

Оглавление

О создателях книги	14
Об авторах.....	14
О научном редакторе.....	14
Предисловие	15
Для кого эта книга.....	15
Что мы рассмотрим	16
Как получить максимум от этой книги.....	17
Скачивание файлов с примерами кода.....	18
Скачивание цветных изображений.....	18
Условные обозначения	18
От издательства	20

Часть I. Перед началом работы

Глава 1. Текущее состояние Python	22
Технические требования.....	23
Где мы находимся и куда движемся	23
Почему и как изменился язык Python	23
Как не отставать от изменений в документации PEP	24
Внедрение Python 3 на момент написания этой книги	25
Основные различия между Python 3 и Python 2	26
Почему это должно нас волновать	26
Основные синтаксические различия и распространенные ошибки.....	27
Популярные инструменты и методы поддержания кросс-версионной совместимости	29
Не только CPython	33
Почему это должно нас волновать	33
Stackless Python	33
Jython	34
IronPython	35
PyPy	36
MicroPython	36
Полезные ресурсы	37
Резюме	38
Глава 2. Современные среды разработки на Python.....	39
Технические требования.....	40
Установка дополнительных пакетов Python с использованием pip.....	40
Изоляция сред выполнения.....	42
venv — виртуальное окружение Python	43
Изоляция среды на уровне системы	46
Виртуальные среды разработки, использующие Vagrant.....	47
Виртуальные среды, использующие Docker	49

Популярные инструменты повышения производительности	59
Пользовательские оболочки Python — ipython, bpython, ptpython и т. д.	60
Включение оболочек в собственные скрипты и программы	62
Интерактивные отладчики	63
Резюме	64

Часть II. Ремесло Python

Глава 3. Современные элементы синтаксиса — ниже уровня класса.....	66
Технические требования.....	67
Встроенные типы языка Python	67
Строки и байты	67
Контейнеры	73
Дополнительные типы данных и контейнеры.....	85
Специализированные контейнеры данных из модуля collections.....	85
Символическое перечисление с модулем enum	86
Расширенный синтаксис.....	88
Итераторы	88
Генераторы и операторы yield.....	91
Декораторы	94
Менеджеры контекста и оператор with.....	105
Функционально-стилевые особенности Python	109
Что такое функциональное программирование	110
Лямбда-функции.....	111
map(), filter() и reduce().....	112
Частичные объекты и функция partial()	115
Выражения генераторов	116
Аннотации функций и переменных	117
Общий синтаксис	117
Возможные способы применения	118
Статическая проверка типа с помощью тьюру	118
Иные элементы синтаксиса, о которых вы, возможно, не знаете.....	119
Оператор for... else...	119
Именованные аргументы.....	120
Резюме	122
Глава 4. Современные элементы синтаксиса — выше уровня класса	123
Технические требования.....	124
Протоколы в языке Python — методы и атрибуты с двойным подчеркиванием	124
Сокращение шаблонного кода с помощью классов данных	126
Создание подклассов встроенных типов	128
ПРМ и доступ к методам из суперклассов.....	131
Классы старого стиля и суперклассы в Python 2	133
Понимание ПРМ в Python	134
Ловушки суперкласса.....	138
Практические рекомендации.....	141
Паттерны доступа к расширенным атрибутам	141
Дескрипторы.....	142
Свойства	147
Слоты	150
Резюме	151

