FS12

Week 03
Object-Oriented Programming

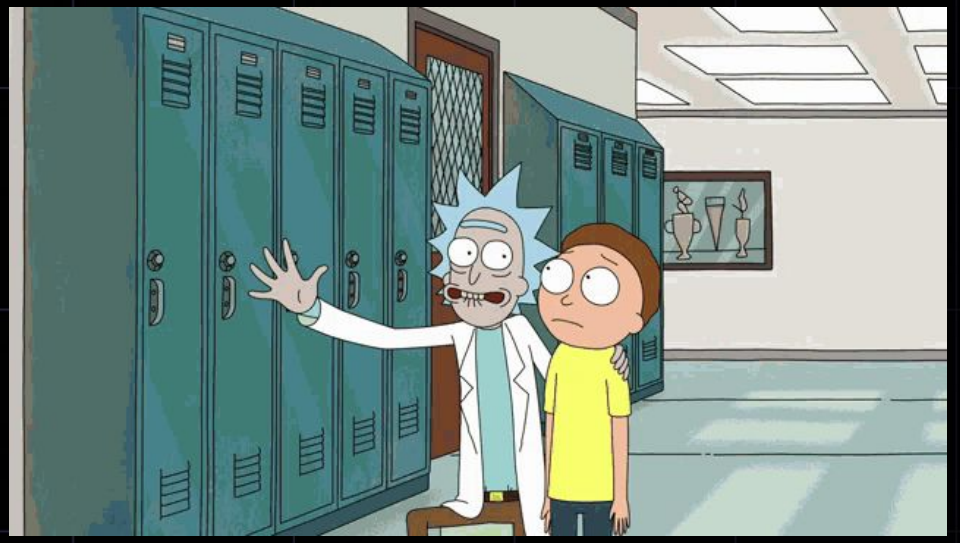
Mikhail Masyagin
Daniil Devyatkin

Recap: Previous Lecture

- Containers: mapping & sequence and it realization in Python;
- I / O & files;
- Programming paradigms: **Imperative Programming** paradigm, **Procedural Programming** paradigm, **Functional Programming** paradigm and sometimes in **Object-Oriented Programming** paradigm;
- Type-hints.

Lesson plan

- Intro to Object-Oriented Programming (OOP);
- Classes: encapsulation, abstraction, inheritance, polymorphism, class attributes and methods;
- Specifics Python OOP properties;
- Magic attributes and methods, MRO;
- Practice example.



Object-Oriented Programming I

- **Object-Oriented Programming** is a programming paradigm based on the concept of **objects**, which can contain self properties (fields).
- The data is in the form of fields (often known as attributes), and the code is in the form of procedures (often known as methods).
- For us as Python developers **Object-Oriented Programming** is about writing code for new objects (classes) and their usage.

Object-Oriented Programming II

- For OOP we use **class** and **object** definition;
- In simple word, **class** is a description of what properties and behavior an object will have.
- **Object** is a sample of class with self properties^ attributes and methods;
- You can imagine cooking: forms for cookies is a class, cookies is a object;
- Let’s go create simple class and object.

Object-Oriented Programming III

- Create dummy-class and object:
- Syntax (Class name begin uppercase):

```
class <name>[()]:
    <some_logic>
class Transport:
    pass
t = Transport()
type(t)
> <class '__main__.Transport'>
```

Object-Oriented Programming IV

- Attributes - what is it?
- We can understand **attributes** as properties (features) of object. For example: for people's features - height, weight, eyes color, gender, etc;
- There are 5 types of attributes: **class attributes** (private, public), **object attributes** (private, protected, public);
- Let's go add attributes to our class;

Object-Oriented Programming IV

```
class Transport:
    cls_pub = "cls_public"
    __cls_prv = "cls_private"
    def __init__(self, i):
        self.pub = f"obj_public_{i}"
        self._pt = f"obj_protected_{i}"
        self.__prv = f"obj_private_{i}"
t = Transport(1)
  ...
```

Object-Oriented Programming IV

```
t1 = Transport(1)
t1.cls_pub > "cls_public"
t1.__cls_prv > AttributeError 'Transport' object has no attribute '_cls_prv'
t1.pub > "obj_public_1"
t1._pt > "obj_protected_1"
t1.__prv)> AttributeError 'Transport' object has no attribute '__prv'
```

