Object-oriented programming (OOP) is a programming paradigm based on the concept of **objects** that interact with each other to perform program functions. Each object can be characterized by a state and behavior. An object's current state is represented by its **fields**, and an object's behavior is represented by its **methods**.

Basic principles of OOP

There are four basic principles of OOP. They are **encapsulation, abstraction, inheritance,** and **polymorphism.**

- **Encapsulation** ensures bundling (=encapsulating) of data and the methods operating on that data into a single unit. It also refers to the ability of an object to hide the internal structure of its properties and methods.
- **Data abstraction** means that objects should provide a simplified, abstract version of their implementations. The details of their internal work usually aren't necessary for the user, so there's no need to represent them. Abstraction also means that only the most relevant features of the object will be presented.
- **Inheritance** is a mechanism for defining parent-child relationships between classes. Often objects are similar, so inheritance allows programmers to reuse common logic and introduce unique concepts into the classes.
- **Polymorphism** literally means "having many forms" and is a concept related to inheritance. It allows programmers to define different implementations for the same method. Thus, the name (or interface) remains the same, but the actions performed may differ. For example, imagine a website that posts three main types of text: news, announcements, and articles. They are somewhat similar in that they all have a headline, some text, and a date. In other ways, they are different: articles have authors, news bulletins have sources, and announcements have a date after which they become irrelevant. It is convenient to write an abstract class with general information for all publications to avoid copying it every time and store what is different in the appropriate derived classes.

These are the key concepts of OOP. Each object-oriented language implements these principles in its own way, but the essence stays the same from language to language.

Objects

The key notion of OOP is, naturally, an **object.** There are a lot of real-world objects around you: pets, buildings, cars, computers, planes, you name it. Even a computer program may be considered as an object.

It's possible to identify some important characteristics of real-world objects. For instance, for a building, we can consider the number of floors, the year of construction, and the total area. Another example is a plane, which can accommodate a certain number of passengers

and transfer you from one city to another. These characteristics constitute the object's **attributes** and **methods**. Attributes characterize the states or data of an object, and methods characterize its behavior.

Classes

Often, many individual objects have similar characteristics. We can say these objects belong to the same **type** or **class**.

A class is another important notion of OOP. A class describes a common structure of similar objects: their fields and methods. It may be considered a template or a blueprint for similar objects. An object is an individual **instance** of a class.

In accordance with the principle of encapsulation mentioned above, any class should be considered as a black box, that is, the user of the class should see and use only the interface part of the class, namely, the list of declared properties and methods, and should not delve into the internal implementation.

Let's look at some examples below.

Example 1. The building class

An abstract building for describing buildings as a type of object (class)

Each building has the same attributes:

- Number of floors (an integer number);
- Area (a floating-point number, square meters);
- Year of construction (an integer number).

Each object of the building type has the same attributes but different values.

For instance:

- Building 1: number of floors = 4, area = 2400.16, year of construction = 1966;

- Building 2: number of floors = 6, area = 3200.54, year of construction = 2001.

It's difficult to determine a building's behavior, but this example demonstrates attributes well.

Example 2. The plane class

Unlike with a building, it is easy to define the behavior of a plane: it can fly and transfer you between two points on the planet.

An abstract plane for describing all planes as a type of object (class)

Each plane has the following attributes:

- Name (a string, for example, "Airbus A320" or "Boeing 777");
- Passenger capacity (an integer number);
- Standard speed (an integer number);
- Current coordinates (they are needed to navigate).

Also, it has a behavior (a method): transferring passengers from one geographical point to another. This behavior changes the state of a plane, namely, its current coordinates.

Objects and classes

In OOP, everything can be viewed as an object; a class, for example, is also an object. Programs are made up of different objects that interact with each other. Object state and behavior are usually combined, but this is not always the case. Sometimes we see objects without a state or methods. This depends on the program's purpose and the object's nature.

