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

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

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 сгенерированные 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 подтипом.

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)
```