

Классы и объекты

Создание классов и объектов

Создание класса в *Python* начинается с инструкции *class*. Вот так будет выглядеть минимальный класс.

```
class C:  
    pass
```

Класс состоит из объявления (инструкция *class*), имени класса (нашем случае это имя *C*) и тела класса, которое содержит атрибуты и методы (в нашем минимальном классе есть только одна инструкция *pass*).

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

имя_объекта = имя_класса()

Статические и динамические атрибуты класса

Как уже было сказано выше, класс может содержать атрибуты и методы. Атрибут может быть статическим и динамическим (уровня объекта класса). Суть в том, что для работы со статическим атрибутом, вам не нужно создавать экземпляр класса, а для работы с динамическим – нужно. Пример:

```
class Rectangle:
    default_color = "green"

    def __init__(self, width, height):
        self.width = width
        self.height = height
```

В представленном выше классе, атрибут *default_color* – это статический атрибут, и доступ к нему, как было сказано выше, можно получить не создавая объект класса *Rectangle* .

```
>>> Rectangle.default_color
'green'
```

width и *height* – это динамические атрибуты, при их создании было использовано ключевое слово *self*. Пока просто примите это как должное, более подробно про *self* будет рассказано ниже. Для доступа к *width* и *height* предварительно нужно создать объект класса *Rectangle*:

```
>>> rect = Rectangle(10, 20)
>>> rect.width
10
>>> rect.height
20
```

Если обратиться через класс, то получим ошибку:

```
>>> Rectangle.width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Rectangle' has no attribute 'width'
```

При этом, если вы обратитесь к статическому атрибуту через экземпляр класса, то все будет ОК, до тех пор, пока вы не попытаетесь его поменять.

Проверим ещё раз значение атрибута default_color:

```
>>> Rectangle.default_color  
'green'
```

Присвоим ему новое значение:

```
>>> Rectangle.default_color = "red"  
>>> Rectangle.default_color  
'red'
```

Создадим два объекта класса Rectangle и проверим, что default_color у них совпадает:

```
>>> r1 = Rectangle(1,2)  
>>> r2 = Rectangle(10, 20)  
>>> r1.default_color  
'red'  
>>> r2.default_color  
'red'
```

Если поменять значение default_color через имя класса Rectangle, то все будет ожидаемо: у объектов r1 и r2 это значение изменится, но если поменять его через экземпляр класса, то у экземпляра будет создан атрибут с таким же именем как статический, а доступ к последнему будет потерян:

Меняем default_color через r1:

```
>>> r1.default_color = "blue"  
>>> r1.default_color  
'blue'
```

При этом у r2 остается значение статического атрибута:

```
>>> r2.default_color  
'red'  
>>> Rectangle.default_color  
'red'
```

Вообще напрямую работать с атрибутами – не очень хорошая идея, лучше для этого использовать свойства.

Методы класса

Добавим к нашему классу метод. Метод – это функция, находящаяся внутри класса и выполняющая определенную работу.

Методы бывают статическими, классовыми (среднее между статическими и обычными) и уровня класса (будем их называть просто словом метод). Статический метод создается с декоратором `@staticmethod`, классовый – с декоратором `@classmethod`, первым аргументом в него передается `cls`, обычный метод создается без специального декоратора, ему первым аргументом передается `self`:

```
class MyClass:
    @staticmethod
    def ex_static_method():
        print("static method")
    @classmethod
    def ex_class_method(cls):
        print("class method")
    def ex_method(self):
        print("method")
```

Статический и классовый метод можно вызвать, не создавая экземпляр класса, для вызова `ex_method()` нужен объект:

```
>>> MyClass.ex_static_method()
static method

>>> MyClass.ex_class_method()
class method

>>> MyClass.ex_method()
```

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

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

```
TypeError: ex_method() missing 1 required positional argument: 'self'
```

```
>>> m = MyClass()
```

```
>>> m.ex_method()
```

```
method
```

Конструктор класса и инициализация экземпляра класса

В Python разделяют конструктор класса и метод для инициализации экземпляра класса. Конструктор класса это метод *new(cls, *args, **kwargs)* для инициализации экземпляра класса используется метод *init(self)*. При этом, как вы могли заметить *new* – это классовый метод, а *init* таким не является. Метод *new* редко переопределяется, чаще используется реализация от базового класса *object* (см. раздел Наследование), *init* же наоборот является очень удобным способом задать параметры объекта при его создании.