For example, there is such a thing as an interface. Not a user interface, but a class that only serves to be inherited from in order to guarantee an interface to its descendant classes. It is a stateless class. Structures exist in C++ for historical reasons. Now a structure in C++ is also a class, but once upon a time, a structure had only properties and did not have any

methods – a type for storing data and nothing else. These are special cases, and they are sometimes useful.

Conclusion

Object-oriented programming is akin to the work of an architect. As an architect, you create templates called **classes** and build individual instances of these classes, known as **objects**. Like the inhabitants of a building, users of a class interact solely with its **fields** and **methods**, without needing to delve into the internal implementation details. This principle is commonly known as **encapsulation**.

As you already know, **object-oriented programming (OOP)** is a programming paradigm that is based on the concept of objects. Objects interact with each other and that is how the program functions. Objects have states and behaviors. Many objects can have similar characteristics and if you need to work with them in your program it may be easier to create a **class** for similar objects. Classes represent the common structure of similar objects: their data and their behavior.

Declaring classes

Classes are declared with the keyword class and the name of the class:

```
# class syntax
class MyClass:
    var = ...  # some variable

    def do_smt(self):
    # some method
```

Generally, a class name starts with a capital letter and it is usually a noun or a noun phrase. The names of the classes follow the **CapWords** convention: meaning that if it's a phrase, all words in the phrase are capitalized and written without underscores between them.

```
# good class name
class MyClass:
    ...

# not so good class name:
class My_class:
    ...
```

Classes allow you to define the data and the behaviors of similar objects. The behavior is described by **methods.** A method is similar to a function in that it is a block of code that has

a name and performs a certain action. Methods, however, are not independent since they are defined within a class. Data within classes are stored in the **attributes** (variables) and there are two kinds of them: **class attributes** and **instance attributes**. The difference between those two will be explained below.

Class attribute

A class attribute is an attribute shared by all instances of the class. Let's consider the class **Book** as an example:

```
# Book class
class Book:
  material = "paper"
  cover = "paperback"
  all_books = []
```

This class has a string variable material with the value "paper", a string variable cover with the value "paperback" and an empty list as an attribute all_books. All those variables are class attributes and they can be accessed using the **dot notation** with the name of the class:

```
Book.material  # "paper"
Book.cover  # "paperback"
Book.all_books  # []
```

Class attributes are defined within the class but outside of any methods. Their value is the same for all instances of that class so you could consider them as the sort of "default" values for all objects.

As for the **instance variables,** they store the data unique to each object of the class. They are defined within the class methods, notably, within the __init__ method. In this topic, we'll deal with the class attributes, but don't worry – you'll have plenty of time to learn more about instance attributes.

Class instance

Now, let's create an instance of a **Book** class. For that we would need to execute this code:

```
# Book instance
my_book = Book()
```

Here we not only created an instance of a **Book** class but also assigned it to the variable **my_book**. The syntax of instantiating a class object resembles a function call: after the class name, we write parentheses.

Our my_book object has access to the contents of the class **Book**: its attributes and methods.

```
print(my_book.material)  # "paper"
print(my_book.cover)  # "paperback"
print(my_book.all_books)  # []
```

Conclusion

Well, those were the very basics of classes in Python. Classes represent the common structure of similar objects, their attributes, and their methods. There are class attributes and instance attributes. Class attributes are common to all instances of the class.

All in all, classes are an extremely useful tool that can help you optimize your code and organize the program in a logical and readable way

Class instance

By now, you already know what classes are and how they're created and used in Python. Now let's get into the details about **class instances.**

A class instance is an object of the class. If, for example, there was a class **River,** we could create such instances as Volga, Seine, and Nile. They would all have the same structure and share all class attributes defined within the class River.

