Unit-III

OOP Concepts

Procedural and Object-oriented Programming

Sr. No.	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
3.	It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
4.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
5.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

OOP Concepts

- Python is an object-oriented programming language, which means that it is based on principle of OOP concept.
- Class
- Object
- Method
- Inheritance
- Encapsulation
- Data Abstraction
- Polymorphism

Class

- A class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it.
- classes serve as a template to create objects
- class can be created by using the keyword class, followed by the class name and :
- Syntax –

```
    class ClassName:
        #statement_suite
```

Object

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

obj = ClassName()

Example

```
class demo:
    # A simple variables or attributes
     var1 = “hello"
      var2 = “welcome"
    # A sample method
    def fun(self):
        print(“Hi", self.var1)
        print(“Hi", self.var2)

# Object instantiation
obj = demo()
# Accessing class attributes and method through objects
print(obj.var1)
obj.fun()
```

SELF

- The self-parameter refers to the current instance of the class and accesses the class variables.
- We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

CONSTRUCTOR

- Constructors are generally used for instantiating an object.
- The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Python the **__init__()** method is called the constructor and is always called when an object is created.
- Syntax –

```
    def __init__(self):
        # body of the constructor
```

CONSTRUCTOR TYPES

- **Default constructor**: The default constructor is a simple constructor which doesn't accept any arguments.
- Its definition has **only one argument i.e self** which is a reference to the instance being constructed.
- Example –

```
class demo:
    # default constructor
    def __init__(self):
        self.str = “Welcome"
    def display(self):
        print(self.str)

# creating object of the class
obj = demo()
# calling the instance method using the object obj
obj.display()
```

Constructor types

- **Parameterized constructor:** constructor with parameters is known as parameterized constructor.
- The parameterized constructor takes its **first argument** as a reference to the instance being constructed known as **self** and the rest of the arguments are provided by the programmer.
- Example –

```
class demo:
    # parameterized constructor
    def __init__(self, n):
        self.name=n
    def display(self):
        print(self.name)

# creating object of the class
obj = demo("Rohan")
# calling the instance method using the object obj
obj.display()
```

DESTRUCTOR

- The __del__() method is a known as a destructor method in Python.
- It is called when all references to the object have been deleted i.e when an object is garbage collected.
- Example –

```
class Employee:
    # Initializing
    def __init__(self):
        print('Employee created.')
    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

PYTHON BUILT-IN CLASS FUNCTIONS

Sr. No.	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

```
class Student:
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
         self.age = age

 # creates the object of the class Student
s = Student("John", 101, 22)
  # prints the attribute name of the object s
print(getattr(s, 'name'))
  # reset the value of attribute age to 23
setattr(s, "age", 23)
 # prints the modified value of age
print(getattr(s, 'age'))
  # prints true if the student contains the attribute with name id
print(hasattr(s, 'id'))
# deletes the attribute age
delattr(s, 'age')
  # this will give an error since the attribute age has been deleted
print(s.age)
```

Built-in class attributes

Sr. No.	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age

    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))

s = Student("John",101,22)

print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

Summary

- 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.
- In Python can make an **instance variable private by adding double underscores before its name**.
- The members of a class that are declared protected are **only accessible to a class derived from it**. Data members of a class are declared **protected** by adding a **single underscore '_'** symbol before the data member of that class.

```
# super class                    //Example of protected variable
class Student:
        # protected data members
        _name = None
        _roll = None
        _branch = None

        # constructor
        def __init__(self, name, roll, branch):
                self._name = name
                self._roll = roll
                self._branch = branch
        # protected member function
        def _displayRollAndBranch(self):
        # accessing protected data members
                print("Roll: ", self._roll)
                print("Branch: ", self._branch)
```

```
# derived class
class std1(Student):
        # constructor
        def __init__(self, name, roll, branch):
                Student.__init__(self, name, roll, branch)
        # public member function
        def displayDetails(self):
        # accessing protected data members of super class
         print("Name: ", self._name)

        # accessing protected member functions of super class
        self._displayRollAndBranch()

# creating objects of the derived class
obj = std1("R2J", 1706256, "Information Technology")
# calling public member functions of the class
obj.displayDetails()
```

Example of protected variable

