Object-Oriented Programming

Objectives

In this module, you will learn about:

- Basic concepts of object-oriented programming (OOP)
- The differences between the procedural and object approaches (motivations and profits)
- Classes, objects, properties, and methods;
- Designing reusable classes and creating objects;
- Inheritance and polymorphism;
- Exceptions as objects.

Basic concepts of object-oriented programming

- The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).
- The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

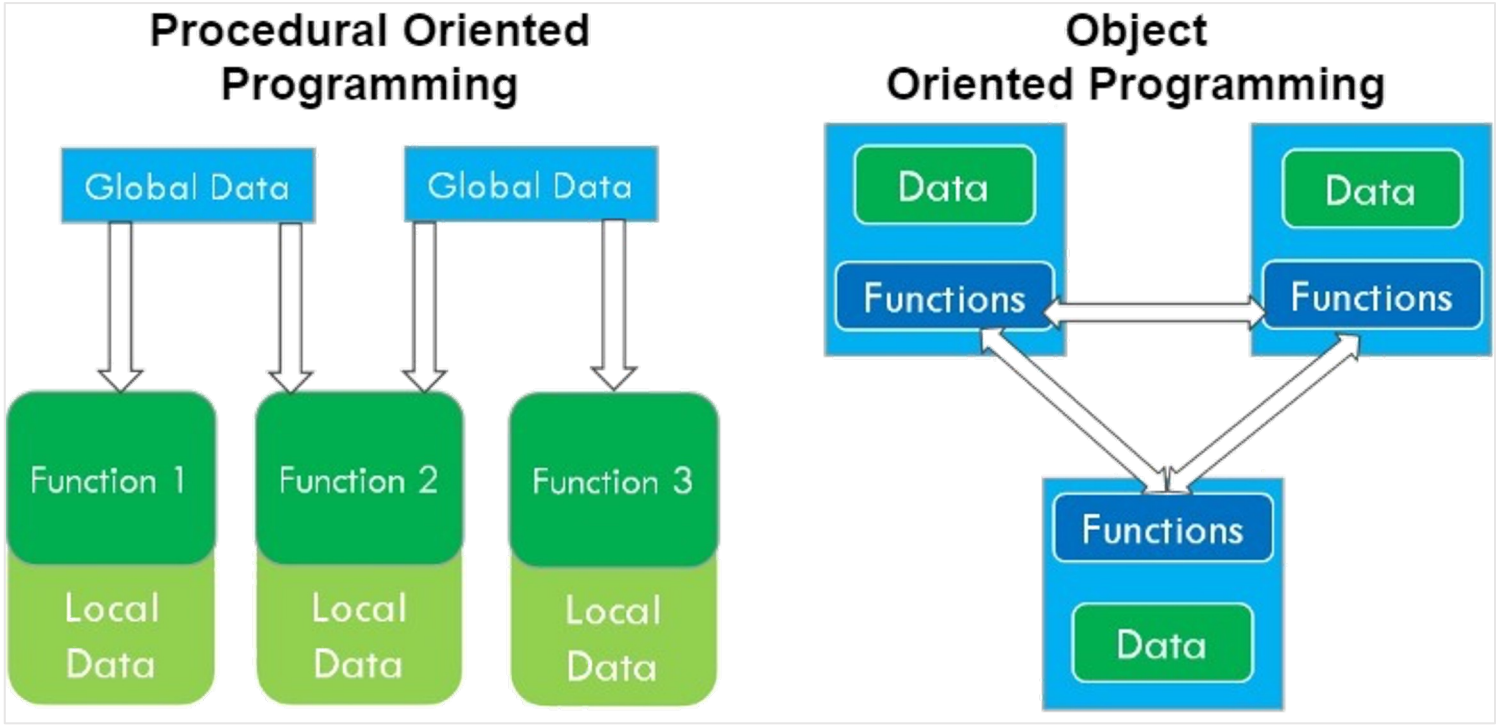
Python is a universal tool for both object and procedural programming. It may be successfully utilized in both spheres.

Procedural vs. the object-oriented approach

- In the **procedural approach**, it's possible to distinguish two different and completely separate worlds: **the world of data, and the world of code**. The world of data is populated with variables of different kinds, while the world of code is inhabited by code grouped into modules and functions.
- Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, i.e., to use the value in an unauthorized manner (e.g., when the sine function gets a bank account balance as a parameter).

