требует упрощения формул за счет объединения членов, то есть произведений чисел и переменных: в выражении $3n^2 + 4n + 5$ членами являются $3n^2$, $4n$ и 5. *Подобные члены* содержат одинаковые переменные, возведенные в одну степень. В выражении $3n^2 + 4n + 6n + 5$ члены $4n$ и $6n$ являются подобными. Выражение можно упростить и переписать в виде $3n^2 + 10n + 5$.

Следует помнить, что поскольку $n \times 1 = n$, выражение вида $3n^2 + 5n + 4$ можно рассматривать в виде $3n^2 + 5n + 4(1)$. Члены этого выражения соответствуют порядкам «О-большое» $O(n^2)$, $O(n)$ и $O(1)$. Мы еще вернемся к этому факту, когда займемся исключением коэффициентов в вычислениях нотации «О-большое».

Эта сводка пригодится нам позднее, когда вы научитесь вычислять порядок сложности нотации «О-большое» для фрагментов кода. Но скорее всего, после раздела «Моментальный анализ сложности» этой главы она вам уже не понадобится. «О-большое» — простая концепция, которая может принести пользу даже без четкого следования математическим правилам.

Определение порядка сложности нотации «О-большое» вашего кода

Чтобы определить порядок сложности нотации «О-большое» для фрагмента кода, необходимо решить четыре задачи: определить n , подсчитать шаги в коде, исключить нижние порядки и исключить коэффициенты.

Для примера найдем сложность «О-большое» для следующей функции `readingList()`:

```
def readingList(books):
    print('Here are the books I will read:')
    numberOfBooks = 0
    for book in books:
        print(book)
        numberOfBooks += 1
    print(numberOfBooks, 'books total.')
```

Напомню, что n представляет размер входных данных, с которыми работает код. В функциях n почти всегда определяется параметром. У функции `readingList()` всего один параметр `books`, и размер `books` станет хорошим кандидатом для n : чем больше размер `books`, тем дольше будет выполняться функция.

Затем подсчитаем шаги выполнения этого кода. Вопрос о том, что считать шагом, не имеет однозначного ответа, но для начала неплохо ориентироваться на строку кода. Количество шагов в цикле равно количеству итераций, умноженному на количество строк в коде цикла. Чтобы понять, о чем я говорю, приведу подсчеты для кода функции `readingList()`:

```
def readingList(books):
    print('Here are the books I will read:') # 1 шаг
    numberOfBooks = 0                       # 1 шаг
    for book in books:                      # n * шагов в цикле
        print(book)                         # 1 шаг
        numberOfBooks += 1                  # 1 шаг
    print(numberOfBooks, 'books total.')    # 1 шаг
```

Каждая строка кода будет рассматриваться как один шаг — кроме цикла `for`. Эта строка выполняется один раз для каждого элемента `books`, а поскольку размер `books` равен n , можно сказать, что он выполняется за n шагов. Мало того, он выполняет все шаги внутри цикла n раз. Так как цикл содержит два шага, общее количество составляет $2 \times n$ шагов. Шаги можно описать так:

```
def readingList(books):
    print('Here are the books I will read:') # 1 шаг
    numberOfBooks = 0                       # 1 шаг
    for book in books:                      # n * 2 шага
        print(book)                         # (уже подсчитано)
        numberOfBooks += 1                  # (уже подсчитано)
    print(numberOfBooks, 'books total.')    # 1 шаг
```

Теперь при вычислении общего количества шагов мы получаем $1 + 1 + (n \times 2) + 1$. Это выражение можно переписать в упрощенном виде $2n + 3$.

Нотация «О-большое» не предназначена для описания конкретных значений; это общий индикатор. По этой причине из подсчетов исключаются нижние порядки. Выражение $2n + 3$ состоит из двух порядков, линейного ($2n$) и постоянного (3). Если оставить только больший из этих порядков, остается $2n$.

Затем из порядка исключаются коэффициенты. В $2n$ коэффициентом является 2. После исключения остается n . В результате мы получаем итоговую метрику нотации «О-большое» для функции `readingList()`: $O(n)$, то есть линейная сложность.

Если задуматься, этот порядок выглядит логично. Функция состоит из нескольких шагов, но в общем случае при увеличении списка `books` в 10 раз время выполнения

также увеличится десятикратно. При увеличении размера `books` с 10 до 100 алгоритм переходит от значения $1 + 1 + (2 \times 10) + 1$, или 23 шага, к $1 + 1 + (2 \times 100) + 1$, или 203 шага. Число 203 равно приблизительно 10×23 , так что время выполнения возрастает пропорционально росту n .

Почему низкие порядки и коэффициенты не важны

Низкие порядки исключаются из подсчета шагов, потому что они становятся менее значимыми с увеличением размера n . Если список `books` из приведенной функции `readingList()` увеличивается с 10 до 10 000 000 000 (10 миллиардов), количество шагов увеличится с 23 до 20 000 000 003. При достаточно больших n эти три дополнительных шага ни на что не влияют.

При увеличении объема данных большой коэффициент меньшего порядка может игнорироваться по сравнению с высокими порядками. При определенном размере n более высокие порядки всегда будут медленнее низких порядков. Допустим, имеется функция `quadraticExample()` с порядком $O(n^2)$ и она состоит из $3n^2$ шагов. Другая функция `linearExample()` с порядком $O(n)$ состоит из $1000n$ шагов. Неважно, что коэффициент 1000 больше коэффициента 3; с ростом n квадратичная операция $O(n^2)$ станет медленнее линейной операции $O(n)$. Реальный код не столь важен, но он может выглядеть примерно так:

```
def quadraticExample(someData):    # n - размер someData
    for i in someData:             # n шагов
        for j in someData:         # n шагов
            print('Something')     # 1 шаг
            print('Something')     # 1 шаг
            print('Something')     # 1 шаг

def linearExample(someData):       # n - размер someData
    for i in someData:             # n шагов
        for k in range(1000):      # 1 * 1000 шагов
            print('Something')     # (уже подсчитано)
```

Функция `linearExample()` имеет большой коэффициент (1000) по сравнению с коэффициентом (3) функции `quadraticExample()`. Если размер входных данных n равен 10, то функция $O(n^2)$ вроде бы работает быстрее — всего 300 шагов по сравнению с 10 000 шагами функции $O(n)$.

Но нотация «О-большое» в основном описывает эффективность алгоритма при большом объеме работы. Когда n достигнет размера 334 и выше, функция `quadraticExample()` всегда будет медленнее функции `linearExample()`. Даже если `linearExample()` равна $1000000n$ шагов, функция `quadraticExample()` все равно будет медленнее, когда n достигнет 333 334. В какой-то момент операция $O(n^2)$ всегда становится медленнее $O(n)$, или низшей операции. Чтобы убедиться в этом,

взгляните на график нотации «О-большое» на рис. 13.3. На графике представлены все основные порядки нотации «О-большое». По оси x представлен размер данных x , а по оси y — время выполнения, необходимое для выполнения операции.

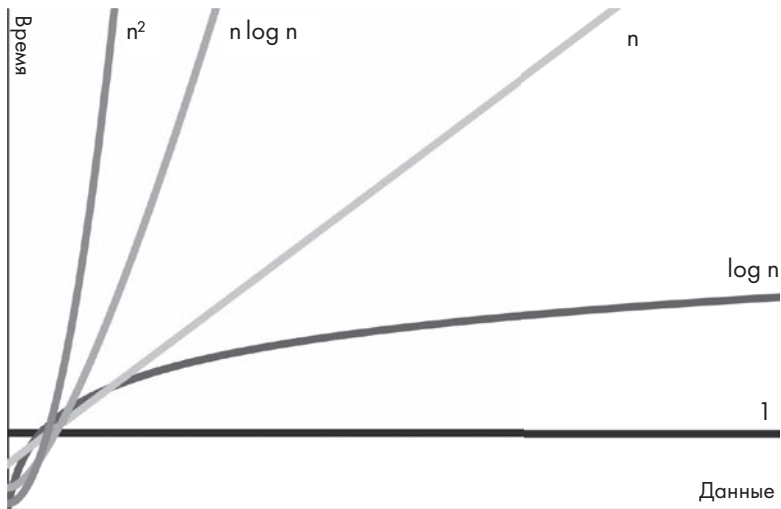


Рис. 13.3. График порядков нотации «О-большое»

Как видно из графика, время выполнения более высоких порядков растет быстрее, чем время низких порядков. Хотя низкие порядки могут иметь более высокие коэффициенты, из-за которых они изначально показывают большие затраты времени, рано или поздно высокие порядки их обойдут.

Примеры анализа «О-большое»

Определим порядки «О-большое» для некоторых примеров функций. В этих примерах будет использоваться параметр `books`, который представляет собой список строк с названиями книг.

Функция `countBookPoints()` вычисляет оценку на основании количества книг в списке. Большинство книг получает одно очко, а книги определенного автора получают два очка:

```
def countBookPoints(books):
    points = 0          # 1 шаг
    for book in books:  # n * шагов в цикле
        points += 1     # 1 шаг

    for book in books:  # n * шагов в цикле
```



```
if 'by Al Sweigart' in book: # 1 шаг
    points += 1              # 1 шаг
return points                # 1 шаг
```

Количество шагов достигает $1 + (n \times 1) + (n \times 2) + 1$, что преобразуется в $3n + 2$ после объединения подобных членов. После исключения составляющих низких порядков и коэффициентов мы приходим к $O(n)$, то есть к линейной сложности, независимо от того, сколько раз перебираются книги — один, два или миллиард.

До настоящего момента все примеры, в которых использовался один цикл, имели линейную сложность, но обратите внимание: все эти циклы выполнялись n раз. Как показывает следующий пример, цикл в коде не всегда подразумевает линейную сложность (которая характеризует циклы, перебирающие входные данные).

Функция `iLoveBooks()` выводит сообщения «I LOVE BOOKS!!!» и «BOOKS ARE GREAT!!!» 10 раз:

```
def iLoveBooks(books):
    for i in range(10):          # 10 * шагов в цикле
        print('I LOVE BOOKS!!!') # 1 шаг
        print('BOOKS ARE GREAT!!!') # 1 шаг
```

Функция содержит цикл `for`, но этот цикл не перебирает список `books` и выполняет 20 шагов независимо от размера `books`. Можно переписать это как $20(1)$. После исключения коэффициента 20 остается $O(1)$, то есть постоянная сложность. Все логично: функция выполняется за одно и то же время независимо от n (размера списка `books`).

Затем рассмотрим функцию `cheerForFavoriteBook()`, которая ищет любимую книгу в списке `books`:

```
def cheerForFavoriteBook(books, favorite):
    for book in books:          # n * шагов в цикле
        print(book)             # 1 шаг
        if book == favorite:    # 1 шаг
            for i in range(100): # 100 * шагов в цикле
                print('THIS IS A GREAT BOOK!!!') # 1 шаг
```

Цикл `for book` перебирает список `books`, для чего требуется n шагов, умноженных на количество шагов в цикле. Этот цикл включает вложенный цикл `for`, который выполняется 100 раз. То есть цикл `for book` выполняется $102 \times n$, или $102n$ раз. После исключения коэффициента выясняется, что `cheerForFavoriteBook()` все еще остается линейной операцией $O(n)$. Коэффициент 102 может показаться слишком большим, чтобы его игнорировать, но примите во внимание следующее: если `favorite` не встречается в списке `books`, эта функция выполнит только $1n$ шагов. Влияние коэффициентов может изменяться в таких пределах, что их трудно рассматривать как содержательные.

Затем функция `findDuplicateBooks()` просматривает список `books` (линейная операция) по одному разу для каждой книги (другая линейная операция):

```
def findDuplicateBooks(books):
    for i in range(books): # n шагов
        for j in range(i + 1, books): # n шагов
            if books[i] == books[j]: # 1 шаг
                print('Duplicate:', books[i]) # 1 шаг
```

Цикл `for i` перебирает весь список `books`, выполняя шаги в цикле n раз. Цикл `j` также перебирает часть списка `books`, хотя из-за исключения коэффициентов это также считается операцией с линейным временем. То есть цикл `for i` выполняет $n \times n$ операций, а именно n^2 . Как следствие, `findDuplicateBooks()` является операцией с полиномиальным временем $O(n^2)$.

Вложенные циклы сами по себе не подразумевают полиномиальной операции, но такой сложностью обладают вложенные циклы, в которых оба цикла выполняют n итераций. В результате будут выполнены n^2 шагов, а это подразумевает операцию $O(n^2)$.

Рассмотрим более сложный пример. Алгоритм бинарного поиска, упоминавшийся ранее, основан на поиске значения (назовем его `needle`) в середине отсортированного списка (назовем его `haystack`)¹. Если элемент `needle` не найден, мы переходим к поиску в первой или второй половине `haystack` в зависимости от того, в какой половине мы рассчитываем найти `needle`. Процесс повторяется, поиск ведется в постоянно уменьшающихся половинах, либо пока не будет найден элемент `needle`, либо пока не будет сделан вывод о том, что в `haystack` его нет. Обратите внимание: бинарный поиск работает только в том случае, если элементы `haystack` следуют в порядке сортировки.

```
def binarySearch(needle, haystack):
    if not len(haystack): # 1 шаг
        return None # 1 шаг
    startIndex = 0 # 1 шаг
    endIndex = len(haystack) - 1 # 1 шаг

    haystack.sort() # ??? шагов

    while start <= end: # ??? steps
        midIndex = (startIndex + endIndex) // 2 # 1 шаг
        if haystack[midIndex] == needle: # 1 шаг
            # Значение needle найдено.
            return midIndex # 1 шаг
        elif needle < haystack[midIndex]: # 1 шаг
            # Искать в первой половине.
```

¹ Needle — иголка, haystack — стог сена (англ). — Примеч. ред.

```
endIndex = midIndex - 1          # 1 шаг
elif needle > haystack[mid]:     # 1 шаг
    # Искать во второй половине.
    startIndex = midIndex + 1     # 1 шаг
```

Две строки `binarySearch()` оценить не так легко. Порядок «О-большое» вызова метода `haystack.sort()` зависит от кода, находящегося в методе Python `sort()`. Найти этот код непросто, но в интернете можно поискать информацию о методе и узнать, что он выполняется с порядком $O(n \log n)$. (Все основные функции сортировки выполняются в лучшем случае за время $O(n \log n)$.) Порядок «О-большое» для некоторых распространенных функций и методов Python описан в этой главе в разделе «Порядок “О-большое” для часто используемых функций».

Цикл `while` анализируется несколько сложнее, чем показанные выше циклы `for`. Чтобы определить, сколько итераций будет выполнено в цикле, необходимо понимать алгоритм бинарного поиска. До начала цикла `startIndex` и `endIndex` покрывают весь диапазон `haystack`, а `midIndex` устанавливается в середину этого диапазона. При каждой итерации цикла `while` происходит одно из двух. Если `haystack[midIndex] == needle`, искомое значение найдено и функция возвращает индекс `needle` в `haystack`. Если `needle < haystack[midIndex]` или `needle > haystack[midIndex]`, диапазон, покрываемый `startIndex` и `endIndex`, сокращается вдвое — за счет изменения либо `startIndex`, либо `endIndex`. Число делений любого списка размера n надвое составляет $\log_2(n)$. (К сожалению, это просто математический факт, который вы должны знать.) Таким образом, у цикла `while` порядок нотации «О-большое» составляет $O(\log n)$.

Но так как порядок $O(n \log n)$ строки `haystack.sort()` выше $O(\log n)$, более низкий порядок $O(\log n)$ исключается, а порядком всей функции `binarySearch()` становится $O(n \log n)$. Если мы можем гарантировать, что `binarySearch()` будет вызываться только с отсортированным списком `haystack`, строку `haystack.sort()` можно опустить и сделать `binarySearch()` функцией $O(\log n)$. Формально это улучшает эффективность функции, но не делает всю программу более эффективной, потому что необходимая работа по сортировке просто перемещается в другую часть программы. Многие реализации бинарного поиска опускают шаг сортировки, поэтому говорят, что алгоритм бинарного поиска обладает логарифмической сложностью $O(\log n)$.

Порядок «О-большое» для часто используемых функций

Анализ эффективности вашего кода должен учитывать порядок сложности всех вызываемых функций. Если вы сами написали функцию, вы сможете проанализировать собственный код. Но чтобы найти порядок «О-большое» для встроенных функций и методов Python, приходится обращаться к спискам вроде приведенного ниже.

В списке показаны порядки некоторых распространенных операций Python для типов последовательностей — таких как строки, кортежи и списки.

- `s[i]` reading и `s[i] = value` assignment — операции $O(1)$.
- `s.append(value)` — операция $O(1)$.
- `s.insert(i, value)` — операция $O(n)$. Вставка значения в последовательность (особенно в начало) требует сдвига всех элементов с индексами выше `i` на одну позицию вверх в последовательности.
- `s.remove(value)` — операция $O(n)$. Удаление значений из последовательности (особенно в начале) требует сдвига всех элементов с индексами выше `i` на одну позицию вниз в последовательности.
- `s.reverse()` — операция $O(n)$, потому что необходимо переставить все элементы последовательности.
- `s.sort()` — операция $O(n \log n)$, потому что алгоритм сортировки Python имеет сложность $O(n \log n)$.
- `value in s` — операция $O(n)$, потому что необходимо проверить каждый элемент.
- `for value in s:` — операция $O(n)$.
- `len(s)` — операция $O(1)$, потому что Python хранит количество элементов в последовательности, чтобы их не приходилось заново пересчитывать при каждом вызове `len()`.

В следующем списке приведены порядки «О-большое» для некоторых распространенных операций Python для типов отображений (таких как словари, множества и фиксированные множества).

- `m[key]` reading и `m[key] = value` assignment — операции $O(1)$.
- `m.add(value)` — операция $O(1)$.
- `value in m` — операция $O(1)$ для словарей; выполняется намного быстрее, чем для последовательностей.
- `for key in m:` — операция $O(n)$.
- `len(m)` — операция $O(1)$, потому что Python хранит количество элементов в отображении, чтобы их не приходилось заново пересчитывать при каждом вызове `len()`.

Если в списках элементы обычно приходится искать от начала до конца, словари используют ключ для вычисления адреса, а время, необходимое для поиска значения

по ключу, остается постоянным. Способ вычисления называется алгоритмом *хеширования*, а адрес — *хеш-кодом*. Тема хеширования выходит за рамки книги, но именно из-за него многие операции отображения выполняются с постоянным временем $O(1)$. Множества также используют хеширование, потому что множества по сути являются словарями, содержащими только ключи (вместо пар «ключ — значение»).

Но помните, что преобразование списка во множество является операцией $O(n)$, так что преобразование списка во множество с последующим обращением к элементам множества не обеспечит никакого выигрыша.

Моментальный анализ сложности

Когда вы освоитесь с выполнением анализа «О-большое», вам не придется выполнять каждый из шагов. Через какое-то время вы сможете просто взглянуть на какие-то характерные особенности кода, чтобы быстро определить его порядок сложности.

Если обозначить переменной n размер данных, с которыми работает код, можно воспользоваться рядом общих правил.

- Если код не обращается ни к каким данным, это $O(1)$.
- Если код последовательно перебирает данные, это $O(1)$.
- Если код содержит два вложенных цикла, каждый из которых перебирает данные, это $O(n^2)$.
- Вызовы функций включаются в подсчеты не как один шаг, а как общее количество шагов кода внутри функции. См. подраздел «Порядок “О-большое” для часто используемых функций», с. 283.
- Если код содержит операцию «разделяй и властвуй», которая многократно делит данные надвое, это $O(\log n)$.
- Если код содержит операцию «разделяй и властвуй», которая выполняется по одному разу для каждого элемента данных, это $O(n \log n)$.
- Если код перебирает все возможные комбинации значений в данных с размером n , это $O(2^n)$ или другой экспоненциальный порядок.
- Если код перебирает все возможные перестановки (то есть варианты упорядочения) значений данных, это $O(n!)$.
- Если код включает сортировку данных, это как минимум $O(n \log n)$.

Эти правила станут хорошей отправной точкой для анализа, но они не заменят реального анализа «О-большое». Помните, что порядок не является окончательным

критерием того, является ли код медленным, быстрым или эффективным. Рассмотрим следующую функцию `waitAnHour()`:

```
import time
def waitAnHour():
    time.sleep(3600)
```

Формально функция `waitAnHour()` является функцией с постоянным временем $O(1)$. Считается, что код с постоянным временем работает быстро, но на ее выполнение требуется целый час! Означает ли это, что код неэффективен? Нет. Трудно представить себе реализацию `waitAnHour()`, которая бы выполнялась быстрее одного часа.

Анализ «О-большое» не заменит профилирования вашего кода. Его цель — дать представление о том, как поведет себя ваш код при возрастающем объеме входных данных.

«О-большое» не имеет значения при малых n ... а значения n обычно малы

Когда вы вооружитесь знанием нотации «О-большое», у вас может возникнуть искушение анализировать каждый написанный вами фрагмент кода. Прежде чем с азартом лупить по каждому гвоздю, попавшему в поле зрения, следует учесть, что нотация «О-большое» полезна прежде всего при большом объеме обрабатываемых данных. В реальных ситуациях объем данных обычно невелик.

В таких случаях проработка нетривиальных, хитроумных алгоритмов с низкими порядками «О-большое» может оказаться нерациональной. Разработчик языка программирования Go Роб Пайк (Rob Pike) сформулировал пять правил программирования, одно из которых гласит: «Хитроумные алгоритмы медленно работают при малых n , а значения n обычно малы». Большинство разработчиков имеет дело не с программами для огромных центров обработки данных или для суперсложных вычислений, а с обычными повседневными программами. В таких ситуациях выполнение вашего кода под управлением профилировщика предоставит более конкретную информацию о быстродействии кода, чем анализ «О-большое».

Итоги

В стандартную библиотеку Python включены два модуля для профилирования: `timeit` и `cProfile`. Функция `timeit.timeit()` полезна для выполнения небольших фрагментов кода с целью сравнения времени их выполнения. Функция `cProfile.run()` компилирует подробный отчет по большим функциям и может выявить любые узкие места.

Важно измерять быстроедействие вашего кода, а не делать предположения относительно него. Хитроумные трюки для ускорения работы программы на самом деле могут замедлить ее. Или есть вероятность, что вы потратите много времени на оптимизацию фрагмента, который будет незначительно влиять на скорость вашей программы. Закон Амдала отражает этот факт в математическом виде: формула описывает, как ускорение работы одного компонента влияет на ускорение программы в целом.

Пожалуй, из всех концепций теории вычислений именно нотация «О-большое» находит наибольшее практическое применение. Для ее понимания необходимы определенные знания математики, но концепция определения закономерности замедления кода с ростом объема данных позволяет описывать алгоритмы без длинных формул.

Известны семь основных порядков сложности нотации «О-большое». Низкие порядки: $O(1)$, или постоянное время, описывает код, время выполнения которого не изменяется с увеличением размера данных n ; $O(\log n)$, или логарифмическое время, описывает код, сложность которого увеличивается на один шаг при увеличении размера данных n вдвое; $O(n)$, или линейное время, описывает код, который замедляется пропорционально росту размера данных n ; $O(n \log n)$, или время $n \cdot \log n$, описывает код, который работает немного медленнее $O(n)$ — многие алгоритмы сортировки относятся к этому порядку. Более высокие порядки работают медленнее, потому что их время выполнения растет намного быстрее размера входных данных: $O(n^2)$, или полиномиальное время, описывает код, время выполнения которого растет в квадратичной зависимости от размера входных данных n ; порядки $O(2^n)$, или экспоненциальное время, и $O(n!)$, или факториальное время, встречаются не так часто — в основном там, где в вычислениях используются комбинации и перестановки соответственно.

Помните: хотя нотация «О-большое» является полезным аналитическим инструментом, она не заменит выполнения кода в профилировщике для выявления узких мест. Однако если вы будете понимать нотацию «О-большое» и закономерности замедления кода с ростом данных, это позволит избежать написания кода, который работает значительно медленнее, чем мог бы.

14

Проекты для тренировки



До настоящего момента в этой книге я рассказывал о приемах написания удобочитаемого питонического кода. Теперь настало время применить эти приемы на практике; рассмотрим исходный код двух игр командной строки: «Ханойская башня» и «Четыре в ряд».

Это небольшие проекты, и они работают в текстовом режиме, чтобы не переусложнять задачу, но они неплохо демонстрируют принципы, о которых я рассказывал ранее. Для форматирования кода я использовал программу `Black`, описанную в разделе «`Black`: бескомпромиссная система форматирования кода», с. 79. Имена переменных были выбраны в соответствии с рекомендациями из главы 4. Код написан в питоническом стиле, о котором шла речь в главе 6. Кроме того, я написал комментарии и `doc`-строки по правилам из главы 11. Так как программы невелики, а объектно-ориентированное программирование (ООП) в книге еще не рассматривалось, я написал эти два проекта без классов — о них я расскажу чуть позже, в главах 15–17.

В этой главе приведен полный исходный код этих двух проектов — с полным анализом. Объяснения относятся не столько к тому, как работает код (для понимания этого достаточно базового понимания синтаксиса Python), а скорее к тому, почему этот код написан именно так, а не иначе. Тем не менее разные разработчики могут иметь разные мнения относительно того, как писать код и какой код следует считать питоническим. Безусловно, ничто не мешает вам оспаривать и критиковать исходники, представленные в этих проектах.

После того как вы прочитаете код проекта в книге, я рекомендую вам самостоятельно ввести этот код и несколько раз выполнить программы, чтобы понять, как они работают. Затем попытайтесь заново реализовать их с нуля. Ваш код не обязательно должен полностью повторять тот, что приведен в книге, но переписывая его, вы

получите представление о принимаемых решениях и компромиссах, на которые приходится идти программисту.

Головоломка «Ханойская башня»

Даны три стержня, на один из которых нанизаны диски, причем диски отличаются размером и лежат меньший на большем. Задача состоит в том, чтобы перенести пирамиду из дисков на другой стержень за наименьшее число ходов (рис. 14.1). При этом действуют три ограничения.

1. Диски можно перемещать по одному.
2. Игрок может перекладывать диски только с верха стопки на верх другой стопки.
3. Нельзя переложить больший диск на меньший.



Рис. 14.1. Головоломка «Ханойская башня»

Решение головоломки часто используется в компьютерной науке как задача для изучения рекурсивных алгоритмов. Наша программа не будет решать головоломку; она будет выводить изображение головоломки, чтобы пользователь мог решить ее. За дополнительной информацией о головоломке «Ханойская башня» обращайтесь на https://ru.wikipedia.org/wiki/Ханойская_башня.

Вывод результатов

Программа рисует башни в ASCII-графике, для изображения дисков используются текстовые символы. Приложение выглядит примитивно по сравнению с современными программами, но такой подход сохраняет простоту реализации, потому что

Для n дисков решение головоломки требует минимум $2^n - 1$ ходов. Таким образом, решение для башни с пятью дисками состоит из 31 хода: AC, AB, CB, AC, BA, BC, AC, AB, CB, CA, BA, CB, AC, AB, CB, AC, BA, BC, AC, BA, CB, CA, BA, BC, AC, AB, CB, AC, BA, BC и AC. Если вам понадобится усложнить пример для самостоятельного решения, увеличьте переменную TOTAL_DISKS в программе с 5 до 6.