```
class C(object):
        def __init__(self):
                self.c = 21

                # d is private instance variable
                self.__d = 42
class D(C):
        def __init__(self):
                self.e = 84
                C.__init__(self)

object1 = D()
# produces an error as d is private instance variable
print(object1.c)
print(object1.__d)
```

INHERITANCE

- Inheritance is one of the most important features of object-oriented programming languages like.
- It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class.
- Inheritance allows you to inherit the properties of a class, i.e., **base class** to another, i.e., **derived class**.
- It provides the reusability of a code. We don’t have to write the same code again and again.

INHERITANCE

- Syntax -

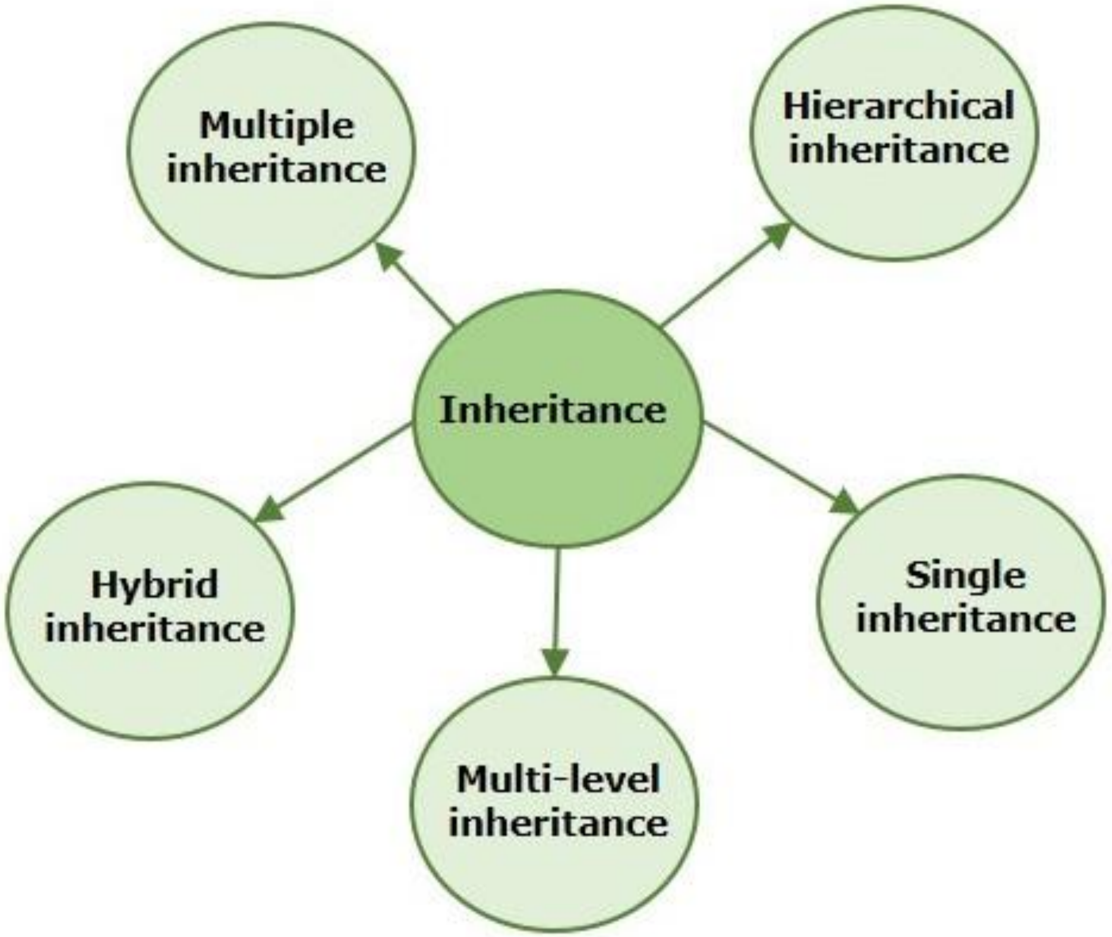
  Class BaseClass:

  {Body}

  Class DerivedClass(BaseClass):

  {Body}

Inheritance types



Inheritance types

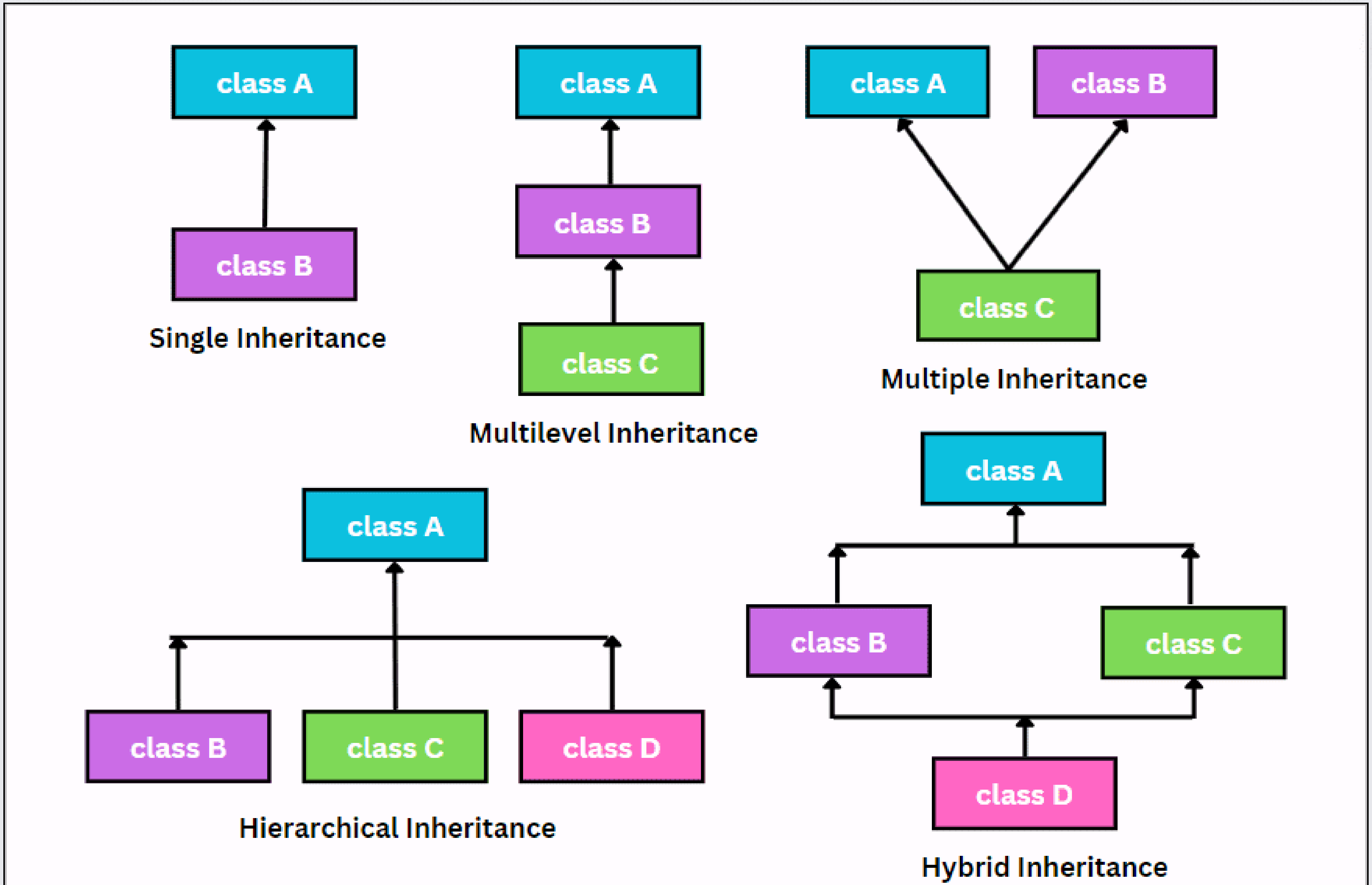


Fig: Types of inheritance in Python

Single Inheritance

