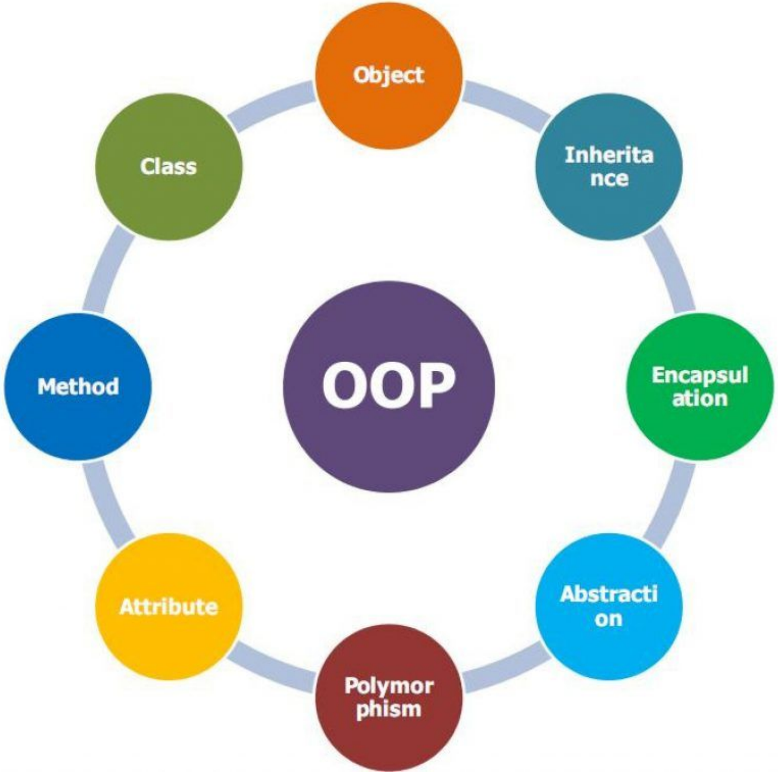
Object Oriented Programming



What is this?

Another programming paradigm, which is very popular and useful in big projects

As comes from the name it is based on objects and classes, so let's study them

Classes

Class is a new datatype which you have programmed. It is very convenient when you need some entity to describe something.

Also classes are a way to further modularize your program

operator programming < functions < classes

Objects

Object is an instance of classes. You can treat class as unmaterialized concept, while object is a concrete realization of this class

Objects have all entity properties defined in the class

Example

Class

Building

Objects

Home

Church

Skyscraper

Almost everything in python is an object...

Easier to say what is not an object - keywords and comments

All other stuff belongs to objects

- data structures
- functions
- modules

Class example

```
# List is a class
print(list)
<class 'list'>

# [1, 2, 3] is an object of list class
print(type([1, 2, 3]))
<class 'list'>
```

Entity

How this essentiality is achieved?

Object can be thought as consisting of 2 parts

- data - described in object attributes
- functionality - described in object methods

Attributes vs Methods

All of them are accessed through `.` after object or class name, methods should be invoked as functions

```
import numpy as np


xs = np.arange(5)

# Attribute is accessed
print(xs.size)
5
# Method
print(xs.mean())
2.0
```

Methods vs Functions

Methods are functions associated with objects. It means that method is specific for class

Functions are realized as a universal feature which can be applied to different classes, and it is realized through the methods too (later about it)

```
xs.mean()

print(xs)
```

Class creation

```
class Dragon:
    pass
```

```
x = Dragon()
print(x)
<__main__.Dragon object at 0x7fea84e7df28>
```

Class morphology

- `class` - keyword to tell python that you are going to define a class
- `Dragon` - name of the class, in UpperCamelCase by convention
- `pass` - just to skip content, here will go class content

To create an object of this class, we just invoke class as function

Class variables

```python
class Dragon:
    animal_type = 5
    can_fly = True



x = Dragon()

print(x.animal_type)
5
print(x.can_fly)
True
# Class variables can be accessed via class, without object
print(Dragon.animal_type)
5
```