Исходный код

Откройте в редакторе или IDE новый файл и введите приведенный ниже код. Сохраните файл с именем `towerofhanoi.py`.

```
"""THE TOWER OF HANOI, by Al Sweigart al@inventwithpython.com
Головоломка с перемещением дисков."""

import copy
import sys

TOTAL_DISKS = 5 # Чем больше дисков, тем сложнее головоломка.

# Изначально все диски находятся на стержне A:
SOLVED_TOWER = list(range(TOTAL_DISKS, 0, -1))

def main():
    """Проводит одну игру Ханойская башня."""
    print(
        """THE TOWER OF HANOI, by Al Sweigart al@inventwithpython.com

Move the tower of disks, one disk at a time, to another tower. Larger
disks cannot rest on top of a smaller disk.

More info at https://en.wikipedia.org/wiki/Tower_of_Hanoi
"""
    )

    """Словарь towers содержит ключи "A", "B" и "C", и значения - списки,
    представляющие стопку дисков. Список содержит целые числа, представляющие
    диски разных размеров, а начало списка представляет низ башни. Для игры
    с 5 дисками список [5, 4, 3, 2, 1] представляет заполненную башню. Пустой
    список list [] представляет башню без дисков. В списке [1, 3] больший диск
    находится на меньшем диске, такая конфигурация недопустима. Список [3, 1]
    допустим, так как меньшие диски могут размещаться на больших."""
    towers = {"A": copy.copy(SOLVED_TOWER), "B": [], "C": []}

    while True: # Один ход для каждой итерации цикла.
        # Вывести башни и диски:
        displayTowers(towers)

        # Запросить ход у пользователя:
        fromTower, toTower = getPlayerMove(towers)

        # Переместить верхний диск с fromTower на toTower:
        disk = towers[fromTower].pop()
        towers[toTower].append(disk)

        # Проверить, решена ли головоломка:
        if SOLVED_TOWER in (towers["B"], towers["C"]):
            displayTowers(towers) # Вывести башни в последний раз.
```

```

        print("You have solved the puzzle! Well done!")
        sys.exit()

def getPlayerMove(towers):
    """Запрашивает ход у пользователя. Возвращает (fromTower, toTower)."""

    while True: # Пока пользователь не введет допустимый ход.
        print('Enter the letters of "from" and "to" towers, or QUIT.')
        print("(e.g., AB to moves a disk from tower A to tower B.)")
        print()
        response = input("> ").upper().strip()

        if response == "QUIT":
            print("Thanks for playing!")
            sys.exit()

        # Убедиться в том, что пользователь ввел допустимые обозначения башен:
        if response not in ("AB", "AC", "BA", "BC", "CA", "CB"):
            print("Enter one of AB, AC, BA, BC, CA, or CB.")
            continue # Снова запросить ход.

        # Более содержательные имена переменных:
        fromTower, toTower = response[0], response[1]

        if len(towers[fromTower]) == 0:
            # Башня fromTower не может быть пустой:
            print("You selected a tower with no disks.")
            continue # Снова запросить ход.
        elif len(towers[toTower]) == 0:
            # На пустую башню можно переместить любой диск:
            return fromTower, toTower
        elif towers[toTower][-1] < towers[fromTower][-1]:
            print("Can't put larger disks on top of smaller ones.")
            continue # Снова запросить ход.
        else:
            # Допустимый ход, вернуть выбранные башни:
            return fromTower, toTower

def displayTowers(towers):
    """Выводит три башни с дисками."""

    # Вывести три башни:
    for level in range(TOTAL_DISKS, -1, -1):
        for tower in (towers["A"], towers["B"], towers["C"]):
            if level >= len(tower):
                displayDisk(0) # Вывести пустой стержень без диска.
            else:
                displayDisk(tower[level]) # Вывести диск.
        print()

    # Вывести обозначения башен A, B и C:

```

```

emptySpace = " " * (TOTAL_DISKS)
print("{0} A{0}{0} B{0}{0} C{n}".format(emptySpace))

def displayDisk(width):
    """Выводит диск заданной ширины. Ширина 0 означает отсутствие диска."""
    emptySpace = " " * (TOTAL_DISKS - width)

    if width == 0:
        # Вывести сегмент стержня без диска:
        print(f"{emptySpace}||{emptySpace}", end="")
    else:
        # Вывести диск:
        disk = "@" * width
        numLabel = str(width).rjust(2, "_")
        print(f"{emptySpace}{disk}{numLabel}{disk}{emptySpace}", end="")

# Если программа была запущена (а не импортирована), начать игру:
if __name__ == "__main__":
    main()

```

Запустите программу и сыграйте несколько раз, чтобы получить представление о том, что она делает, прежде чем читать объяснения исходного кода. Чтобы проверить возможные опечатки, скопируйте код в сетевую программу diff по адресу <https://inventwithpython.com/beyond/diff/>.

Написание кода

А теперь рассмотрим исходный код и посмотрим, как в нем применяются приемы и шаблоны, описанные в книге.

Начало программы:

```

"""THE TOWER OF HANOI, by Al Sweigart al@inventwithpython.com
Головоломка с перемещением дисков."""

```

Программа начинается с многострочного комментария, который служит документацией для модуля `towerofhanoi`. Встроенная функция `help()` использует эту информацию для описания модуля:

```

>>> import towerofhanoi
>>> help(towerofhanoi)
Help on module towerofhanoi:

```

```

NAME
    towerofhanoi

```

```

DESCRIPTION
    THE TOWER OF HANOI, by Al Sweigart al@inventwithpython.com
    Головоломка с перемещением дисков.

```

```
FUNCTIONS
    displayDisk(width)
        Выводит диск заданной ширины.
--snip--
```

При необходимости в doc-строку модуля можно добавить больше слов или даже абзацев. Я ограничился минимумом текста, потому что программа очень проста.

После doc-строки модуля следуют команды `import`:

```
import copy
import sys
```

Black форматирует их как отдельные команды вместо одной команды `import copy, sys`. При таком подходе будет проще отслеживать добавление или удаление импортированных модулей в системах контроля версий (таких как Git), которые отслеживают изменения, вносимые программистом.

Затем определяются константы, требуемые программе:

```
TOTAL_DISKS = 5 # Чем больше дисков, тем сложнее головоломка.

# Изначально все диски находятся на стержне A:
SOLVED_TOWER = list(range(TOTAL_DISKS, 0, -1))
```

Мы определяем константы в начале файла, чтобы сгруппировать их, и делаем их глобальными. Имена записываем в верхнем регистре согласно «змеиной» схеме, чтобы пометить их как константы.

Константа `TOTAL_DISKS` определяет, сколько дисков используется в головоломке. Переменная `SOLVED_TOWER` содержит пример списка, содержащего решение головоломки: в списке указаны все диски, самый большой расположен внизу, а самый маленький — наверху. Это значение генерируется на основании значения `TOTAL_DISKS` и для пяти дисков список имеет вид `[5, 4, 3, 2, 1]`.

Обратите внимание: в файле отсутствуют аннотации типов. Дело в том, что типы всех переменных, параметров и возвращаемых значений автоматически определяются на основании кода. Например, константе `TOTAL_DISKS` присваивается целое значение 5. По нему системы проверки типов (такие как Муру) определяют, что `TOTAL_DISKS` будет содержать только целые числа.

Мы определяем функцию `main()`, которая вызывается программой в конце файла:

```
def main():
    """Проводит одну игру Ханойская башня."""
    print(
        """THE TOWER OF HANOI, by Al Sweigart al@inventwithpython.com
```

Move the tower of disks, one disk at a time, to another tower. Larger disks cannot rest on top of a smaller disk.

More info at https://en.wikipedia.org/wiki/Tower_of_Hanoi

```
"""  
    )
```

Функции также могут содержать doc-строки. Обратите внимание на doc-строку `main()` под командой `def`. Чтобы просмотреть эту doc-строку, выполните команды `import towerofhanoi` и `help(towerofhanoi.main)` из интерактивной оболочки.

Далее следует комментарий, который описывает структуру данных, используемую для представления башни, потому что она занимает центральное место в работе программы:

```
"""Словарь towers содержит ключи "A", "B" и "C" и значения - списки,  
представляющие стопку дисков. Список содержит целые числа, представляющие  
диски разных размеров, а начало списка представляет низ башни. Для игры  
с 5 дисками список [5, 4, 3, 2, 1] представляет заполненную башню. Пустой  
список list [] представляет башню без дисков. В списке [1, 3] больший диск  
находится на меньшем диске, такая конфигурация недопустима. Список [3, 1]  
допустим, так как меньшие диски могут размещаться на больших."""  
towers = {"A": copy.copy(SOLVED_TOWER), "B": [], "C": []}
```

Список `SOLVED_TOWER` используется как *стек* — одна из простейших структур данных, используемых при разработке. Стек представляет собой упорядоченный список значений, а его состояние может изменяться только добавлением (занесением в стек) или удалением (извлечением из стека) значений на вершине (начале) стека. Эта структура данных идеально подходит для представления башен в нашей программе. Список Python можно преобразовать в стек; для этого нужно использовать метод `append()` для включения элементов и метод `pop()` для их извлечения и избегать изменения списка любыми другими способами. Конец списка будет рассматриваться как вершина стека.

Каждое целое число в списке `towers` представляет один диск определенного размера. Например, в игре с пятью дисками список `[5, 4, 3, 2, 1]` представляет полную стопку дисков от самого большого (5) внизу до самого маленького (1) наверху.

Также обратите внимание на то, что в комментарии приведены примеры допустимого и недопустимого списка.

Внутри функции `main()` находится бесконечный цикл, в котором выполняется один ход нашей головоломки:

```
while True: # Один ход для каждой итерации цикла.  
    # Вывести башни и диски:  
    displayTowers(towers)
```

```
# Запросить ход у пользователя:
fromTower, toTower = getPlayerMove(towers)

# Переместить верхний диск с fromTower на toTower:
disk = towers[fromTower].pop()
towers[toTower].append(disk)
```

За один ход игрок смотрит на текущее состояние башен и вводит свой ход. Затем программа обновляет структуру данных `towers`. Подробности выполнения этих операций скрыты в функциях `displayTowers()` и `getPlayerMove()`. Благодаря содержательным именам функция `main()` дает общее представление о том, что делает программа.

Следующие строки проверяют, решил ли игрок головоломку, для чего решение из `SOLVED_TOWER` сравнивается с `towers["B"]` и `towers["C"]`:

```
# Проверить, решена ли головоломка:
if SOLVED_TOWER in (towers["B"], towers["C"]):
    displayTowers(towers) # Вывести башни в последний раз.
    print("You have solved the puzzle! Well done!")
    sys.exit()
```

Сравнивать с `towers["A"]` не нужно, потому что в начале игры стержень уже содержит завершенную башню; чтобы решить головоломку, игрок должен построить башню на стержне В или С. Обратите внимание: `SOLVED_TOWER` используется повторно для генерирования начальных башен и проверки того, решил ли игрок головоломку. Так как `SOLVED_TOWER` является константой, можно быть уверенным в том, что `SOLVED_TOWER` всегда будет иметь значение, присвоенное в начале кода.

Используемое условие эквивалентно `SOLVED_TOWER == towers["B"] or SOLVED_TOWER == towers["C"]`, но записывается короче — эту идиому Python я рассматривал в главе 6. Если условие истинно, значит, игрок решил головоломку и программа завершается. В противном случае цикл продолжается следующим ходом.

Функция `getPlayerMove()` запрашивает у игрока ход и проверяет его по игровым правилам:

```
def getPlayerMove(towers):
    """Запрашивает ход у пользователя. Возвращает (fromTower, toTower)."""
    while True: # Пока пользователь не введет допустимый ход.
        print('Enter the letters of "from" and "to" towers, or QUIT.')
        print("(e.g., AB to moves a disk from tower A to tower B.)")
        print()
        response = input("> ").upper().strip()
```

Мы начинаем бесконечный цикл, который продолжается до выполнения одного из двух условий: либо цикл прерывается командой `return` (с выходом из функции), либо программа будет завершена вызовом `sys.exit()`. Первая часть цикла

предлагает игроку ввести ход в виде двух букв — для башни, с которой перемещается диск и на которую он перемещается.

Обратите внимание на инструкцию `input("> ").upper().strip()`, которая получает ввод с клавиатуры от игрока. Вызов `input("> ")` запрашивает текст у игрока с выводом приглашения `>`. Символ показывает, что игрок должен что-то ввести. Если программа не выведет приглашение, то игрок может подумать, что она зависла.

Строка, полученная от `input()`, преобразуется к верхнему регистру вызовом метода `upper()`. Это позволяет игроку вводить обозначения башен как в верхнем, так и в нижнем регистре — например, 'a' или 'A' для башни A. Затем для строки в верхнем регистре вызывается метод `strip()` для удаления пробельных символов в начале и конце строки на случай, если пользователь случайно добавил пробел при вводе хода. Такие удобства несколько упрощают работу с программой для пользователя.

Все еще внутри функции `getPlayerMove()` мы проверяем данные, введенные пользователем:

```
if response == "QUIT":
    print("Thanks for playing!")
    sys.exit()

# Убедиться в том, что пользователь ввел допустимые обозначения башен:
if response not in ("AB", "AC", "BA", "BC", "CA", "CB"):
    print("Enter one of AB, AC, BA, BC, CA, or CB.")
    continue # Снова запросить ход.
```

Если пользователь вводит 'QUIT' (в произвольном регистре и даже с пробелами в начале или конце строки благодаря вызовам `upper()` и `strip()`), программа завершается. Также функция `getPlayerMove()` могла бы вернуть 'QUIT', чтобы указать вызывающей стороне на необходимость вызвать `sys.exit()`, вместо того чтобы вызывать `sys.exit()` в `getPlayerMove()`. Но это усложнило бы возвращаемое значение `getPlayerMove()`: функция должна возвращать либо кортеж с двумя строками (для хода игрока), либо одну строку 'QUIT'. Функция, которая возвращает значения одного типа данных, более понятна, чем та, которая может возвращать значения многих возможных типов. Эта тема обсуждалась в разделе «Возвращаемые значения всегда должны иметь один тип данных», с. 212.

Из трех башен можно составить только шесть комбинаций «с какой башни — на какую башню». Несмотря на тот факт, что мы жестко зафиксировали все шесть значений в условии, проверяющем ход, такой код читается намного проще, чем что-нибудь вроде конструкции вида `len(response) != 2 or response[0] not in 'ABC' or response[1] not in 'ABC' or response[0] == response[1]`. С учетом этого факта подход с жестко фиксированными вариантами оказывается наиболее прямым.

Как правило, использование «магических» значений вроде "AB", "AC" и т. д. считается нежелательным, потому что они работают, только пока в программе используются три стержня. Но хотя количество дисков можно отрегулировать изменением константы TOTAL_DISKS, крайне маловероятно, что в игре увеличится количество стержней. В данном случае запись всех возможных перемещений в программе допустима.

Две новые переменные `fromTower` и `toTower` создаются как содержательные имена для данных. Они не имеют функционального назначения, но лучше читаются, чем `response[0]` и `response[1]`:

```
# Более содержательные имена переменных:
fromTower, toTower = response[0], response[1]
```

Затем мы проверяем, есть ли для созданных пользователем башен допустимый ход:

```
if len(towers[fromTower]) == 0:
    # Башня fromTower не может быть пустой:
    print("You selected a tower with no disks.")
    continue # Снова запросить ход.
elif len(towers[toTower]) == 0:
    # На пустую башню можно переместить любой диск:
    return fromTower, toTower
elif towers[toTower][-1] < towers[fromTower][-1]:
    print("Can't put larger disks on top of smaller ones.")
    continue # Снова запросить ход.
```

Если ход недопустим, команда `continue` возвращает управление в начало цикла, где игроку снова предлагается ввести ход. Затем мы проверяем, не пуста ли башня `toTower`. Если она пуста, возвращается `fromTower, toTower`, чтобы подчеркнуть, что ход допустим, потому что диск всегда можно поместить на пустой стержень. Первые два условия проверяют, что к моменту проверки третьего условия `towers[toTower]` и `towers[fromTower]` не будут пустыми и не вызовут ошибки `IndexError`. Условия были упорядочены так, чтобы их порядок предотвращал `IndexError` и дополнительные проверки.

Важно, чтобы ваши программы обрабатывали любой недопустимый ввод от пользователя или потенциальные ошибочные ситуации. Пользователи могут не знать, что нужно ввести, или могут допустить опечатку при вводе. Также файлы могут быть неожиданно удалены или в базе данных может произойти сбой. Ваши программы должны обладать достаточной устойчивостью к аномальным ситуациям; в противном случае возможны аварийные завершения или позднее могут возникнуть коварные ошибки. Если ни одно из условий не равно `True`, `getPlayerMove()` возвращает `fromTower, toTower`:

```
else:
    # Допустимый ход, вернуть выбранные башни:
    return fromTower, toTower
```

В Python команды `return` всегда возвращают одно значение. Хотя может показаться, что эта команда `return` возвращает два значения, Python в действительности возвращает один кортеж с двумя значениями, что эквивалентно `return (fromTower, toTower)`. Программисты на языке Python часто опускают круглые скобки в этом контексте. Круглые скобки определяют кортеж в меньшей степени, чем запятые.

Обратите внимание: программа вызывает функцию `getPlayerMove()` только один раз из функции `main()`. Функция не избавляет от дублирования кода — самой распространенной причины для использования функций. Нет никаких причин, по которым весь код `getPlayerMove()` нельзя было бы разместить в функции `main()`. Но функции также могут использоваться как механизм структурирования кода по отдельным блокам, для чего в данном случае и задействована `getPlayerMove()`. С этой функцией `main()` не будет слишком длинной и громоздкой.

Функция `displayTowers()` выводит диски на башнях A, B и C в аргументе `towers`:

```
def displayTowers(towers):
    """Выводит три башни с дисками."""
    # Вывести три башни:
    for level in range(TOTAL_DISKS, -1, -1):
        for tower in (towers["A"], towers["B"], towers["C"]):
            if level >= len(tower):
                displayDisk(0) # Вывести пустой стержень без диска.
            else:
                displayDisk(tower[level]) # Вывести диск.
        print()
```

Для вывода каждого диска в башне функция зависит от функции `displayDisk()`, которая будет рассмотрена следующей. Цикл `for level` проверяет каждый возможный диск, а цикл `for tower` проверяет башни A, B и C.

Функция `displayTowers()` вызывает `displayDisk()` для вывода каждого диска с заданной шириной, или при передаче 0 выводится только стержень без диска:

```
# Вывести обозначения башен A, B и C:
emptySpace = " " * (TOTAL_DISKS)
print("{0} A{0}{0} B{0}{0} C\n".format(emptySpace))
```

На экране выводятся метки A, B и C. Эта информация помогает игроку различать башни, а также подчеркивает, что башни обозначаются буквами A, B и C, а не 1, 2 и 3, или «Левая», «Средняя» и «Правая». Я не стал использовать цифры 1, 2 и 3 для пометки башен, чтобы игроки не путали эти числа с числовыми обозначениями размера дисков.

Переменной `emptySpace` присваивается количество пробелов между метками, которое в свою очередь вычисляется на основании `TOTAL_DISKS`, потому что чем больше дисков в игре, тем больше разделяющее их расстояние. Вместо f-строк, как в `print(f'{emptySpace} A{emptySpace}{emptySpace} B{emptySpace}{emptySpace}`

`C\n')`, используется метод строк `format()`. Это позволяет применить один и тот же аргумент `emptySpace` всюду, где в соответствующей строке встречается `{0}`, в результате чего код получается более коротким и лучше читается, чем в версии с `f`-строкой.

Функция `displayDisk()` выводит один диск заданной ширины. Если диск отсутствует, выводится только стержень:

```
def displayDisk(width):
    """Выводит диск заданной ширины. Ширина 0 означает отсутствие диска."""
    emptySpace = " " * (TOTAL_DISKS - width)
    if width == 0:
        # Вывести сегмент стержня без диска:
        print(f"{emptySpace}||{emptySpace}", end="")
    else:
        # Вывести диск:
        disk = "@" * width
        numLabel = str(width).rjust(2, "_")
        print(f"{emptySpace}{disk}{numLabel}{disk}{emptySpace}", end="")
```

Изображение диска состоит из начального пробела, символов `@` в количестве, равном ширине диска, двух символов ширины (с начальным символом подчеркивания, если ширина задается одной цифрой), еще одной серии символов `@` и завершающего пробела. Чтобы вывести только пустой стержень, достаточно вывести начальный пробел, две вертикальные черты и завершающий пробел. В результате для вывода следующей башни потребуются шесть вызовов `displayDisk()` с шестью разными аргументами ширины:

```
||
@_1@
@@_2@@
@@@_3@@@
@@@@_4@@@@
@@@@@_5@@@@@
```

Обратите внимание на то, как функции `displayTowers()` и `displayDisk()` разделяют обязанности по выводу башен. Хотя функция `displayTowers()` решает, как интерпретировать структуры данных, представляющие каждую башню, она зависит от `displayDisk()` для отображения каждого диска на башне. Разбиение программы на меньшие функции упрощает тестирование каждой части. Если программа неправильно выводит диски, то, скорее всего, проблема в `displayDisk()`. Если диски следуют в неправильном порядке, то, вероятно, проблема в `displayTowers()`. В любом случае объем кода, который необходимо отладить, будет намного меньше.

Для вызова функции `main()` используется стандартная идиома Python:

```
# Если программа была запущена (а не импортирована), начать игру:
if __name__ == "__main__":
    main()
```

Python автоматически присваивает переменной `__name__` значение `'__main__'`, если игрок запускает программу `towerofhanoi.py` напрямую. Но если кто-то импортирует программу как модуль командой `import towerofhanoi`, то `__name__` будет присвоено значение `'towerofhanoi'`. Строка `if __name__ == '__main__':` будет вызывать функцию `main()`, если кто-то запускает нашу программу, тем самым начиная игру. Но если вы хотите просто импортировать программу как модуль, чтобы, допустим, вызывать ее отдельные функции для модульного тестирования, то это условие дает результат `False`, и функция `main()` не вызывается.

Игра «Четыре в ряд»

Это игра для двух игроков: каждый пытается выстроить ряд из четырех своих фишек по горизонтали, по вертикали или по диагонали, помещая свои фишки на самое нижнее свободное место в столбце. В этой игре используется вертикальная доска 7×6 . В нашей реализации игры два игрока-человека (обозначаемые X и O) играют друг с другом (режим игры с компьютером не поддерживается).

Вывод результатов

При запуске программы «Четыре в ряд» вывод будет выглядеть примерно так:

```
Four-in-a-Row, by Al Sweigart al@inventwithpython.com
Two players take turns dropping tiles into one of seven columns, trying
to make four in a row horizontally, vertically, or diagonally.
```

```
1234567
+-----+
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
+-----+
```

```
Player X, enter 1 to 7 or QUIT:
```

```
> 1
```

```
1234567
+-----+
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
|.X. . . .|
+-----+
```

```
Player O, enter 1 to 7 or QUIT:
```

```
--snip--
```

Player 0, enter 1 to 7 or QUIT:

> 4

```

1234567
+-----+
|.....|
|.....|
|...0...|
|X.OO...|
|X.XO...|
|XOXO..X|
+-----+

```

Player 0 has won!

Игрок должен найти такую стратегию, которая позволит ему выстроить четыре фишки в ряд, одновременно не позволяя сделать то же самое противнику.

Исходный код

Откройте в редакторе или IDE новый файл и введите приведенный ниже код. Сохраните файл под именем `fourinarow.py`.

```

"""Four-in-a-Row, by Al Sweigart al@inventwithpython.com
Игра на выстраивание четырех фишек в ряд."""

```

```
import sys
```

```

# Константы, используемые для вывода игрового поля:
EMPTY_SPACE = "." # Точки проще подсчитать, чем пробелы.
PLAYER_X = "X"
PLAYER_O = "O"

```

```

# Примечание: если BOARD_WIDTH изменится, обновите BOARD_TEMPLATE и COLUMN_LABELS.
BOARD_WIDTH = 7
BOARD_HEIGHT = 6
COLUMN_LABELS = ("1", "2", "3", "4", "5", "6", "7")
assert len(COLUMN_LABELS) == BOARD_WIDTH

```

```

# Шаблонная строка для вывода игрового поля:
BOARD_TEMPLATE = """

```

```

1234567
+-----+
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
+-----+"""

```

```

def main():
    """Проводит одну игру Четыре в ряд."""

```

```

print(
    """Four-in-a-Row, by Al Sweigart al@inventwithpython.com

Два игрока по очереди опускают фишки в один из семи столбцов,
стараясь выстроить четыре фишки по вертикали, горизонтали или диагонали.
"""
)
# Подготовка новой игры:
gameBoard = getNewBoard()
playerTurn = PLAYER_X

while True: # Обрабатывает ход игрока.
    # Вывод игрового поля и получение хода игрока:
    displayBoard(gameBoard)
    playerMove = getPlayerMove(playerTurn, gameBoard)
    gameBoard[playerMove] = playerTurn

    # Проверка победы или ничьей:
    if isWinner(playerTurn, gameBoard):
        displayBoard(gameBoard) # В последний раз вывести поле.
        print("Player {} has won!".format(playerTurn))
        sys.exit()

    elif isFull(gameBoard):
        displayBoard(gameBoard) # В последний раз вывести поле.
        print("There is a tie!")
        sys.exit()

    # Ход передается другому игроку:
    if playerTurn == PLAYER_X:
        playerTurn = PLAYER_O
    elif playerTurn == PLAYER_O:
        playerTurn = PLAYER_X

def getNewBoard():
    """Возвращает словарь, представляющий игровое поле.

    Ключи - кортежи (columnIndex, rowIndex) с двумя целыми числами,
    а значения - одна из строк "X", "O" or "." (пробел)."""
    board = {}
    for rowIndex in range(BOARD_HEIGHT):
        for columnIndex in range(BOARD_WIDTH):
            board[(columnIndex, rowIndex)] = EMPTY_SPACE
    return board

def displayBoard(board):
    """Выводит на экран игровое поле и фишки."""

    # Подготовить список, передаваемый строковому методу format() для
    # шаблона игрового поля. Список содержит все фишки игрового поля
    # и пустые ячейки, перечисляемые слева направо, сверху вниз:
    tileChars = []
    for rowIndex in range(BOARD_HEIGHT):
        for columnIndex in range(BOARD_WIDTH):
            tileChars.append(board[(columnIndex, rowIndex)])

```

```

# Выводит игровое поле:
print(BOARD_TEMPLATE.format(*tileChars))

def getPlayerMove(playerTile, board):
    """Предлагает игроку выбрать столбец для размещения фишки.

    Возвращает кортеж (столбец, строка) итогового положения фишки."""
    while True: # Пока игрок не введет допустимый ход.
        print(f"Player {playerTile}, enter 1 to {BOARD_WIDTH} or QUIT:")

        response = input("> ").upper().strip()
        if response == "QUIT":
            print("Thanks for playing!")
            sys.exit()

        if response not in COLUMN_LABELS:
            print(f"Enter a number from 1 to {BOARD_WIDTH}.")
            continue # Снова запросить ход.

        columnIndex = int(response) - 1 # -1, потому что индексы начинаются с 0.

        # Если столбец заполнен, снова запросить ход:
        if board[(columnIndex, 0)] != EMPTY_SPACE:
            print("That column is full, select another one.")
            continue # Снова запросить ход.

        # Начать снизу, найти первую пустую ячейку.
        for rowIndex in range(BOARD_HEIGHT - 1, -1, -1):
            if board[(columnIndex, rowIndex)] == EMPTY_SPACE:
                return (columnIndex, rowIndex)

def isFull(board):
    """Возвращает True, если в `board` не осталось пустых ячеек,
    иначе возвращается False."""
    for rowIndex in range(BOARD_HEIGHT):
        for columnIndex in range(BOARD_WIDTH):
            if board[(columnIndex, rowIndex)] == EMPTY_SPACE:
                return False # Пустая ячейка найдена, вернуть False.
    return True # Все ячейки заполнены.

def isWinner(playerTile, board):
    """Возвращает True, если `playerTile` образует ряд из четырех фишек
    в `board`, в противном случае возвращается False."""

    # Проверить всю доску в поисках четырех фишек в ряд:
    for columnIndex in range(BOARD_WIDTH - 3):
        for rowIndex in range(BOARD_HEIGHT):
            # Проверить четверку направо:
            tile1 = board[(columnIndex, rowIndex)]
            tile2 = board[(columnIndex + 1, rowIndex)]
            tile3 = board[(columnIndex + 2, rowIndex)]
            tile4 = board[(columnIndex + 3, rowIndex)]
            if tile1 == tile2 == tile3 == tile4 == playerTile:
                return True

```



```
for columnIndex in range(BOARD_WIDTH):
    for rowIndex in range(BOARD_HEIGHT - 3):
        # Проверить четверку вниз:
        tile1 = board[(columnIndex, rowIndex)]
        tile2 = board[(columnIndex, rowIndex + 1)]
        tile3 = board[(columnIndex, rowIndex + 2)]
        tile4 = board[(columnIndex, rowIndex + 3)]
        if tile1 == tile2 == tile3 == tile4 == playerTile:
            return True

for columnIndex in range(BOARD_WIDTH - 3):
    for rowIndex in range(BOARD_HEIGHT - 3):
        # Проверить четверку по диагонали направо вниз:
        tile1 = board[(columnIndex, rowIndex)]
        tile2 = board[(columnIndex + 1, rowIndex + 1)]
        tile3 = board[(columnIndex + 2, rowIndex + 2)]
        tile4 = board[(columnIndex + 3, rowIndex + 3)]
        if tile1 == tile2 == tile3 == tile4 == playerTile:
            return True

        # Проверить четверку по диагонали налево вниз:
        tile1 = board[(columnIndex + 3, rowIndex)]
        tile2 = board[(columnIndex + 2, rowIndex + 1)]
        tile3 = board[(columnIndex + 1, rowIndex + 2)]
        tile4 = board[(columnIndex, rowIndex + 3)]
        if tile1 == tile2 == tile3 == tile4 == playerTile:
            return True
    return False

# Если программа была запущена (а не импортирована), начать игру:
if __name__ == "__main__":
    main()
```

Запустите программу и сыграйте несколько раз, чтобы получить представление о том, что делает программа, прежде чем читать объяснения работы исходного кода. Чтобы проверить возможные опечатки, скопируйте код в сетевую программу diff по адресу <https://inventwithpython.com/beyond/diff/>.

