Objects and Classes

What is an object in Python?

An **object** in Python is a specific instance of a class. It represents a collection of data (attributes) and functions (methods) that operate on that data. Here are some key points about objects:

- **Instance of a Class**: When you create an object, you are creating an instance of a class. For example, if you have a class called `Dog`, a specific dog like "Buddy" would be an object of that class.
- **Attributes**: Objects have attributes, which are the data stored in them. For example, a `Dog` object might have attributes like `name`, `breed`, and `age`.
- **Methods**: Objects can also have methods, which are functions defined in the class that can perform actions using the object's attributes. For example, a `Dog` object might have a method called `bark()` that makes the dog bark.
- **Type**: Every object in Python has a type, which is determined by the class it is an instance of. You can check the type of an object using the `type()` function.

In summary, an object is a concrete representation of a class, containing both data and behavior related to that data.

example of object

Here are two examples of objects in Python:

Example 1: Dog Class

```
class Dog:
    def __init__(self, name, breed):
        self.name = name  # Attribute
        self.breed = breed  # Attribute

    def bark(self):  # Method
```

```
        return f"{self.name} says Woof!"

# Creating an object of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")

# Accessing attributes and methods
print(my_dog.name)  # Output: Buddy
print(my_dog.breed)  # Output: Golden Retriever
print(my_dog.bark())  # Output: Buddy says Woof!
```

Example 2: Circle Class

```
class Circle:
    def __init__(self, radius, color):
        self.radius = radius  # Attribute
        self.color = color  # Attribute

    def area(self):  # Method
        return 3.14 * (self.radius ** 2)

# Creating an object of the Circle class
my_circle = Circle(5, "red")

# Accessing attributes and methods
print(my_circle.radius)  # Output: 5
print(my_circle.color)  # Output: red
print(my_circle.area())  # Output: 78.5
```

In these examples:

- `my_dog` is an object of the `Dog` class, with attributes `name` and `breed`, and it can perform the `bark` method.
- `my_circle` is an object of the `Circle` class, with attributes `radius` and `color`, and it can calculate its area using the `area` method.

Objectives

In this reading, you will learn about:

- Fundamental concepts of Python objects and classes.
- Structure of classes and object code.
- Real-world examples related to objects and classes.

Introduction to classes and object

Python is an object-oriented programming (OOP) language that uses a paradigm centered around objects and classes.

Let's look at these fundamental concepts.

Classes

A class is a blueprint or template for creating objects. It defines the structure and behavior that its objects will have.

Think of a class as a cookie cutter and objects as the cookies cut from that template.

In Python, you can create classes using the `class` keyword.

Creating classes

When you create a class, you specify the `attributes` (data) and `methods` (functions) that objects of that class will have.

`Attributes` are defined as variables within the class, and `methods` are defined as functions.

For example,you can design a "Car" class with attributes such as "color" and "speed," along with methods like "accelerate."

Objects

An *object* is a fundamental unit in Python that represents a real-world entity or concept.

Objects can be tangible (like a car) or abstract (like a student's grade).

Every object has two main characteristics:

State

The *attributes or data* that describe the object. For your "Car" object, this might include attributes like "color", "speed", and "fuel level".

Behavior

The *actions or methods* that the object can perform. In Python, methods are functions that belong to objects and can change the object's state or perform specific operations.

Instantiating objects

- Once you've defined a class, you can create individual objects (instances) based on that class.
- Each object is independent and has its own set of attributes and methods.
- To create an object, you use the class name followed by parentheses, so: "my_car = Car()"

Interacting with objects

You interact with objects by calling their methods or accessing their attributes using dot notation.

For example, if you have a Car object named **my_car**, you can set its color with **my_car.color = "blue"** and accelerate it with **my_car.accelerate()** if there's an accelerate method defined in the class.

Structure of classes and object code

> Please don't directly copy and use this code because it is a template for explanation and not for specific results.