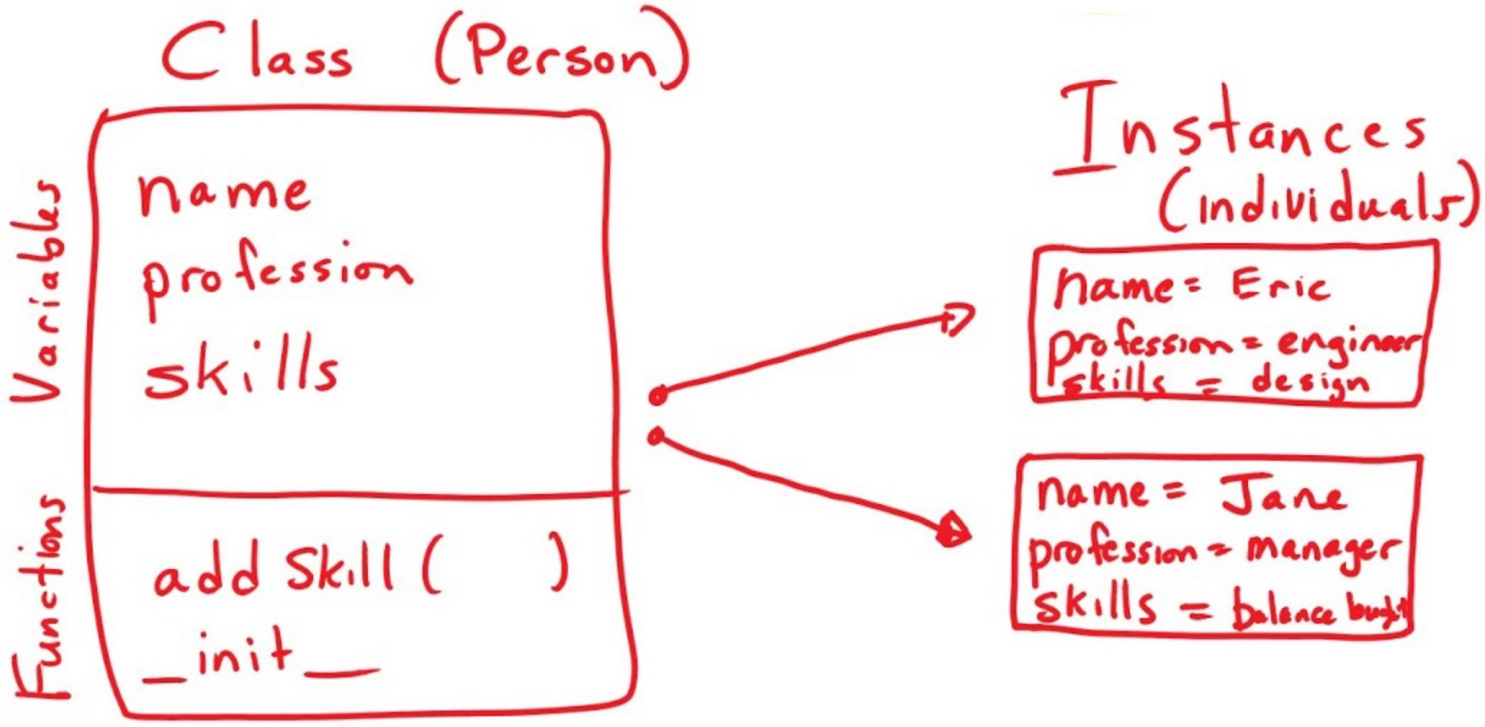
Procedural vs. the object-oriented approach

- The **object approach** suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.



Class and object

- A Class is like an object constructor, or a "blueprint" for creating objects.



Create an object

To create a class, use the keyword class

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Properties

- Python objects, when created, are gifted with a **small set of predefined properties and methods**. Each object has got them, whether you want them or not. One of them is a variable named __dict__ (it's a dictionary).

```
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val

    def set_second(self, val):
        self.second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.third = 5

print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)
```

Console >_

```
{'first': 1}
{'first': 2, 'second': 3}
{'first': 4, 'third': 5}
{'first': 1}
{'first': 2, 'second': 3}
{'first': 4, 'third': 5}
```

Private properties

- Here in Python, we can make an attribute private by adding 2 underscore characters _ in front of our variable name.

```
class Dog:
    def __init__(self):
        self.__name = "Becgie"

dog1 = Dog()

print(dog1.__name)
```

```
AttributeError: 'Dog' object has no attribute '__name'
```

