***Object Oriented Programming In Python***

Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOPs) 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.

**Principles of Object-Oriented Programming (OOPs)**

1. Class
2. Objects
3. Method
4. Polymorphism
5. Encapsulation
6. Inheritance
7. Data Abstraction

**Class:**

The class can be defined as a collection of objects. It is a logical entity that has some specific **attributes** and **methods**.

For example, if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

**Objects:**

The object is an entity that has **state** and **behaviour**. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

**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.

To understand the state, behaviour, and identity let us take the example of the class dog.

* The identity can be considered as the name of the dog.
* State or Attributes can be considered as the breed, age, or colour of the dog.
* The behaviour can be considered as to whether the dog is eating or sleeping.

**Method:**

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

**let us understand some basic keywords that we used while working with objects and classes.**

### ****The self****:

1. 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

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

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

### ****The \_\_init\_\_ method****:

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.

**Now let us define a class and create some objects using the self and \_\_init\_\_ method.**

### Example: Creating Class and objects with methods

Class Dog:

**# class attribute**

    attr1 = "mammal"

**# Instance attribute**

    def \_\_init\_\_(self, name):

        self.name = name

    def bark(self):

        print("My name is {}".format(self.name))

**# Driver code**

**# Object instantiation**

Jimmy = Dog("Jimmy")

Tommy = Dog("Tommy")

**# Accessing class methods**

Jimmy.bark()

Tommy.bark()

**OUTPUT:**

My name is Jimmy

My name is Tommy

**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.

**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
* 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.

**EXAMPLE:**

**# 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, post):

        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', 22101, "Intern")

**# calling a function of the class Person using**

**# its instance**

a.display()

a.details()

**OUTPUT:**

Rahul

22101

My name is Rahul

IdNumber: 22101

Post: Intern

We have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through employee class as seen in the display function in the above code. A child class can also modify the behaviour of the parent class as seen through the details() method.

**Polymorphism:**

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. Which means that one task can be performed in different ways.

**Encapsulation:**

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

**Data Abstraction:**

Abstraction is used to hide internal details and show only functionalities.

Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

***Implementation Of Algorithm***

**Steps For Data Implementation:**

**Step 1:** Data Pre-Processing step.

**Step 2:** Fitting a Decision Tree algorithm to the training set.

**Step 3:** Predicting the test result.

**Step 4:** Test accuracy of the result.

**Step 5:** Visualizing the test set result.

**Step 1: Data Pre-Processing step**

* Importing the necessary libraries and datasets
* Clean and Preprocess the data to make it ready for use in the algorithm. This includes handling missing values and transforming variables if necessary.

**Step 2: Fitting a Decision Tree algorithm to the training set**

* Decide when to stop building the tree. We can stop when all instances belong to the same class or when the tree reaches a certain maximum depth.
* Calculate the Standard Deviation Reduction(SDR) of the target attribute for each subset of the data. The SDR is used to measure the quality of the splits and select the best attribute to spit on.
* Select the best attribute that results in the highest reduction SDR when split.
* Split the data into subsets, one for each possible value of the selected attribute.
* For each subset of the data, create a child node for the selected attribute. Repeat the process until the stopping criteria are met.

**Step 3: Predicting the test result**

* Once the tree has been built, use it to make predictions.
* Start from the root node and follow the branches based on the value of the attributes in the data instances.
* Continue until you reach a leaf node and use the Standard Deviation of the target attribute in the data at the leaf node as the prediction.

**Step 4: Test accuracy of the result**

Evaluate the performance of the model using metrics such as mean squared error, mean absolute error etc. between the actual and predicted values

**Step 5: Visualizing the test set result**

Plot the actual vs predicted values to visualize the accuracy of the model.

**References:**

1. <https://www.geeksforgeeks.org/python-oops-concepts/>

2. <https://www.w3schools.blog/python-oops-concepts>

3. <https://www.javatpoint.com/python-oops-concepts>

4. <https://www.programiz.com/python-programming/object-oriented-programming>

5. <https://youtu.be/V6qFSJ0FXag>

6. <https://youtu.be/rf3WCir02a4>

7. <https://youtu.be/P2ZB8c5Ha1Q>

8. <https://medium.com/geekculture/step-by-step-decision-tree-id3-algorithm-from-scratch-in-python-no-fancy-library-4822bbfdd88f>