8 Оглавление

Глава 5. Элементы метапрограммирования	152
Технические требования.....	152
Что такое метапрограммирование.....	153
Декораторы как средство метапрограммирования.....	153
Декораторы класса	154
Использование <code>__new__()</code> для переопределения процесса создания экземпляра	156
Метаклассы.....	158
Генерация кода.....	165
Резюме	172
Глава 6. Как выбирать имена	173
Технические требования.....	174
PEP 8 и практические рекомендации по именованию.....	174
Почему и когда надо соблюдать PEP 8	174
За пределами PEP 8 — правила стиля внутри команды	175
Стили именования	175
Переменные.....	176
Руководство по именованию	184
Использование префиксов <code>is/has</code> в булевых элементах	184
Использование множественного числа в именах коллекций.....	185
Использование явных имен для словарей	185
Избегайте встроенных и избыточных имен.....	185
Избегайте уже существующих имен	186
Практические рекомендации по работе с аргументами	187
Сборка аргументов по итеративному принципу	187
Доверие к аргументам и тестам	188
Осторожность при работе с магическими аргументами <code>*args</code> и <code>**kwargs</code>	188
Имена классов	190
Имена модулей и пакетов	191
Полезные инструменты	191
Pylint.....	192
pycodestyle и flake8	193
Резюме	194
Глава 7. Создаем пакеты	195
Технические требования.....	195
Создание пакета	196
Странности в нынешних инструментах создания пакетов в Python	196
Конфигурация проекта.....	198
Пользовательская команда <code>setup</code>	207
Работа с пакетами в процессе разработки.....	208
Пакеты пространства имен	209
Почему это полезно	210
Загрузка пакета	214
PyPI — каталог пакетов Python	214
Пакеты с исходным кодом и пакеты сборок.....	216
Исполняемые файлы.....	220
Когда бывают полезны исполняемые файлы	221
Популярные инструменты	221
Безопасность кода Python в исполняемых пакетах	228
Резюме	230

Глава 8. Развёртывание кода	231
Технические требования.....	232
Двенадцатифакторное приложение	232
Различные подходы к автоматизации развёртывания	234
Использование Fabric для автоматизации развёртывания.....	235
Ваш собственный каталог пакетов или зеркало каталогов	239
Зеркала PyPI	240
Объединение дополнительных ресурсов с пакетом Python	241
Общие соглашения и практики	249
Иерархия файловой системы	249
Изоляция	250
Использование инструментов мониторинга процессов	250
Запуск кода приложения в пространстве пользователя.....	252
Использование обратного HTTP-прокси.....	253
Корректная перезагрузка процессов	254
Контрольно-проверочный код и мониторинг	256
Ошибки журнала — Sentry/Raven	256
Метрики систем мониторинга и приложений	260
Работа с журнальными приложениями.....	262
Резюме	267
Глава 9. Расширения Python на других языках	268
Технические требования.....	269
Различия между языками C и C++	269
Необходимость в использовании расширений	272
Повышение производительности критических фрагментов кода	272
Интеграция существующего кода, написанного на разных языках.....	273
Интеграция сторонних динамических библиотек	274
Создание пользовательских типов данных	274
Написание расширений.....	275
Расширения на чистом языке C	276
Написание расширений на Cython	291
Проблемы с использованием расширений	295
Дополнительная сложность.....	296
Отладка	297
Взаимодействие с динамическими библиотеками без расширений	297
Модуль ctypes	298
CFFI	304
Резюме	306

Часть III. Качество, а не количество

Глава 10. Управление кодом	308
Технические требования.....	308
Работа с системой управления версиями	308
Централизованные системы	309
Распределенные системы	312
Распределенные стратегии	313
Централизованность или распределенность.....	314
По возможности используйте Git.....	315
Рабочий процесс GitFlow и GitHub Flow	316

10 Оглавление

Настройка процесса непрерывной разработки	320
Непрерывная интеграция.....	321
Непрерывная доставка.....	325
Непрерывное развертывание	326
Популярные инструменты для непрерывной интеграции.....	326
Выбор правильного инструмента и распространенные ошибки	335
Резюме	338
Глава 11. Документирование проекта	339
Технические требования.....	339
Семь правил технической документации.....	340
Пишите в два этапа	340
Ориентируйтесь на читателя.....	341
Упрощайте стиль.....	342
Ограничивайте объем информации.....	342
Используйте реалистичные примеры кода	343
Пишите по минимуму, но достаточно	344
Используйте шаблоны.....	344
Документация как код	345
Использование строк документации в Python.....	345
Популярные языки разметки и стилей для документации.....	347
Популярные генераторы документации для библиотек Python	348
Sphinx	349
MkDocs.....	352
Сборка документации и непрерывная интеграция	352
Документирование веб-API	353
Документация как прототип API с API Blueprint	354
Самодокументирующиеся API со Swagger/OpenAPI.....	355
Создание хорошо организованной системы документации	356
Создание портфеля документации	356
Ваш собственный портфель документации.....	362
Создание шаблона документации	363
Шаблон для автора	364
Шаблон для читателя	364
Резюме	365
Глава 12. Разработка на основе тестирования	366
Технические требования.....	366
Я не тестирую.....	367
Три простых шага разработки на основе тестирования	367
О каких тестах речь	372
Стандартные инструменты тестирования в Python	375
Я тестирую	380
Ловушки модуля unittest	380
Альтернативы модулю unittest	381
Охват тестирования	388
Подделки и болванки	390
Совместимость среды тестирования и зависимостей	396
Разработка на основе документации	400
Резюме	402