Object-Oriented Programming IV

```
t2 = Transport(2)

print(t2.cls_pub) > ..............

print(t2.__cls_prv)

> ..............

print(t2.pub) > ..............

print(t2._pt) > ..............

print(t2.__prv)

> ..............
```

Object-Oriented Programming IV

```
t2 = Transport(2)
t2.cls_pub > "cls_public"
t2.__cls_prv > AttributeError 'Transport' object has no attribute '_cls_prv'
t2.pub > "obj_public_2"
t2._pt > "obj_protected_2"
t2.__prv > AttributeError 'Transport' object has no attribute '_prv'
```

Object-Oriented Programming IV

```
t2.cls_pub += "_change"
t2.pub += "_change"
t1.cls_pub >
t1.pub >
```

Object-Oriented Programming IV

```
t2.cls_pub += "_change"
t2.pub += "_change"
t1.cls_pub > "cls_public_change"
t1.pub > "obj_public_1"
```

Object-Oriented Programming IV

SOME DIRTY EVENT

PRACTICE

ACCESSING TO PRIVATE AND PROTECTED ATTRIBUTE

Object-Oriented Programming V

- Some information about accessing attributes;
- There are special attributes, which named **magic attributes**. It's special attributes for description our class / object for fully understanding business logic and applying;
- **Magic attribute** is a attribute with double underscore in begin and end of attribute name;
- Let's go to see embedded magic attributes

Object-Oriented Programming V

- Magic attributes:

  __name__ – class name;

  __doc__ – document string;

  __dict__ – class namespace;

  __module__ – name of the module where the class is defined;

  and more…

- Let’s go check it in the practice!

Object-Oriented Programming V

```
class Transport:
    """ My doc """
        def __init__(self): self.pub = "p"
Transport.__name__ > "Transport"
Transport.__doc__ > "My doc"
Transport.__dict__ >
```

{'__module__': '__main__', '__doc__': 'My doc', '__init__': <function __main__.Transport.__init__(self)>, '__dict__': <attribute '__dict__' of 'Transport' objects>, '__weakref__': <attribute '__weakref__' of 'Transport' objects>}

Object-Oriented Programming VI

```
class Transport:
    """ My doc """
        def __init__(self):
            self.pub = "pub"
            self._prt = "prt"
            self.__prv = "prv"
t = Transport()
t.__dict__ > {"pub":"pub", "_prt": "prt",
"_Transport__prv": "prv"}
```

Object-Oriented Programming VII

- How to search for attributes in Python?
- For search object's attribute, Python search:
  - obj.__dict__ (object)
  - obj.__class__.dict__ (object's class)
  - obj.__class__.__mro__ (class parents if we have a inheritance)

Object-Oriented Programming VIII

- Class method is an analogue a function, but you can call it only from your object (class instance) and methods hide in class;
- You can think about function like your object's action: for people's action – running, swimming, drinking (beer), driving and more;
- Anothers words, function is contained in class - method;

Object-Oriented Programming VIII

- There are 3 types of methods : **public**, **private** and **protected**;
- Also methods are separated to 3 types: c**lass methods**, **static methods** and **object methods**;
- Let’s go see!

Object-Oriented Programming VIII

```
class Transport:
      def __init__(self, v):
            self.v = v
            print(f"ctor: {v}")
      def func(self): print(f"obj method v={self.v}")
      @staticmethod
      def gang(): print("static method")
      @classmethod
      def bang(cls): print(f"cls_name: {cls.__name__}")
      def lol(self): print("public")
      def _lol(self): print("protected")
      def __lol(self): print("private")
```

Object-Oriented Programming VIII

```
t = Transport(10) > “ctor: 10”
t.func() > “obj method v=10”
t.gang() > “static method”
t.bang() > “cls_name: Transport”
t.lol() > “public”
t._lol() > “protected”
t.__lol() > AttributeError: 'Transport' object
has no attribute '__lol'
t._Transport__lol() > “private”
t.__init__(v = 1) > “ctor: 1”
```

Object-Oriented Programming IX

- There are a **magic methods** (also like magic attributes);
- **Magic method**s is specially reserved name, which begin and end with 2 underscore and you can define their behavior;
- Example:

  __init__, __del__, __len__, __new__, __getattr__, __str__, __repr__, etc;

Object-Oriented Programming VIII

```
class Transport:
      def __init__(self, hp, length):
            self.hp = hp
            self.l = length
      def __len__(self):
            return self.l
      def __str__(self):
            return f"HP is {self.hp}, len is {self.l}"
      def ride(self):
            print("Brbrbrbrb")
```

