

Введение в объектно- ориентированное программирование (ООП)

ООП в Python

Python поддерживает ООП на сто процентов: все данные в нём являются объектами. Числа всех типов, строки, списки, словари, даже функции, модули, и наконец, сами типы данных — всё это объекты! Все вычисления в Python можно представить как взаимодействия между объектами.

ООП в Python

Объектно-ориентированное программирование (ООП)

ООП – это программирование с помощью классов и объектов.

Давайте сначала разберемся что такое объект. А потом плавно перейдем к такому понятию как класс.

Всё вокруг нас является
объектом.



У объекта есть
свойства
(еще называют параметры)



У объекта есть
методы
(методы – это действия,
то есть что может
делать объект)

Например, машина – это объект.



У любой машины есть такие
свойства: модель, цвет, размер и т.д.

Методы машины:
затормозить ();
нажатьНаГаз ();
открытьДверь ();
закрытьДверь ();
и т.д.

Например, котёнок – это объект.



У любого котенка есть свойства:
порода, имя, цвет, длина шерсти,
возраст и т.д.

Методы котенка:
спать();
кушать();
играть ();
шкодить ();
и т.д.

ООП в Python



Основные понятия

Класс

Описывает модель объекта, его свойства и поведение. Говоря языком программиста, класс — такой тип данных, который создается для описания сложных объектов.

Экземпляр

Для краткости вместо «Объект, порожденный классом „ДОМ“» говорят «экземпляр класса „ДОМ“».

Объект

Хранит конкретные значения свойств и информацию о принадлежности к классу. Может выполнять методы.

Атрибут

Свойство, присущее объекту. Класс объекта определяет, какие атрибуты есть у объекта. Конкретные значения атрибутов — характеристика уже не класса, а конкретного экземпляра этого класса, то есть объекта.

Метод

Действие, которое объект может выполнять над самим собой или другими объектами.

Примеры

Чтобы стало чуть понятнее, давайте разберем на примере: 1, 2, 3, "abc", [10, 20, 30] — объекты. А **int**, **str**, **list** — классы. Да, все типы данных, которые мы изучали ранее, на самом деле — классы:

1, 2, 3 — экземпляры класса **int**

"abc" — класса **str**

[10, 20, 30] — экземпляр класса **list**, в который вложены экземпляры **int**

Чтобы узнать, к какому классу относится тот или иной объект, можно воспользоваться функцией **type**.

Например,

```
>>> type(123)
```

```
# => '<class 'int'>'
```

```
>>> type([1, 2, 3])
```

```
# => '<class 'list'>'
```

Создание классов

Например, пусть программе дан список учащихся, который нужно отсортировать по экзаменационному баллу. Это можно сделать, если хранить в одном списке имена учащихся, в другом списке их баллы, и сортировать список баллов, синхронно переставляя элементы в списке имен учащихся.

Создание классов

Можно использовать кортеж, один из элементов которого будет равен имени учащегося, другой элемент — его баллу. Уже лучше. Кортеж — единое целое.

В этом случае придется поставить так:

```
Vasya = ('Василий', 4)
print('Имя:', Vasya[0])
print('Балл:', Vasya[1])
```

Но элементы кортежей не имеют собственных названий (то есть вовсе неочевидно, балл хранится именно в элементе кортежа с индексом 1). Удобней было бы обозначить атрибуты каждого объекта какими-то осмысленными идентификаторами и далее работать с этими идентификаторами.

Создание классов

```
class Person:  
    pass
```

Определение этого класса состоит из зарезервированного слова `class`, имени класса и пустой инструкции после отступа. Внутри класса с дополнительным уровнем отступов должны определяться его методы. Однако хотя бы одна инструкция должна быть, поэтому приходится использовать пустую инструкцию-заглушку `pass`.

```
Vasya = Person()  
Vasya.name = 'Василий'  
Vasya.score = 4
```

В этом примере мы объявляем новый **класс** объектов: **Person**. Затем мы создаем новый объект класса, при помощи вызова **конструктора** `Person()`. Далее объекту **Vasya** устанавливается два атрибута: **name** (типа **str**) и **score** (типа **int**).

PEP8

Имена классов по стандарту именования PEP-8 должны начинаться с большой буквы. Встроенные классы (**int**, **float**, **str**, **list** и другие) этому правилу не следуют, однако в вашем коде его лучше придерживаться. Так делает большинство программистов на Python.

Атрибуты

Атрибуты можно не только устанавливать, но и читать. При чтении ещё не созданного атрибута будет появляться ошибка **AttributeError**. Вы её часто увидите, допуская неточности в именах атрибутов и методов.