Часть IV. Жажда скорости

Глава 13. Оптимизация — принципы и методы профилирования	404
Технические требования.....	404
Три правила оптимизации.....	405
Сначала — функционал	405
Работа с точки зрения пользователя.....	406
Поддержание читабельности и удобства сопровождения	407
Стратегии оптимизации	408
Пробуем свалить вину на другого	408
Масштабирование оборудования	409
Написание теста скорости.....	410
Поиск узких мест	410
Профилирование использования ЦП	411
Профилирование использования памяти.....	419
Профилирование использования сети	430
Резюме	433
Глава 14. Эффективные методы оптимизации.....	434
Технические требования.....	435
Определение сложности	436
Цикломатическая сложность	437
Нотация «O большое»	438
Уменьшение сложности через выбор подходящей структуры данных	440
Поиск в списке	440
Использование модуля collections	442
Тип deque	442
Тип defaultdict	444
Тип namedtuple	444
Использование архитектурных компромиссов	446
Использование эвристических алгоритмов или приближенных вычислений	446
Применение очереди задач и отложенная обработка.....	447
Использование вероятностной структуры данных	450
Кэширование	451
Детерминированное кэширование	452
Недетерминированное кэширование	455
Сервисы кэширования.....	456
Резюме	460
Глава 15. Многозадачность	461
Технические требования.....	461
Зачем нужна многозадачность	462
Многопоточность	463
Что такое многопоточность	464
Как Python работает с потоками.....	465
Когда использовать многопоточность	466
Многопроцессорная обработка	481
Встроенный модуль multiprocessing	483
Асинхронное программирование	489
Кооперативная многозадачность и асинхронный ввод/вывод	490
Ключевые слова await	491

Модуль <code>asyncio</code> в старых версиях Python	495
Практический пример асинхронного программирования	495
Интеграция синхронного кода с помощью фьючерсов <code>async</code>	498
Резюме	501

Часть V. Техническая архитектура

Глава 16. Событийно-ориентированное и сигнальное программирование	504
Технические требования.....	505
Что такое событийно-ориентированное программирование	505
Событийно-ориентированный != асинхронный.....	506
Событийно-ориентированное программирование в GUI.....	507
Событийно-ориентированная связь.....	509
Различные стили событийно-ориентированного программирования.....	511
Стиль на основе обратных вызовов.....	511
Стиль на основе субъекта	513
Тематический стиль	515
Событийно-ориентированные архитектуры	518
Очереди событий и сообщений	519
Резюме	521
Глава 17. Полезные паттерны проектирования	523
Технические требования.....	524
Порождающие паттерны	524
Синглтон.....	524
Структурные паттерны	527
Адаптер	528
Заместитель	542
Фасад.....	543
Поведенческие паттерны	544
Наблюдатель	544
Посетитель	546
Шаблонный метод.....	548
Резюме	550
Приложение. reStructuredText Primer.....	552
reStructuredText.....	552
Структура раздела	554
Списки	555
Форматирование внутри строк	556
Блок литералов.....	557
Ссылки.....	558

4 Современные элементы синтаксиса — выше уровня класса

В этой главе мы сосредоточимся на современных элементах синтаксиса Python и подробнее поговорим о классах и объектно-ориентированном программировании. Однако мы не будем касаться темы объектно-ориентированных паттернов проектирования, так как им посвящена глава 17. В данной главе мы проведем обзор самых передовых элементов синтаксиса Python, которые позволят улучшить код ваших классов.