Object-Oriented Programming VIII

```
t = Transport(hp = 249, length = 4.5)
t.__len__() / len(t) > 4.5
t.__str__() / str(t) > "HP is 249, len is 4.5"
t.ride() > "Brbrbrbrb"
```

Object-Oriented Programming IX

- Magic methods:

  object.__lt__(self, other)

  object.__le__(self, other)

  object.__eq__(self, other)

  object.__ne__(self, other)

  object.__gt__(self, other)

  object.__ge__(self, other)

  x < y == x.__lt__(y) # <=, ==, !=, >, >=

Object-Oriented Programming IX

- Magic methods:

  object.__add__(self, other)

  object.__mul__(self, other)

  object.__mod__(self, other)

  object.__pow__(self, other[, modulo])

  object.__and__(self, other)

  object.__or__(self, other)

  x + y == x.__add__(y) # -, *, /, **, and, or …

Object-Oriented Programming X

Object-Oriented Programming X

- **Inheritance** is one of the OOP paradigms, which allows you to transfer properties of one class to another. This principle allows you to use fewer lines of code.
- There are base (super) class - **parent**, and derived class - **child**.
- Let’s go see code!

Object-Oriented Programming X

```
class Transport:
    def __init__(self, hp, length):
        self.hp = hp
        self.l = length
    def __len__(self):
        return self.l
    def __str__(self):
        return f"HP is {self.hp}, len is {self.l}"
    def ride(self):
        print("Brbrbrbrb")
    def get_n_doors(self):
        return 4
```

Object-Oriented Programming X

```
class Bus(Transport):
    def __init__(self, hp, length):
        super(Bus, self).__init__()
        self.hp = hp
        self.l = length
    def __len__(self):
        return 100
    def ride(self):
        print("PSH-PSH")
    def get_n_doors(self):
        return 6
```

Object-Oriented Programming X

```
b = Bus(hp = 100, length = 10)
t = Transport(hp = 249, length = 4)
b.__len__() / len(t) >
b.__str__() / str(t) >
b.ride() >
t.__len__() / len(t) >
t.__str__() / str(t) >
t.ride() >
```

Object-Oriented Programming X

```
b = Bus(hp = 100, length = 10)
t = Transport(hp = 249, length = 4)
b.__len__() / len(t) > 100
b.__str__() / str(t) > "HP is 100, len is 10"
b.ride() > "PSH-PSH"
t.__len__() / len(t) > 4.5
t.__str__() / str(t) > "HP is 249, len is 4.5"
t.ride() > "Brbrbrbrb"
```

Object-Oriented Programming X

- All in Python is object -> all Python objects have __dict__ attribute -> we can get access to all data!
- We can redefine magic method’s behavior! It’s no safe.
- All types in Python inherit from object;

Object-Oriented Programming XI

- **Encapsulation** is the principle according to which the internal structure of entities must be combined in a special “shell” and hidden from outside interference (private attributes and methods). Objects can be accessed through special public methods, but their contents cannot be accessed directly;

Object-Oriented Programming XI

Object-Oriented Programming XI

- Encapsulation can be understood as access to attributes and methods of a class (object) - private, protected and public;
- Setter and getter approach to encapsulation.

Object-Oriented Programming XI

- Encapsulation in Python:

```
class Author:
    def __init__(self, name):
            self.__name = name
    def get_name(self):
            return self.__name
    def set_name(self, val):
            self.__name = val
    def del_name(self):
            del self.__name
```

Object-Oriented Programming XI

- Encapsulation in Python:

```
class Author:
    def __init__(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
    def set_name(self, val):
        self.__name = val
    def del_name(self):
        del self.__name
```

```
class Author:
    def __init__(self, name):
        self.name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, val):
        self.__name = val
    @name.deleter
    def name(self, val):
        self.__name = val
```

Object-Oriented Programming XI

- Read / write only properties:

```
class Author:
    def __init__(self, name, password):
        self.__name = name
        self.password = password
        self.password_hash = None
    @property
    def name(self):
        """ name is read-only """
        return self.__name
    @property
    def password(self, val): raise AttributeError("Password is write-only")
    @password.setter
    def password(self, plaintext):
        self.password_hash = make_hash_from_password(plaintext)
```

Object-Oriented Programming XI

- Read / write only properties:

```
a = Author(name = “Alex”, password = “qwert1234”)
a.name > “Alex”
a.name = “Vasya” > AttributeError: can’t set attribute
‘name’
a.password > AttributeError: ‘Password is write-only’
a.password = “my_new_pass” > *password is change*
```

Object-Oriented Programming XIII

- **Polymorphism** is different behavior of the same method in different situations (different arguments). For example, we can add two numbers, and we can add two strings. But the result of the addition will be different.

```
1 + 1 > 2
"1" + "1" > "11"
```

- Remind about **magic methods**!!!