Написание кода

Рассмотрим исходный код программы, как это было сделано в программе «Ханойская башня». Как и в предыдущем случае, я отформатировал код с использованием Black с ограничением длины строки 75 символов.

Начало программы:

```
"""Four-in-a-Row, by Al Sweigart al@inventwithpython.com
Игра на выстраивание четырех фишек в ряд."""
```

```
import sys
```

```
# Константы, используемые для вывода игрового поля:
EMPTY_SPACE = "." # Точки проще подсчитать, чем пробелы.
PLAYER_X = "X"
PLAYER_O = "O"
```

Программа начинается с `doc`-строки, импортирования модулей и присваивания констант, как в программе «Ханойская башня». В программе определяются константы `PLAYER_X` и `PLAYER_O`, чтобы нам не приходилось использовать строки "X" и "O" в программе, а ошибки находились проще. Если при вводе констант будет допущена опечатка (например, `PLAYER_XX`), Python выдаст ошибку `NameError`, и вы моментально узнаете о проблеме. Но если опечатка будет допущена при вводе символов "X" (например, "XX" или "Z"), возникшая ошибка уже не будет столь очевидной. Как объяснялось в разделе «Магические» числа», с. 97, используя константы вместо строковых значений, вы не только получаете описание, но и предупреждаете появление опечаток в исходном коде.

Константы не должны изменяться во время выполнения программы. Тем не менее программист может обновить их значения в будущих версиях программы. По этой причине мы оставляем напоминание программисту, что при изменении значения `BOARD_WIDTH` он должен обновить константы `BOARD_TEMPLATE` и `COLUMN_LABELS`, описанные ниже:

```
# Примечание: если BOARD_WIDTH изменится, обновите BOARD_TEMPLATE и COLUMN_LABELS.
BOARD_WIDTH = 7
BOARD_HEIGHT = 6

COLUMN_LABELS = ("1", "2", "3", "4", "5", "6", "7")
assert len(COLUMN_LABELS) == BOARD_WIDTH
```

Эта константа будет использоваться позднее для проверки правильности столбца, введенного игроком. Если `BOARD_WIDTH` будет присвоено значение, отличное от 7, придется добавить или удалить метки из кортежа `COLUMN_LABELS`. Этого можно было бы избежать, генерируя значение `COLUMN_LABELS` на основании `BOARD_WIDTH` кодом вида `COLUMN_LABELS = tuple([str(n) for n in range(1, BOARD_WIDTH + 1)])`. Однако `COLUMN_LABELS` вряд ли изменится в будущем, потому что стандартно в «Четыре в ряд» играют на доске 7×6 , поэтому я решил записать значение кортежа явно.

Конечно, подобная жесткая фиксация значений в программе является признаком проблем в коде, о чем я уже рассказывал в разделе «Магические» числа», с. 97, но такой код читается лучше, чем альтернатива. Кроме того, команда `assert` предупредит об изменении `BOARD_WIDTH` без обновления `COLUMN_LABELS`.

Как в игре «Ханойская башня», «Четыре в ряд» использует ASCII-графику для рисования игрового поля. В следующих строках содержится одна команда присваивания с многострочным текстом:

```
# Шаблонная строка для вывода игрового поля:
BOARD_TEMPLATE = """
```

```
    1234567
+-----+
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
|{}{}{}{}{}|
+-----+"""
```

Строка содержит фигурные скобки {}, которые метод `format()` заменит содержимым игрового поля. (Эту задачу решает функция `displayBoard()`, описанная ниже.) Так как игровое поле состоит из семи столбцов и шести строк, мы используем семь пар фигурных скобок {} в любой из шести строк для представления каждой ячейки. Как и в случае с `COLUMN_LABELS`, формально мы жестко фиксируем поле с заданным набором столбцов и строк. Если `BOARD_WIDTH` и `BOARD_HEIGHT` будут заменены новыми целыми значениями, многострочный шаблон в `BOARD_TEMPLATE` также потребует обновления.

Также можно было написать код генерирования `BOARD_TEMPLATE` на основании констант `BOARD_WIDTH` и `BOARD_HEIGHT`:

```
BOARD_EDGE = "    "+" + ("-" * BOARD_WIDTH) + "+"
BOARD_ROW = "    |" + ("{}" * BOARD_WIDTH) + "|\\n"
BOARD_TEMPLATE = "\\n        " + "".join(COLUMN_LABELS) + "\\n" + BOARD_EDGE + "\\n"
+ (BOARD_ROW * BOARD_WIDTH) + BOARD_EDGE
```

Но этот код читается намного хуже простого многострочного текста, и размер игрового поля вряд ли будет изменяться, поэтому мы используем простой многострочный текст.

Начнем с функции `main()`, которая будет вызывать все остальные функции, написанные для игры:

```
def main():
    """Проводит одну игру Четыре в ряд."""
    print(
        """Four-in-a-Row, by Al Sweigart al@inventwithpython.com
Два игрока по очереди опускают фишки в один из семи столбцов,
стараясь выстроить четыре фишки по вертикали, горизонтали или диагонали.
"""
    )
    # Подготовка новой игры:
    gameBoard = getNewBoard()
    playerTurn = PLAYER_X
```

Для функции `main()` определяется doc-строка, для просмотра которой можно воспользоваться встроенной функцией `help()`. Функция `main()` также готовит игровое поле к новой игре и выбирает первого игрока.

Внутри функции `main()` выполняется бесконечный цикл:

```
while True: # Обрабатывает ход игрока.
    # Вывод игрового поля и получение хода игрока:
    displayBoard(gameBoard)
    playerMove = getPlayerMove(playerTurn, gameBoard)
    gameBoard[playerMove] = playerTurn
```

Каждая итерация цикла состоит из одного хода. Сначала игровое поле выводится на экран. Затем игрок выбирает столбец, в который опускает фишку, и, наконец, происходит обновление структуры данных игрового поля.

Затем определяется результат хода игрока:

```
# Проверка победы или ничьей:
if isWinner(playerTurn, gameBoard):
    displayBoard(gameBoard) # В последний раз вывести поле.
    print("Player {} has won!".format(playerTurn))
    sys.exit()
elif isFull(gameBoard):
    displayBoard(gameBoard) # В последний раз вывести поле.
    print("There is a tie!")
    sys.exit()
```

Если игрок сделал победный ход, `isWinner()` возвращает `True` и партия завершается. Если игрок заполнил последнюю ячейку, а победитель не определился, `isFull()` возвращает `True` и игра завершается. Обратите внимание: вместо вызова `sys.exit()` можно воспользоваться простой командой `break`. Это бы прервало цикл `while`, а поскольку функция `main()` не содержит кода после цикла, управление вернется к вызову `main()` в конце программы, что приведет к ее завершению. Но я выбрал `sys.exit()`, чтобы наглядно показать программисту, читающему код, что программа немедленно завершится.

Если игра не закончена, следующие строки присваивают `playerTurn` значение, представляющее другого игрока:

```
# Ход передается другому игроку:
if playerTurn == PLAYER_X:
    playerTurn = PLAYER_O
elif playerTurn == PLAYER_O:
    playerTurn = PLAYER_X
```

Команду `elif` можно преобразовать в простую команду `else` без условия. Но вспомните принцип из «Дзен Python»: явное лучше, чем неявное. Этот код явно сообщает,

что если сейчас ход игрока O, то следующим ходит игрок X. Также можно было бы использовать другую формулировку: если сейчас не ход игрока X, то X ходит следующим. И хотя команды `if` и `else` естественно сочетаются с логическими условиями, значения `PLAYER_X` и `PLAYER_O` не эквивалентны `True` или `False`; а значит, `not PLAYER_X` — не то же самое, что `PLAYER_O`. Следовательно, лучше проверять значение `playerTurn` напрямую.

Те же действия можно было выполнить в одной строке:

```
playerTurn = {PLAYER_X: PLAYER_O, PLAYER_O: PLAYER_X}[playerTurn]
```

Здесь используется трюк со словарем, упоминавшийся в подразделе «Использование словарей вместо команды `switch`», с. 129. Но как и многие однострочные команды, она читается хуже прямолинейной конструкции `if` и `elif`.

Затем определяется функция `getNewBoard()`:

```
def getNewBoard():
    """Возвращает словарь, представляющий игровое поле.

    Ключи - кортежи (columnIndex, rowIndex) с двумя целыми числами,
    а значения - одна из строк "X", "O" или "." (пробел)."""
    board = {}
    for rowIndex in range(BOARD_HEIGHT):
        for columnIndex in range(BOARD_WIDTH):
            board[(columnIndex, rowIndex)] = EMPTY_SPACE
    return board
```

Функция возвращает словарь, представляющий игровое поле для игры «Четыре в ряд». Она содержит кортежи `(columnIndex, rowIndex)` для ключей (где `columnIndex` и `rowIndex` являются целыми числами), а также символы 'X', 'O' или '.' для фишки в каждой ячейке поля. Эти строки сохраняются в `PLAYER_X`, `PLAYER_O` и `EMPTY_SPACE` соответственно.

Наша игра «Четыре в ряд» довольно проста, так что использование словаря для представления игрового поля можно считать приемлемым. Впрочем, также можно воспользоваться решением на базе ООП (об ООП мы поговорим в главах с 15 по 17).

Функция `displayBoard()` получает структуру данных игрового поля в аргументе `board` и выводит поле на экран с использованием константы `BOARD_TEMPLATE`:

```
def displayBoard(board):
    """Выводит на экран игровое поле и фишки."""
    # Подготовить список, передаваемый строковому методу format() для
    # шаблона игрового поля. Список содержит все фишки игрового поля
    # и пустые ячейки, перечисляемые слева направо, сверху вниз:
    tileChars = []
```

Напомним, что `BOARD_TEMPLATE` представляет собой многострочный текст, содержащий множество пар фигурных скобок. При вызове метода `format()` для `BOARD_TEMPLATE` эти фигурные скобки будут заменены аргументами, переданными `format()`.

Переменная `tileChars` содержит список таких аргументов. В начале ей присваивается пустой список. Первое значение в `tileChars` заменяет первую пару фигурных скобок в `BOARD_TEMPLATE`, второе значение заменяет вторую пару и т. д. По сути, мы создаем список значений из словаря `board`:

```
for rowIndex in range(BOARD_HEIGHT):
    for columnIndex in range(BOARD_WIDTH):
        tileChars.append(board[(columnIndex, rowIndex)])

# Выводит игровое поле:
print(BOARD_TEMPLATE.format(*tileChars))
```

Вложенные циклы `for` перебирают все возможные комбинации строк и столбцов игрового поля, присоединяя их к списку `tileChars`. После завершения этих циклов значения из `tileChars` передаются как отдельные аргументы метода `format()` с префиксом `*`. В подразделе «Использование `*` при создании вариативных функций» (с. 201) показано, как использовать этот синтаксис для обработки значений в списке как отдельных аргументов функции; код `print(*['cat', 'dog', 'rat'])` эквивалентен `print('cat', 'dog', 'rat')`. Звездочка необходима, потому что метод `format()` ожидает получить один аргумент для каждой пары фигурных скобок, а не один аргумент-список.

Затем записывается функция `getPlayerMove()`:

```
def getPlayerMove(playerTile, board):
    """Предлагает игроку выбрать столбец для размещения фишки.

    Возвращает кортеж (столбец, строка) итогового положения фишки."""
    while True: # Пока игрок не введет допустимый ход.
        print(f"Player {playerTile}, enter 1 to {BOARD_WIDTH} or QUIT:")
        response = input("> ").upper().strip()

        if response == "QUIT":
            print("Thanks for playing!")
            sys.exit()
```

Функция начинается с бесконечного цикла, который ожидает, пока игрок введет допустимый ход. По своей структуре этот код напоминает функцию `getPlayerMove()` из программы «Ханойская башня». Обратите внимание: вызов `print()` в начале цикла `while()` использует f-строку, чтобы нам не пришлось изменять сообщение при обновлении `BOARD_WIDTH`.

Мы проверяем, что ответ игрока представляет столбец; в противном случае команда `continue` передает управление обратно в начало цикла, чтобы запросить у игрока допустимый ход:

```
if response not in COLUMN_LABELS:
    print(f"Enter a number from 1 to {BOARD_WIDTH}.")
    continue # Снова запросить ход.
```

Условие проверки ввода также можно было бы записать в виде `not response.isdecimal()` or `spam < 1` or `spam > BOARD_WIDTH`, но проще воспользоваться условием `not in COLUMN_LABELS`.

Затем необходимо определить, в какую строку упадет фишка, опущенная игроком в выбранный столбец:

```
columnIndex = int(response) - 1 # -1, потому что индексы начинаются с 0.

# Если столбец заполнен, снова запросить ход:
if board[(columnIndex, 0)] != EMPTY_SPACE:
    print("That column is full, select another one.")
    continue # Снова запросить ход.
```

На экран выводятся метки столбцов от 1 до 7. Но индексы (`columnIndex`, `rowIndex`) начинаются с 0, поэтому их значения лежат в диапазоне от 0 до 6. Чтобы устранить это расхождение, строковые значения от '1' до '7' преобразуются в целые значения от 0 до 6.

Индексы строк начинаются с 0 (верх игрового поля) и увеличиваются до 6 (низ игрового поля). Мы проверяем верхнюю ячейку выбранного столбца и смотрим, занята ли она. Если она занята, значит, столбец заполнен, и команда `continue` передает управление обратно в начало цикла, чтобы запросить у игрока другой ход.

Если столбец не заполнен, необходимо найти самую нижнюю свободную ячейку, в которую упадет фишка:

```
# Начать снизу, найти первую пустую ячейку.
for rowIndex in range(BOARD_HEIGHT - 1, -1, -1):
    if board[(columnIndex, rowIndex)] == EMPTY_SPACE:
        return (columnIndex, rowIndex)
```

Цикл `for` начинает с индекса нижней строки, `BOARD_HEIGHT - 1` (или 6), и двигается вверх, пока не найдет первую свободную ячейку. Затем функция возвращает индексы самой нижней пустой ячейки.

Если игровое поле заполнено, игра заканчивается вничью:

```
def isFull(board):
    """Возвращает True, если в `board` не осталось пустых ячеек,
    иначе возвращается False."""
    for rowIndex in range(BOARD_HEIGHT):
        for columnIndex in range(BOARD_WIDTH):
            if board[(columnIndex, rowIndex)] == EMPTY_SPACE:
                return False # Пустая ячейка найдена, вернуть False.
    return True # Все ячейки заполнены.
```

Функция `isFull()` использует пару вложенных циклов `for` для перебора всех ячеек игрового поля. Если функция находит хотя бы одну пустую ячейку, значит, поле еще не заполнено и функция возвращает `False`. Если оба цикла будут выполнены до конца, значит, функция не нашла пустой ячейки и она возвращает `True`.

Функция `isWinner()` проверяет, выиграл ли игрок своим ходом:

```
def isWinner(playerTile, board):
    """Возвращает True, если `playerTile` образует ряд из четырех фишек
    в `board`, в противном случае возвращается False."""

    # Проверить всю доску в поисках четырех фишек в ряд:
    for columnIndex in range(BOARD_WIDTH - 3):
        for rowIndex in range(BOARD_HEIGHT):
            # Проверить четверку направо:
            tile1 = board[(columnIndex, rowIndex)]
            tile2 = board[(columnIndex + 1, rowIndex)]
            tile3 = board[(columnIndex + 2, rowIndex)]
            tile4 = board[(columnIndex + 3, rowIndex)]
            if tile1 == tile2 == tile3 == tile4 == playerTile:
                return True
```

Функция возвращает `True`, если `playerTile` встречается в четырех ячейках подряд по горизонтали, вертикали или диагонали. Чтобы проверить, выполняется ли это условие, необходимо проверить каждый набор из четырех смежных позиций игрового поля. Для этого будет использоваться серия вложенных циклов `for`.

Кортеж `(columnIndex, rowIndex)` определяет отправную точку для проверки. Мы проверяем начальную позицию и ячейки в трех позициях справа от нее в поисках строки `playerTile`. Если начальной позицией является `(columnIndex, rowIndex)`, то позиция справа от нее определяется кортежем `(columnIndex + 1, rowIndex)`, и т. д. Фишки в этих четырех позициях сохраняются в переменных `tile1`, `tile2`, `tile3` и `tile4`. Если все четыре переменные содержат такое же значение, как `playerTile`, значит, четыре фишки в ряд найдены и функция `isWinner()` возвращает `True`.

В разделе «Переменные с числовыми суффиксами», с. 102, я упомянул о том, что имена переменных с последовательными числовыми суффиксами (например, `tile1–tile4` в этой игре) часто являются признаком проблем в коде и их лучше заменить одним списком. Однако в данном контексте такие имена переменных вполне допустимы. Заменять их списком не нужно, потому что в программе «Четыре в ряд» всегда используются ровно четыре такие переменные. Напомню, что запах кода не всегда указывает на наличие проблемы; он означает лишь то, что к коду стоит присмотреться и убедиться в том, что он написан наиболее понятным и удобочитаемым способом. В данном случае использование списка только усложнит код без каких-либо преимуществ, поэтому мы ограничимся использованием имен `tile1`, `tile2`, `tile3` и `tile4`.

Аналогичный процесс используется для проверки вертикальной последовательности из четырех фишек:

```
for columnIndex in range(BOARD_WIDTH):
    for rowIndex in range(BOARD_HEIGHT - 3):
        # Проверить четверку вниз:
        tile1 = board[(columnIndex, rowIndex)]
        tile2 = board[(columnIndex, rowIndex + 1)]
        tile3 = board[(columnIndex, rowIndex + 2)]
        tile4 = board[(columnIndex, rowIndex + 3)]
        if tile1 == tile2 == tile3 == tile4 == playerTile:
            return True
```

Остается проверить последовательность из четырех фишек в ряд по диагонали, идущую вниз и вправо и вниз и влево:

```
for columnIndex in range(BOARD_WIDTH - 3):
    for rowIndex in range(BOARD_HEIGHT - 3):
        # Проверить четверку по диагонали направо вниз:
        tile1 = board[(columnIndex, rowIndex)]
        tile2 = board[(columnIndex + 1, rowIndex + 1)]
        tile3 = board[(columnIndex + 2, rowIndex + 2)]
        tile4 = board[(columnIndex + 3, rowIndex + 3)]
        if tile1 == tile2 == tile3 == tile4 == playerTile:
            return True

        # Проверить четверку по диагонали налево вниз:
        tile1 = board[(columnIndex + 3, rowIndex)]
        tile2 = board[(columnIndex + 2, rowIndex + 1)]
        tile3 = board[(columnIndex + 1, rowIndex + 2)]
        tile4 = board[(columnIndex, rowIndex + 3)]
        if tile1 == tile2 == tile3 == tile4 == playerTile:
            return True
    return False
```

Код аналогичен проверке четырех фишек в ряд по горизонтали, и я не стану повторять объяснения. Если ни одна проверка четырех фишек в ряд ничего не находит, функция возвращает `False`, показывая, что ход `playerTile` не принес победы в игре:

```
return False
```

Остается только вызвать функцию `main()`:

```
# Если программа была запущена (а не импортирована), начать игру:
if __name__ == "__main__":
    main()
```

И снова здесь используется стандартная идиома Python: функция `main()` вызывается в том случае, если программа `fourinarow.py` была запущена напрямую, но не при импортировании `fourinarow.py` в виде модуля.

Итоги

Головоломка «Ханойская башня» и игра «Четыре в ряд» — короткие программы, но так как мы следовали принципам, представленным в книге, их код хорошо читается и прост в отладке. В этих программах применяется ряд полезных практик: они были автоматически отформатированы программой Black, для описания модулей и функций использовались doc-строки, а константы мы разместили в начале файла. Переменные, параметры функций и возвращаемые значения функций ограничиваются одним типом данных, так что аннотации типов (как полезная форма дополнительной документации) оказываются излишними.

В «Ханойской башне» три башни представлены словарем с ключами 'A', 'B' и 'C', значениями которых являются списки целых чисел. Такой подход работает, но если бы программа была сколько-нибудь большой и сложной, для представления этих данных стоило бы воспользоваться классом. Классы и средства ООП в этой главе не использовались, потому что об ООП речь пойдет только в главах 15–17. Просто помните, что для таких структур данных абсолютно нормально использовать класс. Башни выводятся на экран в ASCII-графике, а диски изображаются серией текстовых символов.

Игра «Четыре в ряд» также использует ASCII-графику для вывода представления игрового поля. Изображение поля строится из многострочного текста, хранимого в константе `BOARD_TEMPLATE`. Строка включает 42 пары фигурных скобок `{ }` для каждой ячейки игрового поля 7×6 . Строковый метод `format()` заменяет каждую пару фигурных скобок ячейкой, находящейся в соответствующей позиции. При таком подходе более очевидно, как строка `BOARD_TEMPLATE` строит игровое поле, выводимое на экран.

Несмотря на различия в структурах данных, у этих двух программ много общего. Обе программы выводят свои структуры данных на экран, запрашивают у игрока входные данные, проверяют ввод, а затем используют его для обновления своих структур данных, прежде чем возвращаться к началу цикла. Однако код для выполнения всех этих действий можно написать многими способами. Как сделать свой код удобочитаемым? Удобочитаемость — субъективное ощущение, а не объективная метрика, определяемая степенью соответствия некоторому набору правил. Исходный код, приведенный в этой главе, показывает, что, хотя к любому коду с запахом всегда стоит присмотреться еще раз, признаки проблемы далеко не всегда указывают на существование проблемы, которую необходимо исправить. Удобочитаемость кода важнее бездумного следования политике недопустимости запахов кода в ваших программах.

ЧАСТЬ III

ОБЪЕКТНО- ОРИЕНТИРОВАННЫЙ PYTHON

15

Объектно-ориентированное программирование и классы



*Объектно-ориентированное программирование, или ООП, — механизм языка программирования, позволяющий группировать переменные и функции в новые типы данных, называемые **классами**. На базе классов создаются **объекты**. Распределяя свой код по классам, можно разбить монолитную программу на меньшие части, которые проще понять и отладить.*

В небольших программах ООП добавляет не столько структуру, сколько рутину. Хотя некоторые языки (например, Java) требуют организации всего кода в классах, ООП-функциональность в Python не является обязательной. Программист может воспользоваться классами, если они ему нужны, или забыть про классы, если без них можно обойтись.

В докладе разработчика Python Джека Дидериха (Jack Diederich) «Перестаньте писать классы» на конференции PyCon 2012 (<https://youtu.be/o9pEzgHorH0/>) рассматриваются некоторые ситуации, в которых программисты пишут классы, хотя можно было бы обойтись более простой функцией или модулем.

Как бы то ни было, вам как программисту следует знать основы классов и их использования. Из этой главы вы узнаете, что такое классы, почему они используются в программах и какой синтаксис и концепции программирования лежат в их основе. ООП — обширная тема, и эта глава содержит только краткое введение в нее.

Аналогия из реального мира: заполнение форм

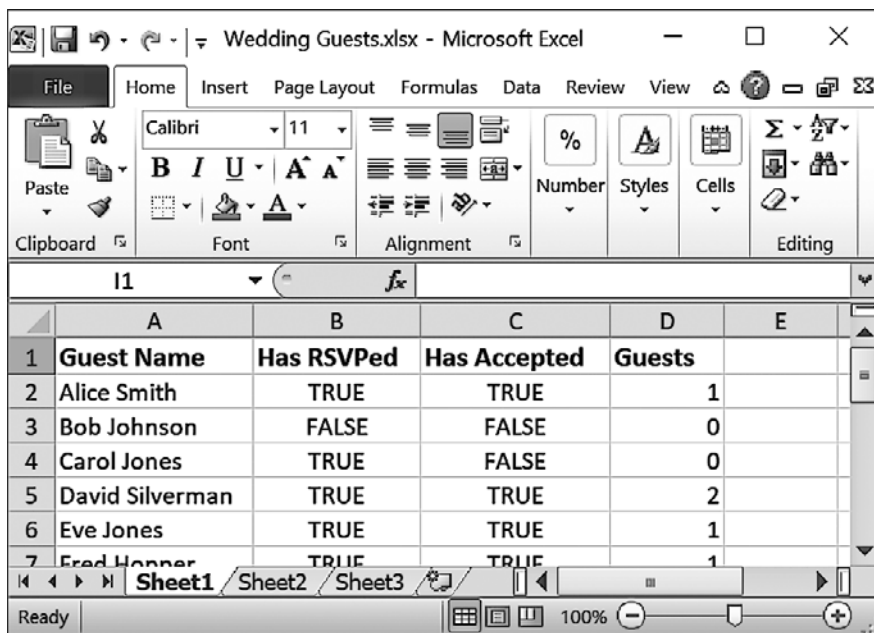
Скорее всего, вам неоднократно приходилось заполнять всевозможные формы, анкеты и бланки, бумажные или электронные: при посещении врача, для покупок в интернете или для приглашения на свадьбу. Формы предоставляют унифицированный механизм сбора необходимой информации людьми или организациями. Разные формы предназначены для сбора разных видов информации. Врач описывает состояние пациента, а на форме планирования свадьбы вы вводите информацию о приглашенных гостях.

В Python термины «класс», «тип» и «тип данных» имеют одинаковый смысл. Класс, как и бумажная или электронная форма, представляет собой заготовку для создания объектов Python (также называемых *экземплярами*). Объекты содержат данные, которые представляют конкретного пациента, покупку в интернет-магазине или гостя на свадьбе. Классы напоминают пустые формы, а объекты, созданные на базе этих классов, — заполненные формы с реальными данными, которые требует форма. Так, на рис. 15.1 форму для подтверждения участия можно сравнить с классом, а заполненную форму — с объектом.

Классы и объекты также можно рассматривать как электронные таблицы (рис. 15.2).



Рис. 15.1. Формы для приглашения гостей на свадьбу напоминают классы, а заполненные формы напоминают объекты



	A	B	C	D	E
1	Guest Name	Has RSVPed	Has Accepted	Guests	
2	Alice Smith	TRUE	TRUE	1	
3	Bob Johnson	FALSE	FALSE	0	
4	Carol Jones	TRUE	FALSE	0	
5	David Silverman	TRUE	TRUE	2	
6	Eve Jones	TRUE	TRUE	1	
7	Fred Hanner	TRUE	TRUE	1	

Рис. 15.2. Электронная таблица с данными гостей. Здесь RSVP означает «répondez s'il vous plaît» («просьба ответить»)

Заголовки столбцов определяют класс, а каждая отдельная строка таблицы — объект.