Модель классов Python, известная нам, сильно эволюционировала в процессе истории Python 2. Долгое время мы жили в мире, в котором две реализации парадигмы объектно-ориентированного программирования сосуществовали на одном языке. Эти две модели были названы *старым* и *новым стилем класса*. Python 3 положил конец этой дихотомии, так что разработчикам доступен только новый стиль. Но по-прежнему важно знать, как обе модели работали в Python 2, поскольку это поможет в случае, если потребуется портировать старый код и написать обратно совместимые приложения. Знание того, как изменилась объектная модель, также поможет понять, почему сейчас она такая, какая есть. Именно по этой причине в следующей главе мы частенько будем говорить о Python 2, несмотря на то что книга посвящена последним версиям Python 3.

В этой главе:

- протоколы языка Python;
- сокращение шаблонного кода с помощью классов данных;
- создание подклассов встроенных типов;
- доступ к методам из суперклассов;
- слоты.

Технические требования

Файлы с примерами кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter4.

Протоколы в языке Python — методы и атрибуты с двойным подчеркиванием

Модель данных Python определяет много специально именованных методов, которые могут быть переопределены в пользовательских классах и тем самым расширять их синтаксис. Вы можете узнать эти методы по обрамляющему их названия *двойному подчеркиванию* — таково соглашение по наименованию для данных методов. Из-за этого они иногда называются *dunder* (сокращение от double underline — «двойное подчеркивание»).

Наиболее распространенный и очевидный пример — метод `__init__()`, который используется для инициализации экземпляра класса:

```
class CustomUserClass:
    def __init__(self, initialization_argument):
        ...
```

Эти методы, определенные отдельно или в комбинации, представляют собой так называемые языковые протоколы. Если объект реализует конкретные протоколы языка, то становится совместимым с конкретными частями синтаксиса Python. В табл. 4.1 приведены наиболее важные протоколы языка Python.

Таблица 4.1

Протокол	Методы	Описание
Протокол вызываемых объектов	<code>__call__()</code>	Объекты можно вызывать с помощью скобок: <code>instance()</code>
Протоколы дескрипторов	<code>__set__()</code> , <code>__get__()</code> и <code>__del__()</code>	Позволяют манипулировать паттерном атрибутов доступа в классах (см. подраздел «Дескрипторы» на с. 142)
Протокол контейнеров	<code>__contains__()</code>	Позволяет проверить, содержит ли объект некое значение через ключевое слово <code>in</code> : <code>value in instance</code>
Протокол итерируемых объектов	<code>__iter__()</code>	Позволяет объектам быть итерируемыми и использоваться для цикла <code>for</code> : <code>for value in instance:</code> ...

Протокол	Методы	Описание
Протоколы последовательности	<code>__len__()</code> , <code>__getitem__()</code>	Позволяют организовать индексацию объектов через синтаксис квадратных скобок и определять их длину с помощью встроенной функции: <code>item = instance[index]</code> <code>length = len(instance)</code>

Это наиболее важные протоколы языка с точки зрения данной главы. Полный список, конечно, гораздо длиннее. Например, в Python есть более 50 таких методов, которые позволяют эмулировать числовые значения. Каждый из этих методов коррелирует с конкретным математическим оператором и поэтому может рассматриваться как отдельный протокол языка. Полный список всех методов с двойным подчеркиванием можно найти в официальной документации модели данных Python (см. docs.python.org/3/reference/datamodel.html).

Языковые протоколы — основа концепции интерфейсов в Python. Одна из реализаций интерфейсов Python — это абстрактные базовые классы, которые позволяют задать произвольный набор атрибутов и методов в определении интерфейса. Такие определения интерфейсов в виде абстрактных классов могут впоследствии служить для проверки совместимости данного объекта с конкретным интерфейсом. Модуль `collections.abc` из стандартной библиотеки Python включает набор абстрактных базовых классов, которые относятся к наиболее распространенному протоколу языка Python. Более подробную информацию об интерфейсах и абстрактных базовых классах см. в пункте «Интерфейсы» на с. 530.

