Object Oriented Python

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Some points on Python class:

- Classes are created by keyword **class**.
- **Attributes** are the **variables** that belong to a class.
- Attributes are always **public** and can be accessed using the dot **(.)** operator. **Eg.:** car .model

Class Definition Syntax:

```
Class ClassName:
  # Statement-1
  .
  .
  # Statement-N
```

- **Creating an Empty Class in Python**

```
class test:
    pass
```

- **Creating a class and object with class and instance attributes**

```
class CDAC:
 # class attribute
course= "PGDBDA"
 # Instance attribute
def __init__(self, name):
    self.name = name
       def course(self):
    print("My Course is {}".format(self.name))
# Object Created
Tarun = CDAC("PGDBDA")
Vikas = CDAC("PGDBDA")
# Accessing class methods
Tarun.course()
```

The Python __init__ Method

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
def __init__(self, name):
  self.name = name
```

__init__ is a special method (constructor) that initializes an instance of the Dog class.

It takes two parameters:

- **self** (referring to the instance being created)
- **name** (representing the **name** of the class - CDAC).

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Python Objects

The object is an entity that has a state and behaviour associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

An object consists of:

State: It is represented by the attributes of an object. It also reflects the properties of an object.

Behaviour: It is represented by the methods of an object. It also reflects the response of an object to other objects.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

Creating an Object

obj = **CDAC()**

Object Methods: How to create object

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

The Python self

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it

If we have a method that takes no arguments, then we still have to have one argument.

The self-Parameter

The self-parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

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

```
 def myfunc(abc):
   print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

- **Python Inheritance**

  Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:
- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

  Types of Inheritance

  Single Inheritance: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

  Multilevel Inheritance: Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

  Hierarchical Inheritance: Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.

  Multiple Inheritance: Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

```
# parent class
class Person(object):

 # __init__ is known as the constructor
 def __init__(self, name, idnumber):
  self.name = name
  self.idnumber = idnumber
```

```
      def display(self):
        print(self.name)
        print(self.idnumber)

       def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

      # child class
            class Employee(Person):
            def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

     def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```