Классы и объекты часто приравнивают к моделям элементов реального мира, но не путайте карту с территорией. Содержимое класса зависит от того, что должна сделать программа. На рис. 15.3 изображены некоторые объекты разных классов, представляющие одного и того же человека. Не считая имени, они содержат совершенно разную информацию.

Кроме того, информация, содержащаяся в классах, зависит от потребностей вашей программы. Во многих учебниках ООП для примера используется класс `Car`, но при этом авторы забывают, что набор сведений, включаемых в класс, полностью зависит от того, какое приложение вы пишете. Не существует обобщенного класса `Car`, который бы включал метод `honkHorn()` (подать сигнал) или атрибут `numberOfCupholders` (количество подставок для стаканов) только потому, что этими характеристиками обладают реальные машины. Может быть, вы пишете веб-приложение для автосалона, видеоигру с гонками или модель дорожного движения. Класс `Car` для автосалона может содержать атрибуты `milesPerGallon` (количество миль на галлон) или `manufacturersSuggestedRetailPrice` (рекомендованная цена производителя), подобно тому как имена этих атрибутов могут быть включены в заголовки столбцов

в электронных таблицах автосалона. Но в видеоигре и модели дорожного движения этих атрибутов не будет, потому что здесь они не актуальны. Класс `Car` для видеоигры может содержать метод `explodeWithLargeFireball()` (эффектно взорваться), но в приложение для автосалона и модели дорожного движения он не попадет... хочется надеяться.

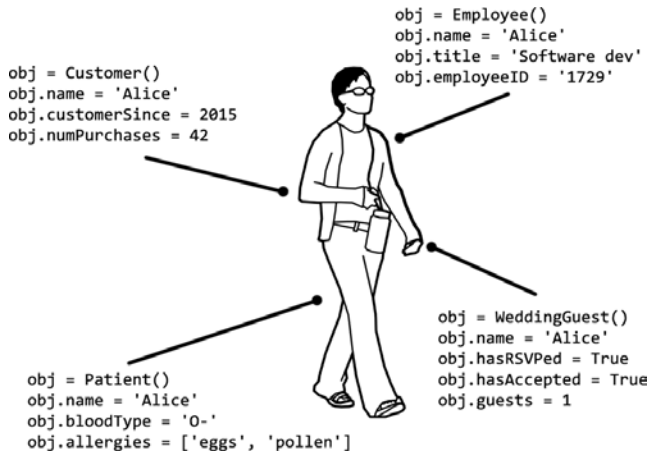


Рис. 15.3. Четыре объекта, созданные на базе разных классов. Объекты представляют одного и того же человека в зависимости от того, какая информация о человеке нужна приложению

Создание объектов на базе классов

Вы уже использовали классы и объекты в Python, даже если не создавали их сами. Вспомните модуль `datetime`, который содержит класс с именем `date`. Объекты класса `datetime.date` (также называемые объектами `datetime.date` или объектами `date`) представляют конкретную дату. Введите следующий фрагмент в интерактивной оболочке, чтобы создать объект класса `datetime.date`:

```

>>> import datetime
>>> birthday = datetime.date(1999, 10, 31) # Передать год, месяц и день.
>>> birthday.year
1999
>>> birthday.month
10
>>> birthday.day
31
>>> birthday.weekday() # weekday() - метод, на что указывают круглые скобки.
6

```

Атрибуты представляют собой переменные, связанные с объектом. Вызов `datetime.date()` создает новый объект `date`, инициализируемый аргументами `1999, 10, 31`, так что объект представляет дату 31 октября 1999 года. Эти аргументы присваиваются атрибутам `year`, `month` и `day` класса `date`; такие атрибуты присутствуют в каждом объекте `date`.

С подобной информацией метод `weekday()` класса может вычислить день недели. В приведенном примере он возвращает `6` — обозначение воскресенья, потому что согласно электронной документации Python возвращаемым значением `weekday()` является целое число от `0` (понедельник) до `6` (воскресенье). В документации также перечислены другие методы, содержащиеся в классе `date`. И хотя объект `date` содержит много атрибутов и методов, это все еще один объект, который можно сохранить в переменной, — такой как `birthday` в приведенном примере.

Создание простого класса: `WizCoin`

Создадим класс `WizCoin`, представляющий набор монет в вымышленной волшебной валюте. В этой валюте используются следующие номиналы: кнаты, сикли (29 кнатов) и галлеоны (17 сиклей, или 493 кната). Помните, что объекты класса `WizCoin` представляют количество монет разного номинала, а не денежную сумму. Условно говоря, этот класс сообщит, что у вас пять четвертаков и один гривенник, а не 1 р. 35 коп.

Создайте файл с именем `wizcoin.py` и введите следующий код для создания класса `WizCoin`. Обратите внимание: имя метода `__init__` начинается и завершается двумя символами подчеркивания (метод `__init__` рассматривается в подразделе «Методы, `__init__()` и `self`» этой главы):

```
class WizCoin:
    def __init__(self, galleons, sickles, knuts):
        """Создание нового объекта WizCoin по значениям galleons, sickles
и knuts."""
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts
        # ВНИМАНИЕ: методы __init__() НИКОГДА не содержат команду return.

    def value(self):
        """The value (in knuts) of all the coins in this WizCoin object."""
        return (self.galleons * 17 * 29) + (self.sickles * 29) + (self.knuts)

    def weightInGrams(self):
        """Возвращает вес монет в граммах."""
        return (self.galleons * 31.103) + (self.sickles * 11.34) + (self.knuts
* 5.0)
```

Программа определяет новый класс с именем `WizCoin` при помощи команды `class` ❶. При создании класса создается новый тип объектов. Использование команды

`class` для определения класса напоминает команду `def`, задающую новые функции. Внутри блока кода, следующего за командой `class`, следуют три определения трех методов: `__init__()` (*сокращение* от initializer) ❷, `value()` ❸ и `weightInGrams()` ❹. Обратите внимание: все методы имеют первый параметр с именем `self`, который будет рассмотрен в следующем разделе.

По общепринятым соглашениям имена модулей (например, `wizcoin` в файле `wizcoin.py`) записываются в нижнем регистре, а имена классов (например, `WizCoin`) начинаются с буквы верхнего регистра. К сожалению, некоторые классы стандартной библиотеки Python — такие как `date` — этому соглашению не следуют.

Чтобы потренироваться в создании новых объектов класса `WizCoin`, введите следующий исходный код в отдельном окне редактора и сохраните файл с именем `wcexample1.py` в одной папке с `wizcoin.py`:

```
import wizcoin

purse = wizcoin.WizCoin(2, 5, 99) # Целые числа передаются __init__(). ❶
print(purse)
print('G:', purse.galleons, 'S:', purse.sickles, 'K:', purse.knuts)
print('Total value:', purse.value())
print('Weight:', purse.weightInGrams(), 'grams')
print()

coinJar = wizcoin.WizCoin(13, 0, 0) # Целые числа передаются __init__(). ❷
print(coinJar)
print('G:', coinJar.galleons, 'S:', coinJar.sickles, 'K:', coinJar.knuts)
print('Total value:', coinJar.value())
print('Weight:', coinJar.weightInGrams(), 'grams')
```

Вызовы `WizCoin()` ❶ и ❷ создают объекты `WizCoin` и выполняют для них код метода `__init__()`. В аргументах `WizCoin()` передаются три целых числа, которые передаются параметрам `__init__()`. Эти аргументы присваиваются атрибутам `self.galleons`, `self.sickles` и `self.knuts` объекта. Подобно тому как функция `time.sleep()` требует сначала импортировать модуль `time` и поставить префикс `time.` перед именем функции, мы также должны импортировать `wizcoin` и поставить префикс `wizcoin.` перед именем функции `WizCoin()`.

Результат выполнения программы выглядит приблизительно так:

```
<wizcoin.WizCoin object at 0x000002136F138080>
G: 2 S: 5 K: 99
Total value: 1230
Weight: 613.906 grams

<wizcoin.WizCoin object at 0x000002136F138128>
G: 13 S: 0 K: 0
Total value: 6409
Weight: 404.339 grams
```

Если вы получите сообщение об ошибке (например, `ModuleNotFoundError: No module named 'wizcoin'`), убедитесь в том, что файлу присвоено имя `wizcoin.py` и он находится в одной папке с `wsexample1.py`.

Объекты `WizCoin` не имеют полезного строкового представления, поэтому при выводе `purse` и `coinJar` выводится адрес памяти в угловых скобках. (Из главы 17 вы узнаете, как изменить выводимое представление.)

Подобно тому как для объекта строки можно вызвать метод `lower()`, мы можем вызвать методы `value()` и `weightInGrams()` для объектов `WizCoin`, присвоенных переменным `purse` и `coinJar`. Эти методы вычисляют результат по значениям атрибутов `galleons`, `sickles` и `knuts` объекта.

Классы и ООП упрощают *сопровождение* кода — то есть код проще читается, изменяется и расширяется в будущем. Изучим методы и атрибуты этого класса более подробно.

Методы, `__init__()` и `self`

Методы представляют собой функции, связанные с объектами некоторого класса. Вспомните, что `lower()` является методом строк — это означает, что он вызывается для объектов строк. Метод `lower()` можно вызвать для строки (например, `'Hello'.lower()`), но вызвать его для списка (например, `['dog', 'cat'].lower()`) не получится. Также обратите внимание на то, что метод указывается после имени объекта: правильным считается код `'Hello'.lower()`, а не `lower('Hello')`. В отличие от метода `lower()` функция `len()` не связана с одним конкретным типом данных; при вызове `len()` можно передавать строки, списки, словари и многие другие типы объектов.

Как было показано в предыдущем разделе, объект создается вызовом имени класса как функции. Эта функция называется *функцией-конструктором* (или просто *конструктором*), потому что она конструирует новый объект. Также говорят, что конструктор создает новый экземпляр класса.

При вызове конструктора Python создает новый объект, а затем выполняет метод `__init__()`. Наличие метода `__init__()` в классе не обязательно, но он почти всегда присутствует. Именно в методе `__init__()` обычно задаются исходные значения атрибутов. Например, я снова приведу метод `__init__()` класса `WizCoin`:

```
def __init__(self, galleons, sickles, knuts):
    """Создание нового объекта WizCoin по значениям galleons, sickles и knuts."""
    self.galleons = galleons
    self.sickles = sickles
    self.knuts = knuts
    # ВНИМАНИЕ: методы __init__() НИКОГДА не содержат команду return.
```

Когда программа `wsexample1.py` вызывает `WizCoin(2, 5, 99)`, Python создает новый объект `WizCoin` и передает три аргумента (2, 5 и 99) вызову `__init__()`. Но

метод `__init__()` получает четыре параметра: `self`, `galleons`, `sickles` и `knuts`. Дело в том, что у каждого метода имеется первый параметр с именем `self`. Когда метод вызывается для объекта, этот объект автоматически передается в параметре `self`. Остальные аргументы присваиваются параметрам как обычно. Если вы увидите сообщение об ошибке вида `TypeError: __init__() takes 3 positional arguments but 4 were given` (`TypeError: __init__()` получает 3 позиционных аргумента, но задано 4), скорее всего, вы забыли добавить параметр `self` в команду `def` метода.

Присваивать первому параметру имя `self` необязательно; имя может быть любым. Однако имя `self` считается общепринятым, и выбор иного имени затруднит чтение вашего кода другими программистами Python. Когда вы читаете код, первый параметр `self` помогает быстро отличить методы от функций. Аналогичным образом, если в коде метода нигде не используется параметр `self`, это указывает на то, что, возможно, метод стоит оформить в виде функции.

Аргументы 2, 5 и 99 в вызове `WizCoin(2, 5, 99)` не присваиваются атрибутам нового объекта автоматически; чтобы это произошло, необходимо включить три команды присваивания в `__init__()`. Часто параметрам `__init__()` присваиваются имена, совпадающие с именами атрибутов, но наличие `self` в `self.galleons` означает, что это атрибут объекта, а `galleons` — параметр. Сохранение аргументов конструктора в атрибутах объекта — одна из типичных задач метода `__init__()` класса. Вызов `datetime.date()` в предыдущем разделе выполнял аналогичную операцию, хотя тогда передавались три аргумента для атрибутов `year`, `month` и `day` создаваемого объекта `date`.

Ранее мы вызвали функции `int()`, `str()`, `float()` и `bool()` для преобразования между типами данных — например, вызов `str(3.1415)` возвращал строковое значение `'3.1415'` для значения с плавающей точкой 3.1415. Ранее в тексте они описывались как функции, но `int`, `str`, `float` и `bool` в действительности являются классами, а `int()`, `str()`, `float()` и `bool()` — конструкторами, которые возвращают новые объекты целого числа, строки, числа с плавающей точкой и логического значения. Руководство по стилю Python рекомендует использовать для имен классов «верблously» схему с первой буквой в верхнем регистре, хотя многие встроенные классы Python этому соглашению не следуют.

Вызов функции-конструктора `WizCoin()` возвращает новый объект `WizCoin`, но метод `__init__()` не может содержать команды `return` с возвращаемым значением. При попытке добавить возвращаемое значение выдается ошибка `TypeError: __init__() should return None` (`TypeError: __init__()` должен возвращать `None`).

Атрибуты

Атрибутами называются переменные, связанные с объектом. В документации Python атрибут описывается как «любое имя, следующее после точки». Вспомните выражение `birthday.year` из предыдущего раздела: атрибут `year` — имя, следующее после точки.

Каждый объект содержит собственный набор атрибутов. Когда программа `wsexample1.py` создает два объекта `WizCoin` и сохраняет их в переменных `purse` и `coinJar`, их атрибуты имеют разные значения. К ним можно обращаться и присваивать значения, как и к любой другой переменной. Чтобы потренироваться в присваивании значений атрибутов, откройте в редакторе окно с новым файлом и введите следующий код, сохранив его в файле `wsexample2.py` в одной папке с файлом `wizcoin.py`:

```
import wizcoin

change = wizcoin.WizCoin(9, 7, 20)
print(change.sickles) # Выводит 7.
change.sickles += 10
print(change.sickles) # Выводит 17.

pile = wizcoin.WizCoin(2, 3, 31)
print(pile.sickles) # Выводит 3.
pile.someNewAttribute = 'a new attr' # Создается новый атрибут.
print(pile.someNewAttribute)
```

При выполнении этой программы результат выглядит так:

```
7
17
3
a new attr
```

Атрибуты объекта также можно сравнить с ключами словаря. Вы можете читать и изменять связанные с ними значения, а также присваивать новые атрибуты объекту. Формально методы также считаются атрибутами класса.

Приватные атрибуты и приватные методы

В таких языках, как C++ или Java, атрибуты могут помечаться как имеющие *приватный* уровень доступа. Это означает, что компилятор или интерпретатор позволит обращаться к атрибутам объектов этого класса только коду методов этого класса. В языке Python такого ограничения не существует. Все атрибуты и методы фактически имеют открытый уровень доступа: код за пределами класса может обратиться к любым атрибутам любых объектов этого класса и изменять их.

Впрочем, приватный доступ полезен. Например, объекты класса `BankAccount` могут содержать атрибут `balance`, который должен быть доступен только для методов класса `BankAccount`. По этим причинам в Python принято начинать имена приватных атрибутов и методов с одного символа подчеркивания. Технически ничто не мешает коду за пределами класса обращаться к приватным атрибутам и методам, но на практике лучше обращаться к ним только из методов класса.

Откройте в редакторе окно с новым файлом, введите следующий код и сохраните его с именем `privateExample.py`. В нем объекты класса `BankAccount` содержат приватные атрибуты `_name` и `_balance`, к которым должны обращаться напрямую только методы `deposit()` и `withdraw()`:

```
class BankAccount:
    def __init__(self, accountHolder):
        # Методы BankAccount могут обращаться к self._balance, но код
        # за пределами класса этого делать не должен:
        self._balance = 0 ❶
        self._name = accountHolder ❷
        with open(self._name + 'Ledger.txt', 'w') as ledgerFile:
            ledgerFile.write('Balance is 0\n')

    def deposit(self, amount):
        if amount <= 0: ❸
            return # Отрицательные "зачисления" недопустимы.
        self._balance += amount
        with open(self._name + 'Ledger.txt', 'a') as ledgerFile: ❹
            ledgerFile.write('Deposit ' + str(amount) + '\n')
            ledgerFile.write('Balance is ' + str(self._balance) + '\n')

    def withdraw(self, amount):
        if self._balance < amount or amount < 0: ❺
            return # Не хватает средств на счете или снимается
            # отрицательная сумма.
        self._balance -= amount
        with open(self._name + 'Ledger.txt', 'a') as ledgerFile: ❻
            ledgerFile.write('Withdraw ' + str(amount) + '\n')
            ledgerFile.write('Balance is ' + str(self._balance) + '\n')

acct = BankAccount('Alice') # Создание учетного счета.
acct.deposit(120) # _balance можно изменять через deposit()
acct.withdraw(40) # _balance можно изменять через withdraw()

# Изменение _name или _balance за пределами BankAccount нежелательно, но возможно:
acct._balance = 1000000000 ❼
acct.withdraw(1000)

acct._name = 'Bob' # Теперь изменяется счет Боба! ❸
acct.withdraw(1000) # Операция регистрируется в BobLedger.txt!
```

При выполнении программы `privateExample.py` создаваемые файлы содержат некорректную информацию, потому что `_balance` и `_name` изменялись за пределами класса, что привело к недействительным состояниям. Файл `AliceLedger.txt` содержит непонятно откуда взявшуюся огромную сумму:

```
Balance is 0
Deposit 120
Balance is 120
Withdraw 40
```

```
Balance is 80
Withdraw 1000
Balance is 999999000
```

Файл `BobLedger.txt` содержит необъяснимый баланс, хотя мы никогда не создавали объект `BankAccount` для пользователя `Bob`:

```
Withdraw 1000
Balance is 999998000
```

Хорошо спроектированные классы в целом автономны, и они должны предоставлять методы для присваивания атрибутам допустимых значений. Атрибуты `_balance` и `_name` помечены как приватные ❶ и ❷, а значение класса `BankAccount` должно изменяться только при помощи методов `deposit()` и `withdraw()`. Эти два метода содержат проверки ❸ и ❹, которые проверяют, что атрибут `_balance` не переводится в недействительное состояние (например, ему не присваивается отрицательное целое значение). Методы также регистрируют каждую операцию для текущего баланса ❺ и ❻.

Код за пределами класса, изменяющий эти атрибуты (например, команды `acct._balance = 1000000000` ❺ или `acct._name = 'Bob'` ❹), может перевести объект в некорректное состояние и создать ошибки. Соблюдение соглашений об использовании префикса `_` для приватного доступа упрощает отладку. Вы точно знаете: ошибку нужно искать в коде класса, а не в коде всей программы.

В отличие от Java и других языков, Python не требует определения открытых `get-` и `set-`методов для приватных атрибутов. Вместо этого в Python используются свойства (см. главу 17).

Функция `type()` и атрибут `__qualname__`

Передав объект встроенной функции `type()`, вы узнаете тип данных объекта по возвращаемому значению этой функции. Объекты, возвращаемые функцией `type()`, называются *объектами типов* (также встречается термин «*объекты классов*»). Вспомните, что термины «тип», «тип данных» и «класс» в Python обозначают одно и то же. Чтобы увидеть, что возвращает функция `type()` для разных значений, введите следующий фрагмент в интерактивной оболочке:

```
>>> type(42) # Объект 42 имеет тип int.
<class 'int'>
>>> int # int - объект типа для целого типа данных.
<class 'int'>
>>> type(42) == int # Проверка типа: является ли 42 целым числом?
True
>>> type('Hello') == int # Проверка типа: имеет ли 'Hello' тип int?
False
>>> import wizcoin
```

```
>>> type(42) == wizcoin.WizCoin # Проверка типа: имеет ли 42 тип WizCoin?
False
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> type(purse) == wizcoin.WizCoin # Проверка типа: имеет ли purse тип WizCoin?
True
```

Обратите внимание: `int` является объектом типа и этот же объект возвращается вызовом `type(42)`, но он также может вызываться как функция-конструктор `int()`: функция `int('42')` не преобразует строковый аргумент `'42'`. Вместо этого она возвращает объект целого числа, соответствующий аргументу.

Допустим, вы хотите сохранить некоторую информацию о переменных в вашей программе, которая позднее пригодится при отладке. В файл журнала можно записывать только строковые данные, но при передаче объекта типа `str()` будет возвращена непонятная строка. Вместо этого следует использовать атрибут `__qualname__`, который имеется у всех объектов типов, для сохранения в журнале более простой и удобочитаемой строки:

```
>>> str(type(42)) # При передаче объекта типа str() возвращает непонятную строку.
"<class 'int'>"
>>> type(42).__qualname__ # Атрибут __qualname__ предоставляет более понятную
информацию.
'int'
```

Атрибут `__qualname__` чаще всего используется для переопределения метода `__repr__()`, более подробно рассматриваемого в главе 17.

Примеры программирования с применением ООП и без него: «Крестики-нолики»

На первый взгляд, трудно понять, как использовать классы в программах. Рассмотрим пример короткой программы для игры «Крестики-нолики», которая не использует классы, а потом перепишем ее с классами.

Откройте в редакторе окно с новым файлом, введите следующую программу и сохраните ее с именем `tictactoe.py`:

```
# tictactoe.py, реализация без ООП.
```

```
ALL_SPACES = list('123456789') # Ключи для словаря с игровым полем.
X, O, BLANK = 'X', 'O', ' ' # Константы для строковых значений.

def main():
    """Проводит игру в крестики-нолики."""
    print('Welcome to tic-tac-toe!')
    gameBoard = getBlankBoard() # Создать словарь с игровым полем.
    currentPlayer, nextPlayer = X, O # X ходит первым, O ходит вторым.
```

```

while True:
    print(getBoardStr(gameBoard)) # Вывести игровое поле на экран.

    # Запрашивать ход, пока игрок не введет число от 1 до 9:
    move = None
    while not isValidSpace(gameBoard, move):
        print(f'What is {currentPlayer}\''s move? (1-9)')
        move = input()
    updateBoard(gameBoard, move, currentPlayer) # Сделать ход.

    # Проверить окончание игры:
    if isWinner(gameBoard, currentPlayer): # Сначала проверяем победу.
        print(getBoardStr(gameBoard))
        print(currentPlayer + ' has won the game!')
        break
    elif isBoardFull(gameBoard): # Затем проверяется ничья.
        print(getBoardStr(gameBoard))
        print('The game is a tie!')
        break
    currentPlayer, nextPlayer = nextPlayer, currentPlayer # Передать ход.
    print('Thanks for playing!')

def getBlankBoard():
    """Создает пустое игровое поле для игры крестики-нолики."""
    board = {} # Поле представляется словарем Python.
    for space in ALL_SPACES:
        board[space] = BLANK # Все поля в исходном состоянии пусты.
    return board

def getBoardStr(board):
    """Возвращает текстовое представление игрового поля."""
    return f'''
    {board['1']}|{board['2']}|{board['3']}  1 2 3
    -+-+
    {board['4']}|{board['5']}|{board['6']}  4 5 6
    -+-+
    {board['7']}|{board['8']}|{board['9']}  7 8 9'''

def isValidSpace(board, space):
    """Возвращает True, если задан допустимый номер клетки,
    и эта клетка пуста."""
    return space in ALL_SPACES or board[space] == BLANK

def isWinner(board, player):
    """Возвращает True, если игрок победил на заданном поле."""
    b, p = board, player # Более короткие имена для удобства.
    # Проверяем 3 знака по 3 строкам, 3 столбцам и 2 диагоналям.
    return ((b['1'] == b['2'] == b['3'] == p) or # Верхняя строка
            (b['4'] == b['5'] == b['6'] == p) or # Средняя строка
            (b['7'] == b['8'] == b['9'] == p) or # Нижняя строка
            (b['1'] == b['4'] == b['7'] == p) or # Левый столбец

```



```

        (b['2'] == b['5'] == b['8'] == p) or # Средний столбец
        (b['3'] == b['6'] == b['9'] == p) or # Правый столбец
        (b['3'] == b['5'] == b['7'] == p) or # Диагональ
        (b['1'] == b['5'] == b['9'] == p)) # Диагональ

def isBoardFull(board):
    """Возвращает True, если заняты все клетки игрового поля."""
    for space in ALL_SPACES:
        if board[space] == BLANK:
            return False # Если есть хотя бы одна пустая клетка, вернуть False.
    return True # Пустых клеток не осталось, вернуть True.

def updateBoard(board, space, mark):
    """Заполняет клетку игрового поля знаком mark."""
    board[space] = mark

if __name__ == '__main__':
    main() # Выполняет main(), если модуль был запущен (а не импортирован).
```

При запуске программы вывод выглядит примерно так:

```

Welcome to tic-tac-toe!
  | |  1 2 3
--+--
  | |  4 5 6
--+--
  | |  7 8 9
What is X's move? (1-9)
1
X| |  1 2 3
--+--
  | |  4 5 6
--+--
  | |  7 8 9
What is O's move? (1-9)
--snip--
X| |O  1 2 3
--+--
 |O|  4 5 6
--+--
X|O|X  7 8 9
What is X's move? (1-9)
4
X| |O  1 2 3
--+--
X|O|  4 5 6
--+--
X|O|X  7 8 9
X has won the game!
Thanks for playing!
```

Для представления девяти клеток игрового поля в программе используется объект словаря. Ключами словаря являются строки '1'–'9', а значениями — строки 'X', 'O' и ' '. Нумерация клеток соответствует расположению цифр на клавиатуре телефона.

Функции в программе `tictactoe.py` делают следующее.

- Функция `main()` содержит код, который создает новую структуру данных игрового поля (хранящуюся в переменной `gameBoard`) и вызывает другие функции программы.
- Функция `getBlankBoard()` возвращает словарь со значениями, инициализированными ' ' (пустое поле).
- Функция `getBoardStr()` получает словарь, представляющий игровое поле, и возвращает представление игрового поля в виде многострочного текста, которое может быть выведено на экран. Именно эта функция формирует текст игрового поля, выводимый игрой.
- Функция `isValidSpace()` возвращает `True`, если ей передан допустимый номер клетки и эта клетка пуста.
- В параметрах функция `isWinner()` получает словарь игрового поля и символ 'X' или 'O'. Она определяет, поставил ли конкретный игрок три знака в ряд на поле.
- Функция `isBoardFull()` проверяет, что на поле не осталось пустых клеток; это означает, что игра закончилась. Функция `updateBoard()` получает в параметрах словарь, пробел и обозначение игрока (X или O) и обновляет словарь.

Обратите внимание: многие функции получают в первом параметре переменную `board`. Это указывает на то, что эти функции связаны друг с другом в том смысле, что все они работают с одной структурой данных.

Когда несколько функций в коде работают с одной структурой данных, обычно лучше сгруппировать их как методы и атрибуты класса. Переработаем программу `tictactoe.py`, чтобы в ней использовался класс `TTTBoard`. В атрибуте `spaces` этого класса будет храниться словарь `board`. Функции, получающие `board` в параметре, станут методами класса `TTTBoard`, а вместо параметра `board` они будут использовать параметр `self`.