Constructor

```
class Dragon:
  animal_type = 5

  def __init__(self, age, name):
      self.age = age
      self.name = name
      self.length = 2
```

`Dragon()` is a constructor - it is used to construct objects of class Dragon

```
dragon_vasya = Dragon(5, 'Vasya')
dragon_izera = Dragon(4, 'Izera')

print(dragon_vasya.age, dragon_izera.age)
5 4
print(dragon_vasya.length, dragon_izera.length)
2 2

print(dragon_izera.name)
Izera
print(dragon_izera.animal_type == dragon_vasya.animal_type)
True
```

Methods

```
class Dragon:
  animal_type = 5

  def __init__(self, age, name):
      self.age = age
      self.name = name
      self.length = 2

  def grow(self):
      """
      Become stronger
      :return:
      """
      self.length += 1
```

```
print(f'Age of {dragon_vasya.name} is {dragon_vasya.age}',
 f'Age of {dragon_izera.name} is {dragon_izera.age}', sep=', ')
Age of Vasya is 5, Age of Izera is 4

print(f'Length of {dragon_vasya.name} is
{dragon_vasya.length}', f'Length of {dragon_izera.name} is
{dragon_izera.length}', sep=', ')
Length of Vasya is 2, Length of Izera is 2

dragon_izera.grow()
```

```
print(f'Age of {dragon_vasya.name} is {dragon_vasya.age}',
 f'Age of {dragon_izera.name} is {dragon_izera.age}', sep=', ')
Age of Vasya is 5, Age of Izera is 5

print(f'Length of {dragon_vasya.name} is
{dragon_vasya.length}', f'Length of {dragon_izera.length} is
 {dragon_izera.age}', sep=', ')
Length of Vasya is 2, Length of 3 is 5
```

self

`self` is a conventional name to denote instance of class (object)

In a class we don't have its objects yet, thus we denote them with name `self`

```python
def grow(self):
```

It means, that class method grow takes exactly one argument which is the instance of this class. It leads us to the next slide

Method invocation

There are 2 ways to invoke method

- `Dragon.grow(dragon_izera)` - self is explicitly passed
- `dragon_izera.grow()` - self is obtained from the object who invokes method

Special methods

All methods surrounded with __ are special, also they are called dunders (double underscore)

They are used to define class behaviour in python

Some of these methods

- __str__ - string representation of the object
- __eq__ - how object equality is tested
- __bool__ - how boolean representation is infered