Object-Oriented Programming XIV

- **Abstraction** is the process of highlighting the general characteristics and functionality of objects or a system, ignoring implementation details.
- Abstraction allows you to develop programs in different programming languages while hiding the complexity and details of the underlying code.
- We don't care about implementation, only functionality matters!

Object-Oriented Programming XIV

```
from abc import ABC, abstractmethod
from math import pi

class User(ABC):
    @abstractmethod
    def page(self):
        pass

    @abstractmethod
    def message(self):
        pass

    @abstractmethod
    def post(self):
        pass
```

```
class VKUser(User):

    def page(self):
        return "I have a page!"

    def message(self):
        return "I sent a message!"

    def post(self):
        return "I shared about my
achievements"
```

```
user = User() > TypeError: Can't instantiate abstract class User with abstract methods message,
page, post

vk_user = VKUser()
vk_user.page() > "I have a page!"
```

Object-Oriented Programming XV

- Design pattern in OOP - an approach to **“high-quality”** and **“correct”** class design. What is **“quality”** and **“correct”**?
- In general, these terms mean the reuse of classes in OOP terms and the addition of new functionality;
- Let’s go see S.O.L.I.D. pattern.

Object-Oriented Programming XV

- S – Single Responsibility. Each class should be responsible for only one operation; (Принцип одиночной ответственности);

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def save_in_db(self):
        pass

    def send_email(self, message):
        pass

    def generate_report(self):
        pass
```

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def save_in_db(self):
        pass

class EmailSender:
    def send_email(self, user, message):
        pass

class ReportGenerator:
    def generate_report(self, user):
        pass
```

Object-Oriented Programming XV

- O – Open-Closed. Classes should be open for extension, but closed for modification; (Принцип открытости-закрытости);

```
from abc import ABC, abstractmethod
from math import pi

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius ** 2

    def info():
        return f"I am a circle with R =
{self.radius}"
```

Object-Oriented Programming XV

- L – Liskov Substitution. If P is a subtype of T, then any objects of type T present in the program can be replaced by objects of type P without negative consequences for the functionality of the program;

```
class A:
    def __init__(): pass
    def func_a1(): pass
    def func_a2(): pass
    def func_a3(): pass

class B(A):
    def __init__(): pass
    def func_b1(): pass
```

```
a_obj = A()
b_obj = B()

...

a.func_a1() # we can replace b.func_a1()

...

a.func_a3() # we can replace b.func_a1()
```

Object-Oriented Programming XV

- I – Interface Segregation. It states that there is no need to create huge classes with many methods. You should create many small classes with fewer methods. (Принцип разделения интерфейсов);

```python
from abc import ABC, abstractmethod
from math import pi

class InputDevice(ABC):
    @abstractmethod
    def read_input(self):
        pass


class OutputDevice(ABC):
    @abstractmethod
    def write_output(self, data):
        pass
```

```python
class Keyboard(InputDevice):
    def read_input(self):
        pass

class Mouse(InputDevice):
    def read_input(self):
        pass

class Monitor(OutputDevice):
    def write_output(self, data):
        pass

class Printer(OutputDevice):
    def write_output(self, data):
        pass
```

Object-Oriented Programming XV

- D — Dependency Inversion. This is a principle that suggests that classes should not directly rely on other classes, but instead depend on abstractions. (Принцип инверсии зависимостей)

```python
from abc import ABC, abstractmethod
from math import pi

class User(ABC):
    @abstractmethod
    def page(self):
        pass

    @abstractmethod
    def message(self):
        pass

    @abstractmethod
    def post(self):
        pass
```

```python
class VKUser(User):

    def page(self):
        return "I have a page!"

    def message(self):
        return "I sent a message!"

    def post(self):
        return "I shared about my achievements"
```

Directed by
ROBERT B. WEIDE