Откройте в редакторе окно с новым файлом, введите следующую программу и сохраните ее с именем `tictactoe_oop.py`:

```
# tictactoe_oop.py, объектно-ориентированная реализация игры.

ALL_SPACES = list('123456789') # Ключи для словаря с игровым полем.
X, O, BLANK = 'X', 'O', ' '     # Константы для строковых значений.
```

```

def main():
    """Проводит игру в крестики-нолики."""
    print('Welcome to tic-tac-toe!')
    gameBoard = TTTBoard() # Создать объект игрового поля.
    currentPlayer, nextPlayer = X, O # X ходит первым, O ходит вторым.

    while True:
        print(gameBoard.getBoardStr()) # Вывести игровое поле на экран.

        # Запрашивать ход, пока игрок не введет число от 1 до 9:
        move = None

        while not gameBoard.isValidSpace(move):
            print(f'What is {currentPlayer}'s move? (1-9)')
            move = input()
        gameBoard.updateBoard(move, currentPlayer) # Сделать ход.

        # Проверить окончание игры:
        if gameBoard.isWinner(currentPlayer): # Сначала проверяем победу.
            print(gameBoard.getBoardStr())
            print(currentPlayer + ' has won the game!')
            break
        elif gameBoard.isBoardFull(): # Затем проверяется ничья.
            print(gameBoard.getBoardStr())
            print('The game is a tie!')
            break
        currentPlayer, nextPlayer = nextPlayer, currentPlayer # Передать ход.
    print('Thanks for playing!')

class TTTBoard:
    def __init__(self, usePrettyBoard=False, useLogging=False):
        """Создает пустое игровое поле для игры крестики-нолики."""
        self._spaces = {} # Поле представляется словарем Python.
        for space in ALL_SPACES:
            self._spaces[space] = BLANK # Все поля в исходном состоянии пусты.

    def getBoardStr(self):
        """Возвращает текстовое представление игрового поля."""
        return f'''
{self._spaces['1']}|{self._spaces['2']}|{self._spaces['3']}  1 2 3
--+--+
{self._spaces['4']}|{self._spaces['5']}|{self._spaces['6']}  4 5 6
--+--+
{self._spaces['7']}|{self._spaces['8']}|{self._spaces['9']}  7 8 9'''

    def isValidSpace(self, space):
        """Возвращает True, если задан допустимый номер клетки
и эта клетка пуста."""
        return space in ALL_SPACES and self._spaces[space] == BLANK

    def isWinner(self, player):
        """Возвращает True, если игрок победил на заданном поле."""

```

```

s, p = self._spaces, player # Более короткие имена для удобства.
# Проверяем 3 знака по 3 строкам, 3 столбцам и 2 диагоналям.
return ((s['1'] == s['2'] == s['3'] == p) or # Верхняя строка
        (s['4'] == s['5'] == s['6'] == p) or # Средняя строка
        (s['7'] == s['8'] == s['9'] == p) or # Нижняя строка
        (s['1'] == s['4'] == s['7'] == p) or # Левый столбец
        (s['2'] == s['5'] == s['8'] == p) or # Средний столбец
        (s['3'] == s['6'] == s['9'] == p) or # Правый столбец
        (s['3'] == s['5'] == s['7'] == p) or # Диагональ
        (s['1'] == s['5'] == s['9'] == p)) # Диагональ

def isBoardFull(self):
    """Возвращает True, если заняты все клетки игрового поля."""
    for space in ALL_SPACES:
        if self._spaces[space] == BLANK:
            return False # Если есть хотя бы одна пустая клетка, вернуть False.
    return True # Пустых клеток не осталось, вернуть True.

def updateBoard(self, space, player):
    """Заполняет клетку игрового поля знаком игрока."""
    self._spaces[space] = player

if __name__ == '__main__':
    main() # Выполняет main(), если модуль был запущен (а не импортирован).

```

С точки зрения функциональности эта программа не отличается от реализации, не использующей ООП. Вывод выглядит идентично. Код, который ранее находился в `getBlankBoard()`, был перемещен в метод `__init__()` класса `TTTBoard`, потому что они выполняют одну задачу инициализации структуры данных игрового поля. Другие функции были преобразованы в методы, параметр `self` заменил старый параметр `board`, потому что они служат одной цели: это блоки кода, работающие со структурой данных игрового поля.

Когда коду этих методов потребуется изменить словарь, хранящийся в атрибуте `_spaces`, он использует выражение `self._spaces`. Если код этих методов должен вызвать другие методы, перед этими вызовами также указывается имя `self` и точка (подобно тому как при вызове метода `coinJars.values()` в разделе «Создание простого класса: `WizCoin`» переменная `coinJars` содержит объект). В этом примере объект, содержащий вызываемый метод, хранится в переменной `self`.

Также обратите внимание на то, что имя атрибута `_spaces` начинается с символа подчеркивания; это означает, что все обращения к нему или его изменение должны выполняться только из кода методов `TTTBoard`. Код за пределами класса должен изменять `_spaces` только косвенно — вызовом соответствующих методов.

Полезно сравнить исходный код двух реализаций игры. Вы можете это сделать в книге или прочитать о параллельном сравнении двух версий на <https://autbor.com/compareoop/>.

«Крестики-нолики» — небольшая программа, и понять ее несложно. А если бы программа состояла из десятков тысяч строк с сотнями разных функций? Программу с несколькими десятками классов проще понять, чем программу с сотнями никак не связанных функций. ООП разбивает сложную программу на более понятные фрагменты.

Трудности проектирования классов для проектов реального мира

Проектирование класса, как и проектирование бумажной или электронной формы, — это обманчиво прямолинейное дело. Формы и классы по своей сути являются упрощенными представлениями реальных объектов. Вопрос в том, как именно упрощать объекты? Например, если вы создаете класс `Customer`, представляющий клиента, в него нужно включить атрибуты имени и фамилии `firstName` и `lastName`, не так ли? Однако в действительности создавать классы для моделирования реальных объектов весьма непросто. В большинстве западных стран фамилия человека указывается после имени, но в Китае — до имени. Если вы не хотите терять более миллиарда потенциальных клиентов, как изменить класс `Customer`? Стоит ли заменить имена `firstName` и `lastName` на `givenName` и `familyName`? Но в некоторых культурах фамилии у людей вообще нет. Например, у бывшего генерального секретаря ООН У Тана (он родом из Бирмы) фамилии нет: Тан — собственное имя, а У — начальный слог собственного имени его отца. Также нужно хранить возраст клиента, но значение атрибута `age` быстро устаревает; вместо этого лучше вычислять возраст каждый раз, когда он потребуется, по дате рождения в атрибуте `birthdate`. Реальный мир сложен, как и проектирование форм и классов для отражения этой сложности в унифицированной структуре, с которой могут работать наши программы. Форматы телефонных номеров также зависят от конкретной страны. ZIP-коды неприменимы к адресам за пределами Соединенных Штатов. Ограничение максимального количества символов в названиях городов может создать проблемы для немецкого городка Шмедесуртервестердейч. В Австралии и Новой Зеландии X считается допустимым гендерным обозначением. Утконос — млекопитающее, которое несет яйца. Арахис — не орех. Хотдог может быть или не быть сэндвичем в зависимости от того, кого вы спросите. Вам как программисту, который пишет программы для реального мира, придется иметь дело со всеми этими сложностями.

Если вы захотите больше узнать обо всем этом, я рекомендую доклад «Schemas for the Real World» Карины Зона (Carina C. Zona) на конференции PyCon 2015 (<https://youtu.be/PYYfVqtcWQY/>) и доклад «Hi! My name is...» Джеймса Беннета (James Bennett) на конференции North Bay Python 2018 (<https://youtu.be/NlebellpdYk/>). Также заслуживают внимания популярные публикации в блоге «Falsehoods Programmers Believe»; в них рассматриваются такие темы, как карты,

адреса электронной почты и другие виды данных, которые программисты часто представляют неправильно. Подборка ссылок на эти статьи доступна на <https://github.com/kdeldycke/awesome-falsehood/>. Кроме того, удачный пример неудачного отражения сложности реального мира показан в видеоролике CGP Grey «Social Security Cards Explained» (<https://youtu.be/Erp8IAUouus/>).

Итоги

ООП — полезный механизм организации вашего кода. Классы позволяют группировать данные и код в новые типы данных. Также на базе классов можно создавать объекты, вызывая их конструкторы (имя класса, вызываемое как функция), которые в свою очередь вызывают метод `__init__()` класса. Методы представляют собой функции, связанные с объектами, а атрибуты — переменные, связанные с объектами. Все методы получают первый параметр `self`, которому присваивается текущий объект при вызове метода. Это позволяет методам присваивать значения атрибутам объекта и вызывать его методы.

Хотя Python не позволяет задать приватный или открытый уровень доступа для атрибутов, в языке принято использовать префикс `_` для любых методов и атрибутов, которые должны вызываться или к которым следует обращаться из собственных методов класса. Соблюдение этого соглашения поможет предотвратить некорректное использование класса и перевод его в недействительное состояние, которое может привести к ошибкам. Вызов `type(obj)` возвращает объект класса для типа `obj`. Объекты класса включают атрибут `__qualname__`, который содержит строку с удобочитаемой формой имени класса.

Возможно, к этому моменту у вас возник вопрос: зачем вообще нужны хлопоты с классами, атрибутами или методами, когда все то же доступно с помощью функций? ООП — полезный механизм организации кода в нечто большее, чем обычный файл `.py` с сотней функций. Разбивая программу на несколько хорошо спроектированных классов, вы можете сосредоточиться на каждом классе по отдельности.

Методология ООП ориентирована на структуры данных и методы работы с этими структурами данных. Эта методология не является обязательной для всех программ, и, безусловно, злоупотребления ООП тоже возможны. Однако ООП позволяет использовать некоторые нетривиальные механизмы, о которых мы поговорим в следующих двух главах. Глава 16 посвящена первому из этих механизмов — *наследованию*.

16

Объектно-ориентированное программирование и наследование



Определение функции и вызов ее в нескольких местах программы избавляет от копирования исходного кода. Отказ от дублирования кода — полезная практика, потому что, если вам потребуется изменить этот код (чтобы исправить ошибку или добавить новые возможности), изменения достаточно внести только в одном месте. Также без дублирования кода программа становится более короткой и удобочитаемой.

Наследование (inheritance) представляет собой метод повторного использования кода, который применяется к классам. Это механизм организации классов в системе отношений «родитель — потомок», в которой дочерний класс наследует копию методов своего родительского класса, что избавляет вас от необходимости дублировать код метода в нескольких классах.

Многие программисты считают наследование переоцененным и даже опасным из-за дополнительной сложности, которую добавляют в программу большие иерархии наследования. Когда вы встречаете в блогах публикации типа «Наследование — зло», не стоит полагать, что они совершенно не обоснованы: да, наследованием можно злоупотреблять. Тем не менее разумное применение наследования может сильно сэкономить время при организации вашего кода.

Как работает наследование

Чтобы создать новый дочерний класс, укажите имя существующего родительского класса в круглых скобках в команде `class`. Чтобы потренироваться в создании

дочерних классов, откройте в редакторе окно нового файла, введите следующую программу и сохраните ее с именем `inheritanceExample.py`:

```
class ParentClass:                                ❶
    def printHello(self):                          ❷
        print('Hello, world!')

class ChildClass(ParentClass):                     ❸
    def someNewMethod(self):
        print('ParentClass objects don't have this method.')

class GrandchildClass(ChildClass):                 ❹
    def anotherNewMethod(self):
        print('Only GrandchildClass objects have this method.')

print('Create a ParentClass object and call its methods:')
parent = ParentClass()
parent.printHello()

print('Create a ChildClass object and call its methods:')
child = ChildClass()
child.printHello()
child.someNewMethod()

print('Create a GrandchildClass object and call its methods:')
grandchild = GrandchildClass()
grandchild.printHello()
grandchild.someNewMethod()
grandchild.anotherNewMethod()

print('An error:')
parent.someNewMethod()
```

При выполнении этой программы результат выглядит примерно так:

```
Create a ParentClass object and call its methods:
Hello, world!
Create a ChildClass object and call its methods:
Hello, world!
ParentClass objects don't have this method.
Create a GrandchildClass object and call its methods:
Hello, world!
ParentClass objects don't have this method.
Only GrandchildClass objects have this method.
An error:
Traceback (most recent call last):
  File "inheritanceExample.py", line 35, in <module>
    parent.someNewMethod() # ParentClass objects don't have this method.
AttributeError: 'ParentClass' object has no attribute 'someNewMethod'
```


Мы создали три класса с именами `ParentClass` ❶, `ChildClass` ❸ и `GrandchildClass` ❹. `ChildClass` *субклассирует* `ParentClass`; это означает, что `ChildClass` содержит те же методы, что и `ParentClass`. Мы говорим, что `ChildClass` наследует методы от `ParentClass`. Кроме того, класс `GrandchildClass` субклассирует `ChildClass`, поэтому он содержит все методы `ChildClass` и его родителя `ParentClass`.

Используя механизм наследования, мы фактически скопировали код метода `printHello()` ❷ в классы `ChildClass` и `GrandchildClass`. Любые изменения, вносимые в код `printHello()`, воздействуют не только на `ParentClass`, но и на `ChildClass` и `GrandchildClass`. Происходящее можно сравнить с изменением кода функции, который обновляет все ее вызовы. Эти отношения показаны на рис. 16.1. Обратите внимание: на диаграмме классов стрелка ведет от субкласса к базовому классу. Такое обозначение отражает тот факт, что класс всегда знает свой базовый класс, но не знает свои субклассы.

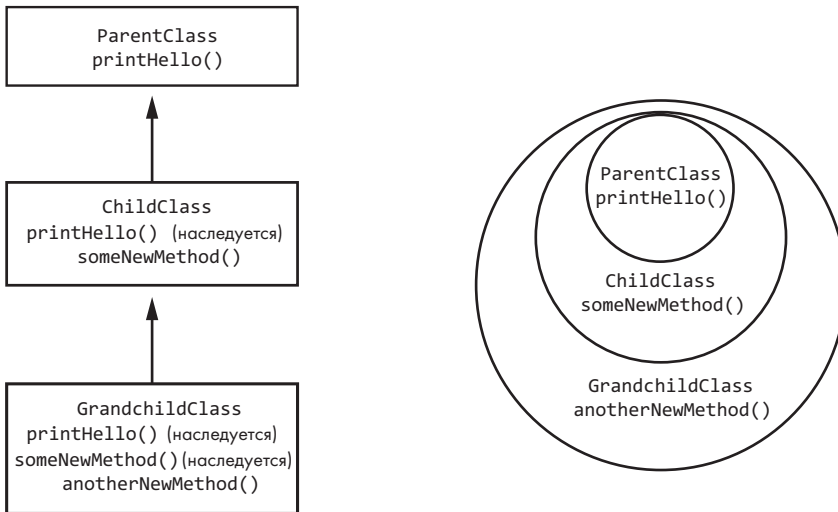


Рис. 16.1. Иерархическая диаграмма (слева) и диаграмма Венна (справа), изображающие отношения трех классов и их методов

Обычно говорят, что родительский и дочерний классы образуют отношения «является <частным случаем>». Объект `ChildClass` является объектом `ParentClass`, потому что он содержит все те же методы, которые содержит объект `ParentClass`, а также некоторые дополнительные методы. Отношения являются односторонними: объект `ParentClass` не является объектом `ChildClass`. Если объект `ParentClass` попытается вызвать метод `someNewMethod()`, существующий только для объектов `ChildClass` (и субклассов `ChildClass`), Python выдает ошибку `AttributeError`.

Программисты часто считают, что взаимосвязанные классы должны образовывать некоторую иерархию из реального мира. В учебниках ООП связи между родительскими, дочерними и «внучатыми» классами часто объясняются на примере иерархий Животное ► Птица ► Ласточка, Фигура ► Прямоугольник ► Квадрат и т. д. Но я напому, что главной целью наследования является повторное использование кода. Если вашей программе нужен класс с набором методов, который является полным надмножеством методов другого класса, наследование позволит избежать копирования кода.

Дочерние классы также иногда называются *производными классами*, или *субклассами*, а родительские классы — *базовыми классами*, или *суперклассами*.

Переопределение методов

Дочерние классы наследуют все методы своих родительских классов. Но дочерний класс может переопределить унаследованный метод, предоставляя собственный метод с собственным кодом. Имя переопределяющего метода дочернего класса совпадает с именем метода родительского класса.

Для демонстрации этой концепции вернемся к игре «Крестики-нолики», созданной в предыдущей главе. На этот раз мы создадим новый класс `MiniBoard`, который субклассирует `TTTBoard` и переопределяет `getBoardStr()` для вывода уменьшенного изображения игрового поля. Программа предлагает игроку выбрать стиль игрового поля. Копировать остальные методы `TTTBoard` не нужно, потому что `MiniBoard` наследует их.

Добавьте следующий фрагмент в конец файла `tictactoe_oop.py`, чтобы создать дочерний класс, производный от `TTTBoard`, а затем переопределить метод `getBoardStr()`:

```
class MiniBoard(TTTBoard):
    def getBoardStr(self):
        """Возвращает уменьшенное текстовое представление игрового поля."""
        # Пробелы заменяются символами '.'
        for space in ALL_SPACES:
            if self._spaces[space] == BLANK:
                self._spaces[space] = '.'

        boardStr = f'''
            {self._spaces['1']}{self._spaces['2']}{self._spaces['3']} 123
            {self._spaces['4']}{self._spaces['5']}{self._spaces['6']} 456
            {self._spaces['7']}{self._spaces['8']}{self._spaces['9']} 789'''

        # Символы '.' снова заменяются пробелами.
        for space in ALL_SPACES:
            if self._spaces[space] == '.':
                self._spaces[space] = BLANK
        return boardStr
```

Как и метод `getBoardStr()` класса `TTTBoard`, метод `getBoardStr()` класса `MiniBoard` создает многострочное представление игрового поля, которое выводится при передаче функции `print()`. Но эта строка намного компактнее, в ней отсутствуют линии между знаками X и O, а пустые клетки обозначаются точками.

Измените строку `main()` так, чтобы она создавала экземпляр объекта `MiniBoard` вместо объекта `TTTBoard`:

```
if input('Use mini board? Y/N: ').lower().startswith('y'):
    gameBoard = MiniBoard() # Создать объект MiniBoard.
else:
    gameBoard = TTTBoard() # Создать объект TTTBoard.
```

Не считая изменения одной строки в `main()`, остальной код программы работает так же, как прежде. Теперь при запуске программы вывод выглядит примерно так:

```
Welcome to Tic-Tac-Toe!
Use mini board? Y/N: y
    ... 123
    ... 456
    ... 789
What is X's move? (1-9)
1
    X.. 123
    ... 456
    ... 789
What is O's move? (1-9)
--snip--
    XXX 123
    .OO 456
    O.X 789
X has won the game!
Thanks for playing!
```

Программа легко адаптируется для включения обеих реализаций классов игрового поля. Конечно, если вам нужна *только* мини-версия игрового поля, вы могли легко заменить код метода `getBoardStr()` для `TTTBoard`. Но если вам нужны *обе* версии, наследование позволяет легко создать два класса за счет повторного использования общего кода.

Если бы мы не использовали наследование, можно было бы, скажем, добавить в `TTTBoard` новый атрибут с именем `useMiniBoard` и включить в `getBoardStr()` команду `if-else` для выбора одного из двух вариантов игрового поля (обычного или компактного). Для простого изменения такое решение могло бы сработать. Но что, если субкласс `MiniBoard` должен переопределить 2, 3 или даже 100 методов? Как быть, если вы захотите создать несколько разных субклассов `TTTBoard`? Отказ

от наследования вызовет стремительное размножение команд `if-else` внутри методов и заметно усложнит код. Использование субклассов и переопределения методов позволяет лучше разбить код на субклассы, обрабатывающие эти разные сценарии использования.

Функция `super()`

Переопределенный метод дочернего класса часто бывает похож на метод родительского класса. И хотя наследование является средством повторного использования кода, переопределение метода может заставить вас переписать код метода родительского класса как часть кода метода дочернего класса. Чтобы предотвратить дублирование кода, встроенная функция `super()` позволяет переопределяющему методу вызвать исходный метод родительского класса.

Например, создадим новый класс с именем `HintBoard`, который субклассирует `TTTBoard`. Новый класс переопределяет `getBoardStr()`, чтобы после вывода игрового поля также добавлялась подсказка о том, может ли X или O выиграть при своем следующем ходе. Это означает, что метод `getBoardStr()` класса `HintBoard` должен сделать все, что делает метод `getBoardStr()` класса `TTTBoard` для вывода игрового поля. Вместо повторения кода можно воспользоваться вызовом `super()`, чтобы вызвать метод `getBoardStr()` класса `TTTBoard` из метода `getBoardStr()` класса `HintBoard`. Добавьте следующий фрагмент в конец файла `tictactoe_oop.py`:

```
class HintBoard(TTTBoard):
    def getBoardStr(self):
        """Возвращает текстовое представление игрового поля с подсказкой."""
        boardStr = super().getBoardStr() # Вызвать getBoardStr() в TTTBoard.❶

        xCanWin = False
        oCanWin = False
        originalSpaces = self._spaces # Сохранить _spaces. ❷
        for space in ALL_SPACES: # Проверить каждую клетку:
            # Смоделировать ход X в эту клетку:
            self._spaces = copy.copy(originalSpaces)
            if self._spaces[space] == BLANK:
                self._spaces[space] = X
            if self.isWinner(X):
                xCanWin = True
            # Смоделировать ход O в эту клетку:
            self._spaces = copy.copy(originalSpaces) ❸
            if self._spaces[space] == BLANK:
                self._spaces[space] = O
            if self.isWinner(O):
                oCanWin = True
        if xCanWin:
            boardStr += '\nX can win in one more move.'
        if oCanWin:
```

```

        boardStr += '\n0 can win in one more move.'
    self._spaces = originalSpaces
    return boardStr

```

Сначала `super().getBoardStr()` ❶ выполняет код метода `getBoardStr()` родительского класса `TTTBoard`, который возвращает строку с игровым полем. Строка временно сохраняется в переменной с именем `boardStr`. Так как представление игрового поля было сгенерировано повторным использованием метода `getBoardStr()` класса `TTTBoard`, оставшийся код этого метода занимается генерированием подсказки. Затем метод `getBoardStr()` присваивает переменным `xCanWin` и `oCanWin` значение `False` и сохраняет словарь `self._spaces` в переменной `originalSpaces` ❷. Далее цикл `for` перебирает все клетки поля от 1 до 9. Внутри цикла атрибуту `self._spaces` присваивается копия словаря `originalSpaces`, и если текущая клетка перебора пуста, в нее помещается знак X. Таким образом моделируется ход X в эту пустую клетку следующим ходом. Вызов `self.isWinner()` определит, принесет ли этот ход выигрыш, и если принесет — `xCanWin` присваивается `True`. Затем те же шаги повторяются для O, чтобы определить, сможет ли O выиграть ходом в эту клетку ❸. Этот метод использует модуль `copy` для создания копии словаря в `self._spaces`, поэтому в начало `tictactoe.py` необходимо добавить следующую строку:

```
import copy
```

Затем измените строку `main()`, чтобы она создавала экземпляр `HintBoard` вместо объекта `TTTBoard`:

```
gameBoard = HintBoard() # Создать объект игрового поля.
```

Кроме одной измененной строки в `main()`, оставшаяся часть программы работает точно так же, как прежде. Если запустить программу, результат будет выглядеть так:

```

Welcome to Tic-Tac-Toe!
--snip--
  X| | 1 2 3
  +-+-
  | |0 4 5 6
  +-+-
  | |X 7 8 9
X can win in one more move.
What is O's move? (1-9)
5
  X| | 1 2 3
  +-+-
  |0|0 4 5 6
  +-+-
  | |X 7 8 9
O can win in one more move.
--snip--
The game is a tie!
Thanks for playing!

```

В конце метода, если `xCanWin` или `oCanWin` содержит `True`, в строку `boardStr` включается дополнительное сообщение. Наконец, функция возвращает `boardStr`.

Не в каждом переопределенном методе необходимо использовать `super()`! Если переопределенный метод класса делает что-то совершенно отличное от переопределенного метода родительского класса, вызывать переопределенный метод с использованием `super()` необязательно. Функция `super()` особенно полезна, когда класс содержит несколько родительских методов, как объясняется в разделе «Множественное наследование» этой главы.

Предпочитайте композицию наследованию

Наследование — эффективный механизм повторного использования кода. Возможно, вам захочется немедленно начать применять его во всех ваших классах. Тем не менее базовые классы не всегда настолько тесно связаны с subclasses. С созданием нескольких уровней наследования в код добавляется не столько порядок, сколько рутина. И хотя наследование может использоваться для классов, связанных отношениями «является <частным случаем>» (иначе говоря, когда дочерний класс является разновидностью родительского класса), часто для классов с отношениями «является» предпочтительнее использовать механизм, называемый *композицией*. Композиция — прием проектирования классов, основанный на включении объектов в класс (вместо наследования классов этих объектов). Именно это происходит при добавлении атрибутов в классы. При проектировании классов с возможностью применения наследования следует предпочесть композицию наследованию. Собственно, именно это происходило во всех примерах этой и предыдущей главы.

- Объект `WizCoin` «содержит» количества монет разного номинала.
- Объект `TTTBoard` «содержит» набор из девяти клеток.
- Объект `MiniBoard` «содержит» объект `TTTBoard`, так что он тоже «содержит» набор из девяти клеток.
- Объект `HintBoard` «содержит» объект `TTTBoard`, так что он тоже «содержит» набор из девяти клеток.

Вернемся к классу `WizCoin` из предыдущей главы. Если мы создали класс `WizardCustomer` для представления клиентов волшебной лавки, эти клиенты будут носить с собой некую сумму денег, которая представляется классом `WizCoin`. Однако эти классы не связаны отношениями «является» — объект `WizardCustomer` не может рассматриваться как разновидность объекта `WizCoin`. При использовании наследования получится довольно неуклюжий код:

```
import wizcoin

class WizardCustomer(wizcoin.WizCoin): ❶
    def __init__(self, name):
        self.name = name
        super().__init__(0, 0, 0)

wizard = WizardCustomer('Alice')
print(f'{wizard.name} has {wizard.value()} knuts worth of money.')
print(f'{wizard.name}\''s coins weigh {wizard.weightInGrams()} grams.')
```

В этом примере `WizardCustomer` наследует методы объекта `WizCoin` ❶ — такие как `value()` и `weightInGrams()`. Формально `WizardCustomer`, наследующий объекту `WizCoin`, может делать то же самое, что и объект `WizardCustomer`, содержащий объект `WizCoin` в атрибуте. Однако имена `wizard.value()` и `wizard.weightInGrams()` выглядят странно; создается впечатление, что они возвращают сумму и вес волшебника, а не сумму и вес его монет. Кроме того, если позднее потребуется добавить метод `weightInGrams()` для веса волшебника, имя уже будет занято. Гораздо лучше включить объект `WizCoin` в атрибут, потому что волшебник-клиент «содержит» некоторое количество монет:

```
import wizcoin

class WizardCustomer:
    def __init__(self, name):
        self.name = name
        self.purse = wizcoin.WizCoin(0, 0, 0) ❶

wizard = WizardCustomer('Alice')
print(f'{wizard.name} has {wizard.purse.value()} knuts worth of money.')
print(f'{wizard.name}\''s coins weigh {wizard.purse.weightInGrams()} grams.')
```

Вместо того чтобы наследовать методы от `WizCoin` в классе `WizardCustomer`, мы включаем в класс `WizardCustomer` атрибут `purse` ❶, который содержит объект `WizCoin`. При использовании композиции любые изменения в методах класса `WizCoin` не приведут к изменению методов класса `WizardCustomer`. Этот механизм обеспечивает большую гибкость для будущих архитектурных изменений в обоих классах и упрощает сопровождение кода в будущем.

Обратная сторона наследования

Главный недостаток наследования связан с тем, что любые будущие изменения в родительском классе обязательно наследуются всеми его дочерними классами. В большинстве случаев такое жесткое связывание — именно то, что требуется.

Но в некоторых ситуациях требования к коду плохо вписываются в модель наследования.