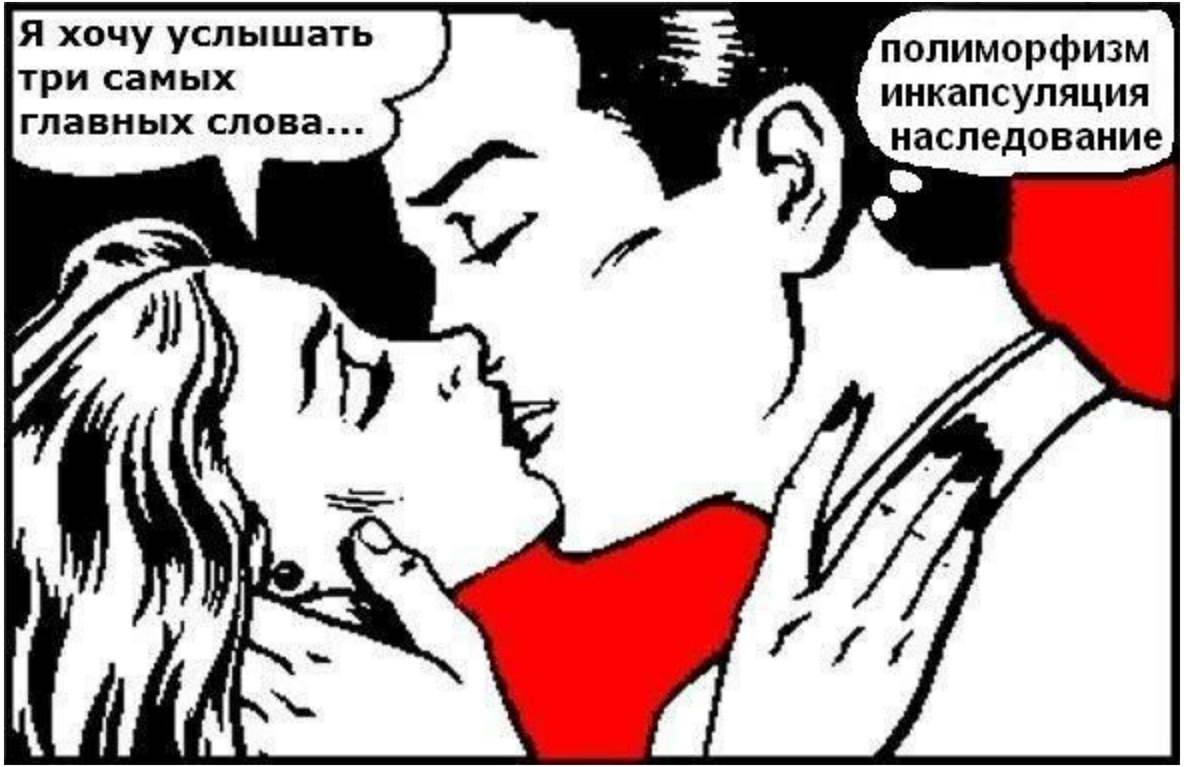
```
class Dragon:
    animal_type = 5
    def __init__(self, age, name):
        self.age = age
        self.name = name
        self.length = 2

    def __str__(self):
        age = f'Age {self.age}'
        length = f'Length {self.length}'

        return '\n'.join((self.name, age, length))

dragon_izera = Dragon(4, 'Izera')
print(dragon_izera)
Izera
Age 4
Length 2
```

Contract

All these methods form a so-called contract - expectations how you class will behave, which result can be returned

It is important to keep contract when you have an application or library

Class features


Я хочу услышать три самых главных слова...
полиморфизм инкапсуляция наследование

Encapsulation

It is hiding information in the object - other programmers will not have access to hided (private) attributes

Why is it used?

To obliviate load on your class users - accessing 0-9 attributes is easier than 9-∞

Python doesn't have robust encapsulation - you can always access private field if you know how

```
class Dragon:
    public_animal_type = 5
    __private_dragon_info = ''

    def __init__(self, age, name):
        self.age = age
        self.name = name
        self.length = 2

    def __change_private_dragon_info(self, info):
        self.__private_dragon_info = info

dragon_izera = Dragon(4, 'Izera')
dragon_izera.__private_dragon_info
AttributeError: 'Dragon' object has no attribute
'__private_dragon_info'
```

Abstraction

Implementation details of your class functioning are hidden (abstracted) from your users

In other words you don't have to know how the class do everything, you can just use it

```
class Dragon:
   public_animal_type = 5
   __private_dragon_info = ''

   def __init__(self, age, name):
       self.age = age
       self.name = name
       self.length = 2

   def do_science(self):
       # hard stuff
       return result

dragon_izera = Dragon(4, 'Izera')
new_results = dragon_izera.do_science()
```

Inheritance

Common ancestor with Amphibia - Common ancestor with Reptiles - Mammals

In this scheme Common ancestor with Amphibia is the ancestor of the following taxa

In python we have quite the same situation

```python
class PreDragon:
   has_scale = True
   can_fly = False

   def __init__(self, age, name):
       self.age = age
       self.name = name


class Dragon(PreDragon):
   public_animal_type = 5
   can_fly = True

   def __init__(self, age, name):
       super().__init__(age, name)
       self.length = 2
```

```
dragon_izera = Dragon(4, 'Izera')

print(dragon_izera.age, dragon_izera.length)
4 2

print(dragon_izera.has_scale, dragon_izera.can_fly)
True True
```

Redefining attribute or method in the child class is called overload