**Objects & classes :**

Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of **objects** and **classes**.

An object is any entity that has **attributes** and **behaviors**. For example, a parrot is an object. It has

* **attributes** - name, age, color, etc.
* **behavior** - dancing, singing, etc.

Similarly, a class is a blueprint for that object.

**Python Class and Object**

class Parrot:

# class attribute

name = ""

age = 0

# create parrot1 object

parrot1 = Parrot()

parrot1.name = "Blu"

parrot1.age = 10

# create another object parrot2

parrot2 = Parrot()

parrot2.name = "Woo"

parrot2.age = 15

# access attributes

print(f"{parrot1.name} is {parrot1.age} years old")

print(f"{parrot2.name} is {parrot2.age} years old")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Blu is 10 years old

Woo is 15 years old

In the above example, we created a class with the name Parrot with two attributes: name and age.

Then, we create instances of the Parrot class. Here, parrot1 and parrot2 are references (value) to our new objects.

We then accessed and assigned different values to the instance attributes using the objects name and the . notation.

To learn more about classes and objects, visit [Python Classes and Objects](https://www.programiz.com/python-programming/class)

**Python Inheritance**

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

**Example 2: Use of Inheritance in Python**

# base class

class Animal:

def eat(self):

print( "I can eat!")

def sleep(self):

print("I can sleep!")

# derived class

class Dog(Animal):

def bark(self):

print("I can bark! Woof woof!!")

# Create object of the Dog class

dog1 = Dog()

# Calling members of the base class

dog1.eat()

dog1.sleep()

# Calling member of the derived class

dog1.bark();

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat!

I can sleep!

I can bark! Woof woof!!

Here, dog1 (the object of derived class Dog) can access members of the base class Animal. It's because Dog is inherited from Animal.

# Calling members of the Animal class

dog1.eat()

dog1.sleep()

To learn more about inheritance, visit [Python Inheritance](https://www.programiz.com/python-programming/inheritance).

**Python Encapsulation**

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and methods inside a single class.

It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve **data hiding**.

In Python, we denote private attributes using underscore as the prefix i.e single \_ or double \_\_. For example,

class Computer:

def \_\_init\_\_(self):

self.\_\_maxprice = 900

def sell(self):

print("Selling Price: {}".format(self.\_\_maxprice))

def setMaxPrice(self, price):

self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Selling Price: 900

Selling Price: 900

Selling Price: 1000

In the above program, we defined a Computer class.

We used \_\_init\_\_() method to store the maximum selling price of Computer. Here, notice the code

c.\_\_maxprice = 1000

Here, we have tried to modify the value of \_\_maxprice outside of the class. However, since \_\_maxprice is a private variable, this modification is not seen on the output.

As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

**Polymorphism**

Polymorphism is another important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Let's see an example,

class Polygon:

# method to render a shape

def render(self):

print("Rendering Polygon...")

class Square(Polygon):

# renders Square

def render(self):

print("Rendering Square...")

class Circle(Polygon):

# renders circle

def render(self):

print("Rendering Circle...")

# create an object of Square

s1 = Square()

s1.render()

# create an object of Circle

c1 = Circle()

c1.render()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Rendering Square...

Rendering Circle...

In the above example, we have created a superclass: Polygon and two subclasses: Square and Circle. Notice the use of the render() method.

The main purpose of the render() method is to render the shape. However, the process of rendering a square is different from the process of rendering a circle.

Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.