Представьте, что в программе моделирования дорожного движения используются классы `Car`, `Motorcycle` и `LunarRover`. Они содержат похожие методы — такие как `startIgnition()` и `changeTire()`. Вместо того чтобы копировать этот код в каждый класс, можно создать родительский класс `Vehicle`, которому будут наследовать классы `Car`, `Motorcycle` и `LunarRover`. Если теперь вам потребуется исправить ошибку, скажем, в методе `changeTire()`, изменения придется вносить только в одном месте. Это особенно полезно, если классу `Vehicle` наследуют десятки разных классов, связанных с транспортными средствами. Код этих классов будет выглядеть примерно так:

```
class Vehicle:
    def __init__(self):
        print('Vehicle created.')
    def startIgnition(self):
        pass # Здесь размещается код зажигания.
    def changeTire(self):
        pass # Здесь размещается код замены шин.

class Car(Vehicle):
    def __init__(self):
        print('Car created.')

class Motorcycle(Vehicle):
    def __init__(self):
        print('Motorcycle created.')

class LunarRover(Vehicle):
    def __init__(self):
        print('LunarRover created.')
```

Но все будущие изменения в `Vehicle` также будут распространяться и на эти классы. Что произойдет, если понадобится добавить метод `changeSparkPlug()`? У машин и мотоциклов есть свечи зажигания, но у луноходов (`LunarRover`) их нет. Предпочитая композицию наследованию, можно создать отдельные классы `CombustionEngine` (двигатель внутреннего сгорания) и `ElectricEngine` (электрический двигатель). Затем мы проектируем класс `Vehicle`, так чтобы он содержал атрибут `engine` — либо `CombustionEngine`, либо `ElectricEngine` — с соответствующими методами:

```
class CombustionEngine:
    def __init__(self):
        print('Combustion engine created.')
    def changeSparkPlug(self):
        pass # Здесь размещается код замены свечи зажигания.

class ElectricEngine:
```



```
def __init__(self):
    print('Electric engine created.')

class Vehicle:
    def __init__(self):
        print('Vehicle created.')
        self.engine = CombustionEngine() # Используется по умолчанию.
--snip--

class LunarRover(Vehicle):
    def __init__(self):
        print('LunarRover created.')
        self.engine = ElectricEngine()
```

Возможно, вам придется переписать большие объемы кода, особенно если программа содержит несколько классов, наследующих существующему классу `Vehicle`: все вызовы `vehicleObj.changeSparkPlug()` должны быть преобразованы в `vehicleObj.engine.changeSparkPlug()` для каждого объекта класса `Vehicle` или его subclasses. Так как столь значительные изменения могут привести к появлению ошибок, возможно, вы предпочтете, чтобы метод `changeSparkPlug()` для `LunarVehicle` не делал ничего. В этом случае в питоническом стиле следует присвоить `changeSparkPlug` значение `None` внутри класса `LunarVehicle`:

```
class LunarRover(Vehicle):
    changeSparkPlug = None
    def __init__(self):
        print('LunarRover created.')
```

В строке `changeSparkPlug = None` используется синтаксис, описанный в разделе «Атрибуты классов» этой главы. В следующем фрагменте переопределяется метод `changeSparkPlug()`, унаследованный от `Vehicle`, и при вызове его с объектом `LunarRover` происходит ошибка:

```
>>> myVehicle = LunarRover()
LunarRover created.
>>> myVehicle.changeSparkPlug()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Эта ошибка позволит быстро и непосредственно узнать о проблеме, если вы попытаетесь вызвать этот неподходящий метод с объектом `LunarRover`. Любые дочерние классы `LunarRover` также унаследуют значение `None` для `changeSparkPlug()`. Сообщение об ошибке `TypeError: 'NoneType' object is not callable` (`TypeError: объект 'NoneType' не может вызываться`) информирует, что программист класса `LunarRover` намеренно задал для метода `changeSparkPlug()` значение `None`. Если бы такой метод не существовал изначально, то вы бы получили сообщение об ошибке

`NameError: name 'changeSparkPlug' is not defined` error message (`NameError`: имя `'changeSparkPlug'` не определено).

Применение наследования для создания классов нередко оборачивается сложностью и противоречиями. Часто вместо наследования лучше воспользоваться композицией.

Функции `isinstance()` и `issubclass()`

Если вы хотите узнать тип объекта, можно передать объект встроенной функции `type()`, о которой я упоминал в предыдущей главе. Но если вы выполняете проверку типа для объекта, лучше воспользоваться более гибкой встроенной функцией `isinstance()`. Функция `isinstance()` вернет `True`, если объект относится к заданному классу или одному из его subclasses. Введите следующий фрагмент в интерактивной оболочке:

```
>>> class ParentClass:
...     pass
...
>>> class ChildClass(ParentClass):
...     pass
...
>>> parent = ParentClass() # Создать объект ParentClass.
>>> child = ChildClass()   # Создать объект ChildClass.
>>> isinstance(parent, ParentClass)
True
>>> isinstance(parent, ChildClass)
False
>>> isinstance(child, ChildClass)    ❶
True
>>> isinstance(child, ParentClass)   ❷
True
```

Функция `isinstance()` указывает, что объект `ChildClass` из `child` является экземпляром `ChildClass` ❶ и экземпляром `ParentClass` ❷. Выглядит логично, так как объект `ChildClass` может рассматриваться как разновидность объекта `ParentClass`.

Во втором аргументе также можно передать кортеж объектов классов, чтобы проверить, относится ли первый аргумент к одному из классов в кортеже:

```
>>> isinstance(42, (int, str, bool)) # True, если 42 имеет тип int, str или bool.
True
```

Встроенная функция `issubclass()` — она используется реже — может сообщить, является ли объект класса, переданный в первом аргументе, subclassом (или относится к тому же классу), что и объект класса, переданный во втором аргументе:

```
>>> issubclass(ChildClass, ParentClass) # ChildClass субклассирует ParentClass.
True
>>> issubclass(ChildClass, str) # ChildClass не субклассирует str.
False
>>> issubclass(ChildClass, ChildClass) # ChildClass имеет тип ChildClass.
True
```

Как и в случае с `isinstance()`, во втором аргументе `issubclass()` можно передать кортеж аргументов, чтобы проверить, является ли первый аргумент субклассом одного из классов в кортеже. Принципиальное отличие `isinstance()` и `issubclass()` заключается в том, что `issubclass()` передаются два объекта классов, а `isinstance()` передается объект и объект класса.

Методы классов

Метод класса связывается с классом, а не с его отдельными объектами (в отличие от обычных методов). Метод класса можно узнать в коде по двум признакам: по декоратору `@classmethod` перед командой `def` метода и по использованию `cls` в первом параметре, как показано в следующем примере.

```
class ExampleClass:
    def exampleRegularMethod(self):
        print('This is a regular method.')

    @classmethod
    def exampleClassMethod(cls):
        print('This is a class method.')

# Вызов метода класса без создания экземпляра:
ExampleClass.exampleClassMethod()
```

```
obj = ExampleClass()
# С предыдущей строкой эти две строки эквивалентны:
obj.exampleClassMethod()
obj.__class__.exampleClassMethod()
```

Параметр `cls` работает как `self`, не считая того, что `self` содержит ссылку на объект, а параметр `cls` — ссылку на класс объекта. Это означает, что код метода класса не может обратиться к атрибутам отдельных объектов или вызывать обычные методы объекта. Метод класса может вызывать только другие методы класса или обращаться к атрибутам класса. Мы используем имя `cls`, потому что `class` является ключевым словом Python и, как и любые другие ключевые слова (такие как `if`, `while` или `import`), оно не может использоваться для имен параметров. Методы класса часто вызываются через объект класса (например, `ExampleClass.exampleClassMethod()`). Но их также можно вызывать с любым экземпляром класса — например, `obj.exampleClassMethod()`.

Методы классов используются не так часто. Самый распространенный сценарий использования — определение альтернативных конструкторов, кроме `__init__()`. Например, что если функция-конструктор может получать либо строку с данными, необходимыми новому объекту, либо строку с именем файла, который содержит данные для нового объекта? Список параметров метода `__init__()` не должен быть длинным и запутанным. Вместо этого можно воспользоваться методом класса, возвращающим новый объект.

Для примера создадим класс `AsciiArt`. Как было показано в главе 14, ASCII-графика формирует изображение из символов текста.

```
class AsciiArt:
    def __init__(self, characters):
        self._characters = characters

    @classmethod
    def fromFile(cls, filename):
        with open(filename) as fileObj:
            characters = fileObj.read()
        return cls(characters)

    def display(self):
        print(self._characters)

# Другие методы AsciiArt...

face1 = AsciiArt(' _____\n' +
                 '|  . .  |\n' +
                 '|  \  _/  |\n' +
                 '|  _____|')

face1.display()

face2 = AsciiArt.fromFile('face.txt')
face2.display()
```

Класс `AsciiArt` содержит метод `__init__()`, которому может передаваться набор символов в виде строки. Он также содержит метод класса `fromFile()`, которому может передаваться строка с именем текстового файла, содержащего ASCII-графику. Оба метода создают объекты `AsciiArt`.

Если запустить эту программу и если существует файл `face.txt` с изображением в ASCII-графике, результат будет выглядеть примерно так:

```
|  _____|
|  . .  |\n
|  \  _/  |\n
|  _____|
```

```
|  _____|
|  . .  |\n
|  \  _/  |\n
|  _____|
```

Метод класса `fromFile()` несколько упрощает код по сравнению с выполнением всех операций в `__init__()`.

У методов классов есть и другое преимущество: субкласс `AsciiArt` может наследовать свой метод `fromFile()` (и переопределить его при необходимости). Это объясняет, почему в метод `fromFile()` класса `AsciiArt` включен вызов `cls(characters)`, а не `AsciiArt(characters)`. Вызов `cls()` также будет работать в субклассах `AsciiArt` в неизменном виде, потому что класс `AsciiArt` не фиксируется в методе. Но вызов `AsciiArt()` всегда будет вызывать метод `__init__()` класса `AsciiArt` вместо метода `__init__()` субкласса. Рассматривайте `cls` как объект, представляющий класс.

По аналогии с тем, как обычные методы всегда должны использовать параметр `self` где-то в коде, метод класса всегда использует свой параметр `cls`. Если в коде вашего метода класса параметр `cls` никогда не применяется, это признак того, что метод класса, возможно, стоило бы оформить в виде обычной функции.

Атрибуты классов

Атрибут класса представляет собой переменную, которая принадлежит классу, а не объекту. Атрибуты класса создаются внутри класса, но вне любых методов, подобно тому как глобальные переменные создаются в файле `.py`, но за пределами любых функций.

Ниже приведен пример атрибута класса с именем `count`, который отслеживает количество созданных объектов `CreateCounter`:

```
class CreateCounter:
    count = 0 # This is a class attribute.

    def __init__(self):
        CreateCounter.count += 1

print('Objects created:', CreateCounter.count) # Выводит 0.
a = CreateCounter()
b = CreateCounter()
c = CreateCounter()
print('Objects created:', CreateCounter.count) # Выводит 3.
```

Класс `CreateCounter` содержит один атрибут класса с именем `count`. Все объекты `CreateCounter` совместно используют этот атрибут (вместо того, чтобы иметь собственные атрибуты `count`). Это объясняет, почему строка `CreateCounter.count += 1` в функции-конструкторе подсчитывает все созданные объекты `CreateCounter`.

При запуске этой программы результат выглядит так:

```
Objects created: 0
Objects created: 3
```

Атрибуты классов используются редко. Даже пример с подсчетом количества созданных объектов `CreateCounter` проще реализуется на базе глобальной переменной вместо атрибута класса.

Статические методы

Статический метод не имеет параметра `self` или `cls`. По сути статические методы представляют собой обычные функции, потому что они не могут обращаться к атрибутам и методам класса или его объектов. Необходимость в использовании статических методов в Python возникает очень редко. Если вы решите воспользоваться ими, вам наверняка стоит подумать об использовании обычной функции.

Чтобы определить статический метод, поставьте декоратор `@staticmethod` перед соответствующей командой `def`. Пример статического метода:

```
class ExampleClassWithStaticMethod:
    @staticmethod
    def sayHello():
        print('Hello!')
```

Объект не создается, перед sayHello() указывается имя класса:
`ExampleClassWithStaticMethod.sayHello()`

Между статическим методом `sayHello()` в классе `ExampleClassWithStaticMethod` и функцией `sayHello()` почти нет различий. Вполне возможно, что вы предпочтете функцию, потому что ее можно вызывать без указания имени класса.

Статические методы чаще встречаются в других языках, где отсутствуют гибкие возможности языка Python. Статические методы в Python моделируют функциональность других языков, но не обладают особой практической ценностью.

Когда использовать объектно-ориентированные статические средства и средства уровня классов

Вам редко могут понадобиться методы классов, атрибуты классов и статические методы. Если вы думаете: «А нельзя ли вместо этого использовать функцию или глобальную переменную?», то скорее всего вам не стоит использовать метод класса, атрибут класса или статический метод. Они рассматриваются в этой книге только по одной причине: чтобы вы узнали их, если они встретятся вам в коде, но пользоваться ими я не рекомендую. Они могут пригодиться, если вы создаете собственный фреймворк с нетривиальным семейством классов, которые должны субклассифицироваться программистами, использующими фреймворк. Скорее всего, при написании обычных приложений Python они вам не понадобятся.

Дополнительную информацию об этих возможностях, о том, почему они вам нужны (или не нужны), вы можете почерпнуть в публикациях Филипа Дж. Эби (Phillip J. Eby) «Python Is Not Java» (<https://dirtsimple.org/2004/12/python-is-not-java.html>) и Райана Томайко (Ryan Tomayko) «The Static Method Thing» (<https://tomayko.com/blog/2004/the-static-method-thing>).

Термины объектно-ориентированного программирования

В объяснениях ООП часто встречаются специальные термины: наследование, инкапсуляция, полиморфизм и т. д. Нельзя сказать, что знать их абсолютно необходимо, но следует хотя бы понимать их смысл на базовом уровне. Наследование уже упоминалось ранее, а о других терминах я расскажу сейчас.

Инкапсуляция

Слово «инкапсуляция» имеет два распространенных и взаимосвязанных определения. Первое: инкапсуляцией называется объединение взаимосвязанных данных и кода в одно целое. В сущности, именно это и делают классы: они объединяют взаимосвязанные атрибуты и методы. Например, наш класс `WizCoin` инкапсулирует три целых числа (`knuts`, `sickles` и `galleons`) в одном объекте `WizCoin`.

Второе определение: инкапсуляцией называется механизм сокрытия информации, позволяющий объектам скрывать сложные подробности реализации, то есть внутреннее устройство объекта. Пример такого рода встречался в подразделе «Приватные атрибуты и приватные методы», с. 324, где объекты `BankAccount` предоставляют методы `deposit()` и `withdraw()` для сокрытия подробностей работы с атрибутами `_balance`. Функции позволяют осуществить похожую цель создания «черного ящика» — например, алгоритм вычисления квадратного корня функцией `math.sqrt()` не виден пользователю. Все, что вам нужно знать, — эта функция возвращает квадратный корень того числа, которое ей было передано.

Полиморфизм

Полиморфизм позволяет рассматривать объекты одного типа как объекты другого типа. Например, функция `len()` возвращает длину переданного ей аргумента. Функции `len()` можно передать строку, чтобы узнать, сколько символов она содержит, но `len()` также можно передать список или словарь, чтобы узнать, сколько элементов или пар «ключ — значение» они содержат. Эта разновидность полиморфизма называется *параметрическим полиморфизмом*, или *обобщением*, потому что она может работать с объектами многих разных типов.

Термином «полиморфизм» также иногда обозначают *ситуативный* (ad hoc) *полиморфизм*, или *перегрузку операторов*, когда операторы (такие как + или *) демонстрируют разное поведение в зависимости от типа объектов, с которыми они работают. Например, оператор + выполняет математическое сложение для двух целых чисел или чисел с плавающей точкой, но с двумя строками он выполняет конкатенацию. Перегрузке операторов посвящена глава 17.

Когда наследование не используется

Наследование легко приводит к излишнему переусложнению классов. Как выразился Лучано Рамальо (Luciano Ramalho), «Упорядочение объектов в аккуратную иерархию апеллирует к нашему чувству порядка; программисты делают это просто для развлечения». Мы создаем классы, субклассы и субсубклассы, когда того же эффекта можно было бы достичь с одним классом или парой функций. Но вспомните принцип из «Дзен Python» (глава 6): «Простое лучше, чем сложное».

Использование ООП позволяет разделить код на меньшие единицы (в данном случае классы), которые проще понять, чем один большой файл .py с сотнями функций, не следующих в каком-то определенном порядке. Наследование полезно, когда вы определяете несколько функций, работающих с одной структурой данных словаря или списка. Объединение таких функций в класс принесет пользу.

Однако возможны и ситуации, когда создавать класс или использовать наследование не обязательно.

- Если ваш класс состоит из методов, в которых совсем не используются параметры `self` или `cls`, удалите класс и примените функции вместо методов.
- Если вы создали родителя, который имеет только один дочерний класс, но объекты родительского класса нигде не используются, их можно объединить в один класс.
- Если вы создаете более трех или четырех уровней субклассирования, вероятно, вы злоупотребляете наследованием. Переработайте субклассы и уберите лишние.

Как и в примере с двумя версиями программы «Крестики-нолики» из предыдущей главы, вы можете отказаться от использования классов и при этом иметь вполне работоспособную, избавленную от ошибок программу. Не думайте, что программу непременно нужно проектировать в виде сложной паутины классов. Простое работоспособное решение лучше сложного, которое не работает. Джоэл Спольски (Joel Spolsky) пишет об этом в своем сообщении в блоге «Don't Let the Astronaut Architects Scare You» (<https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/>).

Вы должны знать, как работают такие объектно-ориентированные концепции, как наследование, потому что они помогают организовать код, а также упрощают разработку и отладку. Благодаря своей гибкости Python не только предоставляет объектно-ориентированные средства, но и не требует обязательного их использования, если они плохо подходят для целей вашей программы.

Множественное наследование

Многие языки программирования позволяют классу иметь не более одного (непосредственного) родительского класса. Python позволяет использовать несколько родительских классов при помощи механизма, называемого *множественным наследованием*. Например, можно определить класс `Airplane` с методом `flyInTheAir()` и класс `Ship` с методом `floatOnWater()`. После этого можно создать класс `FlyingBoat`, наследующий как классу `Airplane`, так и `Ship`; для этого оба класса следует перечислить в команде `class` с разделением запятыми. Откройте в редакторе окно с новым файлом и сохраните следующий код с именем `flyingboat.py`:

```
class Airplane:
    def flyInTheAir(self):
        print('Flying...')

class Ship:
    def floatOnWater(self):
        print('Floating...')

class FlyingBoat(Airplane, Ship):
    pass
```

Объекты `FlyingBoat`, которые мы создаем, наследуют методы `flyInTheAir()` и `floatOnWater()`, как показывает следующий сеанс интерактивной оболочки:

```
>>> from flyingboat import *
>>> seaDuck = FlyingBoat()
>>> seaDuck.flyInTheAir()
Flying...
>>> seaDuck.floatOnWater()
Floating...
```

Множественное наследование работает вполне прямолинейно — при условии, что имена методов родительских классов различны. Такие классы называются *примесями* (mixins). (Это просто термин для разновидности классов; в Python нет ключевого слова `mixin`.) Но что произойдет, если ваш класс наследует нескольким сложным классам, содержащим методы с совпадающими именами?

Вспомните классы игрового поля для игры «Крестики-нолики» `MiniBoard` и `HintTTTBoard`, рассмотренные ранее в этой главе. А если вам нужен класс, который

выводит компактное игровое поле, а также подсказки? С множественным наследованием можно повторно использовать существующие классы. Добавьте следующий фрагмент в конец файла `tictactoe_oop.py` до команды `if`, в которой вызывается функция `main()`:

```
class HybridBoard(HintBoard, MiniBoard):  
    pass
```

Этот класс не содержит ничего — он только повторно использует код, унаследованный от `HintBoard` и `MiniBoard`. Затем измените код в функции `main()`, чтобы в нем создавался объект `HybridBoard`:

```
gameBoard = HybridBoard() # Создать объект игрового поля.
```

Оба родительских класса, `MiniBoard` и `HintBoard`, содержат метод с именем `getBoardStr()`. Какой же из методов унаследует `HybridBoard`? При выполнении эта программа выводит компактное игровое поле, а также выводит подсказку:

```
--snip--  
      X.. 123  
      .O. 456  
      X.. 789  
X can win in one more move.
```

Похоже, Python, как по волшебству, объединил метод `getBoardStr()` класса `MiniBoard` с методом `getBoardStr()` класса `HintBoard`, чтобы он делал и то и другое! Но это произошло потому, что я написал их так, чтобы они работали друг с другом. Более того, если поменять порядок классов в команде `class` класса `HybridBoard`:

```
class HybridBoard(MiniBoard, HintBoard):
```

подсказки полностью пропадут:

```
--snip--  
      X.. 123  
      .O. 456  
      X.. 789
```

Чтобы понять, почему это происходит, необходимо изучить *порядок разрешения методов* (MRO, Method Resolution Order) языка Python и то, как работает функция `super()`.

Порядок разрешения методов

Наша программа «Крестики-нолики» теперь содержит четыре класса для представления игрового поля: три с определенными методами `getBoardStr()` и четвертый с унаследованным методом `getBoardStr()`, как показано на рис. 16.2.

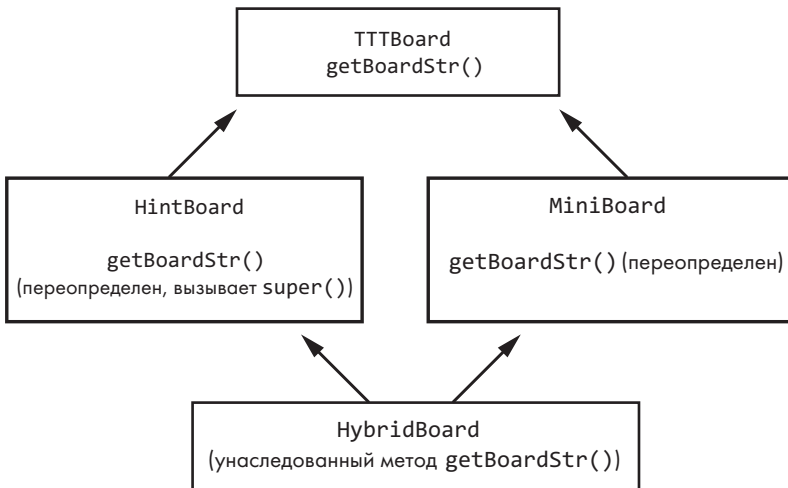


Рис. 16.2. Четыре класса в программе «Крестики-нолики»

Когда вы вызываете `getBoardStr()` для объекта `HybridBoard`, Python знает, что в классе `HybridBoard` нет метода с таким именем, и поэтому проверяет родительский класс. Но класс имеет два родительских класса, и в обоих присутствует метод `getBoardStr()`. Какая версия будет вызвана?

С помощью MRO класса `HybridBoard` вы можете получить упорядоченный список классов, который Python проверяет при наследовании методов либо при вызове функции `super()` из метода. Для просмотра MRO класса `HybridBoard` вызовите его метод `mro()` в интерактивной оболочке:

```
>>> from tictactoe_oop import *
>>> HybridBoard.mro()
[<class 'tictactoe_oop.HybridBoard'>, <class 'tictactoe_oop.HintBoard'>,
 <class 'tictactoe_oop.MiniBoard'>, <class 'tictactoe_oop.TTTBoard'>, <class
 'object'>]
```

Из возвращаемого значения видно, что при вызове метода для `HybridBoard` Python сначала проверяет его в классе `HybridBoard`. Если метод не найден, Python проверяет класс `HintBoard`, затем класс `MiniBoard` и, наконец, класс `TTTBoard`. В конце каждого списка MRO находится встроенный класс `object`, родительский для всех классов Python.

При одиночном наследовании MRO определяется просто: достаточно взять цепочку родительских классов. При множественном наследовании задача усложняется. В Python MRO следует алгоритму C3, подробности которого выходят за рамки книги. Но чтобы определить MRO, достаточно запомнить два правила.

- Python проверяет дочерние классы до родительских классов.
- Python проверяет классы, задействованные в наследовании, слева направо в порядке их перечисления в команде `class`.

Если вызвать `getBoardStr()` для объекта `HybridBoard`, Python сначала проверит класс `HybridBoard`. Затем, поскольку в родителях класса слева направо указаны `HintBoard` и `MiniBoard`, Python проверяет `HintBoard`. Этот родительский класс содержит метод `getBoardStr()`, поэтому `HybridBoard` наследует и вызывает его.

Но этим дело не заканчивается: далее метод вызывает `super().getBoardStr()`. Имя функции Python `super()` выбрано неудачно, потому что возвращает она не родительский класс, а следующий класс в MRO. Это означает, что при вызове `getBoardStr()` для объекта `HybridBoard` следующим классом в MRO после `HintBoard` будет `MiniBoard`, а не родительский класс `TTTBoard`. Таким образом, вызов `super().getBoardStr()` вызывает метод `getBoardStr()` класса `MiniBoard`, который возвращает компактное игровое поле. Оставшийся код `getBoardStr()` класса `HintBoard` после этого вызова `super()` присоединяет текст подсказки к строке.

Если изменить команду `class` класса `HybridBoard` так, чтобы сначала в ней был указан класс `MiniBoard`, а потом `HintBoard`, в списке MRO класс `MiniBoard` будет предшествовать `HintBoard`. Это означает, что `HybridBoard` унаследует версию `getBoardStr()` от `MiniBoard`, в которой нет вызова `super()`. Именно этот порядок становится причиной ошибки с выводом компактного игрового поля без подсказок: без вызова `super()` метод `getBoardStr()` класса `MiniBoard` не вызовет метод `getBoardStr()` класса `HintBoard`.

Множественное наследование позволяет реализовать значительную функциональность в небольшом объеме кода, но легко приводит к возникновению переусложненного, трудного для понимания кода. Отдавайте предпочтение одиночному наследованию, примесям или решениям без наследования. Этих решений часто более чем достаточно для достижения целей вашей программы.

Итоги

Наследование — механизм повторного использования кода. Оно позволяет создавать дочерние классы, наследующие методы родительских классов. Вы можете переопределить методы, чтобы предоставить для них новый код, а также воспользоваться функцией `super()` для вызова исходных методов родительского класса. Дочерний класс связан с родительским классом отношениями типа «является <частным случаем>», потому что дочерний класс может рассматриваться как разновидность объекта родительского класса.

В Python использовать классы и наследование не обязательно. Некоторые программисты считают, что преимущества наследования не окупают сложность от его неумеренного использования. Часто решения с композицией (вместо наследования) оказываются более гибкими, потому что они реализуют отношение «содержит» с объектом одного класса и объектами других классов вместо прямого наследования методов этих других классов. Это означает, что объект одного класса может содержать объект другого класса. Например, объект `Customer` может содержать атрибут `birthdate`, которому присваивается объект `Date`, вместо субклассирования `Date` классом `Customer`.

Подобно тому как функция `type()` возвращает тип переданного ей объекта, функции `isinstance()` и `issubclass()` возвращают тип и информацию о наследовании для переданного им объекта.

Классы содержат методы и атрибуты объектов, но в них также могут присутствовать методы класса, атрибуты класса и статические методы. И хотя эти возможности используются редко, они позволяют реализовать некоторые объектно-ориентированные приемы, которые не обеспечиваются глобальными переменными или функциями.

Python позволяет классу наследовать нескольким родительским классам, хотя иногда это приводит к появлению трудного для понимания кода. Функция `super()` и методы класса определяют способ наследования методов на основании порядка разрешения методов (MRO). Чтобы просмотреть список MRO в интерактивной оболочке, вызовите метод `mro()` для класса.

В этой и предыдущей главе я рассказал об общих концепциях ООП. В следующей главе мы займемся приемами ООП, специфическими для Python.

17

ООП в Python: свойства и dunder-методы



Средства ООП включены во многие языки, но Python предоставляет ряд уникальных средств ООП, включая свойства и специальные методы. Умение пользоваться этими питоническими средствами помогает написать лаконичный и удобочитаемый код.

Свойства позволяют выполнять заданный код при каждом чтении, изменении или удалении атрибута объекта; тем самым гарантируется, что объект не окажется в некорректном состоянии. В других языках такой код часто называется *getter*- или *setter*-методами. *Dunder-методы* (называемые также *магическими* методами) предоставляют возможность использовать ваши объекты со встроенными операторами Python — такими как оператор `+`. Например, так можно объединить два объекта `datetime.datetime` (`datetime.datetime(days=2)` и `datetime.datetime(days=3)`) для создания нового объекта `datetime.datetime(days=5)`.

Кроме других примеров, мы продолжим расширять класс `WizCoin`, работать с которым начали в главе 15, и добавим в него свойства и перегрузку операторов с использованием *dunder*-методов. Эти возможности сделают объекты `WizCoin` более выразительными и простыми для применения в приложениях, импортирующих модуль `wizcoin`.

Свойства

В классе `BankAccount`, использованном в главе 15, атрибут `_balance` мы поместили как приватный, для чего в начало его имени вставили символ подчеркивания `_`.

Однако следует помнить, что пометка атрибута как приватного всего лишь является условным соглашением: с технической точки зрения все атрибуты Python являются открытыми, то есть они доступны для кода за пределами класса. Ничто не мешает коду намеренно (в том числе и злонамеренно) изменить атрибут `_balance` и присвоить ему некорректное значение.

Однако вы можете предотвратить случайные изменения приватных атрибутов при помощи свойств. В Python свойства представляют собой атрибуты, которым специально назначаются методы *getter*, *setter* и *deleter*, управляющие их чтением, изменением и удалением. Например, если атрибут должен принимать только целые значения, попытка присвоить ему строку `'42'` с большой вероятностью создаст ошибку. Свойство вызывает *setter*-метод для выполнения кода, который исправит (или по крайней мере исправит на ранней стадии) попытку некорректного присваивания. Если у вас возникает мысль: «А хорошо бы, чтобы при каждом обращении к атрибуту при его изменении командой присваивания или удалении командой `del` выполнялся какой-нибудь код», — используйте свойства.

Преобразование атрибута в свойство

Начнем с создания простого класса, который содержит обычный атрибут вместо свойства. Откройте в редакторе окно с новым файлом, введите следующий код и сохраните его с именем `regularAttributeExample.py`:

```
class ClassWithRegularAttributes:
    def __init__(self, someParameter):
        self.someAttribute = someParameter

obj = ClassWithRegularAttributes('some initial value')
print(obj.someAttribute) # Выводит 'some initial value'
obj.someAttribute = 'changed value'
print(obj.someAttribute) # Выводит 'changed value'
del obj.someAttribute # Удаляет атрибут someAttribute.
```

Класс `ClassWithRegularAttributes` содержит обычный атрибут с именем `someAttribute`. Метод `__init__()` присваивает `someAttribute` строку `'some initial value'`, но затем значение атрибута напрямую заменяется строкой `'changed value'`. При выполнении этой программы результат выглядит так:

```
some initial value
changed value
```

Вывод показывает, что `someAttribute` можно легко присвоить любое значение. Недосток обычных атрибутов заключается в том, что значение, присваиваемое `someAttribute`, может оказаться некорректным. Гибкость — отличное качество, но некорректное значение `someAttribute` может стать причиной ошибок в программе.

Перепишем этот класс с использованием свойств. Выполните следующие действия, чтобы создать свойство для атрибута `someAttribute`.

1. Переименуйте атрибут, чтобы имя начиналось с префикса `_: _someAttribute`.
2. Создайте метод с именем `someAttribute` и добавьте декоратор `@property`. Этот getter-метод получает параметр `self`, который получают все методы.
3. Создайте другой метод с именем `someAttribute` и декоратором `@someAttribute.setter`. Этот setter-метод получает параметры `self` и `value`.
4. Создайте еще один метод с именем `someAttribute` и добавьте декоратор `@someAttribute.deleter`. Этот deleter-метод получает параметр `self`, который получают все методы.

Откройте в редакторе окно с новым файлом, введите следующий код и сохраните его с именем `propertiesExample.py`:

```
class ClassWithProperties:
    def __init__(self):
        self.someAttribute = 'some initial value'

    @property
    def someAttribute(self): # Get-метод
        return self._someAttribute

    @someAttribute.setter
    def someAttribute(self, value): # Set-метод
        self._someAttribute = value

    @someAttribute.deleter
    def someAttribute(self): # Del-метод
        del self._someAttribute

obj = ClassWithProperties()
print(obj.someAttribute) # Выводит 'some initial value'
obj.someAttribute = 'changed value'
print(obj.someAttribute) # Выводит 'changed value'
del obj.someAttribute # Удаляет атрибут _someAttribute.
```

Вывод этой программы совпадает с выводом кода `regularAttributeExample.py`, потому что фактически они делают одно и то же: выводят исходный атрибут объекта, затем обновляют атрибут и снова выводят его.

Но заметим, что код за пределами класса никогда не обращается к атрибуту `_someAttribute` напрямую (ведь он является приватным). Вместо этого внешний код обращается к свойству `someAttribute`. Возможно, структура такого свойства кому-то покажется немного абстрактной: оно образуется из getter-, setter- и deleter-метода. Когда мы переименовываем атрибут с именем `someAttribute` в `_someAttribute`

и одновременно создаем для него `getter`-, `setter`- и `deleter`-методы, мы получаем результат, который называется свойством `someAttribute`.

В этом контексте атрибут `_someAttribute` называется *резервным полем* (или *резервной переменной*); это атрибут, на основе которого создается свойство. Резервная переменная используется многими, но не всеми свойствами. Свойство без резервной переменной мы создадим в разделе «Свойства, доступные только для чтения» позднее в этой главе.

Вы никогда не вызываете `getter`-, `setter`- и `deleter`-методы в своем коде, потому что Python делает это за вас в следующих обстоятельствах.

- Когда Python выполняет код, который обращается к свойству (например, `print(obj.someAttribute)`), во внутренней реализации он вызывает `getter`-метод и использует возвращенное значение.
- Когда Python выполняет команду присваивания для свойства (например, `obj.someAttribute = 'changed value'`), во внутренней реализации он вызывает `setter`-метод, передавая строку `'changed value'` для параметра `value`.
- Когда Python выполняет команду `del` для свойства (например, `del obj.someAttribute`), во внутренней реализации он вызывает `deleter`-метод.

Код методов `getter`, `setter` и `deleter` работает с резервной переменной напрямую. Методы `getter`, `setter` и `deleter` не должны работать со свойством, потому что это может привести к ошибкам. Один из возможных примеров: когда метод `getter` обращается к свойству, это заставляет метод вызвать самого себя, что заставляет его снова обратиться к свойству и так далее, пока в программе не произойдет фатальный сбой. Откройте в редакторе окно с новым файлом, введите следующий код и сохраните его с именем `badPropertyExample.py`:

```
class ClassWithBadProperty:
    def __init__(self):
        self.someAttribute = 'some initial value'

    @property
    def someAttribute(self): # Get-метод.
        # Пропущен символ _ в `self._someAttribute`, из-за чего
        # свойство используется снова, а get-метод вызывается повторно:
        return self.someAttribute # Снова вызывается get-метод!

    @someAttribute.setter
    def someAttribute(self, value): # Set-метод.
        self._someAttribute = value

obj = ClassWithBadProperty()
print(obj.someAttribute) # Ошибка, get-метод вызывает get-метод.
```

При попытке выполнить этот код метод `getter` непрерывно вызывает самого себя, пока Python не выдаст исключение `RecursionError`:

```
Traceback (most recent call last):
  File "badPropertyExample.py", line 16, in <module>
    print(obj.someAttribute) # Ошибка, get-метод вызывает getter-метод.
  File "badPropertyExample.py", line 9, in someAttribute
    return self.someAttribute # Снова вызывается getter-метод!
  File "badPropertyExample.py", line 9, in someAttribute
    return self.someAttribute # Снова вызывается getter-метод!
  File "badPropertyExample.py", line 9, in someAttribute
    return self.someAttribute # Снова вызывается getter-метод!
  [предыдущая строка повторяется еще 996 раз]
RecursionError: maximum recursion depth exceeded
```

Для предотвращения рекурсии код методов `getter`, `setter` и `deleter` должен всегда работать с резервной переменной (с префиксом `_` в имени), а не со свойством. Код за пределами этих методов должен использовать переменную, хотя, как и с соглашением о символе `_` для обозначения приватного доступа, ничто не мешает написать код для обращения к резервной переменной.

Использование методов `setter` для проверки данных

Свойства чаще всего используются для проверки самих данных или формата их хранения. Допустим, вы не хотите, чтобы код за пределами класса мог присвоить атрибуту произвольное значение; некорректные значения способны вызвать ошибки. В свойствах можно добавить проверки, которые присваивают атрибутам только проверенные значения. Такие проверки помогают перехватывать ошибки на более ранней стадии разработки, потому что при присваивании некорректного значения выдается исключение.

Обновим файл `wizcoin.py` из главы 15 и преобразуем атрибуты `galleons`, `sickles` и `knuts` в свойства. Мы заменим `setter`-методы этих свойств так, чтобы допустимыми были только положительные целые числа. Наши объекты `WizCoin` представляют набор монет, а количество монет не может быть дробным или отрицательным. Если код за пределами класса попытается задать свойствам `galleons`, `sickles` или `knuts` недопустимое значение, будет выдано исключение `WizCoinException`.

Откройте файл `wizcoin.py`, сохраненный в главе 15, и приведите его к следующему виду:

```
class WizCoinException(Exception):          ❶
    """Выдается модулем wizcoin при некорректном использовании."""    ❷
    pass

class WizCoin:
```

```

def __init__(self, galleons, sickles, knuts):
    """Создание нового объекта WizCoin по значениям galleons,
       sickles и knuts."""
    self.galleons = galleons      ❸
    self.sickles = sickles
    self.knuts = knuts
    # ВНИМАНИЕ: методы __init__() НИКОГДА не содержат команду return.

--snip--

@property
def galleons(self):              ❹
    """Возвращает количество галлеонов в объекте."""
    return self._galleons

@galleons.setter
def galleons(self, value):      ❺
    if not isinstance(value, int): ❻
        raise WizCoinException('galleons attr must be set to an int, not a ❼
' + value.__class__.__qualname__)
    if value < 0:                 ❸
        raise WizCoinException('galleons attr must be a positive int, not
' + value.__class__.__qualname__)
    self._galleons = value

--snip--

```

С новыми изменениями добавляется класс `WizCoinException`, наследующий от встроенного класса `Python Exception`. Дос-строка класса описывает, как он используется в модуле `wizcoin` ❷. Такой подход считается правильным для модулей Python: объекты класса `WizCoin` выдают это исключение при неправильном использовании. Если объект `WizCoin` выдает исключения любых других классов (например, `ValueError` или `TypeError`), это с большой вероятностью указывает на наличие ошибки в классе `WizCoin`.

В методе `__init__()` свойствам `self.galleons`, `self.sickles` и `self.knuts` присваиваются соответствующие параметры ❸.

В конце файла после методов `total()` и `weight()` добавляется getter-метод ❹ и setter-метод ❺ для атрибута `self._galleons`. Getter-метод просто возвращает значение `self._galleons`. Setter-метод проверяет, является ли значение, присваиваемое свойству `galleons`, целым ❻ и положительным ❸ числом. Если хотя бы одна из проверок завершается неудачей, выдается исключение `WizCoinException` с сообщением об ошибке. Эта проверка предотвращает возможность того, что `_galleons` когда-либо будет присвоено недопустимое значение — при условии, что в коде всегда используется свойство `galleons`.

Все объекты Python автоматически получают атрибут `__class__`, который содержит ссылку на объект класса для заданного объекта. Другими словами, `value.__class__` содержит тот же объект класса, который возвращается вызовом

`type(value)`. Этот объект класса содержит атрибут с именем `__qualname__`, который содержит строку с именем класса (а именно *полное* имя класса, включающее имена любых классов, в которые вложен объект класса; о вложенных классах я в этой книге не рассказываю, а применяются они не так часто). Например, если в `value` хранится объект `date`, возвращенный вызовом `datetime.date(2021, 1, 1)`, то `value.__class__.__qualname__` будет содержать строку `'date'`.

Сообщения исключений используют `value.__class__.__qualname__` ⁷ для получения строки с именем объекта значения. С именем класса сообщение об ошибке становится более полезным для программиста, который его читает, потому что оно указывает не только на то, что аргумент `value` имеет неправильный тип, но и на то, к какому типу он относится и какой тип должен иметь.

Коды getter- и setter-методов для `_galleons` также необходимо скопировать, чтобы они использовались для атрибутов `_sickles` и `_knuts`. Эти коды остаются теми же, не считая того, что в качестве резервных переменных они используют атрибуты `_sickles` и `_knuts` вместо `_galleons`.

Свойства, доступные только для чтения

Вашим объектам могут понадобиться свойства, доступные только для чтения, которым нельзя присвоить новые значения оператором `=`. Чтобы сделать свойство доступным только для чтения, исключите setter- и deleter-методы.

Например, метод `total()` в классе `WizCoin` возвращает ценность объекта в кнатах. Из обычного метода его можно преобразовать в свойство, доступное только для чтения, потому что не существует разумного способа задать ценность объекта `WizCoin`. В конце концов, если вы присвоили `total` целое значение `1000`, что это означает — 1000 кнатов? Или 1 галлеон и 493 кната? Или какую-нибудь другую комбинацию? По этой причине мы сделаем `total` свойством, доступным только для чтения, для чего в файл `wizcoin.py` добавим код, выделенный жирным шрифтом:

```
@property
def total(self):
    """Общая ценность (в кнатах) всех монет в объекте WizCoin."""
    return (self.galleons * 17 * 29) + (self.sickles * 29) + (self.knuts)

# Обратите внимание на отсутствие setter- и deleter-методов для `total`.
```

После добавления декоратора функции `@property` перед `total()` Python будет вызывать метод `total()` при каждом обращении к `total`. Из-за отсутствия setter- и deleter-методов Python выдаст исключение `AttributeError`, если какой-либо код попытается изменить или удалить свойство `total`, используя его в присваивании или в команде `del` соответственно. Обратите внимание: значение свойства `total` зависит от значений свойств `galleons`, `sickles` и `knuts`; это свойство не базируется

на резервной переменной с именем `_total`. Введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> purse.total
1141
>>> purse.total = 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Возможно, вам не понравится, что при попытке изменения свойства, доступного только для чтения, программа немедленно аварийно завершается, но лучше действовать так. Если ваша программа способна менять свойство, доступное только для чтения, это наверняка приведет к ошибке в какой-то момент выполнения вашей программы. Если ошибка произойдет намного позднее изменения свойства, это усложнит поиск исходной причины. Немедленный фатальный сбой поможет быстрее обнаружить проблему.

Не путайте свойства, доступные только для чтения, с константами. Имена констант записываются в верхнем регистре, но программист сам отвечает за то, чтобы они не изменялись. Предполагается, что их значение остается неизменным на протяжении одного запуска программы. Свойство, доступное только для чтения, как и любой атрибут, связывается с объектом. Свойство, доступное только для чтения, невозможно напрямую присвоить или удалить. С другой стороны, значение, которое будет получено при обращении к нему, может изменяться. Свойство `total` класса `WizCoin` изменяется с изменением свойств `galleons`, `sickles` и `knuts`.

Когда использовать свойства

Как было показано в предыдущих разделах, свойства расширяют возможности управления использованием атрибутов класса; использование свойств — питонический стиль написания кода. Такие имена методов, как `getSomeAttribute()` или `setSomeAttribute()`, сигнализируют, что вместо них с большой вероятностью стоит использовать свойства.

Это не означает, что каждый метод, начинающийся с `get` или `set`, нужно немедленно заменить свойством. В некоторых ситуациях следует использовать метод даже в том случае, если его имя начинается с `get` или `set`. Вот несколько примеров.

- Медленные операции, занимающие более одной или двух секунд, например загрузка или отправка файла.
- Операции с побочными эффектами, например изменения других атрибутов или объектов.

- Операции, требующие дополнительных аргументов get- или set-операциям, например вызовы методов вида `emailObj.getFileAttachment(filename)`.

Программисты часто рассматривают методы как глаголы (в том смысле, что методы выполняют некоторое действие), а атрибуты и свойства — как существительные (в том смысле, что они представляют некоторый элемент или объект). Если ваш код не просто читает или задает некоторое значение, а выполняет действие, возможно, лучше использовать getter- или setter-метод. В конечном итоге решение зависит от того, что кажется более правильным вам как программисту.

У свойств Python есть одно огромное преимущество: их не обязательно использовать при создании класса. Вы можете задать обычные атрибуты, а если потом решите переключиться на свойства, атрибуты можно преобразовать в свойства без нарушения работоспособности какого-либо кода за пределами класса. При создании свойства, имя которого совпадает с именем атрибута, можно переименовать атрибут и добавить в него префикс `_`; программа будет работать так же, как и прежде.

Dunder-методы Python

Python содержит ряд особых *методов*, имена которых начинаются и заканчиваются двойным подчеркиванием `__(dunder)`. Такие методы также иногда называются *магическими*. Вам уже знаком dunder-метод `__init__()`, но в Python есть и другие. Они часто используются для *перегрузки* операторов — то есть добавления нестандартного поведения, позволяющего использовать объекты ваших классов с операторами Python, такими как `+` или `>=`. Другие dunder-методы позволяют объектам классов работать со встроенными функциями Python, такими как `len()` или `repr()`.

Как методы `__init__()` или getter-, setter- и deleter-методы свойств, специальные методы почти никогда не должны вызываться напрямую. Python вызывает их автоматически при использовании объектов с операторами или встроенными функциями. Например, если создать метод `__len__()` или `__repr__()` для вашего класса, он будет автоматически вызываться при передаче объекта этого класса функциям `len()` или `repr()` соответственно. Такие методы описаны в официальной документации Python на <https://docs.python.org/3/reference/datamodel.html>.

Далее мы рассмотрим разные виды dunder-методов, расширим наш класс `WizCoin` и используем в нем эти методы.

Dunder-методы строкового представления

Dunder-методы `__repr__()` и `__str__()` используются для создания строковых представлений объектов, которые обычно неизвестны Python. Обычно Python создает строковые представления объектов двумя способами. Встроенная функция

`repr()` возвращает код Python, при выполнении которого создается копия объекта. Встроенная функция `str()` возвращает строку, понятную для человека и содержащую ясную, полезную информацию об объекте. Например, чтобы просмотреть два представления объекта `datetime.date`, введите следующий фрагмент в интерактивной оболочке:

```
>>> import datetime
>>> newyears = datetime.date(2021, 1, 1)    ❶
>>> repr(newyears)
'datetime.date(2021, 1, 1)'                ❷
>>> str(newyears)
'2021-01-01'                                ❸
>>> newyears                                ❹
datetime.date(2021, 1, 1)
```

В этом примере строковое представление `'datetime.date(2021, 1, 1)'` объекта `datetime.date` ❷ содержит строку кода Python, создающую копию этого объекта ❶. Копия является точным представлением объекта. С другой стороны, строка `'2021-01-01'` ❸ выдает значение объекта в том виде, в котором оно хорошо воспринимается человеком. Если просто ввести объект в интерактивной оболочке ❹, будет выведена `repr`-строка. `Str`-строка объекта обычно выводится для пользователей, тогда как `repr`-строка используется в техническом контексте, например в сообщениях об ошибках и журнальных файлах.

Python знает, как выводить объекты встроенных типов, например целые числа и строки. Однако Python не знает, как следует выводить объекты тех классов, которые создаете вы. Если функция `repr()` не знает, как создать строку с представлением объекта, по умолчанию такая строка содержит адрес памяти и имя класса в угловых скобках: `'<wizcoin.WizCoin object at 0x00000212B4148EE0>'`. Чтобы создать такую строку для объекта `WizCoin`, введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> str(purse)
'<wizcoin.WizCoin object at 0x00000212B4148EE0>'
>>> repr(purse)
'<wizcoin.WizCoin object at 0x00000212B4148EE0>'
>>> purse
<wizcoin.WizCoin object at 0x00000212B4148EE0>
```

Эти строки плохо читаются и не содержат полезной информации для пользователя. Чтобы указать Python, какую строку следует использовать, можно реализовать dunder-методы `__repr__()` и `__str__()`. Метод `__repr__()` указывает, какую строку должен возвращать Python для объекта, переданного встроенной функции `repr()`. Метод `__str__()` указывает, какую строку должен возвращать Python для объекта,

переданного встроенной функции `str()`. Добавьте следующий фрагмент в конец файла `wizcoin.py`:

```
--snip--
def __repr__(self):
    """Возвращает строку с выражением, создающим объект."""
    return f'{self.__class__.__qualname__}({self.galleons}, {self.sickles}, {self.knuts})'

def __str__(self):
    """Возвращает строковое представление объекта, понятное для человека."""
    return f'{self.galleons}g, {self.sickles}s, {self.knuts}k'
```

Когда мы передаем `purse` функциям `repr()` и `str()`, Python вызывает dunder-методы `__repr__()` и `__str__()`. В нашем коде dunder-методы не вызываются.

Обратите внимание: f-строки, содержащие объект в фигурных скобках, неявно вызывают `str()` для получения строкового представления объекта. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> repr(purse) # Автоматически вызывает __repr__() класса WizCoin.
'WizCoin(2, 5, 10)'
>>> str(purse) # Автоматически вызывает __str__() класса WizCoin.
'2g, 5s, 10k'
>>> print(f'My purse contains {purse}.') # Вызывает __str__() класса WizCoin.
My purse contains 2g, 5s, 10k.
```

Когда мы передаем объект `WizCoin` в переменной `purse` функциям `repr()` и `str()`, во внутренней реализации Python вызывает методы `__repr__()` и `__str__()` класса `WizCoin`. Мы запрограммировали эти методы так, чтобы они возвращали более понятные и содержательные строки. Если ввести в интерактивной оболочке текст герг-строки `'WizCoin(2, 5, 10)'`, будет создан объект `WizCoin` с такими же атрибутами, как у объекта в `purse`. Str-строка содержит представление значения объекта, более понятное для человека: `'2g, 5s, 10k'`. Если вы используете объект `WizCoin` в f-строке, Python использует str-представление объекта.

Если объекты `WizCoin` настолько сложны, что создать их копию одним вызовом функции-конструктора невозможно, герг-строка заключается в угловые скобки — это показывает, что текст не рассматривается как код Python. Именно это происходит со строками обобщенного представления вроде `<wizcoin.WizCoin object at 0x00000212B4148EE0>`. При вводе этой строки в интерактивной оболочке выдается ошибка `SyntaxError`, так что ее невозможно спутать с кодом Python, создающим копию объекта.

Внутри метода `__repr__()` мы используем `self.__class__.__qualname__` вместо того, чтобы жестко фиксировать строку `'WizCoin'`; при субклассировании `WizCoin`

унаследованный метод `__repr__()` будет использовать имя subclasses, а не `'WizCoin'`. Кроме того, если класс `WizCoin` будет переименован, метод `__repr__()` автоматически использует обновленное имя.

Но str-строка объекта `WizCoin` выдает значения атрибута в удобной, компактной форме. Я настоятельно рекомендую реализовать методы `__repr__()` и `__str__()` во всех ваших классах.

КОНФИДЕНЦИАЛЬНАЯ ИНФОРМАЦИЯ В REPR-СТРОКАХ

Как упоминалось ранее, str-строки обычно выводятся для пользователей, а repr-строки используются в техническом контексте (например, в журналах). Но repr-строка может создать проблемы безопасности, если создаваемый объект содержит конфиденциальную информацию: пароли, медицинские сведения или данные личного порядка. В таком случае убедитесь, что метод `__repr__()` не включает эту информацию в возвращаемую строку. На случай, если в программе произойдет фатальный сбой, полезно включать переменные в файл журнала для упрощения отладки. Часто журналы не рассматриваются как конфиденциальная информация. В нескольких инцидентах, связанных с нарушением безопасности, в общедоступных журналах случайно были опубликованы пароли, номера кредитных карт, домашние адреса и другая закрытая информация. Помните об этом, когда будете писать методы `__repr__()` для вашего класса.

Числовые dunder-методы

Числовые *dunder-методы*, также называемые *математическими dunder-методами*, перегружают математические операторы Python: `+`, `-`, `*`, `/` и т. д. В настоящее время мы не можем выполнить такую операцию, как сложение двух объектов `WizCoin`, оператором `+`. Если вы попытаетесь это сделать, Python выдаст исключение `TypeError`, потому что не знает, как суммировать объекты `WizCoin`. Чтобы убедиться в этом, введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> tipJar = wizcoin.WizCoin(0, 0, 37)
>>> purse + tipJar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'WizCoin' and 'WizCoin'
```

Вместо того чтобы писать метод `addWizCoin()` для класса `WizCoin`, вы можете использовать dunder-метод `__add__()` и заставить объекты `WizCoin` работать с оператором `+`. Добавьте следующий фрагмент в конец файла `wizcoin.py`:

```
--snip--
def __add__(self, other):      ❶
    """Суммирует денежные величины двух объектов WizCoin."""
    if not isinstance(other, WizCoin):    ❷
        return NotImplemented

    return WizCoin(other.galleons + self.galleons, other.sickles + ❸
self.sickles, other.knuts + self.knuts)
```

Когда объект `WizCoin` стоит в левой части оператора `+`, Python вызывает метод `__add__()` ❶ и передает значение в правой части оператора `+` в параметре `other`. (Параметру можно назначить любое имя, но традиционно используется имя `other`.)

Помните, что методу `__add__()` можно передать объект любого типа, поэтому в метод необходимо включить проверку типа ❷. Например, бессмысленно прибавлять целое число или число с плавающей точкой к объекту `WizCoin`, потому что мы не знаем, к какому атрибуту его следует прибавить — `galleons`, `sickles` или `knuts`.

Метод `__add__()` создает новый объект `WizCoin` с атрибутами, равными сумме `galleons`, `sickles` и `knuts` объектов `self` и `other` ❸. Так как все три атрибута содержат целые числа, их можно суммировать оператором `+`. Теперь оператор `+` для класса `WizCoin` перегружен, и его можно использовать с объектами `WizCoin`.

Перегрузка оператора `+` позволяет создавать более понятный код. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10) # Создает объект WizCoin.
>>> tipJar = wizcoin.WizCoin(0, 0, 37) # Создает другой объект WizCoin.
>>> purse + tipJar # Создает новый объект WizCoin с суммой.
WizCoin(2, 5, 47)
```

Если в `other` передается объект неправильного типа, dunder-метод должен не выдавать исключение, а возвращать встроенное значение `NotImplemented`. Например, в следующем фрагменте в `other` передается целое число:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> purse + 42 # Объекты WizCoin и целые числа не могут суммироваться.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'WizCoin' and 'int'
```

Возвращение `NotImplemented` приказывает Python попробовать вызвать другие методы для выполнения этой операции (за подробностями обращайтесь к подразделу «Отраженные числовые dunder-методы» этой главы). Во внутренней реализации Python вызывает метод `__add__()` со значением `42` для параметра `other`, но этот метод тоже возвращает `NotImplemented`, что заставляет Python выдать исключение `TypeError`.

И хотя операции прибавления или вычитания целых чисел из объектов `WizCoin` не имеют смысла, будет разумно разрешить умножение объектов `WizCoin` на положительные целые числа; для этого определяется dunder-метод `__mul__()`. Добавьте следующий фрагмент в конец файла `wizcoin.py`:

```
--snip--
def __mul__(self, other):
    """Умножает количество монет на неотрицательное целое число."""
    if not isinstance(other, int):
        return NotImplemented
    if other < 0:
        # Умножение на отрицательное целое число приведет
        # к отрицательному количеству монет, что недопустимо.
        raise WizCoinException('cannot multiply with negative integers')

    return WizCoin(self.galleons * other, self.sickles * other, self.knuts * other)
```

Этот метод `__mul__()` позволяет умножать объекты `WizCoin` на положительные целые числа. Если `other` является целым числом, то это тип данных, которые ожидает получить метод `__mul__()`, и возвращать `NotImplemented` не нужно. Но если целое число — отрицательное, то умножение объекта `WizCoin` на него приведет к отрицательному количеству монет в объекте `WizCoin`. Так как это противоречит нашему подходу к проектированию класса, мы выдаем исключение `WizCoinException` с содержательным сообщением об ошибке.