Создадим реализацию класса *Rectangle* с измененным конструктором и инициализатором, через который задается ширина и высота прямоугольника:

```
class Rectangle:
```

```
    def __new__(cls, *args, **kwargs):
```

```
        print("Hello from __new__")
```

```
        return super().__new__(cls)
```

```
def __init__(self, width, height):  
    print("Hello from __init__")  
    self.width = width  
    self.height = height
```

```
>>> rect = Rectangle(10, 20)
```

```
Hello from __new__
```

```
Hello from __init__
```

```
>>> rect.width
```

```
10
```

```
>>> rect.height
```

```
20
```

Что такое self?

До этого момента вы уже успели познакомиться с ключевым словом `self`. `self` – это ссылка на текущий экземпляр класса, в таких языках как Java, C# аналогом является ключевое слово `this`. Через `self` вы получаете доступ к атрибутам и методам класса внутри него:

```
class Rectangle:
```

```
    def __init__(self, width, height):
```

```
        self.width = width
```

```
        self.height = height
```

```
    def area(self):
```

```
        return self.width * self.height
```

В приведенной реализации метод `area` получает доступ к атрибутам `width` и `height` для расчета площади. Если бы в качестве первого параметра не было указано `self`, то при попытке вызвать `area` программа была бы остановлена с ошибкой.

Уровни доступа атрибута и метода

Если вы знакомы с языками программирования Java, C#, C++ то, наверное, уже задались вопросом: “а как управлять уровнем доступа?”. В перечисленных языках вы можете явно указать для переменной, что доступ к ней снаружи класса запрещен, это делается с помощью ключевых слов (`private`, `protected` и т.д.). В Python таких возможностей нет, и любой может обратиться к атрибутам и методам вашего класса, если возникнет такая необходимость. Это существенный недостаток этого языка, т.к. нарушается один из ключевых принципов ООП – инкапсуляция. Хорошим тоном считается, что для чтения/изменения/удаления какого-то атрибута должны использоваться специальные методы, которые называются **getter/setter/deleter**, их можно реализовать, но ничего не помешает изменить атрибут напрямую. При этом есть соглашение, что метод или атрибут, который начинается с нижнего подчеркивания, является скрытым, и снаружи класса трогать его не нужно (хотя сделать это можно).

Внесем соответствующие изменения в класс `Rectangle`:

```
class Rectangle:

    def __init__(self, width, height):
        self._width = width
        self._height = height

    def get_width(self):
        return self._width

    def set_width(self, w):
        self._width = w

    def get_height(self):
        return self._height
```

```
def set_height(self, h):  
    self._height = h  
  
def area(self):  
    return self._width * self._height
```

В приведенном примере для доступа к `_width` и `_height` используются специальные методы, но ничего не мешает вам обратиться к ним (атрибутам) напрямую.

```
>>> rect = Rectangle(10, 20)  
>>> rect.get_width()  
10  
>>> rect._width  
10
```

Если же атрибут или метод начинается с двух подчеркиваний, то тут напрямую вы к нему уже не обратитесь (простым образом). Модифицируем наш класс `Rectangle`:

```
class Rectangle:  
  
    def __init__(self, width, height):  
        self.__width = width  
        self.__height = height  
  
    def get_width(self):  
        return self.__width  
  
    def set_width(self, w):  
        self.__width = w  
  
    def get_height(self):  
        return self.__height  
  
    def set_height(self, h):  
        self.__height = h  
  
    def area(self):
```



```
return self.__width * self.__height
```

Попытка обратиться к `__width` напрямую вызовет ошибку, нужно работать только через `get_width()`:

```
>>> rect = Rectangle(10, 20)

>>> rect.__width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute '__width'

>>> rect.get_width()
10
```

Попытка обратиться к `__width` напрямую вызовет ошибку, нужно работать только через `get_width()`:

```
>>> rect = Rectangle(10, 20)

>>> rect.__width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute '__width'

>>> rect.get_width()
10
```

Но на самом деле это сделать можно, просто этот атрибут теперь для внешнего использования носит название: *Rectangle_width*:

```
>>> rect._Rectangle__width
10

>>> rect._Rectangle__width = 20

>>> rect.get_width()
20
```

Свойства

Свойством называется такой метод класса, работа с которым подобна работе с атрибутом. Для объявления метода свойством необходимо использовать декоратор `@property`.

Важным преимуществом работы через свойства является то, что вы можете осуществлять проверку входных значений, перед тем как присвоить их атрибутам.