Атрибуты

```
print(Vasya.name, Vasya.score) # Василий 4  
print(Vasya.address)  
# Ошибка AttributeError, нет атрибута address
```

Атрибуты

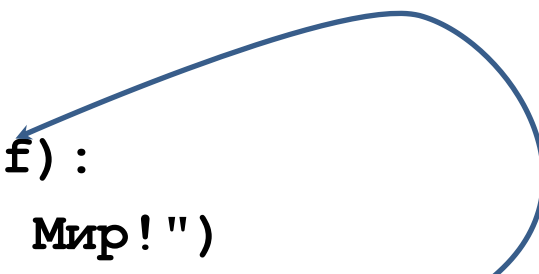
Обращение к атрибуту какого-либо объекта осуществляется при помощи **dot-нотации**: имя_класса.имя_атрибута.

Помимо полей у классов бывают методы: функции, которые можно применять к экземплярам класса. Например, у списков есть метод **sort**. Вызов метода также осуществляется при помощи dot-нотации, например: **A.sort()**.

Методы класса

Создаём метод класса

```
class Greeter:  
    def hello_world(self):  
        print("Привет, Мир!")
```



```
greet = Greeter()
```

```
greet.hello_world() # выведет "Привет, Мир!"
```



Методы класса

Создаём метод класса

Мы написали метод, с синтаксисом вызова которого вы хорошо знакомы по методу строк **split** или методу списков **append**.

При создании собственных методов обратите внимание на два момента:

- Метод должен быть определён внутри класса (добавляется уровень отступов);
- У методов всегда есть хотя бы один аргумент, и первый по счёту аргумент должен называться **self**.

Аргумент **self**

В него передается тот объект, который вызвал этот метод. Поэтому **self** ещё часто называют «контекстным объектом».

greet.hello_world() преобразуется в вызов
hello_world(greet)

Этот факт объясняет особую важность параметра **self** и то, почему он всегда должен быть первым в любом методе объекта, который вы пишете. Вызывая метод, вы не должны передавать значение для **self** явно — интерпретатор сделает это за вас

Пример: class Person

Можно рассматривать методы, как функции, у которых первым параметром является экземпляр класса. Методы так и объявляются: как функции внутри описания класса, первым параметром которой является экземпляр класса.

Вот пример объявления класса **Person** и метода **print**, выводящего информацию о полях **name** и **score**:

```
class Person:
    def print(self):
        print(self.name, self.score)
```

Пример: class Person

Теперь для вызова метода **print** для объекта **Vasya** нужно вызвать:
Vasya.print() .

При этом не нужно задавать первый параметр **self**: в качестве этого параметра автоматически будет передан объект, для которого был вызван метод.

Методы могут принимать дополнительные параметры, как и обычные функции. Эти параметры описываются после параметра **self**.

Специальные методы

Наш метод `print` предполагает, что у объекта есть поля `name` и `score`, иначе он завершится с ошибкой. Хочется быть уверенным, что у любого объекта класса `Person` есть эти поля. Для этого проще всего создать эти поля при создании объекта, т.е. при вызове функции `Person`. Для этого можно использовать **конструктор**: метод, который автоматически вызывается при создании объекта. Конструктором является метод с именем `__init__`:

```
class Person:
    def __init__(self):
        self.name = ''
        self.score = 0
```

При создании объекта функцией `Person` будет автоматически вызван конструктор `__init__` (явно вызывать его не нужно), который полю `name` объекта, для которого он вызван, присвоит пустую строку, а полю `score` присвоит значение 0.

Специальные методы

Конструктор может создавать объект, инициализируя поля объекта некоторыми параметрами, используя передаваемые ему значения, а не значения по умолчанию. Для этого конструктору можно передавать параметры:

```
class Person:
    def __init__(self, name, score):
        self.name = name
        self.score = score
```

В данном случае мы используем одинаковые имена (**name**, **score**) для обозначения передаваемых параметров и полей класса. Это сделано для удобства — имена могут и различаться.

Теперь мы сможем создавать новый объект с заданными полями так:

```
Person('Иванов', 5)
```

Специальные методы

Но поскольку конструктор теперь обязательно принимает два дополнительных параметра мы лишились возможности вызывать конструктор без параметров, что также бывает удобно. Можно вернуть эту особенность, если установить для параметров, передаваемых конструктору, значения по умолчанию:

```
class Person:
    def __init__(self, name='', score=0):
        self.name = name
        self.score = score
```

Теперь мы можем вызывать конструктор как с параметрами (**Person('Иванов', 5)**), так и без параметров (**Person()**), в последнем случае параметрам будут переданы значения “по умолчанию”, указанные в описании конструктора.