However, initially, all instances of the class would be identical to one another. Most of the time that is not what we want. To customize the initial state of an instance, the **__init__** method is used.

def __init__()

The __init__ method is a **constructor**. Constructors are a concept from the object-oriented programming. A class can have one and only one constructor. If __init__ is defined within a class, it is automatically invoked when we create a new class instance. Take our class River as an example:

```
class River:
  # list of all rivers
  all_rivers = []

  def __init__(self, name, length):
```

```
        self.name = name
        self.length = length
        # add current river to the list of all rivers
        River.all_rivers.append(self)

volga = River("Volga", 3530)
seine = River("Seine", 776)
nile = River("Nile", 6852)

# print all river names
for river in River.all_rivers:
    print(river.name)
# Output:
# Volga
# Seine
# Nile
```

We created three instances (or objects) of the class River: volga, seine, and nile. Since we defined **name** and **length** parameters for the __init__, they must be explicitly passed when creating new instances. So something like volga = River() would cause an error. Look at this visualization of the code to see how it works almost in real-time!

The __init__ method specifies what attributes we want the instances of our class to have from the very beginning. In our example, they are **name** and **length.**

self

You may have noticed that our __init__ method had another argument besides name and length: **self**. The self argument represents a particular instance of the class and allows us to access its attributes and methods. In the example with __init__, we basically create attributes for the particular instance and assign the values of method arguments to them. It is important to use the self parameter inside the method if we want to save the values of the instance for later use.

Most of the time we also need to write the self parameter in other methods because when the method is called the first argument that is passed to the method is the object itself. Let's add a method to our River class and see how it works. The syntax of the methods is not of importance at the moment, just pay attention to the use of the self:

```
class River:
    all_rivers = []
```

```
    def __init__(self, name, length):
        self.name = name
        self.length = length
        River.all_rivers.append(self)

    def get_info(self):
        print("The length of the {0} is {1} km".format(self.name, self.length))
```

Now if we call this method with the objects we've created we will get this:

```
volga.get_info()
# The length of the Volga is 3530 km
seine.get_info()
# The length of the Seine is 776 km
nile.get_info()
# The length of the Nile is 6852 km
```

As you can see, for each object the get_info() method printed its particular values and that was possible because we used the self-keyword in the method.

Note that when we actually call an object's method we don't write the self argument in the brackets. The self parameter (that represents a particular instance of the class) is passed to the instance method **implicitly** when it is called. So there are actually two ways to call an instance method: self.method() or class.method(self). In our example it would look like this:

```
# self.method()
volga.get_info()
# The length of the Volga is 3530 km

# class.method(self)
River.get_info(volga)
# The length of the Volga is 3530 km
```

Instance attributes

Classes in Python have two types of attributes: class attributes and instance attributes. You should already know what class attributes are, so here we'll focus on the instance attributes instead. **Instance attributes** are defined within methods and they store instance-specific information.

In the class River, the attributes **name** and **length** are instance attributes since they are defined within a method (__init__) and have self before them. Usually, instance attributes

are created within the __init__ method since it's the constructor, but you can define instance attributes in other methods as well. However, it's not recommended so we advise you to stick to the __init__.

Instance attributes are available only from the scope of the object which is why this code will produce a mistake:

```
print(River.name)  # AttributeError
```

Instance attributes, naturally, are used to distinguish objects: their values are different for different instances.

```
volga.name  # "Volga"
seine.name  # "Seine"
nile.name   # "Nile"
```

So when deciding which attributes to choose in your program, you should first decide whether you want it to store values unique to each object of the class or, on the contrary, the ones shared by all instances.

Summary

In this topic, you've learned about class instances.

If classes are an abstraction, a template for similar objects, a **class instance** is a sort of example of that class, a particular object that follows the structure outlined in the class. In your program, you can create as many objects of your class as you need.

To create objects with different initial states, classes have a constructor __init__ that allows us to define necessary parameters. Reference to a particular instance within methods is done through the self keyword. Within the __init__ method, we define instance attributes that are different for all instances.

Most of the time, we'll deal in our programs not with classes as such but rather with their instances, so knowing how to create them and work with them is very important!