Class declaration (class ClassName)

- The `class` keyword is used to declare a class in Python.
- `ClassName` is the name of the class, typically following CamelCase naming conventions.

```
class ClassName:
```

Class attributes (class_attribute = value)

- Class attributes are variables shared among all class instances (objects).
- They are defined within the class but outside of any methods.

```python
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value
```

Constructor method (def init(self, attribute1, attribute2, ...):)

- The `__init__` method is a special method known as the constructor.
- It initializes the **instance attributes** (also called instance variables) when an object is created.
- The `self` parameter is the first parameter of the constructor, referring to the instance being created.
- **attribute1**, **attribute2**, and so on are parameters passed to the constructor when creating an object.
- Inside the constructor, `self.attribute1`, `self.attribute2`, and so on are used to assign values to instance attributes.

```python
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        pass
        # ...
```

Instance attributes (self.attribute1 = attribute1)

- Instance attributes are variables that store data specific to each class instance.
- They are initialized within the `__init__` method using the self keyword followed by the attribute name.
- These attributes hold unique data for each object created from the class.

```
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...
```

Instance methods (def method1(self, parameter1, parameter2, ...):)

- Instance methods are functions defined within the class.
- They operate on the instance's data (instance attributes) and can perform actions specific to instances.
- The **self** parameter is required in instance methods, allowing them to access instance attributes and call other methods within the class.

```
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...

    # Instance methods (functions)
    def method1(self, parameter1, parameter2, ...):
        # Method logic
        pass
```

> *Using the same steps you can define multiple instance methods.*

```python
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...

    # Instance methods (functions)
    def method1(self, parameter1, parameter2, ...):
        # Method logic
        pass

    def method2(self, parameter1, parameter2, ...):
        # Method logic
        pass
```

> *Note: Now, you have successfully created a dummy class.*

Creating objects (Instances)

- To create objects (instances) of the class, you call the class like a function and provide arguments the constructor requires.
- Each object is a distinct instance of the class, with its own instance attributes and the ability to call methods defined in the class.

```python
# Create objects (instances) of the class
object1 = ClassName(arg1, arg2, ...)
object2 = ClassName(arg1, arg2, ...)
```

Calling methods on objects

- In this section, you will call methods on objects, specifically `object1` and `object2`.
- The methods **method1** and **method2** are defined in the ClassName **class**, and you're calling them on **object1** and **object2** respectively.
- You pass values **param1_value** and **param2_value** as arguments to these methods. These arguments are used within the method's logic.

Method 1: Using dot notation

- This is the most straightforward way to call an object's method. In this, use the dot notation **(object.method())** to invoke the method on the object directly.
- For example, `result1 = object1.method1(param1_value, param2_value, ...)` calls method1 on object1.

```
# Calling methods on objects
# Method 1: Using dot notation
result1 = object1.method1(param1_value, param2_value, ...)
result2 = object2.method2(param1_value, param2_value, ...)
```

Method 2: Assigning object methods to variables

- Here's an alternative way to call an object's method by assigning the method reference to a variable.
- `method_reference = object1.method1` assigns the method **method1** of **object1** to the variable **method_reference**.
- Later, call the method using the variable like this: **result3 = method_reference(param1_value, param2_value, ...)**.

```
# Method 2: Assigning object methods to variables
method_reference = object1.method1  # Assign the method to a variable
result3 = method_reference(param1_value, param2_value, ...)
```

Accessing object attributes

- Here, you are accessing an object's attribute using dot notation.
- `attribute_value = object1.attribute1` retrieves the value of the attribute **attribute1** from **object1** and assigns it to the variable **attribute_value**.

```
# Accessing object attributes
attribute_value = object1.attribute1  # Access the attribute using dot notation
```

Modifying object attributes

- You will modify an object's attribute using dot notation.
- `object1.attribute2 = new_value` sets the attribute **attribute2** of **object1** to the new value **new_value**.

```
# Modifying object attributes
object1.attribute2 = new_value  # Change the value of an attribute using dot notation
```

Accessing class attributes (shared by all instances)

- Finally, access a class attribute shared by all class instances.
- `class_attr_value = ClassName.class_attribute` accesses the class attribute `class_attribute` from the ClassName `class` and assigns its value to the variable. `class_attr_value`.

```
# Accessing class attributes (shared by all instances)
class_attr_value = ClassName.class_attribute
```

Real-world example

Let's write a python program that simulates a simple car class, allowing you to create car instances, accelerate them, and display their current speeds.

1. Let's start by defining a `Car` class that includes the following attributes and methods:

- Class attribute `max_speed`, which is set to **120 km/h**.
- Constructor method `__init__` that takes parameters for the **car's make, model, color, and an optional speed (defaulting to 0)**. This method initializes instance attributes for make, model, color, and speed.
- Method `accelerate(self, acceleration)` that allows the car to accelerate. If the acceleration does not exceed the `max_speed`, update the **car's speed** attribute. Otherwise, set the speed to the **max_speed**.
- Method `get_speed(self)` that returns the current speed of the car.

```python
class Car:
    # Class attribute (shared by all instances)
    max_speed = 120  # Maximum speed in km/h

    # Constructor method (initialize instance attributes)
    def __init__(self, make, model, color, speed=0):
        self.make = make
        self.model = model
        self.color = color
        self.speed = speed  # Initial speed is set to 0

    # Method for accelerating the car
    def accelerate(self, acceleration):
        if self.speed + acceleration <= Car.max_speed:
            self.speed += acceleration
        else:
            self.speed = Car.max_speed

    # Method to get the current speed of the car
    def get_speed(self):
        return self.speed
```

2. Now, you will instantiate two objects of the `Car` class, each with the following characteristics:

- car1: **Make = "Toyota", Model = "Camry", Color = "Blue"**
- car2: **Make = "Honda", Model = "Civic", Color = "Red"**

```
# Create objects (instances) of the Car class
car1 = Car("Toyota", "Camry", "Blue")
car2 = Car("Honda", "Civic", "Red")
```

3. Using the `accelerate` method, you will increase the speed of car1 by 30 km/h and car2 by 20 km/h.

```
# Accelerate the cars
car1.accelerate(30)
car2.accelerate(20)
```

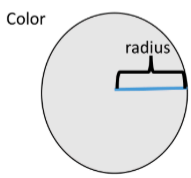
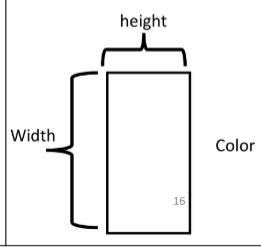
4. Lastly, you will display the current speed of each car by utilizing the `get_speed` method.

```
# Print the current speeds of the cars
print(f"{car1.make} {car1.model} is currently at {car1.get_speed()} km/h.")
print(f"{car2.make} {car2.model} is currently at {car2.get_speed()} km/h.")
```

Example with Explanation step by step

Creating a Class

The first step in creating a class is giving it a name. In this notebook, we will create two classes: Circle and Rectangle. We need to determine all the data that make up that class, which we call *attributes*. Think about this step as creating a blue print that we will use to create objects. In figure 1 we see two classes, Circle and Rectangle. Each has their attributes, which are variables. The class Circle has the attribute radius and color, while the Rectangle class has the attribute height and width. Let's use the visual examples of these shapes before we get to the code, as this will help you get accustomed to the vocabulary.

Class Circle	**Class Rectangle**
Attributes: radius, Color	Attributes: Color, height and Width
	

Instances of a Class: Objects and Attributes

An instance of an object is the realisation of a class, and in Figure 2 we see three instances of the class circle. We give each object a name: red circle, yellow circle, and green circle. Each object has different attributes, so let's focus on the color attribute for each object.

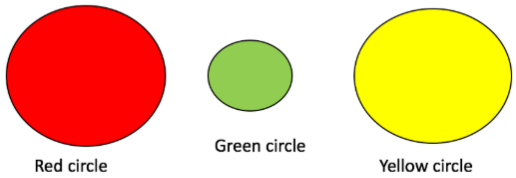


Figure 2: Three instances of the class Circle, or three objects of type Circle.

The colour attribute for the red Circle is the colour red, for the green Circle object the colour attribute is green, and for the yellow Circle the colour attribute is yellow.

Methods

Methods give you a way to change or interact with the object; they are functions that interact with objects. For example, let's say we would like to increase the radius of a circle by a specified amount. We can create a method called **add_radius(r)** that increases the radius by **r**. This is shown in figure 3, where after applying the method to the "orange circle object", the radius of the object increases accordingly. The "dot" notation means to apply the method to the object, which is essentially applying a function to the information in the object.

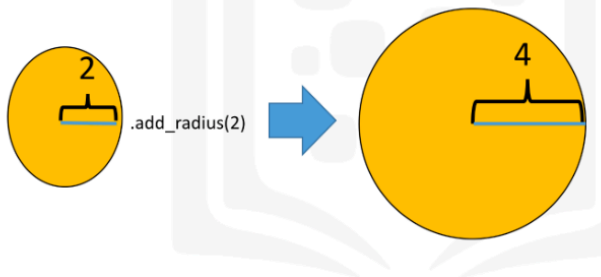


Figure 3: Applying the method "add_radius" to the object orange circle object.

Creating a Class

Now we are going to create a class Circle, but first, we are going to import a library to draw the objects:

```
# Import the library

import matplotlib.pyplot as plt
%matplotlib inline
```

The first step in creating your own class is to use the `class` keyword, then the name of the class as shown in Figure 4. In this course the class parent will always be object:

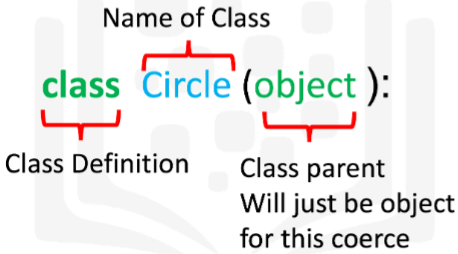
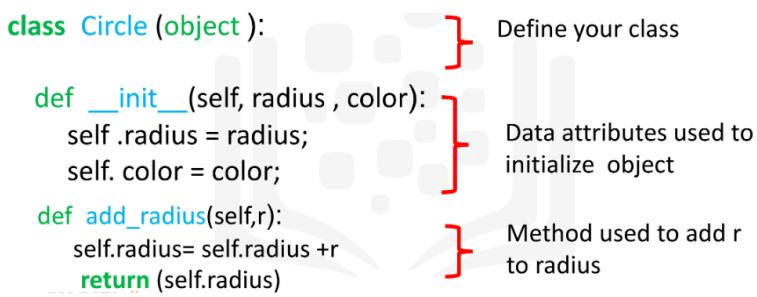


Figure 4: Creating a class Circle.

The next step is a special method called a constructor `__init__`, which is used to initialize the object. The inputs are data attributes. The term `self` contains all the attributes in the set. For example the `self.color` gives the value of the attribute color and `self.radius` will give you the radius of the object. We also have the method `add_radius()` with the parameter `r`, the method adds the value of `r` to the attribute radius. To access the radius we use the syntax `self.radius`. The labeled syntax is summarized in Figure 5:



The actual object is shown below. We include the method `drawCircle` to display the image of a circle. We set the default radius to 3 and the default colour to blue:

```
# Create a class Circle

class Circle(object):

    # Constructor
    def __init__(self, radius=3, color='blue'):
        self.radius = radius
        self.color = color

    # Method
    def add_radius(self, r):
        self.radius = self.radius + r
        return(self.radius)

    # Method
    def drawCircle(self):
        plt.gca().add_patch(plt.Circle((0, 0), radius=self.radius, fc=self.color))
        plt.axis('scaled')
        plt.show()
```

Creating an instance of a class Circle

Let's create the object `RedCircle` of type Circle to do the following:

```
# Create an object RedCircle

RedCircle = Circle(10, 'red')
```

We can use the `dir` command to get a list of the object's methods. Many of them are default Python methods.

```
# Find out the methods can be used on the object RedCircle

dir(RedCircle)
```

We can look at the data attributes of the object:

```
# Print the object attribute radius

RedCircle.radius
```

```
# Print the object attribute color

RedCircle.color
```

We can change the object's data attributes:

```
# Set the object attribute radius

RedCircle.radius = 1
RedCircle.radius
```

We can draw the object by using the method `drawCircle()`:

```
# Call the method drawCircle

RedCircle.drawCircle()
```

We can increase the radius of the circle by applying the method `add_radius()`. Let's increases the radius by 2 and then by 5:

```
# Use method to change the object attribute radius

print('Radius of object:',RedCircle.radius)
RedCircle.add_radius(2)
print('Radius of object of after applying the method add_radius(2):',RedCircle.radius)
RedCircle.add_radius(5)
print('Radius of object of after applying the method add_radius(5):',RedCircle.radius)
```

Let's create a blue circle. As the default colour is blue, all we have to do is specify what the radius is:

```
# Create a blue circle with a given radius

BlueCircle = Circle(radius=100)
```

As before, we can access the attributes of the instance of the class by using the dot notation:

```
# Print the object attribute radius

BlueCircle.radius
```

```
# Print the object attribute color

BlueCircle.color
```

We can draw the object by using the method `drawCircle()`:

```
# Call the method drawCircle

BlueCircle.drawCircle()
```

Compare the x and y axis of the figure to the figure for `RedCircle`; they are different.