2

Objects in Python

У нас есть дизайн в руках и мы готовы превратить этот дизайн в рабочую программу! Конечно, обычно так не бывает. Мы увидим примеры и подсказки для хорошего дизайна программного обеспечения на протяжении всей книги, но наше внимание сосредоточено на объектно-ориентированном программировании. Итак, давайте посмотрим на синтаксис Python, который позволяет нам создавать объектно-ориентированное программное обеспечение.

После завершения этой главы мы поймем следующее:

* Подсказки по типу Python
* Создание классов и создание экземпляров объектов в Python
* Организация занятий в пакеты и модули
* Как предположить, что люди не засоряют данные объекта, делая недействительным внутреннее состояние
* Работа со сторонними разработчиками packages available from the Python Package Index, PyPI

Эта глава также продолжит наше тематическое исследование, перейдя к дизайну некоторых из

Классы.

Ввод подсказки по типам

Прежде чем мы сможем внимательно присмотреться к созданию классов, нам нужно немного поговорить о том, что такое класс и как мы уверены, что используем его правильно. Центральная идея здесь заключается в том, что все в Python является объектом.

Когда мы пишем литеральные значения, такие как "Hello, world!" или 42, мы фактически создаем экземпляры встроенных классов. Мы можем запустить интерактивный Python и использовать встроенную функцию type() в классе, который определяет свойства этих объектов:

**>>>** type("Hello, world!")

<class 'str'>

**>>>** type(42)

<class 'int'>

Смысл *объектно-ориентированного* программирования заключается в решении задачи посредством взаимодействий объектов. Когда мы пишем 6\*7, умножение двух объектов обрабатывается методом встроенного класса int. Для более сложного поведения нам часто приходится писать уникальные, новые классы.

Вот первые два основных правила работы объектов Python:

* Все в Python является объектом
* Каждый объект определяется как экземпляр по крайней мере одного класса

Эти правила имеют много интересных последствий. Определение класса, которое мы пишем с помощью оператора class, создает новый объект типа class. Когда мы создаем **экземпляр** класса, объект класса будет использоваться для создания и инициализации объекта экзем­пляра.

В чем разница между классом и типом? Оператор class позволяет определять новые типы. Поскольку оператор class — это то, что мы используем, мы будем называть их классами по всему тексту. Смотрите объекты, типы, классы и экземпляры Python - глоссарий Эли Бендерского:

https://eli.thegreenplace.net/2012/03/30/python-objects-typesclasses-and-instances-a-glossary для этой полезной цитаты:

*«Термины «класс» и «тип» являются примером двух имен, относящихся к одному и тому же понятию».*

Мы будем следовать общепринятому использованию и называть подсказки типа аннотаций.

Есть еще одно важное правило:

* Переменная — это ссылка на объект. Представьте себе желтую записку с именем, нацарапанным на ней, шлепнутой по вещи.

Это не кажется слишком потрясающим, но на самом деле это довольно круто. Это означает, что информация о типе — что такое объект — определяется классом (классами), связанными с объектом. Информация об этом типе никак не присоединяется к переменной. Это приводит к тому, что код, подобный следующему, является допустимым, но очень запутанным Python:

**>>>** a\_string\_variable = "Hello, world!"

**>>>** type(a\_string\_variable)

<class 'str'>

**>>>** a\_string\_variable = 42

**>>>** type(a\_string\_variable)

<class 'int'>

Мы создали объект с помощью встроенного класса, str. Мы присвоили объекту длинное имя, a\_string\_variable. Затем мы создали объект, используя другой встроенный класс, int. Мы присвоили этому объекту такое же имя (Предыдущий строковый объект больше не имеет ссылок и перестает существовать).