Сделаем реализацию класса `Rectangle` с использованием свойств:

```
class Rectangle:

    def __init__(self, width, height):
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self):
        return self.__height

    @height.setter
    def height(self, h):
        if h > 0:
            self.__height = h
```

```
    else:
        raise ValueError

    @height.deleter
    def height(self):
        del self.__height

    def area(self):
        return self.__width * self.__height
```

Теперь работать с **width** и **height** можно так, как будто они являются атрибутами:

```
>>> rect = Rectangle(10, 20)

>>> rect.width
10

>>> rect.height
20
```

Можно не только читать, но и задавать новые значения свойствам:

```
>>> rect.width = 50

>>> rect.width
50

>>> rect.height = 70

>>> rect.height
70
```

Если вы обратили внимание: в setter'ах этих свойств осуществляется проверка входных значений, если значение меньше нуля, то будет выброшено исключение `ValueError`:

```
>>> rect.width = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "test.py", line 28, in width
    raise ValueError
ValueError
```

Организация доступа к атрибутам объекта

Доступ к атрибутам объекта можно также осуществлять с помощью встроенных функций `getattr`, `setattr` и `delattr` извне так и внутри объекта с помощью встроенных методов `__getattr__`, `__setattr__` и `__delattr__`.

Наследование

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка может быть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в Python можно его использовать. По умолчанию все классы в Python являются наследниками от `object`, явно этот факт указывать не нужно.

Синтаксически создание класса с указанием его родителя выглядит так:

`class имя_класса(имя_родителя1, [имя_родителя2,..., имя_родителя_n])`

Переработаем наш пример так, чтобы в нем присутствовало наследование:

```
class Figure:
```

```
def __init__(self, color):  
    self.__color = color
```

```
@property  
def color(self):  
    return self.__color
```

```
@color.setter  
def color(self, c):  
    self.__color = c
```

```
class Rectangle(Figure):
```

```
def __init__(self, width, height, color):  
    super().__init__(color)  
    self.__width = width  
    self.__height = height
```

```
@property  
def width(self):  
    return self.__width
```

```
@width.setter  
def width(self, w):  
    if w > 0:  
        self.__width = w  
    else:  
        raise ValueError
```

```
@property  
def height(self):  
    return self.__height
```

```
@height.setter  
def height(self, h):  
    if h > 0:  
        self.__height = h  
    else:  
        raise ValueError
```

```
def area(self):  
    return self.__width * self.__height
```

Родительским классом является Figure, который при инициализации принимает цвет фигуры и предоставляет его через свойства. Rectangle – класс наследник от Figure. Обратите внимание на его метод *init*: в нем первым делом вызывается конструктор (хотя это не совсем верно, но будем говорить так) его родительского класса:

```
super().__init__(color)_
```

super – это ключевое слово, которое используется для обращения к родительскому классу.

Теперь у объекта класса Rectangle помимо уже знакомых свойств width и height появилось свойство color:

```
>>> rect = Rectangle(10, 20, "green")  
  
>>> rect.width  
10  
  
>>> rect.height  
20  
  
>>> rect.color  
'green'  
  
>>> rect.color = "red"  
  
>>> rect.color  
'red'
```

Полиморфизм

Как уже было сказано во введении в рамках ООП полиморфизм, как правило, используется с позиции переопределения методов базового класса в классе наследнике. Проще всего это рассмотреть на примере. Добавим в наш базовый класс метод `info()`, который печатает сводную информацию по объекту класса `Figure` и переопределим этот метод в классе `Rectangle`, добавим в него дополнительные данные:

```
class Figure:

    def __init__(self, color):
        self.__color = color

    @property
    def color(self):
        return self.__color

    @color.setter
    def color(self, c):
        self.__color = c

    def info(self):
        print("Figure")
        print("Color: " + self.__color)

class Rectangle(Figure):

    def __init__(self, width, height, color):
        super().__init__(color)
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
```

```
    else:
        raise ValueError

@property
def height(self):
    return self.__height

@height.setter
def height(self, h):
    if h > 0:
        self.__height = h
    else:
        raise ValueError

def info(self):
    print("Rectangle")
    print("Color: " + self.color)
    print("Width: " + str(self.width))
    print("Height: " + str(self.height))
    print("Area: " + str(self.area()))

def area(self):
    return self.__width * self.__height
```

Посмотрим, как это работает

```
>>> fig = Figure("orange")

>>> fig.info()
Figure
Color: orange

>>> rect = Rectangle(10, 20, "green")

>>> rect.info()
Rectangle
Color: green
Width: 10
Height: 20
Area: 200
```


Таким образом, класс наследник может расширять функционал класса родителя.

Источники

1. [Python. Урок 14. Классы и объекты](#)
2. [Python ООП](#)
3. [Built-in Functions](#)
4. [Customizing attribute access](#)

Дополнительная литература

1. [Объектная модель Python. Классы, поля и методы](#)
2. [Волшебные методы, переопределение методов. Наследование](#)
3. [Настройка доступа к атрибутам класса Python](#)
4. [Python ООП](#)

Revision #16

Created 1 April 2023 02:20:23 by Антон Сергеевич Абраменко

Updated 6 May 2024 09:05:58 by Антон Сергеевич Абраменко