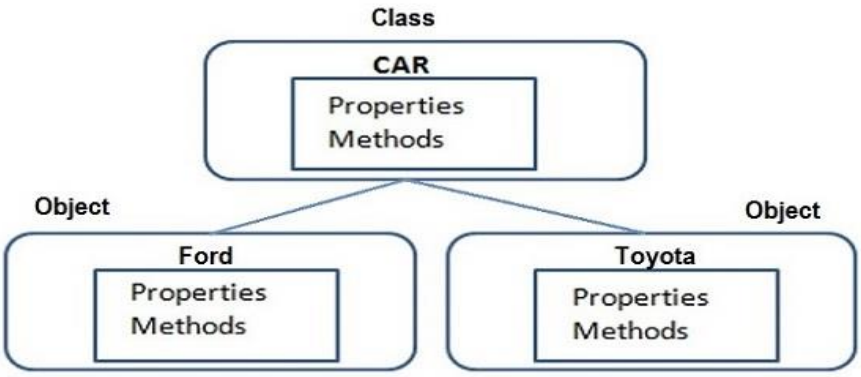
Unit – 2

Object –Oriented Programming in Python

2.1 Object Oriented Programming Concepts

OOP's is an abbreviation that stands for **Object-oriented programming** . It is defined as a programming model that uses the concept of **objects** which refers to real-world entities with state and behavior.

In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.



2.1.1 Attributes

- Name, class, subjects, marks, etc., of student
- Name, designation, department, salary, etc., of employee
- Invoice number, customer, product code and name, price and quantity, etc., in an invoice
- Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

2.1.2 Behavior

Behaviors are actions that can occur on an object. The behaviors that can be performed on a specific class of object are called **methods**. At the programming level, methods are like functions in structured programming, but they *magically* have access to all the data associated with this object.

Processing attributes associated with an object.

- Compute percentage of student's marks
- Calculate incentives payable to employee
- Apply GST to invoice value
- Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.

In OOP, class defines what the attributes its object has are, and how is its behavior. Object, on the other hand, is an instance of the class.

2.1.3 Principles of OOPs Concepts

Object-oriented programming paradigm is characterized by the following principles −

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism

2.2 Classes

In Python, a **class** is a user defined entity (data types) that defines the type of data an object can contain and the actions it can perform. It is used as a template for creating objects. For instance, if we want to define a class for Smartphone in a Python program

we can use the type of data like RAM, ROM, screen-size and actions like call and message.

- **Creating Classes in Python**

The **class** keyword is used to create a new class in Python. The name of the class immediately follows the keyword **class** followed by a colon as shown below − class

ClassName:

- **Class Attribute**

  The properties or variables defined inside a class are called as **Attributes**. An attribute provides information about the type of data a class contains. There are two types of attributes in Python namely **instance attribute** and **class attribute**.

  The instance attribute is defined within the constructor of a Python class and is unique to each instance of the class. And, a class attribute is declared and initialized outside the constructor of the class.

- **Class Attributes (Variables)**

  Class attributes are those variables that belong to a class and whose value is share

  among all the instances of that class. A class attribute remains the same for every instance of the class.

  Class attributes are defined in the class but outside any method. They cannot be initialized inside **__init__()** constructor. They can be accessed by the name of the class in addition to the object. In other words, a class attribute is available to the class as well as its object.

- **Accessing Class Attributes**

  The object name followed by dot notation (.) is used to access class attributes.

Example

The below example demonstrates how to access the attributes of a Python class.

```
class Employee:

name = "ABC XYZ"

 age = "30"

# instance of the class

emp = Employee()

# accessing class attributes

print("Name of the Employee:", emp.name)
```

print("Age of the Employee:", emp.age)

Output

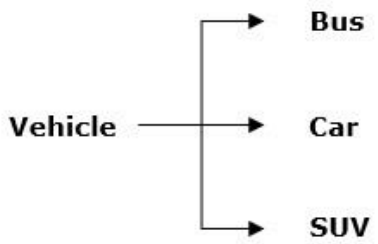
Name of the Employee: ABC XYZ
Age of the Employee: 30

2.3 Objects

An **object** is refered to as an instance of a given Python class. Each object has its own attributes and methods, which are defined by its class.

When a class is created, it only describes the structure of obejcts. The memory is allocated when an object is instantiated from a class.

Example



In the above figure, **Vehicle** is the class name and **Car**, **Bus** and **SUV** are its objects.

2.3.1 Creating Objects of Classes in Python

To create instances of a class, you call the class using class name and pass in whatever arguments its **__init__** method accepts.

EXAMPLE

This would create first object of Employee class

emp1 = Employee("LMN", 2000)

This would create second object of Employee class

emp2 = Employee("PQR", 5000)

2.3.2 Accessing Attributes of Objects in Python

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows –

```
emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)
```

- Now, putting all the concepts together –

```
class Employee:

 empCount = 0

  def __init__(self, name, salary):

  self.name = name

  self.salary = salary

  Employee.empCount += 1

   def displayCount(self):

  print ("Total Employee %d" % Employee.empCount)

  def displayEmployee(self):

  print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("LMN", 2000)

emp2 = Employee("PQR", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)
```

When the above code is executed, it produces the following result –

Name : LMN , Salary: 2000

Name : PQR , Salary: 5000

Total Employee 2

2.4 Inheritance

Inheritance is one of the most important features of object-oriented programming languages like Python. It is used to inherit the properties and behaviours of one class to another. The class that inherits another class is called a **child class** and the class that gets inherited is called a **base class or parent class**.

