11. Introduction to Object-Oriented Programming

October 2, 2025

1 11. Introduction to Object-Oriented Programming

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

1.1 11.1 Classes and Instances

In OOP, a **class** is a blueprint for creating objects. It defines the attributes and behaviors that objects of that class will have. An **object** is an instance of a class. It represents a specific instance of the class, with its own unique data.

For example, `0` is an instance of the class `int`.

1.2 11.2 Defining the class

The simplest definition for a class consists of the following:

- keyword `class`
- the name of the class (usually camel-cased)
- an optional docstring
- the body of the class

[2]:
```
class MyClass:
    """Docstring for MyClass"""
    pass
```

1.3 11.3 Instantiating the class

Instantiating the class, or creating an object of that type, is done by *calling* the class:

[3]:
```
my_obj = MyClass()
type(my_obj)
```

[3]:
```
__main__.MyClass
```

1.3.1 Exercises 11.3

1. Create a `BankAccount` class.
2. Create a `BankAccount` instance and inspect its type.

1.4 11.4 Class members: attributes

Any definining characteristics that we wish to store on objects can be kept on attributes. They are also known as data members, and there are two types of data members:

- class variables (shared by all instances of a class)
- instance variables (particular to every instance)

Most often, instance variables are defined in a special initializing method, which is called after the instance is created. Methods are just like normal functions with the exception that the first argument to each method is `self` - the current instance.

[9]:
```
class Vehicle:
    # Class attribute
    wheels = 0

    def __init__(self, color, make, model=None):
        # Instance attributes
        self.color = color
        self.make = make
        self.model = model
        self.speed = 0
```

When instantiating a class, we must send the arguments `__init__` expects, minus the `self` argument which is implicitly passed.

[10]:
```
vehicle = Vehicle("Red", "Toyota", "Camry")
```

[11]:
```
other_vehicle = Vehicle("Black", "Tesla")
```

1.4.1 Accessing attributes

We can access the attributes of an object using dot notation.

[6]:
```
vehicle.color
```

[6]:
```
'Red'
```

[7]:
```
vehicle.color = "White"
vehicle.color
```

[7]:
```
'White'
```

Class attributes can be accessed both from the class and from any instance:

[8]:
```
Vehicle.wheels
```

[8]:
```
0
```

1.4.2 Exercises 11.4

1. Modify `BankAccount` class to receive two parameters on initialisation:
   - `bank_name` - a string to identify the bank (required)
   - `balance` - a number that holds the current balance (optional, default 0)
2. Create two `BankAccount` instances (different bank names, one with balance specified, the other one without) and inspect their attributes.

1.5 11.5 Class members: methods

Methods are functions defined inside classes. Instance methods always receive the current instance (`self`) as the first parameter. `self` has to be specified in the method definition.

[9]:
```
class Vehicle:
    # Class attribute
    wheels = 0

    def __init__(self, color, make, model):
        # Instance attributes
        self.color = color
        self.make = make
        self.model = model
        self.speed = 0

    def accelerate(self, increment):
        self.speed += increment

    def brake(self, decrement):
        self.speed -= decrement

    def display_speed(self):
        print("Current speed:", self.speed)
```

1.5.1 Calling methods

When calling an instance method, you don't need to send the current object as its first parameter; Python knows who the current instance is because you are using it to call the method: `current_obj.method(params)`.

[21]:
```
vehicle = Vehicle("Red", "Toyota", "Camry")
vehicle.accelerate(20)
vehicle.display_speed()
```

```
Current speed: 20
```

[22]:
```
vehicle.brake(10)
print(vehicle.speed)
```

```
10
```

1.6 Exercises 11.5

1. Create two methods in `BankAccount` class, one to withdraw money and another one to deposit money into the account. The withdraw method will not allow withdrawing more money than available: it will raise `ValueError` exception and not change the balance.
2. Test the two methods with different inputs on the existing instances.

1.7 11.6 Inheritance

Instead of starting from scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

In order to call a parent method from a child method, we need to use `super()`. `super()` returns a temporary object of the superclass that then allows you to call that superclass's methods.

[11]:
```
class Car(Vehicle):
    # Class attribute specific to Car
    wheels = 4

    def __init__(self, color, make, model, fuel_type):
        # Call the constructor of the superclass
        super().__init__(color, make, model)
        # Additional instance attribute specific to Car
        self.fuel_type = fuel_type

    def honk_horn(self):
        print("Beep beep!")
```

[12]:
```
car = Car("Red", "Toyota", "Camry", "diesel")
```

[13]:
```
car.accelerate(50)
```

[14]:
```
car.honk_horn()
```

```
Beep beep!
```

[15]:
```
car.display_speed()
```

```
Current speed: 50
```

Car.wheels

1.8 Exercises 11.6

1. Create a `CreditBankAccount` class that inherits `BankAccount` and receives one extra argument at initialisation (`overdraft`) which allows for the balance to go below zero (but not under `-overdraft`).

2. Override parent `withdraw` method so that the new rule is implemented.
3. Create a `CreditBankAccount` instance and try calling `withdraw` and `deposit` methods.

1.9 11.7 Built-in functions useful in OOP

Functions to check relationships of classes and instances:

`isinstance(obj, cls)` - verifies if `obj` is an instance of `cls`

`issubclass(cls1, cls2)` - verifies if `cls1` is a subclass of `cls2`

[16]:
```
isinstance(car, Car)
```

[16]: True

[17]:
```
isinstance(car, Vehicle)
```

[17]: True

[18]:
```
issubclass(Car, Vehicle)
```

[18]: True