Специальные метод `__repr__`

Метод `__repr__` должен возвращать текстовую строку, содержащую код (на языке Python), создающий объект, равный данному. Естественно, метод `__repr__` должен содержать вызов конструктора, которому передаются в качестве параметров все строки исходного объекта, то есть он должен возвращать строку вида

```
"Person('Иванов', 5)"
```

Пример метода `__repr__` (для экономии места опустим описание конструктора `__init__`):

```
class Person:  
    def __repr__(self):  
        return "Person('" + self.name + "', " + self.score + ")"
```

Таким образом, метод `__repr__` возвращает строку с описанием объекта, которое может быть воспринято интерпретатором языка Питон.

Специальный метод `__str__`

Метод `__str__` возвращает строку, являющуюся описанием объекта в том виде, в котором его удобно будет воспринимать человеку. Здесь не нужно выводить имя конструктора, можно, например, просто вернуть строку с содержимым всех полей:

```
class Person:
```

```
    def __str__(self):  
        return self.name + ' ' + str(self.score)
```

Метод `__str__` будет вызываться, когда вызывается функция `str` от данного объекта, например, `str(Vasya)`.

Т.е. создавая метод `__str__` вы даете указание Питону, как преобразовывать данный объект к типу `str`.

Поскольку функция `print` использует именно функцию `str` для вывода объекта на экран, то определение метода `__str__` позволит выводить объекты на экран удобным способом: при помощи `print`.

Специальные методы

Есть и другие Специальные методы, которые можно определить в описании класса.

Специальные методы имеют для интерпретатора особое значение. Имена специальных методов и их смысл определены создателями языка: создавать новые нельзя, можно только реализовывать существующие. Названия всех специальных методов начинаются и заканчиваются на два подчёркивания.

Пример такого метода—уже знакомый нам `__init__`. Он предназначен для инициализации экземпляров и автоматически вызывается интерпретатором после создания экземпляра объекта.

Специальные методы

Остальные специальные методы также вызываются в строго определённых ситуациях. Так , например, всякий раз, когда интерпретатор встречает запись вида:

`x + y,`

он заменяет её на

`x.__add__(y) ,`

и для реализации сложения нам достаточно определить в классе экземпляра `x` метод `__add__`.

Специальный метод `__add__`

Рассмотрим класс Point (точка),

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

Если рассматривать точку как радиус-вектор, то хотелось бы определить для точек удобную операцию +. Например, чтобы можно было записать так:

```
A = Point(1, 2)
B = Point(3, 4)
C = A + B
```

Для этого необходимо **переопределить** функцию. Для этого нужно определить метод `__add__` класса Point, у которого два параметра: неявная ссылка `self` на экземпляр класса, для которого она будет вызвана и явная ссылка `other` на правый операнд:

```
def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)
```

Функции `type` и `isinstance`

В задачах вам надо будет проверять принадлежность объекта какому-либо классу.

Функция `type` возвращает класс, к которому принадлежит объект.

Пример:

```
if type(a) == int:
    print('a - целое число')
elif type(a) == str:
    print('a - строка')
```

Функция `isinstance`, у которой два параметра: объект и класс. Возвращает `True`, если объект принадлежит классу или `False` в противном случае.

Пример:

```
if isinstance(a, int):
    print('a - целое число')
elif isinstance(a, str):
    print('a - строка')
```

Стиль Имя́н Классов

на примере класса робота-почтальона

СтильИмя́нКлассов

стиль_имён_методов_и_атрибутов

```
class RoboticMailDelivery:
    def __init__(self):
        self.house_flat_pairs = []

    def add_mail(self, house_number, flat_number):
        '''Добавить информацию о доставке письма по номеру дома
        house_number, квартира flat_number.'''
        self.house_flat_pairs.append((house_number, flat_number))

    def flat_numbers_for_house(self, house_number):
        '''Вернуть список квартир в доме house_number,
        в которые нужно доставить письма.'''
        flat_numbers = []
        for h, f in self.house_flat_pairs:
            if h == house_number:
                flat_numbers.append(f)
        return flat_numbers
```

вызов метода
объекта-атрибута

Стиль Имя́н Классов PEP 8

на примере класса робота-почтальона

- Имя класса должно начинаться с большой буквы, между словами не должно быть прочерка, каждое слово внутри имени должно начинаться с большой буквы.

RoboticMailDelivery

- Для имен атрибутов и методов применяются те же правила, что и для имен переменных и функций. Имя должно быть записано в нижнем регистре, слова внутри имени разделяются подчеркиванием:

flat_numbers_for_house, house_flat_pairs.

- **Важно!** Документация с описанием методов записывается в `"""многострочных строках"""` перед первой инструкцией как в функциях, так и в методах.