Вот два шага, показанные рядом, показывающие, как переменная перемещается от объекта к объекту:



Рисунок 2.1: Имена переменных и объекты

Различные свойства являются частью объекта, а не переменной. Когда мы проверяем тип переменной с помощью type(), мы видим тип объекта, на который ссылается переменная в данный момент. Переменная не имеет собственного типа; это не что иное, как имя. Аналогичным образом, запрос id() переменной показывает идентификатор объекта, на который ссылается переменная. Таким образом, очевидно, что имя a\_string\_variable немного вводит в заблуждение, если мы присваиваем имя целочисленному объекту.

Проверка типа

Давайте продвинем отношения между объектом и типом на шаг дальше и рассмотрим еще некоторые последствия этих правил. Вот определение функции:

**>>> def odd**(n):

**... return** n % 2 != 0

**>>>** odd(3)

True

**>>>** odd(4)

False

Эта функция выполняет небольшое вычисление для переменной параметра n. Он вычисляет остаток после деления, по модулю. Если мы разделим нечетное число на два, у нас останется одно. Если мы разделим четное число на два, у нас останется ноль. Эта функция возвращает истинное значение для всех нечетных чисел.

Что произойдет, если мы не сможем предоставить номер? Что ж, давайте просто попробуем и посмотрим (распространенный способ изучения Python!). Введя код в интерактивной подсказке, мы получим что-то вроде этого:

**>>>** odd("Hello, world!")

Traceback (most recent call last):

File "<doctestexamples.md[9]>", line 1, in <module>

odd("Hello, world!")

File "<doctestexamples.md[6]>", line 2, in odd

return n % 2 != 0

TypeError: not all arguments converted during string formatting

Это важное следствие сверхгибких правил Python: ничто не мешает нам сделать что-нибудь глупое, что может вызвать исключение. Это важный совет:

Python не мешает нам использовать несуществующие методы объектов.

В нашем примере оператор %, предоставляемый классом str, работает не так, как оператор %, предоставляемый классом int, вызывая исключение. Для строк оператор % используется не очень часто, но он выполняет интерполяцию: "a=%d" % 113 вычисляет строку 'a=113'; если слева нет спецификации формата, такой как %d, исключением является TypeError. Для целых чисел это остаток от деления: 355 % 113 возвращает целое число 16.

Обратите внимание на разницу между классом и объектом в этом обсуждении. Мы можем думать о методе как о функции, прикрепленной к классу. Параметр self ссылается на конкретный экземпляр класса. При вызове метода для двух разных объектов один и тот же метод вызывается дважды, но в качестве параметра self передаются два разных объекта.

Обратите внимание, что при вызове метода p.reset() мы явно не передаем в него аргумент self. Python автоматически заботится об этой части за нас. Он знает, что мы вызываем метод для объекта p, поэтому он автоматически передает этот объект, p, методу класса Point.

Для некоторых это может помочь думать о методе как о функции, которая является частью класса. Вместо того, чтобы вызывать метод для объекта, мы могли бы вызвать функцию, определенную в классе, явно передавая наш объект в качестве аргумента self:

…

Выходные данные такие же, как и в предыдущем примере, потому что внутри произошел точно такой же процесс. Это не очень хорошая практика программирования, но она может помочь укрепить ваше понимание аргумента о себе.

Что произойдет, если мы забудем включить аргумент self в наше определение класса? Python выйдет на свободу с сообщением об ошибке, как показано ниже:

…

Сообщение об ошибке не так ясно, как могло бы быть («Эй, глупо, вы забыли определить метод с параметром self» может быть более информативным). Просто помните, что когда вы видите сообщение об ошибке, в котором указываются отсутствующие аргументы, первое, что нужно проверить, это не забыли ли вы параметр self в определении метода.

Больше аргументов

Как передать методу несколько аргументов? Давайте добавим новый метод, который позволяет перемещать точку в произвольное положение, а не только в начало координат. Мы также можем включить метод, который принимает другой объект Point в качестве входных данных и возвращает расстояние между ними:

…

Мы определили класс с двумя атрибутами, x и y, и тремя отдельными методами: move(), reset() и calculate\_distance().

Метод move() принимает два аргумента, x и y, и задает значения для объекта self. Метод reset() вызывает метод move(), так как сброс – это просто перемещение в определенное известное расположение.

Метод calculate\_distance() вычисляет евклидово расстояние между двумя точками. (Существует ряд других способов смотреть на расстояние. В главе 3 «Когда объекты похожи», мы рассмотрим некоторые альтернативы.) На данный момент мы надеемся, что вы понимаете математику. Определение, которая является функцией math.hypot(). В Python мы будем использовать self.x, но математики часто предпочитают писать .

Ниже приведен пример использования этого определения класса. Здесь показано, как вызвать метод с аргументами: включить аргументы в скобки и использовать ту же точечную нотацию для доступа к имени метода в экземпляре. Мы просто выбрали несколько случайных позиций для тестирования методов. Тестовый код вызывает каждый метод и выводит результаты на консоль:

…

The assert statement is a marvelous test tool; the program will bail if the expression after assert evaluates to False (or zero, empty, or None). In this case, we use it to ensure that the distance is the same regardless of which point called the other point's calculate\_distance() method. We'll see a lot more use of assert in *Chapter 13*, *Testing Object-Oriented Programs*, where we'll write more rigorous tests.

Утверждение assert является замечательным инструментом тестирования; программа будет освобождена, если выражение после assert вычисляет значение False (или ноль, пусто или None). В этом случае мы используем его, чтобы гарантировать, что расстояние одинаково независимо от того, какая точка вызывает метод calculate\_distance() другой точки. Мы увидим гораздо больше использования assert *в главе 13, Тестирование объектно-ориентированных программ,* где мы напишем более строгие тесты.

Инициализация объекта

If we don't explicitly set the x and y positions on our Point object, either using move or by accessing them directly, we'll have a broken Point object with no real position. What will happen when we try to access it?

Если мы явно не установим позиции x и y для нашего объекта Point, либо с помощью move, либо путем прямого доступа к ним, у нас будет объект «Сломанная точка» (broken Point) без реального положения. Что произойдет, когда мы попытаемся получить к нему доступ?

Что ж, давайте просто попробуем и посмотрим. *Попробуйте и посмотрите*, это чрезвычайно полезный инструмент для изучения Python. Откройте своего интерактивного переводчика и печатайте (Использование интерактивной подсказки, в конце концов, является одним из инструментов, которые мы использовали для написания этой книги).

В следующем интерактивном сеансе показано, что произойдет, если мы попытаемся получить доступ к отсутствующему атрибуту. Если вы сохранили предыдущий пример в виде файла или используете примеры, распространяемые вместе с книгой, вы можете загрузить его в интер­претатор Python с помощью команды python -i more\_arguments.py:

…

Ну, по крайней мере, это породило полезное исключение. Мы подробно рассмотрим исключения *в главе 4, Ожидание неожиданного*. Вы, вероятно, видели их раньше (особенно вездесущий SyntaxError, что означает, что вы ввели что-то неправильно!). На этом этапе просто имейте в виду, что это означает, что что-то пошло не так.

Выходные данные полезны для отладки. В интерактивном интерпретаторе он сообщает нам, что ошибка произошла в *строке 1*, что верно только частично (в интерактивном сеансе одновременно выполняется только одна инструкция). Если бы мы запускали сценарий в файле, он сообщал бы нам точный номер строки, что облегчало бы поиск кода- нарушителя. Кроме того, он сообщает нам, что ошибка является AttributeError, и дает полезное сообщение, сообщающее нам, что означает эта ошибка.

Мы можем поймать и восстановиться после этой ошибки, но в этом случае кажется, что мы должны были указать какое-то значение по умолчанию. Возможно, каждый новый объект должен быть сброшен reset() по умолчанию, или, может быть, было бы неплохо, если бы мы могли заставить пользователя сказать нам, какими должны быть эти позиции при создании объекта.

Интересно, что ***mypy*** не может определить, должно ли y быть атрибутом объекта Point. Атрибуты — по определению — динамичны, поэтому нет простого списка, который является частью определения класса. Тем не менее, Python имеет некоторые широко распространенные соглашения, которые могут помочь назвать ожидаемый набор атрибутов.

Большинство объектно-ориентированных языков программирования имеют концепцию **конструктора**, специального метода, который создает и инициализирует объект при его создании. Python немного отличается; он имеет конструктор и инициализатор. Метод конструктора, \_\_new\_\_(), редко используется, если вы не делаете что-то очень экзотическое. Итак, мы начнем наше обсуждение с гораздо более распространенного метода инициализации, \_\_init\_\_().

Метод инициализации Python такой же, как и любой другой метод, за исключением того, что он имеет специальное имя \_\_init\_\_. Ведущие и замыкающие двойные подчеркивания означают, что это особый метод, который интерпретатор Python будет рассматривать как особый случай.

Никогда не называйте свой собственный метод с ведущими и задними двойными подчеркиваниями. Это может ничего не значить для Python сегодня, но всегда есть вероятность, что дизайнеры Python добавят функцию, которая имеет специальное назначение с этим именем в будущем. Когда они это сделают, ваш код сломается.

Добавим функцию инициализации в наш класс Point, которая требует, чтобы пользователь предоставлял координаты x и y при создании экземпляра объекта Point:

…

Теперь наш объект Point никогда не может обойтись без координат x и y! Если мы попытаемся построить экземпляр Point без включения правильных параметров инициализации, он завершится ошибкой недостаточного количества аргументов, подобной той, которую мы получили ранее, когда забыли аргумент self в определении метода.

В большинстве случаев мы помещаем наши инициализации в функцию \_\_init\_\_(). Очень важно убедиться, что все атрибуты инициализированы в методе \_\_init\_\_(). Это помогает инструменту ***mypy***, предоставляя все атрибуты в одном очевидном месте. Это также помогает людям читать ваш код; это избавляет их от необходимости читать все приложение, чтобы найти таинственные атрибуты, установленные за пределами определения класса.

While they're optional, it's generally helpful to include type annotations on the method parameters and result values. After each parameter name, we've included the expected type of each value. At the end of the definition, we've included the twocharacter -> operator and the type returned by the method.

Хотя они являются необязательными, обычно полезно включать аннотации типов в параметры метода и значения результатов. После имени каждого параметра мы включили ожидаемый тип каждого значения. В конце определения мы включили оператор twocharacter -> и тип, возвращаемый методом.

Ввод подсказок и значений по умолчанию

Как мы уже отмечали несколько раз, подсказки являются необязательными. Они ничего не делают во время выполнения. Однако есть инструменты, которые могут изучить подсказки, чтобы проверить согласованность. Инструмент ***mypy*** широко используется для проверки подсказок типа.

Если мы не хотим делать два аргумента обязательными, мы можем использовать тот же синтаксис, который используют функции Python для предоставления аргументов по умолчанию. Синтаксис аргумента ключевого слова добавляет знак равенства после имени каждой переменной. Если вызывающий объект не предоставляет этот аргумент, вместо него используется аргумент по умолчанию. Переменные по-прежнему будут доступны функции, но они будут иметь значения, указанные в списке аргументов. Вот пример:

class Point:

def \_\_init\_\_(self, x: float = 0, y: float = 0) -> None:

self.move(x, y)

Определения для отдельных параметров могут быть длинными, что приводит к очень длинным строкам кода. В некоторых примерах вы увидите, что эта одна логическая строка кода расширена до нескольких физических строк. Это зависит от того, как Python объединяет физические линии для соответствия (). Мы можем написать это, когда строка станет длинной:

class Point:

def \_\_init\_\_(

self,

x: float = 0,

y: float = 0

) -> None:

self.move(x, y)

Этот стиль используется не очень часто, но он действителен и делает строки короче и легче для чтения.

Подсказки по типу и значения по умолчанию удобны, но мы можем сделать еще больше, чтобы предоставить класс, который прост в использовании и легко расширяется при возникновении новых требований. Мы добавим документацию в виде docstrings.

Объяснение себя с помощью строк документов

Python может быть чрезвычайно простым для чтения языком программирования; некоторые могут сказать, что это само документирование. Однако при выполнении объектно-ориентированного программирования важно написать документацию API, которая четко суммирует то, что делает каждый объект и метод. Поддержание документации в актуальном состоянии затруднено; лучший способ сделать это – записать его прямо в наш код.

Python поддерживает это с помощью **docstrings**. Каждый заголовок класса, функции или метода может иметь стандартную строку Python в качестве первой строки с отступом внутри определения (строки, которая заканчивается двоеточием).

Docstrings — это строки Python, заключенные в апострофы (') или кавычки ("). Часто строки документов довольно длинные и охватывают несколько строк (руководство по стилю предполагает, что длина строки не должна превышать 80 символов), которые могут быть отформатированы как многострочные строки, заключенные в соответствующие символы тройного апострофа (''') или тройной кавычки (""").

Docstring должен четко и кратко обобщать цель класса или метода, который он описывает. Он должен объяснять любые параметры, использование которых не сразу очевидно, а также является хорошим местом для включения кратких примеров того, как использовать API. Также следует отметить любые предостережения или проблемы, о которых должен знать ничего не подозревающий пользователь API.

Одна из лучших вещей, которые можно включить в docstring, – это конкретный пример. Такие инструменты, как **doctest**, могут найти и подтвердить правильность этих примеров. Все примеры в этой книге проверяются с помощью инструмента doctest.

Чтобы проиллюстрировать использование docstrings, мы закончим этот раздел нашим полностью документированным классом Point:

class Point:

"""

Represents a point in two-dimensional geometric coordinates

>>> p\_0 = Point()

>>> p\_1 = Point(3, 4)

>>> p\_0.calculate\_distance(p\_1)

5.0

"""

def \_\_init\_\_(self, x: float = 0, y: float = 0) -> None:

"""

Initialize the position of a new point. The x and y

coordinates can be specified. If they are not, the

point defaults to the origin.

:param x: float x-coordinate

:param y: float x-coordinate

"""

self.move(x, y)

def move(self, x: float, y: float) -> None:

"""

Move the point to a new location in 2D space.

:param x: float x-coordinate

:param y: float x-coordinate

"""

self.x = x

self.y = y

def reset(self) -> None:

"""

Reset the point back to the geometric origin: 0, 0

"""

self.move(0, 0)

def calculate\_distance(self, other: "Point") -> float:

"""

Calculate the Euclidean distance from this point

to a second point passed as a parameter.

:param other: Point instance

:return: float distance

"""

return math.hypot(self.x - other.x, self.y - other.y)

Попробуйте ввести или загрузить (помните, что это python -i point.py) этот файл в интерактивный интерпретатор. Затем введите help(Point)<enter> в командной строке Python.

...

Мало того, что наша документация так же отполирована, как и документация по встроенным функциям, но мы можем запустить python -m doctest point\_2.py, чтобы подтвердить пример, показанный в docstring.

Кроме того, мы также можем запустить ***mypy***, чтобы проверить подсказки типа. Используйте mypy –-strict src/\*.py для проверки всех файлов в папке src. Если проблем нет, приложение ***mypy*** не выдает никаких выходных данных. (Помните, что ***mypy*** не является частью стандартной установки, поэтому вам нужно будет добавить его. Проверьте предисловие для получения информации о дополнительных пакетах, которые необходимо установить.)