If you have to design a new class whose most of the attributes are already well defined in an existing class, then why redefine them? Inheritance allows capabilities of existing class to be reused and if required extended to design a new class.

- Creating a Parent Class

The class whose attributes and methods are inherited is called as parent class. It is defined just like other classes i.e. using the class keyword.

Syntax

The syntax for creating a parent class is shown below –

```
class ParentClassName:

{class body}
```

- Creating a Child Class

Classes that inherit from base classes are declared similarly to their parent class, however, we need to provide the name of parent classes within the parentheses.

Syntax

Following is the syntax of child class –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):

{sub class body}
```

2.4.1 Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance

1. **Single Inheritance**

This is the simplest form of inheritance where a child class inherits attributes and methods from only one parent class.

Example

The below example shows single inheritance concept in Python −

```
# parent class

class Parent:

def parentMethod(self):

print ("Calling parent method")


# child class

class Child(Parent):

def childMethod(self):

print ("Calling child method")


# instance of child

c = Child()

# calling method of child class

c.childMethod()

# calling method of parent class
```

```
c.parentMethod()
```

On running the above code, it will print the following result −

```
Calling child method
Calling parent method
```

2. Multiple Inheritance

Multiple inheritance in Python allows you to construct a class based on more than one parent classes. The Child class thus inherits the attributes and method from all parents. The child can override methods inherited from any parent.

Syntax

```
class parent1:
   #statements


class parent2:
   #statements


class child(parent1, parent2):
   #statements
```

3. Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class. There exists multiple layers of inheritance. We can imagine it as a grandparent-parent-child relationship.

```
# parent class
class Universe:
   def universeMethod(self):
```

```
    print ("I am in the Universe")


# child class
class Earth(Universe):
  def earthMethod(self):
    print ("I am on Earth")


# another child class
class India(Earth):
  def indianMethod(self):
    print ("I am in India")


# creating instance
person = India()
# method calls
person.universeMethod()
person.earthMethod()
person.indianMethod()
```

When we execute the above code, it will produce the following result −

```
I am in the Universe
I am on Earth
I am in India
```

2.5 Polymorphism

Polymorphism is defined as the circumstance of occurring in several forms. It refers to the usage of a single type entity (method, operator, or object) to represent several types in various contexts. Polymorphism is made from 2 words – '**poly**' and '**morphs**.' The word 'poly' means 'many' and 'morphs' means 'many forms.' Polymorphism, in a nutshell, means having multiple forms. To put it simply, polymorphism allows us to do the same activity in a variety of ways.

2.5.1 Ways of implementing Polymorphism in Python

- Operator Overloading
- Method Overriding
- Method Overloading

1) Method Overriding in Python

Polymorphism is most commonly associated with inheritance. In Python, child classes, like other programming languages, inherit methods and attributes from the parent class. Method Overriding is the process of redefining certain methods and attributes to fit the child class. This is especially handy when the method inherited from the parent class does not exactly fit the child class. In such circumstances, the method is re-implemented in the child class. Method Overriding refers to the technique of re-implementing a method in a child class.

2) Method Overloading in Python

Method overloading occurs when a class contains many methods with the same name. The types and amount of arguments passed by these overloaded methods vary. Python does not support method overloading or compile-time polymorphism. If there are multiple methods with the same name in a class or Python script, the method specified in the latter one will override the earlier one.

Python does not use function arguments in method signatures, hence method overloading is not supported.

3) Operator Overloading in Python

Operator overloading is another type of polymorphism in which the same operator performs various operations depending on the operands. Python allows for operator overloading.

- **Polymorphism in + operator:**

We already know that the '+' operator is frequently used in Python programs. However, it does not have a single application. When you wish to add two integers, concatenate two strings, or extend two lists, you use the same + sign. The + operator behaves differently depending on the type of object on which it is used.

Example

```
a = 10
b = 20
print('Addition of 2 numbers:', a + b)

str1 = 'Hello '
str2 = 'Python'
print('Concatenation of 2 strings:', str1 + str2)

list1 = [1, 2, 3]
list2 = ['A', 'B']
print('Concatenation of 2 lists:', list1 + list2)
```

Output

```
Addition of 2 numbers: 30
Concatenation of 2 strings: Hello Python
Concatenation of 2 lists: [1, 2, 3, 'A', 'B']
```

The + operator is used to execute arithmetic addition on integer data types. Similarly, the + operator is used to execute concatenation on string data types. In the case of lists, it returns a new list that contains elements from both original lists.

- **Polymorphism in * operator:**

The * operator is used to multiply 2 numbers if the data elements are numeric values. If one of the data types is a string, and the other is numeric, the string is printed that many times as that of the 2nd variable in the multiplication process.

Example

```
a = 10
b = 5
print('Multiplication of 2 numbers:', a * b)

num = 3
mystr = 'Python'
print('Multiplication of string:', num * mystr)
```

Output

```
Multiplication of 2 numbers: 50
Multiplication of string: PythonPythonPython
```

2.5.2 Benefits of Polymorphism

- **Code Reusability**: Polymorphism allows methods to be reused across different classes.
- **Flexibility and Extensibility**: It enables the design of flexible and extensible code that can work with new classes without modification.
- **Simplifies Code**: By allowing different classes to be treated uniformly through a common interface, polymorphism can simplify code structure and logic.