```
# Here, we will create the base class or the Parent class
class A:
    def func_1(self):
        print ("This function is defined inside the parent class.")

# now, we will create the Derived class
class B(A):
    def func_2(self):
        print ("This function is defined inside the child class.")

object = B()
object.func_1()
object.func_2()
```

Multilevel Inheritance

```
class Base:
    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

# Inherited or Sub class
class Child(Base):

    # Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

    # To get name
    def getAge(self):
        return self.age
```

Multilevel Inheritance

```
# Inherited or Sub class

class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

    # To get address
    def getAddress(self):
        return self.address


g = GrandChild("Geek1", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())
```

Multiple Inheritance

```
# Here, we will create the Base class 1
class Mother1:
    mothername1 = ""
    def mother1(self):
        print(self.mothername1)

# Here, we will create the Base class 2
class Father1:
    fathername1 = ""
    def father1(self):
        print(self.fathername1)

# now, we will create the Derived class
class Son1(Mother1, Father1):
    def parents1(self):
        print ("Father name is :", self.fathername1)
        print ("Mother name is :", self.mothername1)

s1 = Son1()
s1.fathername1 = "Rajesh"
s1.mothername1 = "Shreya"
s1.parents1()
```

Hierarchical Inheritance

```
# Here, we will create the Base class
class Parent1:
    def func_1(self):
        print ("This function is defined inside the parent class.")
# Derived class1
class Child_1(Parent1):
    def func_2(self):
        print ("This function is defined inside the child 1.")
# Derivied class2
class Child_2(Parent1):
    def func_3(self):
        print ("This function is defined inside the child 2.")

object1 = Child_1()
object2 = Child_2()
object1.func_1()
object1.func_2()
object2.func_1()
object2.func_3()
```

THE SUPER( ) FUNCTION

- super() function is used to refer to the parent class or superclass.
- It allows you to **call methods defined in the superclass** from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

```
# parent class
class Person():
def __init__(self, name, age):
        self.name = name
        self.age = age

def display(self):
        print(self.name, self.age)

# child class
class Student(Person):
def __init__(self, name, age, dob):
        self.sName = name
        self.sAge = age
        self.dob = dob
        # inheriting the properties of parent class
        super().__init__("Rahul", age)

def displayInfo(self):
        print(self.sName, self.sAge, self.dob)

obj = Student("Mayank", 23, "16-03-2000")
obj.display()
obj.displayInfo()
```

Polymorphism

- Polymorphism is a very important concept in programming.
- The word polymorphism means having many forms.
- It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Function Polymorphism

- Same function name and different parameters and data types.

Example 1: Polymorphic len() function

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Function Polymorphism

Example 2: Polymorphic function

```
def add(x, y, z = 0):
    return x + y+z

print(add(2, 3))
print(add(2, 3, 4))
```

Class Polymorphism

- Same function name and different classes

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.type()
```

Method overriding

- Same function name and different classes

```
class Bird
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

obj_spr = sparrow()
obj_spr.intro()
obj_spr.flight()       //override base class Bird method flight
```

ABSTRACTION

- Abstraction is used to hide the internal functionality of the function from the users.
- The users only interact with the basic implementation of the function, but inner working is hidden.
- User is familiar with that **"what function does"** but they don't know **"how it does.“**
- In Python, abstraction can be achieved by using abstract classes and interfaces.
- A class that consists of one or more abstract method is called the abstract class.
- Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass.

Abstract Base Classes

- An abstract base class is the common application program of the interface for a set of subclasses.
- It can be used by the third-party, which will provide the implementations such as with plugins.
- It is also beneficial when work with the large code-base hard to remember all the classes.
- Unlike the other high-level language, Python doesn't provide the abstract class itself.
- Need to import the abc module, which provides the base for defining Abstract Base classes (ABC).

- Abstraction classes are meant to be the blueprint of the other class.
- An abstract class can be useful while designing large functions.
- An abstract class is also helpful to provide the standard interface for different implementations of components.
- Python provides the **abc** module to use the abstraction in the Python program.

*from abc **import** ABC*

***class** ClassName(ABC):*

import the ABC class from the **abc** module.

- The ABC works by decorating methods of the base class as abstract.
- It registers concrete classes as the implementation of the abstract base.
- Use the ***@abstractmethod*** decorator (Decorators are used to modify the behavior of functions or methods) to define an abstract method or if you don't provide the definition to the method, it automatically becomes the abstract method.
- An Abstract class can contain the both method normal and abstract method.
- An Abstract cannot be instantiated; cannot create objects for the abstract class.

```
from abc import ABC, abstractmethod
class Car(ABC):   //abstract class
    def mileage(self):    //abstract method
        pass
  class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")

t= Tesla()
t.mileage()
s = Suzuki()
s.mileage()
```