Delete Object Properties

- You can delete properties on objects by using the del keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

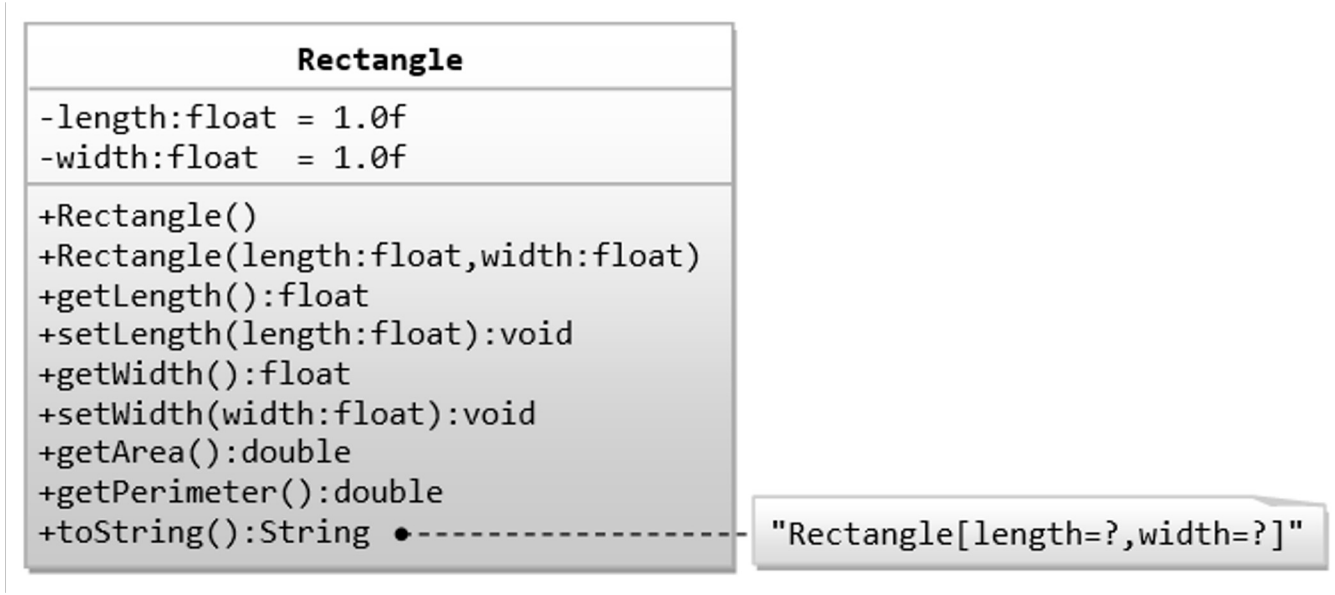
Methods

- As you already know, a **method is a function embedded inside a class**.
- The first (or only) parameter is usually named self. We suggest that you follow the convention - it's commonly used, and you'll cause a few surprises by using other names for it.

```
class Classy:
    def method(self, par):
        print("method:", par)
```

Exercises

1. Write a Python class named Student with two attributes student_name, marks.
2. Class diagram:



Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Inheritance

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)
```

```
class Student(Person):
  pass
```

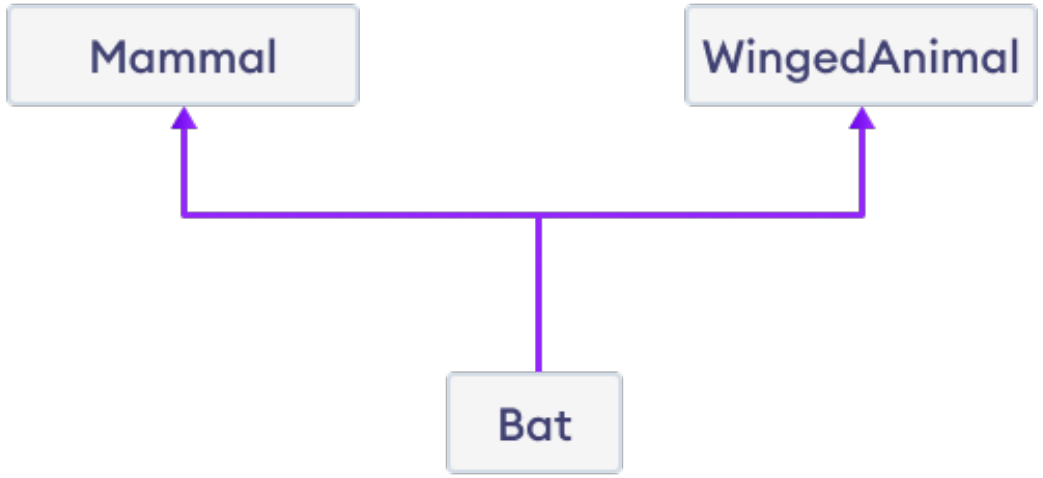
```
x = Student("Mike", "Olsen")
x.printname()
```

