**Object Oriented Programming**

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

* attributes
* behavior

Let's take an example:

A parrot is an object, as it has the following properties:

* name, age, color as attributes
* singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

**Class**

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

class Parrot:

pass

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

**Object**

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

obj = Parrot()

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

**Example 1: Creating Class and Object in Python**

class Parrot:

# class attribute

species = "bird"

# instance attribute

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)

# access the class attributes

print("Blu is a {}".format(blu.\_\_class\_\_.species))

print("Woo is also a {}".format(woo.\_\_class\_\_.species))

# access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))

**Output**

Blu is a bird

Woo is also a bird

Blu is 10 years old

Woo is 15 years old

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the \_\_init\_\_ method of the class. It is the initializer method that is first run as soon as the object is created.

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

We can access the class attribute using \_\_class\_\_.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

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

**Methods**

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

**Example 2 : Creating Methods in Python**

class Parrot:

# instance attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instance method

def sing(self, song):

return "{} sings {}".format(self.name, song)

def dance(self):

return "{} is now dancing".format(self.name)

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

print(blu.sing("'Happy'"))

print(blu.dance())

**Output**

Blu sings 'Happy'

Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance methods because they are called on an instance object i.e blu.

**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 3: Use of Inheritance in Python**

# parent class

class Bird:

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

print("Bird is ready")

def whoisThis(self):

print("Bird")

def swim(self):

print("Swim faster")

# child class

class Penguin(Bird):

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

# call super() function

super().\_\_init\_\_()

print("Penguin is ready")

def whoisThis(self):

print("Penguin")

def run(self):

print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()

**Output**

Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the \_\_init\_\_() method. This allows us to run the \_\_init\_\_() method of the parent class inside the child class.

**Encapsulation**

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single \_ or double \_\_.

**Example 4: Data Encapsulation in Python**

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()

**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 an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

**Example 5: Using Polymorphism in Python**

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

**Output**

Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying\_test() function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying\_test() function, it ran effectively.

**Key Points to Remember:**

* Object-Oriented Programming makes the program easy to understand as well as efficient.
* Since the class is sharable, the code can be reused.
* Data is safe and secure with data abstraction.
* Polymorphism allows the same interface for different objects, so programmers can write efficient code.