То же соглашение об именах применяется для определенных атрибутов пользовательских функций и хранения различных метаданных об объектах Python. Рассмотрим эти атрибуты:

- ❑ `__doc__` — перезаписываемый атрибут, который содержит документацию функции. По умолчанию заполняется функцией `docstring`;
- ❑ `__name__` — перезаписываемый атрибут, содержащий имя функции;
- ❑ `__qualname__` — перезаписываемый атрибут, который содержит полное имя функции, то есть полный путь к объекту (с именами классов) в глобальной области видимости модуля, в котором определен объект;
- ❑ `__module__` — перезаписываемый атрибут, содержащий имя модуля, к которому принадлежит функция;
- ❑ `__defaults__` — перезаписываемый атрибут, который содержит значения аргументов по умолчанию, если у функции есть таковые;
- ❑ `__code__` — перезаписываемый атрибут, содержащий код объекта компиляции функции;

- `__globals__` — атрибут только для чтения, который содержит ссылку на словарь глобальных переменных сферы действия этой функции. Сфера действия — пространство имен модуля, где определена эта функция;
- `__dict__` — перезаписываемый атрибут, содержащий словарь атрибутов функции. Функции в Python являются объектами первого класса, поэтому могут иметь любые произвольные аргументы, так же как и любой другой объект;
- `__closure__` — атрибут только для чтения, который содержит кортеж клеток со свободными переменными функции. Позволяет создавать параметризованные функции декораторов;
- `__annotations__` — перезаписываемый атрибут, который содержит аргумент функции и возвращает аннотации;
- `__kwdefaults__` — перезаписываемый атрибут, содержащий значение аргументов по умолчанию для именованных аргументов, если у функции они есть.

Далее рассмотрим, как сократить шаблонный код с помощью классов данных.

Сокращение шаблонного кода с помощью классов данных

Прежде чем углубиться в обсуждение классов Python, сделаем небольшое отступление. Мы обсудим относительно новые дополнения языка Python — а именно, классы данных. Модуль `dataclasses`, введенный в Python 3.7, включает в себя декоратор и функцию, которая позволяет легко добавлять генерированные специальные методы в пользовательские классы.

Рассмотрим следующий пример. Мы разрабатываем программу, выполняющую некие геометрические вычисления, и нам нужен класс, который позволяет хранить информацию о двумерных векторах. Мы будем выводить данные векторов на экран и выполнять простые математические операции, такие как сложение, вычитание и проверка равенства. Нам уже известно, что для этой цели можно использовать специальные методы. Мы можем реализовать наш класс `Vector` следующим образом:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Два вектора с оператором +"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )
```

```

def __sub__(self, other):
    """Вычитание векторов оператором -"""
    return Vector(
        self.x - other.x,
        self.y - other.y,
    )

def __repr__(self):
    """Возвращает текстовое представление вектора"""
    return f"<Vector: x={self.x}, y={self.y}>"

def __eq__(self, other):
    """Сравнение векторов на равенство"""
    return self.x == other.x and self.y == other.y

```

Ниже приведен пример интерактивной сессии, где показано поведение программы при использовании обычных операторов:

```

>>> Vector(2, 3)
<Vector: x=2, y=3>
>>> Vector(5, 3) + Vector(1, 2)
<Vector: x=6, y=5>
>>> Vector(5, 3) - Vector(1, 2)
<Vector: x=4, y=1>
>>> Vector(1, 1) == Vector(2, 2)
False
>>> Vector(2, 2) == Vector(2, 2)
True

```

Данная реализация вектора довольно проста, но в ней много повторяющегося кода, от которого можно было бы избавиться. Если в вашей программе используется много подобных простых классов, которые не требуют сложной инициализации, то понадобится много кода только для методов `__init__()`, `__repr__()` и `__eq__()`.

С помощью модуля `dataclasses` мы можем сделать код класса `Vector` намного короче:

```

from dataclasses import dataclass

@dataclass
class Vector:
    x: int
    y: int

    def __add__(self, other):
        """Два вектора с оператором +"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )

    def __sub__(self, other):
        """Вычитание векторов оператором -"""

```

```

        return Vector(
            self.x - other.x,
            self.y - other.y,
        )
    
```

Декоратор класса `dataclass` считывает аннотации атрибута класса `Vector` и автоматически создает методы `__init__()`, `__repr__()` и `__eq__()`. Проверка на равенство по умолчанию предполагает равенство двух экземпляров, если все соответствующие атрибуты равны друг другу.

Но это не все. Классы данных предлагают множество полезных функций. Они легко совместимы с другими протоколами Python. Предположим, мы хотим, чтобы наши экземпляры класса `Vector` были неизменяемыми. В таком случае они могут быть использованы в качестве ключей словаря или входить во множество. Вы можете сделать это, просто добавив в декоратор аргумент `frozen=True`, как в примере ниже:

```

@dataclass(frozen=True)
class FrozenVector:
    x: int
    y: int

```

Такой замороженный класс `Vector` становится совершенно неизменяемым, и вы не сможете изменить ни один из его атрибутов. Но складывать и вычитать векторы все еще можно, как и показано в примере, поскольку эти операции просто создают новый объект `Vector`.

В завершение разговора о классах данных в этой главе отметим, что вы можете задать значения для определенных атрибутов по умолчанию с помощью конструктора `field()`. Можно использовать и статические значения, и конструкторы других объектов. Рассмотрим следующий пример:

```

>>> @dataclass
... class DataClassWithDefaults:
...     static_default: str = field(default="this is static default value")
...     factory_default: list = field(default_factory=list)
...
>>> DataClassWithDefaults()
DataClassWithDefaults(static_default='this is static default value',
factory_default=[])

```

В следующем разделе мы поговорим о подклассах встроенных типов.

Создание подклассов встроенных типов

Создать подклассы встроенных типов в Python довольно просто. Встроенный тип `object` — общий предок для всех встроенных типов, а также всех пользовательских классов, не имеющих явно указанного родительского класса. Благодаря этому каждый раз, когда вам нужно реализовать класс, который ведет себя почти как один из встроенных типов, лучше всего сделать его подтипов.

Теперь рассмотрим код класса под названием `distinctdict`, где используется именно такой метод. Это будет подкласс обычного типа `dict`. Этот новый класс будет вести себя в основном так же, как обычный тип Python `dict`. Но вместо того, чтобы допускать наличие нескольких ключей с одним значением, при добавлении значения он вызывает подкласс `ValueError` со справочным сообщением.

Как уже было сказано, встроенный тип `dict` является объектом подкласса:

```
>>> isinstance(dict(), object)
True
>>> issubclass(dict, object)
True
```

Это значит, что мы могли бы легко определить собственный словарь в виде подкласса:

```
class distinctdict(dict):
    ...
```

Описанный ранее подход будет работать как надо, поскольку подклассы из типов `dict`, `list` и `str` были разрешены, начиная с версии Python 2.2. Но, как правило, лучше всего создавать подкласс с помощью модулей `collections`:

- `collections.UserDict`;
- `collections.UserList`;
- `collections.UserString`.

С этими классами, как правило, легче работать, поскольку обычные объекты `dict`, `list` и `str` сохраняются в виде атрибутов данных этих классов.

Ниже приведен пример реализации типа `distinctdict`, который отменяет часть свойств словаря, чтобы он теперь мог содержать только уникальные значения:

```
from collections import UserDict

class DistinctError(ValueError):
    """Выдается, когда в distinctdict добавляется дубликат"""

class distinctdict(UserDict):
    """Словарь, в который нельзя добавлять дублирующиеся значения"""
    def __setitem__(self, key, value):
        if value in self.values():
            if (
                (key in self and self[key] != value) or
                key not in self
            ):
                raise DistinctError(
                    "This value already exists for different key"
                )
        super().__setitem__(key, value)
```