ПРИМЕЧАНИЕ

Не изменяйте объект `self` в математическом dunder-методе. Метод всегда должен создавать и возвращать новый объект. Оператор `+` и другие числовые операторы всегда должны давать в результате новый объект, вместо того чтобы изменять значение объекта на месте.

Чтобы увидеть dunder-метод `__mul__()` в действии, введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10) # Создать объект WizCoin.
>>> purse * 10 # Умножить объект WizCoin на целое число.
WizCoin(20, 50, 100)
>>> purse * -2 # Умножение на отрицательное целое число приводит к ошибке.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\Al\Desktop\wizcoin.py", line 86, in __mul__
    raise WizCoinException('cannot multiply with negative integers')
wizcoin.WizCoinException: cannot multiply with negative integers
```

В табл. 17.1 приведен полный список числовых dunder-методов. Как правило, вам не требуется полный список этих методов для ваших классов. Вы сами решаете, какие методы для вас актуальны.

Таблица 17.1. Числовые dunder-методы

Dunder-метод	Операция	Оператор или встроенная функция
<code>__add__()</code>	Сложение	<code>+</code>
<code>__sub__()</code>	Вычитание	<code>-</code>
<code>__mul__()</code>	Умножение	<code>*</code>
<code>__matmul__()</code>	Матричное умножение (в Python 3.5 и выше)	<code>@</code>
<code>__truediv__()</code>	Деление	<code>/</code>
<code>__floordiv__()</code>	Целочисленное деление	<code>//</code>
<code>__mod__()</code>	Остаток	<code>%</code>
<code>__divmod__()</code>	Деление с остатком	<code>divmod()</code>
<code>__pow__()</code>	Возведение в степень	<code>**</code> , <code>pow()</code>
<code>__lshift__()</code>	Сдвиг влево	<code>>></code>
<code>__rshift__()</code>	Сдвиг вправо	<code><<</code>
<code>__and__()</code>	Поразрядная операция AND	<code>&</code>
<code>__or__()</code>	Поразрядная операция OR	<code> </code>
<code>__xor__()</code>	Поразрядная исключающая операция OR	<code>^</code>
<code>__neg__()</code>	Отрицание	Унарный <code>-</code> , как в <code>-42</code>
<code>__pos__()</code>	Тождество	Унарный <code>+</code> , как в <code>+42</code>
<code>__abs__()</code>	Абсолютное значение (модуль)	<code>abs()</code>
<code>__invert__()</code>	Поразрядное инвертирование	<code>~</code>
<code>__complex__()</code>	Комплексная форма числа	<code>complex()</code>
<code>__int__()</code>	Целая форма числа	<code>int()</code>
<code>__float__()</code>	Форма числа с плавающей точкой	<code>float()</code>
<code>__bool__()</code>	Логическая форма	<code>bool()</code>
<code>__round__()</code>	Округление	<code>round()</code>
<code>__trunc__()</code>	Целая часть	<code>math.trunc()</code>
<code>__floor__()</code>	Округление в меньшую сторону	<code>math.floor()</code>
<code>__ceil__()</code>	Округление в большую сторону	<code>math.ceil()</code>

Некоторые из этих методов актуальны для нашего класса `WizCoin`. Попробуйте написать собственные реализации методов `__sub__()`, `__pow__()`, `__int__()`, `__float__()` и `__bool__()`. Пример реализации доступен на <https://autbor.com/wizcoinfull>. Полное описание числовых dunder-методов в документации Python доступно на <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>.

Числовые dunder-методы позволяют использовать объекты ваших классов со встроенными математическими операторами Python. Если вы пишете методы с именами вида `multiplyBy()`, `convertToInt()` и т. д., описывающими задачу, которая выполняется существующим оператором или встроенной функцией, используйте числовые dunder-методы (а также отраженные dunder-методы и dunder-методы присваивания на месте, описанные в следующих двух разделах).

Отраженные числовые dunder-методы

Python вызывает числовые dunder-методы, когда объект находится в левой части математического оператора. Но если объект располагается в правой части математического оператора, то вызываются *отраженные числовые dunder-методы* (их также называют *обратными числовыми dunder-методами*).

Отраженные числовые dunder-методы полезны, потому что программисты, использующие ваш класс, не всегда будут записывать объект в левой части оператора, что может привести к непредвиденному поведению. Для примера рассмотрим, что произойдет, если `purse` содержит объект `WizCoin`, а Python вычисляет выражение `2 * purse`, когда `purse` находится в правой части оператора.

1. Так как 2 является целым числом, вызывается метод `__mul__()` класса `int`, которому в параметре `other` передается `purse`.
2. Метод `__mul__()` класса `int` не знает, как обрабатывать объекты `WizCoin`, поэтому он возвращает `NotImplemented`.
3. Пока Python не выдает ошибку `TypeError`. Так как `purse` содержит объект `WizCoin`, вызывается метод `__rmul__()` класса `WizCoin`, которому в параметре `other` передается 2.
4. Если `__rmul__()` возвращает `NotImplemented`, Python выдает ошибку `TypeError`. В противном случае `__rmul__()` возвращает объект, полученный в результате вычисления выражения `2 * purse`.

А вот выражение `purse * 2`, в котором `purse` находится в левой части оператора, работает иначе.

1. Так как `purse` содержит объект `WizCoin`, вызывается метод `__mul__()` класса `WizCoin`, которому в параметре `other` передается 2.

2. Метод `__mul__()` создает новый объект `WizCoin` и возвращает его.
3. Этот возвращенный объект — то, что возвращает выражение `purse * 2`.

Числовые dunder-методы и отраженные числовые dunder-методы имеют одинаковый код, если они обладают свойством *коммутативности*. Коммутативные операции (такие как сложение) дают одинаковый результат при записи в прямом и обратном направлении: $3 + 2$ — то же самое, что $2 + 3$.

Но существуют и другие операции, которые коммутативными не являются: $3 - 2$ и $2 - 3$ дают разные результаты. Любая коммутативная операция может просто вызывать исходный числовой dunder-метод каждый раз, когда вызывается отраженный числовой dunder-метод. Например, добавьте следующий фрагмент в конец файла `wizcoin.py`, чтобы определить отраженный числовой dunder-метод для операции умножения:

```
--snip--
def __rmul__(self, other):
    """Умножает количество монет на неотрицательное целое число."""
    return self.__mul__(other)
```

Умножение целого числа на объект `WizCoin` коммутативно: $2 * \text{purse}$ — то же самое, что $\text{purse} * 2$. Вместо того чтобы копировать и вставлять код из `__mul__()`, мы просто вызываем `self.__mul__()` и передаем параметр `other`.

После обновления файла `wizcoin.py` проверьте, как работает отраженный dunder-метод умножения. Для этого введите следующий фрагмент в интерактивную оболочку:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> purse * 10 # Вызывает __mul__() с параметром `other`, равным 10.
WizCoin(20, 50, 100)
>>> 10 * purse # Вызывает __rmul__() с параметром `other`, равным 10.
WizCoin(20, 50, 100)
```

Помните, что в выражении $10 * \text{purse}$ Python сначала вызывает метод `__mul__()` класса `int`, чтобы узнать, могут ли целые числа умножаться на объекты `WizCoin`. Конечно, встроенный класс `int` ничего не знает о создаваемых нами классах, поэтому он возвращает `NotImplemented`. Это значение сигнализирует Python о том, чтобы следующим для обработки операции был вызван метод `__rmul__()` класса `WizCoin` (если он существует). Если вызовы `__mul__()` класса `int` и `__rmul__()` класса `WizCoin` вернут `NotImplemented`, Python выдает исключение `TypeError`.

Только объекты `WizCoin` способны суммироваться друг с другом. Это гарантирует, что метод `__add__()` первого объекта `WizCoin` обработает операцию, поэтому реализовать `__radd__()` не нужно. Например, в выражении `purse + tipJar` метод `__add__()`

для объекта `purse` вызывается с передачей `tipJar` в параметре `other`. Так как этот вызов не возвращает `NotImplemented`, Python не пытается вызвать метод `__radd__()` объекта `tipJar` с передачей `purse` в параметре `other`.

В табл. 17.2 приведен полный список доступных отраженных dunder-методов.

Таблица 17.2. Отраженные числовые dunder-методы

Dunder-метод	Операция	Оператор или встроенная функция
<code>__radd__()</code>	Сложение	<code>+</code>
<code>__rsub__()</code>	Вычитание	<code>-</code>
<code>__rmul__()</code>	Умножение	<code>*</code>
<code>__rmatmul__()</code>	Матричное умножение (в Python 3.5 и выше)	<code>@</code>
<code>__rtruediv__()</code>	Деление	<code>/</code>
<code>__rfloordiv__()</code>	Целочисленное деление	<code>//</code>
<code>__rmod__()</code>	Остаток	<code>%</code>
<code>__rdivmod__()</code>	Деление с остатком	<code>divmod()</code>
<code>__rpow__()</code>	Возведение в степень	<code>**</code> , <code>pow()</code>
<code>__rlshift__()</code>	Сдвиг влево	<code>>></code>
<code>__rrshift__()</code>	Сдвиг вправо	<code><<</code>
<code>__rand__()</code>	Поразрядная операция AND	<code>&</code>
<code>__ror__()</code>	Поразрядная операция OR	<code> </code>
<code>__rxor__()</code>	Поразрядная исключающая операция OR	<code>^</code>

Полное описание отраженных dunder-методов доступно в документации Python на <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>.

Dunder-методы присваивания на месте (in-place)

Числовые и отраженные dunder-методы всегда создают новые объекты (вместо изменения объектов на месте). Dunder-методы присваивания на месте, вызываемые расширенными операторами присваивания (такими как `+=` и `*=`), изменяют объект на месте без создания нового объекта. (У этого правила есть исключение, о котором я расскажу в конце раздела.) Имена таких dunder-методов начинаются с `i` (например, `__iadd__()` и `__imul__()` для операторов `+=` и `*=` соответственно).

Например, при выполнении кода Python `purse *= 2` мы не предполагаем, что метод `__imul__()` класса `WizCoin` создаст и вернет новый объект `WizCoin` с вдвое большим количеством монет, а затем присвоит его переменной `purse`. Вместо этого метод `__imul__()` изменяет существующий объект `WizCoin` в `purse`, чтобы он содержал вдвое большее количество монет. Это тонкое, но важное отличие, которое необходимо учитывать, если ваши классы должны перегружать расширенные операторы присваивания.

Наши объекты `WizCoin` уже перегружают операторы `+` и `*`, поэтому определим dunder-методы `__iadd__()` и `__imul__()`, чтобы они также перегружали операторы `+=` и `*=`. В выражениях `purse += tipJar` и `purse *= 2` вызываются методы `__iadd__()` и `__imul__()`, при этом в параметре `other` передаются `tipJar` и `2` соответственно. Добавьте следующий фрагмент в конец файла `wizcoin.py`:

```
--snip--
def __iadd__(self, other):
    """Прибавляет монеты из другого объекта WizCoin к этому объекту."""
    if not isinstance(other, WizCoin):
        return NotImplemented

    # Объект `self` изменяется на месте:
    self.galleons += other.galleons
    self.sickles += other.sickles
    self.knuts += other.knuts
    return self # Dunder-методы присваивания на месте
                # почти всегда возвращают self.

def __imul__(self, other):
    """Умножает galleons, sickles и knuts этого объекта
    на неотрицательное целое число."""
    if not isinstance(other, int):
        return NotImplemented
    if other < 0:
        raise WizCoinException('cannot multiply with negative integers')

    # Класс WizCoin создает изменяемые объекты. НЕ СОЗДАВАЙТЕ новый
    # объект, как в следующем закомментированном коде:
    # return WizCoin(self.galleons * other, self.sickles * other,
    # self.knuts * other)

    # Объект `self` изменяется на месте:
    self.galleons *= other
    self.sickles *= other
    self.knuts *= other
    return self # Dunder-методы присваивания на месте
                # почти всегда возвращают self.
```

Объекты `WizCoin` могут использовать оператор `+=` с другими объектами `WizCoin` и оператор `*=` с положительными целыми числами. Заметим, что после проверки

параметра `other` методы присваивания на месте изменяют объект `self`, вместо того чтобы создавать новый объект `WizCoin`. Чтобы увидеть, как расширенные операторы присваивания изменяют объекты `WizCoin` на месте, введите следующий код в интерактивной оболочке:

```
>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10)
>>> tipJar = wizcoin.WizCoin(0, 0, 37)
>>> purse + tipJar    ❶
WizCoin(2, 5, 46)     ❷
>>> purse
WizCoin(2, 5, 10)
>>> purse += tipJar   ❸
>>> purse
WizCoin(2, 5, 47)
>>> purse *= 10       ❹
>>> purse
WizCoin(20, 50, 470)
```

Оператор `+` ❶ вызывает dunder-методы `__add__()` или `__radd__()` для создания и возвращения новых объектов ❷. Исходные объекты, с которыми работал оператор `+`, остаются без изменений. Dunder-методы присваивания на месте ❸ и ❹ должны изменять объект на месте, при условии что объект является изменяемым (то есть это объект, значение которого может изменяться). Исключение делается для неизменяемых объектов, так как такие объекты по определению не могут изменяться. В таком случае dunder-метод присваивания на месте должен создать и вернуть новый объект, как и числовые и отраженные dunder-методы.

Мы не сделали атрибуты `galleons`, `sickles` и `knuts` доступными только для чтения; это означает, что они могут изменяться. Таким образом, объекты `WizCoin` являются изменяемыми. Большинство классов, которые вы напишете, будут создавать изменяемые объекты, поэтому вам стоит проектировать dunder-методы присваивания на месте, так чтобы они изменяли объект на месте.

Если вы не реализуете dunder-метод присваивания на месте, Python вызовет числовой dunder-метод. Например, если в классе `WizCoin` отсутствует метод `__imul__()`, выражение `purse *= 10` вызовет `__mul__()` и присвоит возвращаемое значение переменной `purse`. Так как объекты `WizCoin` являются изменяемыми, это неожиданное поведение способно породить коварные ошибки.

Dunder-методы сравнения

Метод Python `sort()` и функция `sorted()` используют эффективный алгоритм сортировки, для выполнения которого достаточно простого вызова. Но если вы хотите сравнивать и сортировать объекты создаваемого вами класса, необходимо сообщить Python, как должны сравниваться два таких объекта — для этого нужно

реализовать dunder-методы сравнения. За кулисами Python вызывает dunder-методы сравнения каждый раз, когда ваши объекты используются в выражениях с операторами сравнения `<`, `>`, `<=`, `>=`, `==` и `!=`.

Прежде чем исследовать dunder-методы сравнения, рассмотрим шесть функций модуля `operator`, которые выполняют те же операции, что и шесть операций сравнения. Наши dunder-методы сравнения будут вызывать эти функции. Введите следующий фрагмент в интерактивной оболочке:

```
>>> import operator
>>> operator.eq(42, 42)          # "Равно"; то же, что 42 == 42
True
>>> operator.ne('cat', 'dog')   # "Не равно"; то же, что 'cat' != 'dog'
True
>>> operator.gt(10, 20)         # "Больше"; то же, что 10 > 20
False
>>> operator.ge(10, 10)         # "Больше или равно"; то же, что 10 >= 10
True
>>> operator.lt(10, 20)         # "Меньше"; то же, что 10 < 20
True
>>> operator.le(10, 20)         # "Меньше или равно"; то же, что 10 <= 20
True
```

Модуль `operator` предоставляет версии операторов сравнения в виде функций. Их реализации очень просты. Например, можно написать собственную функцию `operator.eq()` всего в две строки:

```
def eq(a, b):
    return a == b
```

Реализация оператора сравнения в виде функции полезна, потому что, в отличие от операторов, функции можно передавать как аргументы при вызове других функций. Мы воспользуемся этой возможностью и реализуем вспомогательный метод для наших dunder-методов сравнения.

Сначала добавьте следующий фрагмент в начало файла `wizcoin.py`. Эти команды импортирования открывают доступ к функциям модуля `operator` и позволяют проверить, является ли аргумент `other` нашего метода последовательностью, для чего он сравнивается с `collections.abc.Sequence`:

```
import collections.abc
import operator
```

Затем добавьте следующий фрагмент в конец файла `wizcoin.py`:

```
--snip--
def _comparisonOperatorHelper(self, operatorFunc, other): ❶
    """A helper method for our comparison dunder methods."""
```

```

if isinstance(other, WizCoin): ❷
    return operatorFunc(self.total, other.total)
elif isinstance(other, (int, float)): ❸
    return operatorFunc(self.total, other)
elif isinstance(other, collections.abc.Sequence): ❹
    otherValue = (other[0] * 17 * 29) + (other[1] * 29) + other[2]
    return operatorFunc(self.total, otherValue)
elif operatorFunc == operator.eq:
    return False
elif operatorFunc == operator.ne:
    return True
else:
    return NotImplemented
def __eq__(self, other): # "Равно"
    return self._comparisonOperatorHelper(operator.eq, other) ❶
def __ne__(self, other): # "Не равно"
    return self._comparisonOperatorHelper(operator.ne, other) ❷
def __lt__(self, other): # "Меньше"
    return self._comparisonOperatorHelper(operator.lt, other) ❸
def __le__(self, other): # "Меньше или равно"
    return self._comparisonOperatorHelper(operator.le, other) ❹
def __gt__(self, other): # "Больше"
    return self._comparisonOperatorHelper(operator.gt, other) ❺
def __ge__(self, other): # "Больше или равно"
    return self._comparisonOperatorHelper(operator.ge, other) ❻

```

Наши dunder-методы сравнения вызывают метод `_comparisonOperatorHelper()` ❶ и передают соответствующую функцию из модуля `operator` для параметра `operatorFunc`. При вызове `operatorFunc()` вызывается функция, которая была передана в операторе `operatorFunc` — `eq()` ❶, `ne()` ❷, `lt()` ❸, `le()` ❹, `gt()` ❺ или `ge()` ❻ — из модуля `operator`. В противном случае нам пришлось бы продублировать код в `_comparisonOperatorHelper()` в каждом из шести dunder-методов сравнения.

ПРИМЕЧАНИЕ

Функции (или методы), получающие другие функции в аргументах (как `_comparisonOperatorHelper()`), называются функциями высшего порядка.

Теперь наши объекты `WizCoin` можно сравнивать с другими объектами `WizCoin` ❷, с целыми числами и числами с плавающей точкой ❸, а также со значениями-последовательностями из трех чисел, представляющими значения `galleons`, `sickles` и `knuts` ❹. Чтобы увидеть эту возможность в действии, введите следующий фрагмент в интерактивной оболочке:

```

>>> import wizcoin
>>> purse = wizcoin.WizCoin(2, 5, 10) # Создать объект WizCoin.
>>> tipJar = wizcoin.WizCoin(0, 0, 37) # Создать другой объект WizCoin.
>>> purse.total, tipJar.total # Проверить значение в кнатах.

```

```

(1141, 37)
>>> purse > tipJar # Сравнение объектов WizCoin при помощи оператора.
True
>>> purse < tipJar
False
>>> purse > 1000 # Сравнение с целым числом.
True
>>> purse <= 1000
False
>>> purse == 1141
True
>>> purse == 1141.0 # Сравнение с числом с плавающей точкой.
True
>>> purse == '1141' # Объект WizCoin не равен никакому строковому значению.
False
>>> bagOfKnuts = wizcoin.WizCoin(0, 0, 1141)
>>> purse == bagOfKnuts
True
>>> purse == (2, 5, 10) # Возможно сравнение с кортежем из 3 целых чисел.
True
>>> purse >= [2, 5, 10] # Возможно сравнение со списком из 3 целых чисел.
True
>>> purse >= ['cat', 'dog'] # Должно привести к ошибке.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "C:\Users\AI\Desktop\wizcoin.py", line 265, in __ge__
        return self._comparisonOperatorHelper(operator.ge, other)
    File "C:\Users\AI\Desktop\wizcoin.py", line 237, in _
comparisonOperatorHelper
    otherValue = (other[0] * 17 * 29) + (other[1] * 29) + other[2]
IndexError: list index out of range

```

Наш вспомогательный метод вызывает `isinstance(other, collections.abc.Sequence)`, чтобы проверить, имеет ли `other` тип данных последовательности, например кортеж или список. Обеспечив возможность сравнения объектов `WizCoin` с последовательностями, можно писать код вида `purse >= [2, 5, 10]` для быстрого сравнения.

Не существует *отраженных* dunder-методов сравнения (таких как `__req__()` или `__rne__()`), которые вам было бы необходимо реализовать. Вместо этого `__lt__()` и `__gt__()` отражают друг друга, `__le__()` и `__ge__()` отражают друг друга, а `__eq__()` и `__ne__()` отражают сами себя. Дело в том, что следующие отношения остаются истинными независимо от значений в левой и правой части оператора:

- `purse > [2, 5, 10]` то же, что `[2, 5, 10] < purse`
- `purse >= [2, 5, 10]` то же, что `[2, 5, 10] <= purse`
- `purse == [2, 5, 10]` то же, что `[2, 5, 10] == purse`
- `purse != [2, 5, 10]` то же, что `[2, 5, 10] != purse`

СРАВНЕНИЯ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

При сравнении двух объектов встроенных типов последовательностей (таких, как строки, списки или кортежи) Python назначает более высокий приоритет более ранним элементам последовательности. Другими словами, более поздние элементы сравниваются только в том случае, если более ранние элементы имеют равные значения. Например, введите следующий фрагмент в интерактивной оболочке:

```
>>> 'Azriel' < 'Zelda'
True
>>> (1, 2, 3) > (0, 8888, 9999)
True
```

Строка 'Azriel' предшествует строке 'Zelda' (то есть меньше нее), потому что 'A' предшествует 'Z'. Кортеж (1, 2, 3) следует после (0, 8888, 9999) (то есть больше него), потому что 1 больше 0. А теперь введите следующий фрагмент в интерактивной оболочке:

```
>>> 'Azriel' < 'Aaron'
False
>>> (1, 0, 0) > (1, 0, 9999)
False
```

Строка 'Azriel' не предшествует 'Aaron', потому что, хотя элемент 'A' в 'Azriel' равен 'A' в 'Aaron', следующий элемент 'z' в 'Azriel' не предшествует 'a' в 'Aaron'. То же относится к кортежам (1, 0, 0) и (1, 0, 9999): первые два элемента каждого кортежа равны, поэтому третий элемент (0 и 9999 соответственно) определяет, что (1, 0, 0) предшествует (1, 0, 9999).

Таким образом, нам приходится принять проектировочное решение относительно класса WizCoin. Должен ли объект WizCoin(0, 0, 9999) предшествовать объекту WizCoin(1, 0, 0) или же следовать после него? Если количество галлеонов более значимо, чем количество сиклей или кнатов, то объект WizCoin(0, 0, 9999) должен предшествовать WizCoin(1, 0, 0). А может, объекты должны сравниваться на основании их значений кнатов? Тогда объект WizCoin(0, 0, 9999) (равно 9999 кнатам) должен следовать после WizCoin(1, 0, 0) (равно 493 кнатам). В программе `wizcoin.py` я решил использовать значение объекта в кнатах, потому что такое поведение соответствует принципам сравнения объектов WizCoin с целыми числами и числами с плавающей точкой. Вам придется неоднократно принимать подобные решения при проектировании собственных классов.

После того как вы реализуете dunder-методы сравнения, функция Python `sort()` автоматически использует их для сортировки объектов. Введите следующий фрагмент в интерактивной оболочке:

```
>>> import wizcoin
>>> oneGalleon = wizcoin.WizCoin(1, 0, 0) # 493 кната.
>>> oneSickle = wizcoin.WizCoin(0, 1, 0)  # 29 кнатов.
>>> oneKnut = wizcoin.WizCoin(0, 0, 1)    # 1 кнат.
>>> coins = [oneSickle, oneKnut, oneGalleon, 100]
>>> coins.sort() # Отсортировать по возрастанию.
>>> coins
[WizCoin(0, 0, 1), WizCoin(0, 1, 0), 100, WizCoin(1, 0, 0)]
```

В табл. 17.3 приведен полный список доступных dunder-методов сравнения.

Таблица 17.3. Dunder-методы сравнения и функции модулей `operator`

Dunder-метод	Операция	Оператор сравнения	Функция модуля <code>operator</code>
<code>__eq__()</code>	Равно	<code>==</code>	<code>operator.eq()</code>
<code>__ne__()</code>	Не равно	<code>!=</code>	<code>operator.ne()</code>
<code>__lt__()</code>	Меньше	<code><</code>	<code>operator.lt()</code>
<code>__le__()</code>	Меньше или равно	<code><=</code>	<code>operator.le()</code>
<code>__gt__()</code>	Больше	<code>></code>	<code>operator.gt()</code>
<code>__ge__()</code>	Больше или равно	<code>>=</code>	<code>operator.ge()</code>

Реализация этих методов доступна на <https://autbor.com/wizcoinfull>. Полное описание dunder-методов сравнения доступно в документации Python на https://docs.python.org/3/reference/datamodel.html#object.__lt__.

Dunder-методы сравнения позволяют использовать операторы сравнения Python с объектами вашего класса (вместо того, чтобы создавать ваши собственные методы). Если вы создаете методы с именами вида `equals()` или `isGreaterThan()`, это не питонический стиль и признак того, что вам стоит использовать dunder-методы сравнения.

Итоги

В языке Python объектно-ориентированные средства реализуются не так, как в других языках (например, Java или C++). Вместо явного определения

getter- и setter-методов в Python используются свойства, которые позволяют проверять атрибуты или делать их доступными только для чтения.

Python также позволяет перегружать операторы при помощи dunder-методов, имена которых начинаются и заканчиваются двойными символами подчеркивания `__`. Основные математические операторы перегружаются числовыми и отраженными числовыми dunder-методами. Эти методы дают возможность использовать встроенные операторы Python для работы с объектами созданных вами классов. Если методы не способны обработать тип данных объекта в другой части оператора, они возвращают встроенное значение `NotImplemented`. Такие методы создают и возвращают новые объекты, тогда как dunder-методы присваивания на месте (которые перегружают расширенные операторы присваивания) изменяют объект на месте. Dunder-методы сравнения не только реализуют шесть операторов сравнения Python для объектов, но и позволяют функции Python `sort()` сортировать объекты ваших классов. Для реализации этих dunder-методов можно воспользоваться функциями `eq()`, `ne()`, `lt()`, `le()`, `gt()` и `ge()` модуля `operator`.

Свойства и dunder-методы помогают писать классы, которые хорошо читаются и последовательно работают. Они избавят вас от необходимости писать значительный объем шаблонного кода, что обязательно в других языках (таких как Java). Если вам захочется больше узнать о написании питонического кода, более подробно о некоторых концепциях этой главы (и не только) рассказано в паре докладов Рэймонда Хеттингера (Raymond Hettinger) на PyCon: «Transforming Code into Beautiful, Idiomatic Python» (<https://youtu.be/OSGv2VnC0go/>) и «Beyond PEP 8 — Best Practices for Beautiful, Intelligible Code» (<https://youtu.be/wf-BqAjZb8M/>).

Об эффективном использовании Python еще можно сказать очень много. В книгах «Fluent Python» (O'Reilly Media, 2021) Лучано Рамальо (Luciano Ramalho)¹ и «Effective Python» (Addison-Wesley Professional, 2019) Бретта Слаткина (Brett Slatkin)² более подробно рассказывается о синтаксисе и передовых практиках Python. Эти книги стоит прочитать каждому, кто хочет больше знать о Python.

¹ Рамальо Л. Python. К вершинам мастерства.

² Слаткин Б. Секреты Python. 59 рекомендаций по написанию эффективного кода.

Эл Свейгарт

Python. Чистый код для продолжающих

Перевел с английского *Е. Матвеев*

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>Ю. Леонова</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, М. Молчанова</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: 07.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.05.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 30,960. Тираж 1000. Заказ 0000.