```
Mike Olsen
```

Uses of Inheritance

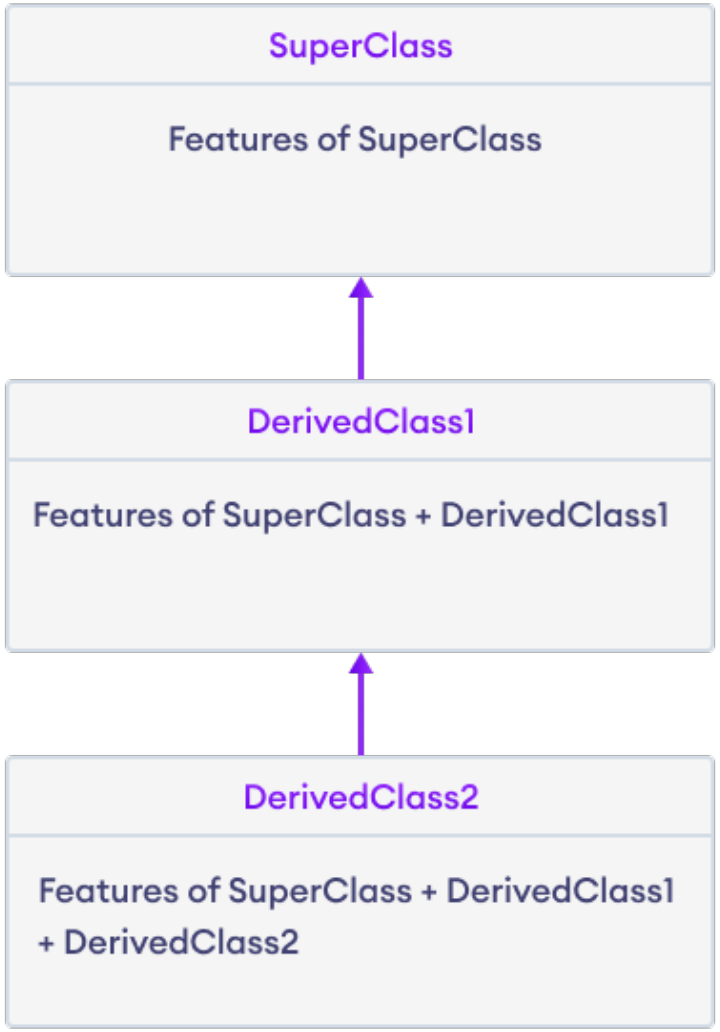
1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
3. Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

Python Multiple Inheritance




```
class Mammal:
    def mammal_info(self):
        print("Mammals can give direct birth.")

class WingedAnimal:
    def winged_animal_info(self):
        print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):
    pass

# create an object of Bat class
b1 = Bat()

b1.mammal_info()
b1.winged_animal_info()
```

Python Multilevel Inheritance



Method Resolution Order (MRO) in Python

- If two superclasses have the same method name and the derived class calls that method, Python uses the MRO to search for the right method to call.

```
class SuperClass1:
    def info(self):
        print("Super Class 1 method called")

class SuperClass2:
    def info(self):
        print("Super Class 2 method called")

class Derived(SuperClass1, SuperClass2):
    pass

d1 = Derived()
d1.info()

# Output: "Super Class 1 method called"
```

Override

```
1   class Animal:
2     def makeSound(self):
3       print("Make any sound")
4
5   class Dog(Animal):
6     def makeSound(self):
7       print("Woof woof woof")
8
9   a = Animal()
10  a.makeSound()
11
12  d = Dog()
13  d.makeSound()
```

```
Make any sound
Woof woof woof
```

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

Abstraction

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC