Python Merged Notes

Unit – 1,2,3

Classes and Objects

Introduction

Problem solving using Advance Python

B.Tech 2nd semester

Unit 1

NIET Greater Noida

GET FUTURE READY

Content

- Introduction: Python classes and objects
- User-defined classes
- Encapsulation
- Data hiding
- Class variable and Instance variables
- Instance methods
- Class methods
- Static methods
- Constructor in python
- Parametrized constructor
- Magic methods in python
- Object as an argument
- Instances as Return Values, namespaces

Introduction : OOPs concept

- Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviours are bundled into individual objects.
- object-oriented programming is an approach for modelling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on.
- In this approach , data and functions are combined to form a class.

For example:

Object	Data or attribute	Functions/methods
Person	Name, age ,gender	Speak(),walk() etc.
Polygon	Vertices ,border ,color	Draw(),erase() etc.
Computer	Brand, resolution,price	Display(),printing() etc.

Introduction : Python Classes and Objects

Python Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- A class creates a new type and object is an instance of the class.
- The python library is based on the concept of classes and objects.

Introduction : Python Classes and Objects

Create a Class

•Starts with a keyword class followed by the class_name and a colon:

•Similar to function definition.

Syntax:

```
class  class_name:
        <statement-1>
         <statement-2>
        ……
         <statement-n>
```

Example:

Create a class named MyClass, with a property named x:

```
class MyClass:
        x = 5
print(MyClass)
```

Output:

```
<class '__main__.MyClass'>
```

Introduction : Python Classes and Objects

Create Object:

- Creating an object or instance of a class is known as class instantiation
- We can use the class name to create objects:

Syntax:

```
Object_name =class_name()
# using this syntax ,an empty object of a class is created.
```

Accessing a class member:

- The object can access class variables and class methods using dot(.) operator.

Example:

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

Output:

```
5
```

Introduction : Python Classes and Objects

The __init__() Method:

- The example above are classes and objects in their simplest form, and are not really useful in real life applications.
- All classes have a method called __init__(), which is always executed when the class is being initiated.
- Use the __init__() Method to assign values to object properties, or other operations that are necessary to do when the object is being created:
- The __init__() Method is called automatically every time the class is being used to create a new object.
- The __init__() method is useful to initialize the variables of the class object.

Introduction : Python Classes and Objects

Example:

Create a class named Person, use the __init__() method to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Output:

```
John
36
```

Another way:

```
class Person:
    def __init__(self, name, age):
        print("In class method")
        self.name = name
        self.age = age
        print("the values are",name,age)
p1=Person("john",36)
```

Output:

```
In class method
The values are:
John 36
```

Class Variables and Instance Variables

Variables are essentially symbols that stand in for a value we're using in a program. Object-oriented programming allows for variables to be used at the class level or the instance level.

Class Variables

- Declared inside the class definition (but outside any of the instance methods).
- They are not tied to any particular object of the class, hence shared across all the objects of the class.
- Modifying a class variable affects all objects instance at the same time.

Class Variables and Instance Variables

Instance Variable or object variable —

- Object variable or instance variable owned by each object created for a class.
- Declared inside the constructor method of class (the __init__ method).
- They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.
- The object variable is not shared between objects

Class Variables and Instance Variables

```
class Car:
        wheels = 4    # <- Class variable
        def __init__(self, name):
            self.name = name    # <- Instance variable
```

Above is the basic, no-frills *Car* class defined. Each instance of it will have class variable *wheels* along with the instance variable *name*.

Let's instantiate the *class* to access the variables:

```
mercedes=Car("mercedes")
print(mercedes.wheel)# access of class variable through object
print(Car.wheel) )# access of class variable through class
print(mercedes.name) )# access of instance variable through object
```

Difference between Class Variables and Instance Variables

- Class variable:

1. Class variable owned by class.
2. All the objects of the class will share the class variable.
3. Any change made to the class variable will be reflected in all other objects.

- Instance variable:

1. Instance variable owned by object.
2. The instance variable is not shared between objects.
3. Any change made to the Instance variable will not be reflected in all other objects.

Important point:

1. **Generally , Class variable is used to define constant with a particular class or provide default attribute.**
2. **Another use of class variable is to count the number of objects created.**

Program to demonstrate the use of class or static variable

```
class Fruits(object):
    count = 0
def __init__(self, name, count):
    self.name = name
    self.count = count
Fruits.count = Fruits.count + count

def main():
    apples = Fruits("apples", 3);
    pears = Fruits("pears", 4);
    print (apples.count)
    print (pears.count)
    print (Fruits.count)
    print (apples.__class__.count) # This is Fruit.count
    print (type(pears).count) # So is this
if __name__ == '__main__':
main()
```

Output:

3
4
7
7
7

Demonstrate the use of variable defined on the class level

```
class example:
        staticVariable = 9 # Access through class

print (example.staticVariable) # Gives 9

#Access through an instance
instance = example()
print(instance.staticVariable) #Again gives 9

#Change within an instance
instance.staticVariable = 12
print(instance.staticVariable) # Gives 12
print(example.staticVariable) #Gives 9
```

Output:

9
9
12
9

Introduction : Python Classes and Objects(Self Parameter)

The self Parameter or argument:

- The self argument refers to the object itself.
- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.
- We can have .__init__() any number of parameters, but the first parameter will always be a variable called *self.* When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new **attributes** can be defined on the object.

Introduction : Python Classes and Objects(Self Parameter)

•Creating object using self argument:

Example:

```
Class Myclass:
        a=10
        def func(self):
                return “using self”
# instance an object
ob=Myclass()
print(ob.func()) # output : using self
print(Myclass.func(ob)) # output : using self
```

Introduction : Python Classes and Objects

Example:

Use the words mysillyobject and abc instead of self:

```
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()
```

Output:

Hello my name is John

User Defined Classes or problems:

Q1.program modifying a mutable type attributes:

```
Class Number:
        evens=[]
        odd=[]
        def __init__(self,num):
                self.num=num
                if num%2==0:
                        Number.evens.append(num)
                else:
                        Number.odds.append(num)
N1=Number(21)
N1=Number(32)
N1=Number(43)
N1=Number(54)
N1=Number(65)
print("even Numbers are:",Number.evens)
print("odd Numbers are:",Number.odds)
```

Output:
Even Numbers are: [32,54]
Odd Numbers are: [21,43,65]

Python Encapsulation

Python 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 __.

Python Encapsulation

Example : 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()
```

Python Encapsulation

Example : Data Encapsulation in Python

```
# 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
```

Python Encapsulation

Example : Data Encapsulation in Python

- In the above program, we defined a Computer class.
- We used __init__() method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes.
- As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

Data Hiding

Data Hiding:

- An object's attributes may or may not be visible outside the class definition**.**
- In Python, we use double underscore before the attributes name to make them inaccessible/private or to hide them.
- The attributes with prefix double underscore not visible outside the class.

Data Hiding

Example:

```
class MyClass:
    __hiddenVar = 12
    def add(self, increment):
        self.__hiddenVar += increment
        print (self.__hiddenVar)
myObject = MyClass()
myObject.add(3)
myObject.add (8)
print(Myobject.__hiddenVar)# Error as not accessible outside
print (myObject._MyClass__hiddenVar)
```

Output:

```
15
23
23
```

Data Hiding

Example:

```
class Car:
    __maxspeed = 0
    __name = ""
      def __init__(self):
           self.__maxspeed = 200
          self.__name = "Supercar"
      def drive(self):
          print('driving. maxspeed ' + str(self.__maxspeed))
redcar = Car()
redcar.drive()
redcar.__maxspeed = 10  # will not change variable because its private
redcar.drive()
```

Note: To change the value of a private variable, a setter method is used

Data Hiding

Example:

```
class Car:
    __maxspeed = 0
    __name = ""
      def __init__(self):
           self.__maxspeed = 200
          self.__name = "Supercar"
      def drive(self):
          print('driving. maxspeed ' + str(self.__maxspeed))
      def setMaxSpeed(self,speed):
          self.__maxspeed = speed
redcar = Car()
redcar.drive()
redcar.setMaxspeed = 320  # will change variable
redcar.drive()
```

Introduction to Methods:

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Example:

Insert a function that prints a greeting, and execute it on the p1 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()
```

Output:

```
Hello my name is John
```

Instance methods, Class method, static methods

Python's data model, Python offers three types of methods namely *instance*, *class* and *static methods*.

Instance or object methods:

- They are most widely used methods.
- Instance method receives the instance of the class as the first argument, which by convention is called self, and points to the instance of class
- However it can take any number of arguments. Using the self parameter, we can access the other attributes and methods on the same object and can change the object state.
- Also, using the self.__class__ attribute, we can access the class attributes, and can change the class state as well.
- **Therefore, instance methods gives us control of changing the object as well as the class state.**
- A built-in example of an instance method is str.upper()

```
>>> "welcome".upper() #  called on the str object
'WELCOME'
```

Example :Instance Methods

Q. Write a program to deposit or withdraw money in a bank account.

```
Class Account:
    def __init__(self):
        self.balance=0
        print("New Account Created")
    def deposit(self):
        amount=float(input("enter amount to deposit:"))
        self.balance+=amount
    def withdraw(self):
        amount=float(input("enter amount to withdraw:"))
        if amount> self.balance:
            print("Insufficient balance")
        else:
            self.balance-=amount
            print(New Balance", self.balance)
    def enquiry(self):
        print("Balance",self.balance)
```

```
account=Account()
account.deposit()
account.withdraw()
account.enquiry()
```

Output:

New Account created
enter amount to deposit: 1000
New Balance:1000.000000
enter amount to withdraw:25.23
New Balance:974.770000
Balance:974.770000

Decorators Before :Class and static Methods

- Decorators are very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class.
- Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.
- Before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

Decorators Before :Class and static Methods

```
# defining a decorator
def hello_decorator(func):
    # inner1 is a Wrapper function in
    # which the argument is called
    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")
        # calling the actual function now
        # inside the wrapper function.
        func()
        print("This is after function execution")
    return inner1
```

```
# defining a function, to be called inside wrapper

def function_to_be_used():
        print("This is inside the function !!")

# passing 'function_to_be_used' inside the
# decorator to control its behavior

function_to_be_used=hello_decorator(function_to_be_used)

# calling the function

function_to_be_used()
```

Decorators Before :Class and static Methods

```
def hello_decorator(func):

    def inner1():
        print("Hello, this is before function execution")

        func()

        print("This is after function execution")
    return inner1


def function_to_be_used():
    print("This is inside the function!!")


function_to_be_used = hello_decorator(function_to_be_used)

function_to_be_used()
```

step 2 step 3 step 4 step 1 step 5

```
def hello_decorator(func):

    def inner1():
        print("Hello, this is before function execution")

        func()

        print("This is after function execution")
    return inner1


def function_to_be_used():
    print("This is inside the function!!")


function_to_be_used = hello_decorator(function_to_be_used)

function_to_be_used()
```

step 6 step 7 step 8 step 11 step 9 step 10 step 12

Decorators Before :Class and static Methods

```
# defining a decorator
def hello_decorator(func):
        def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")
    return inner1
```

```
# defining a function, to be called inside wrapper

@hello_decorator
def function_to_be_used():
        print("This is inside the function !!")

# passing 'function_to_be_used' inside the
# decorator to control its behavior


# calling the function

function_to_be_used()
```

Class Methods

- A class method accepts the class as an argument to it which by convention is called cls. It take the cls parameter, which points to the class instead of the object of it. It is declared with the @classmethod decorator.
- Class methods are bound to the class and not to the object of the class. **They can alter the class state that would apply across all instances of class but not the object state.**

Note: 1.class methods are called by class .

2. First argument of the class method cls , not the self

- Syntax for Class Method.

```
class my_class:
  @classmethod
 def function_name(cls, arguments):
    #Function Body
    return value
```

Class Methods

#Normal Calling

```
class Rectangle:
        def __init__(self,length,breadth):
                self.length=length
                self.breadth=breadth
        def area(self):
                return self.length * self.breadth
        def Square(cls,side):
                return cls(side,side)
Rectangle.Square=classmethod(Rectangle.Square)
S=Rectangle.Square(10)
print("Area=",S.area())
```

Output:

Area=100

 # Using Decorators

```
class Rectangle:
        def __init__(self,length,breadth):
                self.length=length
                self.breadth=breadth
        def area(self):
                return self.length * self.breadth
        @classmethod
        def Square(cls,side):
                return cls(side,side)
S=Rectangle.Square(10)
print("Area=",S.area())
```

Output:

Area=100

Static Methods

- A static method is marked with a @staticmethod decorator to flag it as *static*.
- It does not receive an implicit first argument (neither self nor cls).
- It can also be put as a method that "does't know its class".
  Hence a static method is merely attached for convenience to the class object.
- A static method can be called either on the class or an instance.
- **Hence static methods can neither modify the object state nor class state. They are primarily a way to namespace our methods.**
- Syntax for Static Method

```
class my_class:
    @staticmethod
    def function_name(arguments):
        #Function Body
        return value
```

Static Methods

```
class Choice:
        def __init__(self, subjects):
                self.subjects=subjects
        @static method
        def validate_subject(subjects):
                if "CSA" in subjects:
                        print("This option is no longer available")
                else:
                        return True
Subjects=["DS","CSA","Foc","OS","TOC"]
if all(Choice.validate_subjects(i) for I in subjects):
    ch=Choice(subjects)
    print("You have been alloted the subjects:",subjects)
```

OUTPUT:

This option is no longer available.

NIET Greater Noida

GET FUTURE READY

What are the differences between Classmethod and StaticMehtod?

Class Method	Static Method
The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Class method can access and modify the class state.	Static Method cannot access or modify the class state.
The class method takes the class as parameter to know about the state of that class.	Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.
@classmethod decorator is used here.	@staticmethod decorator is used here.

Constructor (CO1)

- 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 class is created.
- In Python the **__init__()** method is called the constructor and is always called when an object is created.
- def __init__(self):

  # body of the constructor

Constructor (CO1)

Types of constructors :

Default constructor :The default constructor is simple constructor which doesn't accept any arguments.

It's definition has only one argument which is a reference to the instance being constructed.

Parameterized constructor :Constructor with parameters is known as parameterized constructor.

The parameterized constructor take 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.

Constructor (CO1)

```
class default_Constructor:
# default constructor
    def __init__(self):
        self.dC = “Hello Python Default Constructor"
# a method for printing data members
    def print_ default_Constructor(self):
        print(self.dC)
# creating object of the class
obj = default_Constructor()
# calling the instance method using the object obj
obj. print_ default_Constructor()
```

Constructor (CO1)

```
class Addition:
    first = 0
    second = 0
    answer = 0
# parameterized constructor
    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of two numbers = " +
str(self.answer))
    def calculate(self):
        self.answer = self.first + self.second

# creating object of the class
# this will invoke parameterized constructor
obj = Addition(1000, 2000)

# perform Addition
obj.calculate()

# display result
obj.display()
```

Magic Methods

- Magic methods in Python are the special methods that start and end with the double underscores.
- They are also called **dunder** methods.
- Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.

For example, when you add two numbers using the + operator, internally, the **__add__()** method will be called.

Magic Methods

- Built-in classes in Python define many magic methods.
- Use the **dir()** function to see the number of magic methods inherited by a class.

For example, the following lists all the attributes and methods defined in the int class.

```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Magic Methods

the __add__ method is a magic method which gets called when we add two numbers using the + operator.

Consider the following example.

```
>>> num=10
>>> num + 5
15
>>> num.__add__(5)
15
```

Magic Methods

__new__() method

Languages such as Java and C# use the new operator to create a new instance of a class. In Python the __new__() magic method is implicitly called before the __init__() method. The __new__() method returns a new object, which is then initialized by __init__().

```
class Employee:
    def __new__(cls):
        print ("__new__ magic method is called")
        inst = object.__new__(cls)
                return inst
    def __init__(self):
        print ("__init__ magic method is called")
        self.name=Python'
```

Magic Methods

__str__() method

- Another useful magic method is __str__().
- It is overridden to return a printable string representation of any user defined class.
- We have seen str() built-in function which returns a string from the object parameter.

For example, str(12) returns '12'. When invoked, it calls the __str__() method in the int class.

Magic Methods

```
>>> num=12
>>> str(num)
'12'
>>> #This is equivalent to
>>> int.__str__(num)
'12’
```

Magic Methods

Override the __str__() method in the Employee class to return a string representation of its object.

```
class Employee:
    def __init__(self):
        self.name='Swati'
        self.salary=10000
    def __str__(self):
        return 'name='+self.name+' salary=$'+str(self.salary)
```

Magic Methods

See how the str() function internally calls the __str__() method defined in the Employee class.

This is why it is called a magic method!

```
>>> e1=Employee()
>>> print(e1)
name=Swati salary=$10000
```

Magic Methods

__add__() method

In following example, a class named distance is defined with two instance attributes - ft and inch.

The addition of these two distance objects is desired to be performed using the overloading + operator.

To achieve this, the magic method __add__() is overridden, which performs the addition of the ft and inch attributes of the two objects.

The __str__() method returns the object's string representation.

Magic Methods

```
class distance:
    def __init__(self, x=None,y=None):
        self.ft=x
        self.inch=y
    def __add__(self,x):
        temp=distance()
        temp.ft=self.ft+x.ft
        temp.inch=self.inch+x.inch
        if temp.inch>=12:
            temp.ft+=1
            temp.inch-=12
            return temp
    def __str__(self):
        return 'ft:'+str(self.ft)+' in: '+str(self.inch)
```

```
>>> d1=distance(3,10)
>>> d2=distance(4,4)
>>> print("d1= {} d2={}".format(d1, d2))
d1= ft:3 in: 10 d2=ft:4 in: 4
>>> d3=d1+d2
>>> print(d3)
ft:8 in: 2
```

Magic Methods

__ge__() method

The following method is added in the distance class to overload the >= operator.

```
class distance:
    def __init__(self, x=None,y=None):
        self.ft=x
        self.inch=y
    def __ge__(self, x):
        val1=self.ft*12+self.inch
        val2=x.ft*12+x.inch
        if val1>=val2:
            return True
        else:
            return False
```

Magic Methods

This method gets invoked when the >= operator is used and returns True or False. Accordingly, the appropriate message can be displayed.

```
>>> d1=distance(2,1)
>>> d2=distance(4,10)
>>> d1>=d2
False
```

Namespaces in Python (CO1)

- It is combination of name (refers to the name of object) and space (location from where object is accessed).
- A namespace is a collection of currently defined symbolic names along with information about the object that each name references.
- A namespace can be as a dictionary in which the keys are the object names, and the values are the objects themselves.
- Each key-value pair maps a name to its corresponding object.

In a Python program, there are four types of namespaces:

  - Built-In
  - Global
  - Enclosing
  - Local

Namespaces in Python (CO1)

1. The Built-in namespace:
   - The **built-in namespace** contains the names of all of Python's built-in objects.
   - These are always available when Python is running.
   - we can list the objects in the built-in namespace with the following command:

```
>>> dir(__builtins__)
```

Note:

1. **The Python interpreter creates the built-in namespace when it starts up.**
2. **This namespace remains in existence until the interpreter terminates.**

Namespaces in Python (CO1)

2. The Global namespace:

- The **global namespace** contains any names defined at the level of the main program.
- Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

```
x = 10                  # x has global namespace
def myfun():
    print(x)
myfun()
```

Namespaces in Python (CO1)

- Any variable defined inside a function using global keyword has also global namespace.

```
#Example
x = 10
def myfun():
    global x    # defined x using global keyword
    x = 12      # access the global variable, does not creates the local variable
    print("Inside function definition",x)
myfun()
print("Outside function definition",x)
```

Output

```
Inside function definition 12
Outside function definition 12
```

Namespaces in Python (CO1)

3. Local namespace
   - The **local namespace** contains any names defined at the block or function level of the main program.
   - Python creates the local namespace when the execution of function body starts and terminates after the execution of body of function.

```
#Example
x = 10
def myfun():
    x = 12        # Creates the local variable x
    print("Inside function definition",x)
myfun()
print("Outside function definition",x)
```

Output

```
Inside function definition 12
Outside function definition 10
```

Namespaces in Python (CO1)

4. Enclosing namespace

- It refers the to the definition of inner() function and other statement nested inside the definition of outer() function.
- It occurs only inside the nested function.
- inner() function can access the enclosing namespace variable (also known as the non-local variable with respect to the inner() function) but can't modify it.
- If inner() function try to modify the enclosing variable, then a local variable is declared inside the inner() function namespace.

Namespaces in Python (CO1)

```
# Example of enclosing namespace
x = 10
def outer():
    x = 12          # Enclosing namespace starts here, x is defined inside the enclosing namespace
    def inner():
        print("Inside the inner function", x)   # It refers to the enclosing namespace variable
    inner()
    print("Inside outer definition", x)   # Enclosing namespace ends here
outer()
print("Outside function definition", x)
```

Output

```
Inside function definition 12
Inside outer definition 12
Outside function definition 10
```

Namespaces in Python (CO1)

```
# Example of inner() trying to modify the enclosing variable
x = 10
def outer():
    x = 12          # Enclosing namespace starts here, x is defined inside the enclosing namespace
    def inner():
        x = 7       # x now declared as local variable of inner
        print("Inside the inner function", x)   # It refers to the enclosing namespace variable
    inner()
    print("Inside outer definition", x)   # Enclosing namespace ends here
outer()
print("Outside function definition", x)
```

Output

```
Inside function definition 7
Inside outer definition 12
Outside function definition 10
```

Namespaces in Python (CO1)

```
# Example of inner() function modifying the enclosing variable using non-local keyword.
x = 10                  # x is declared in global namespace
def outer():
    x = 12              # Enclosing namespace starts here, x is defined inside the enclosing namespace
    def inner():
        nonlocal x      # x is defined using global keyword
        x = 7           # x now refers to enclosing variable of outer() function
        print("Inside the inner function", x)   # It refers to the enclosing namespace variable
    inner()
    print("Inside outer definition", x)   # Enclosing namespace ends here
outer()
print("Outside function definition", x)
```

Output

```
Inside function definition 7
Inside outer definition 7
Outside function definition 10
```

Namespaces in Python (CO1)

Namespace resolution rules.

1. First the environment looks for the variable (object) in local namespace.
2. If variable is not declared inside the local namespace, then it looks for the variable defined inside the enclosing namespace.
3. If variable is not declared inside the enclosing namespace, then it looks for the variable defined inside the global namespace.
4. If variable is not defined inside the global namespace, then it looks for the variable (object) in builtins namespace.
5. Finally, If variable is not present in builtins namespace, then “NameError” exception is raised by the environment.

Example: Namespaces (CO1)

```
a = 'global a'
y = 'global y'

def test_namespace():
    a = 'enclosing a'

    def inner_namespace():
        a = 'local a'
        print(a)
        print(y)

    inner_namespace()

    print(a)

test_namespace()

print(a)
```

Global

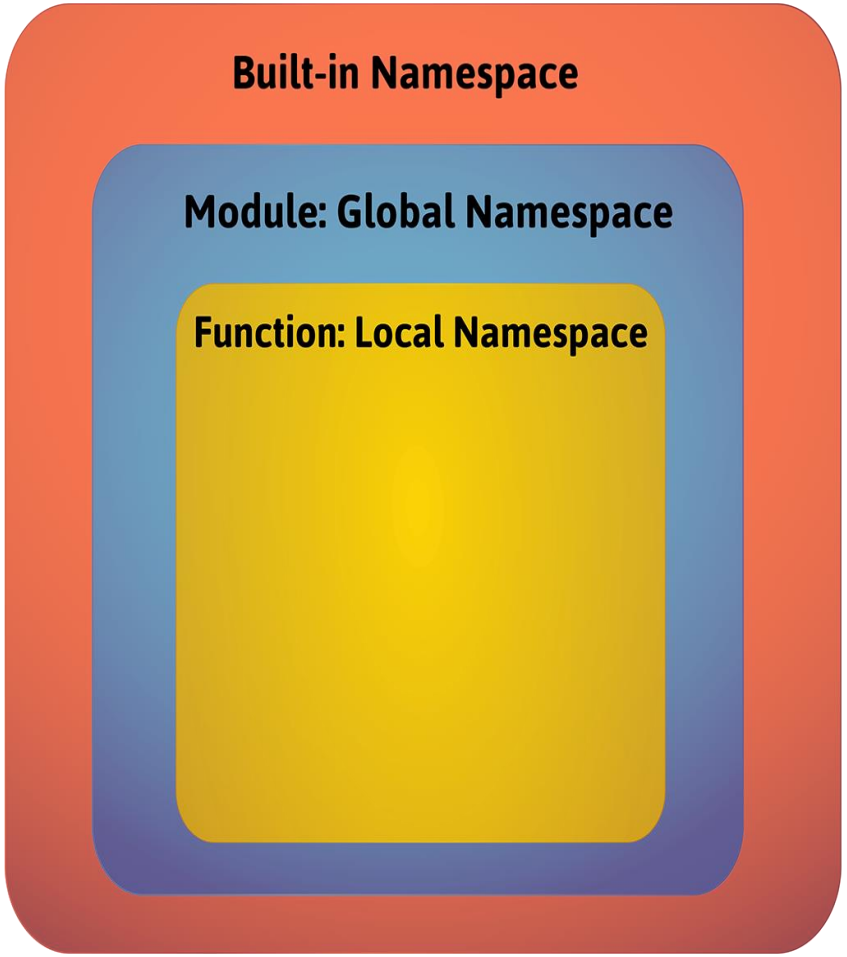
Enclosing

Local

```
local a
global y
enclosing a
global a
```

Namespace in Python

- When we start the interpreter, a python namespace is created for as long as we don’t exit. This holds all built-in names. Because of which python functions like print() and id() are always available
- Each module creates its own global namespace in python. So is the main module .
- When we call a function, a local python namespace is created for all the names in the function.



Variable Scope

- A variable scope is the region in a program where the variable is available for use.
- A namespace is in variable scope in a part of a program, if it lets you access the python namespace without having to use a prefix.
- At any instant, we have at least three nested python scopes:
  - Current function's variable scope- has local names
  - Module's variable scope- has global names
  - The outermost variable scope- has built-in names

Variable Scope

- **Local Scope:**

  A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

- **Enclosed (Nonlocal)Scope:**

  A nested function creates a nested variable scope inside the outer function's scope.

  It can be modified in inner function through nonlocal keyword

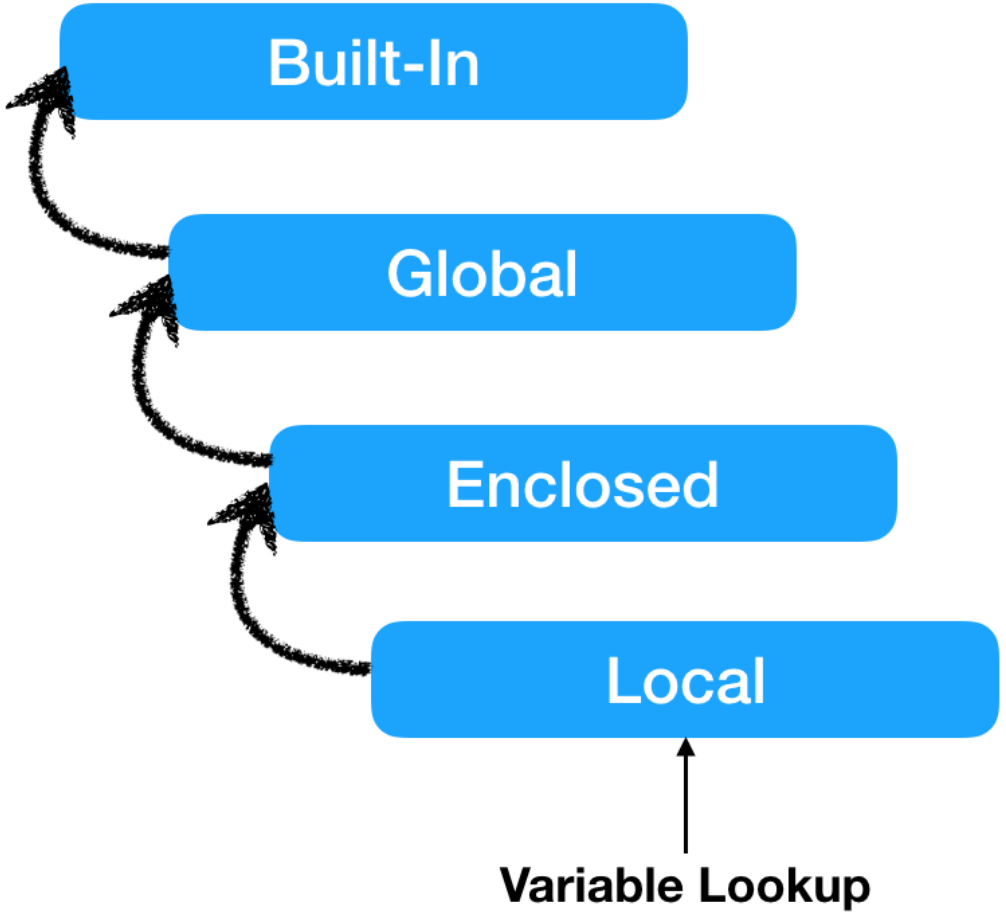
- **Global Scope:**

  A variable created in the main body (module)of the Python code is a global variable and belongs to the global scope. Global variables are available from within any scope, global and local. global keyword is used to modify it inside a function.

- **Built-in Scope:**

  Built-in scope is a special Python scope that's created or loaded whenever you run a script or open python interpreter such variables are accessible from anywhere in the program .

Order in Which Python will look up for a Variable Scope



Garbage Collection (CO1)

- It is the process by Python periodically reclaims unwanted memory.
- Garbage collection in Python is automatic i.e. it deletes all the objects that are not needed or have gone out of scope to free the memory space.
- It runs in the background during the program execution.
- It immediately reclaims its memory as soon as the object's reference count reaches to zero.

Garbage Collection (CO1)

- Object's reference count increases when it is created, and its aliases are created.
- That is, when object is assigned a new name, or it is referenced within in a list, tuple, or other data structure.
- The object's reference count decreases when it is being assigned to some other reference or its reference goes out of scope.
- The object' reference count can be decreased manually by deleting it with the del command.

Garbage Collection (CO1)

```
# Example
A = 100        # Creates object A
B = A          # Object's reference count increases by 1-object assigned
C = [1,2,B]    # Object's reference count increases by 1-object used in list
B = 200        # Object's reference count decreases by 1-Reassignment of B
C[2] = 3       # Object's reference count decreases by 1- not used in list
del A          # Object's reference count is zero – object removed from memory
```

Object Oriented Concepts

Unit: 2

Problem Solving Using Advanced Python

Course Details
(B Tech 2nd Sem /1st Year)

CONTENTS

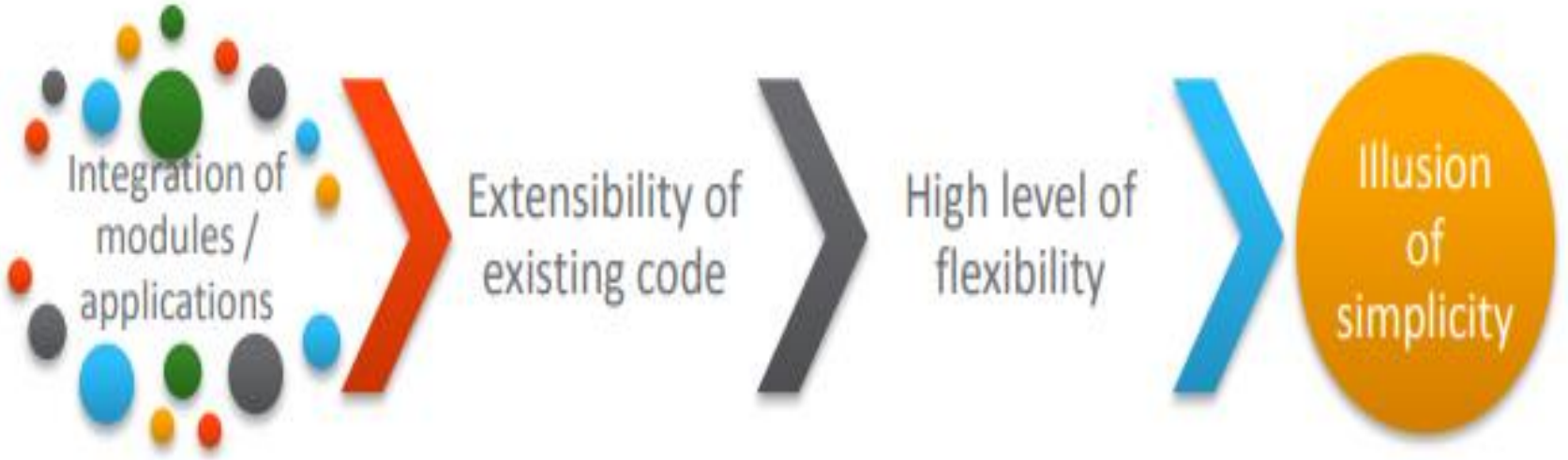
- Introduction to the Specialization
- Inheritance
- Types of Inheritance
- Invoking the parent Class Method
- Method Overriding
- Abstract Class
- MRO & Super()
- Polymorphism
- Introspection: Introspection types , Introspection objects
- Introspection scopes, Inspect modules, Introspect tools

Introduction to the Specialization(CO2)

- Python is used in Data Science Field.
- Python is implemented in IOT Field.
- Python Specialization in Artificial Intelligence tools.
- The python tools is also used in web Designing ,web crawling.
- Python is easily implemented in Machine Learning.
- We can specialized in Python Data Science tools.
- It is mostly used in Cloud Computing platform also.
- The specialization of Python is storing Data.
- The Specialization of Python is in Libraries of python.

Need for Object Oriented Approach (CO2)

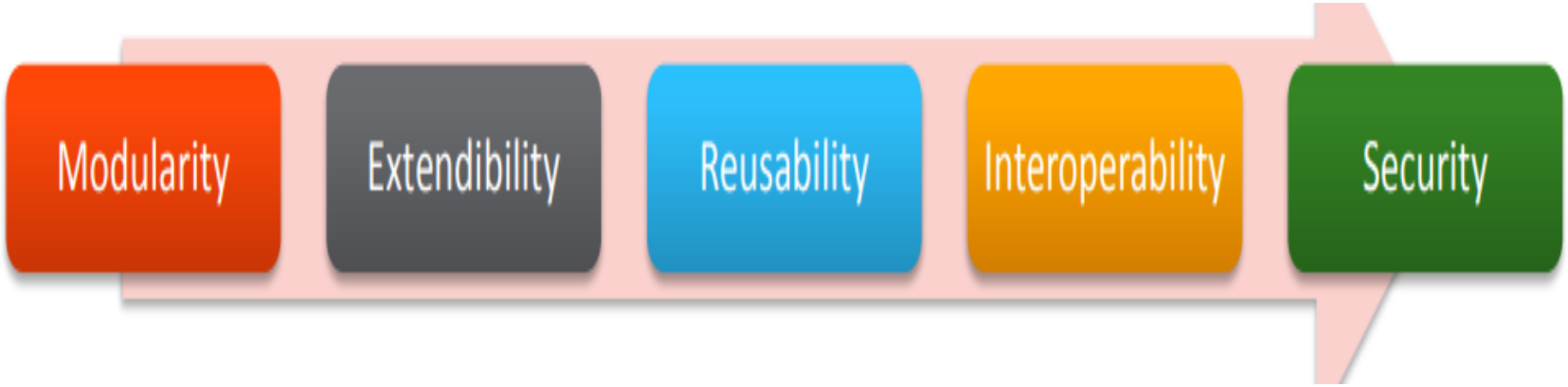
- Challenges in developing a business application.



- If these challenges are not addressed, it may lead to software crisis.

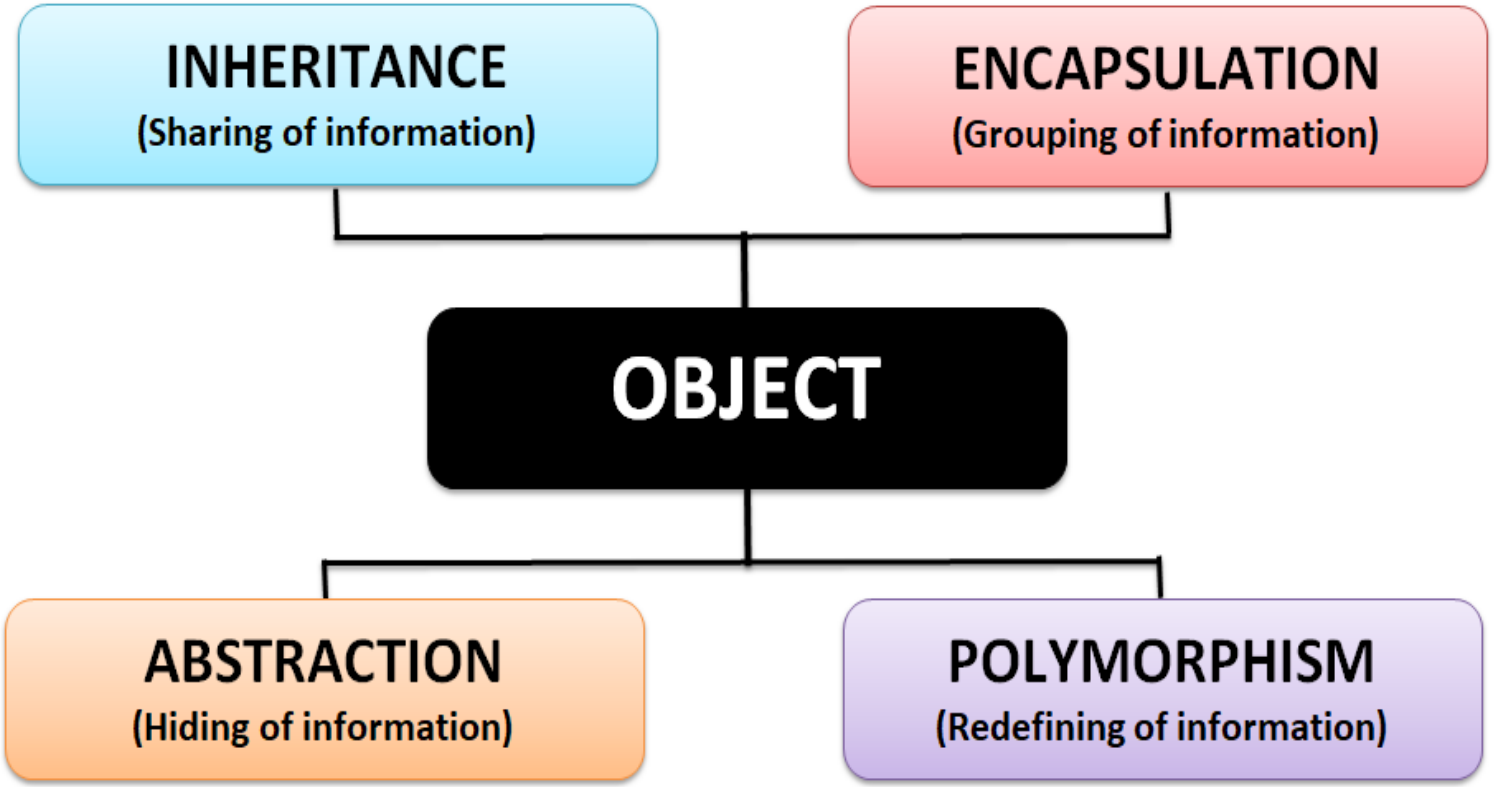
Need for Object Oriented Approach (CO2)

- Features needed in the business application to meet these challenges.



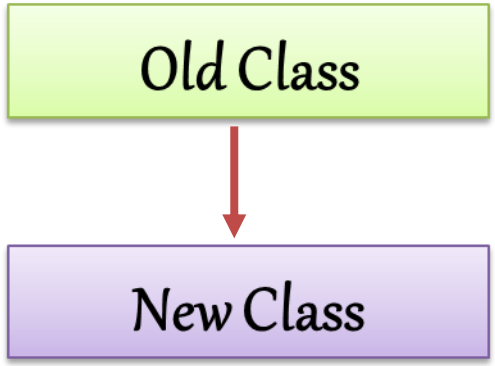
- Challenges can be addressed using object-oriented approach.

Pillars of OOPS (CO2)



Inheritance (CO2)

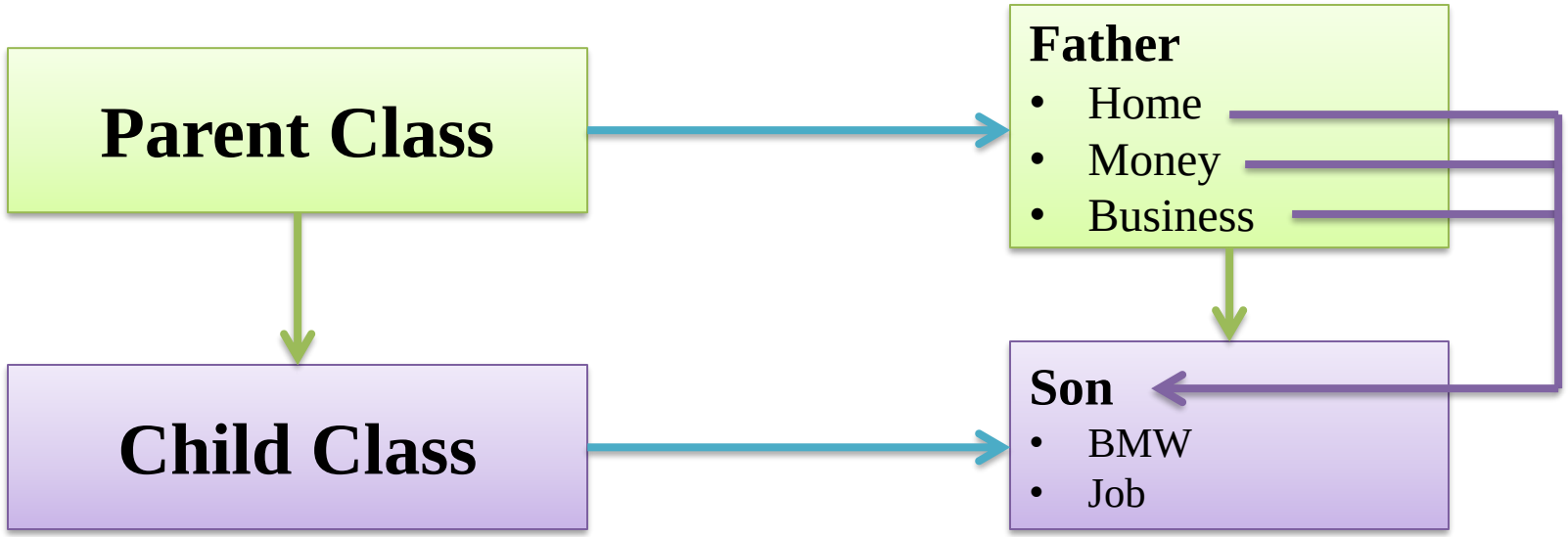
- The mechanism of deriving a new class from an old one (existing class) such that the new class inherit all the members (variables and methods) of old class is called inheritance or derivation.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.



Super Class and Sub Class (CO2)

The old class is referred to as the Super class and the new one is called the Sub class.

- Parent Class - Base Class or Super Class
- Child Class - Derived Class or Sub Class



Advantage of Inheritance (CO2)

- All classes in python are built from a single super class called ‘object’ so whenever we create a class in python, object will become super class for them internally.

```
class Mobile(object):
class Mobile:
```

- The main advantage of inheritance is code reusability.
- Python also allows the classes to inherit commonly used attributes and methods from other classes through inheritance.
- 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.

Type of Inheritance (CO2)

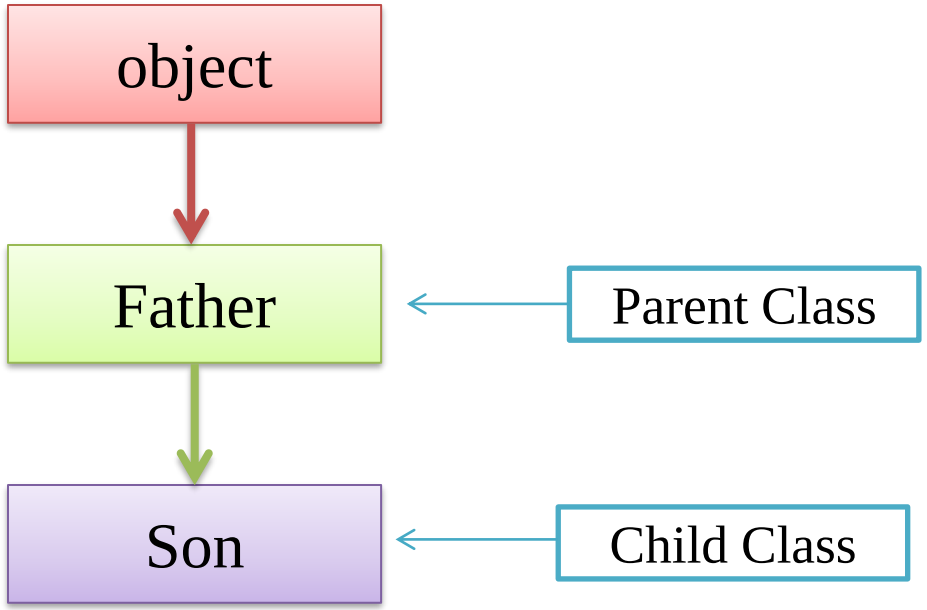
- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Multiple Inheritance

Declaration of Child Class(CO2)

```
class ChildClassName (ParentClassName) :
        members of Child class

class Mobile (object) :
        members of Child class

class Mobile :
        members of Child class
```

Single Inheritance(CO2)

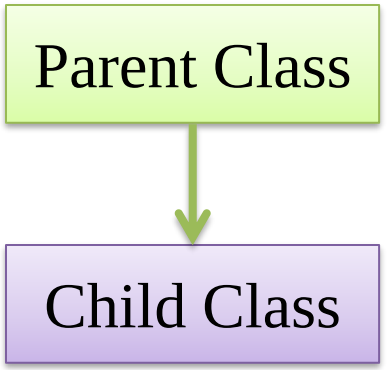
If a class is derived from one base class (Parent Class), it is called Single Inheritance.



Single Inheritance Syntax(CO2)

Syntax:-

```
class ParentClassName(object):
        members of Parent Class

class ChildClassName(ParentClassName):
        members of Child Class
```

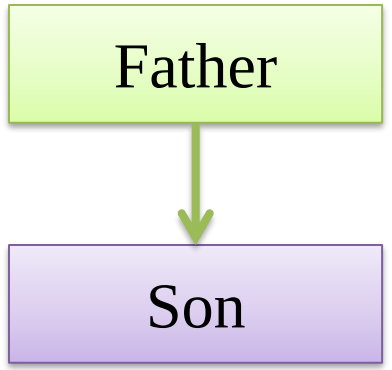


Example:-

```
class Father:
        members of class Father

class Son (Father):
        members of class Son
```

Parent Class

Child Class



Single Inheritance(CO2)

- We can access Parent Class Variables and Methods using Child Class Object
- We can also access Parent Class Variables and Methods using Parent Class Object
- We can not access Child Class Variables and Methods using Parent Class Object

Single Inheritance Program (CO2)

```
class Father:                                          # Parent Class
        money = 1000

        def show(self):
                print("Parent Class Instance Method")

        @classmethod
        def showmoney(cls):
                print("Parent Class Class Method:", cls.money)

        @staticmethod
        def stat():
                a = 10
                print("Parent Class Static Method:", a)

class Son(Father):                                     # Child Class
        def disp(self):
                print("Child Class Instance Method")

s = Son()
s.disp()
s.show()
s.showmoney()
s.stat()
```

Single Inheritance Program Output (CO2)

Output

```
= RESTART: C:\Users\admin\Desktop\ALL N
nce\1. SingleInheritance.py
Child Class Instance Method
Parent Class Instance Method
Parent Class Class Method: 1000
Parent Class Static Method: 10
>>>
```

Daily MCQs

1. A class can serve as base class for many derived classes.

A. True

B. False

Answer: A

2. Which of the following is not a type of inheritance?

A. Double-level

B. Multi-level

C. Single-level

D. Multiple

Answer: A

3. What does single-level inheritance mean?

A. A subclass derives from a class which in turn derives from another class

B. A single superclass inherits from multiple subclasses

C. A single subclass derives from a single superclass

D. Multiple base classes inherit a single derived class

Answer: C

The self Parameter (CO2)

- The self parameter is a reference to the current instance of the class and is used to access variables that belongs to the class.
- self represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

the __init__() Function(CO2)

- We want to add the __init__() function to the child class (instead of the pass keyword).

  Example

- Add the __init__() function to the Student class:

```
class Student(Person):
        def __init__(self, fname, lname):
                #add properties etc.
```

- When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.
- **Note:** The child's __init__() function **overrides** the inheritance of the parent's __init__() function.

Example of the __init__() Function(CO2)

- To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)

x = Student("Abhijit", "Niet")
x.printname()
```

Output

```
Nidhi Niet
```

Daily MCQs

1. What will be the output of below Python code?

```
class Student:
   def __init__(self,name,id):
       self.name=name
       self.id=id
        print(self.id)
std=Student("Simon",1)
std.id=2
print(std.id)
```

A. 1 1 B. 1 2 C. 2 1 D. 2 2

Answer: B

2. Which of the following is correct with respect to OOP concept in Python?

A. Objects are real world entities while classes are not real.
B. Classes are real world entities while objects are not real.
C. Both objects and classes are real world entities.
D. Both object and classes are not real.

Answer: A

Constructor in Inheritance(CO2)

By default, The constructor in the parent class is available to the child class.

```
class Father:
        def __init__(self):
            self.money = 2000
            print("Father Class Constructor")

class Son (Father):
        def disp(self):
            print("Son Class Instance Method:",self.money)

s = Son( )
s.disp()
```

What will happen if we define constructor in both classes ?

Constructor Overriding(CO2)

- If we write constructor in the both classes, parent class and child class then the parent class constructor is not available to the child class.
- In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.
- Constructor overriding is used when programmer want to modify the existing behavior of a constructor.

Constructor Overriding Program(CO2)

```
class Father:
        def __init__(self):
                self.money = 2000
                print("Father Class Constructor")
class Son(Father):
        def __init__(self):
                self.money = 5000
                print("Son Class Constructor")
        def disp(self):
                print(self.money)
s = Son()
s.disp()
```

How can we call parent class constructor ?

Constructor with super( ) Method(CO2)

- If we write constructor in the both classes, parent class and child class then the parent class constructor is not available to the child class.
- In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.
- **super ( )** method is used to call parent class constructor or methods from the child class.

Daily MCQs

1. When a child class inherits from only one parent class, it is called?

A. single inheritance
B. singular inheritance
C. Multiple inheritance
D. Multilevel inheritance

Answer: A

2. The child's __init__() function overrides the inheritance of the parent's __init__() function.

A. TRUE
B. FALSE
C. Can be true or false
D. Can not say

Answer: A

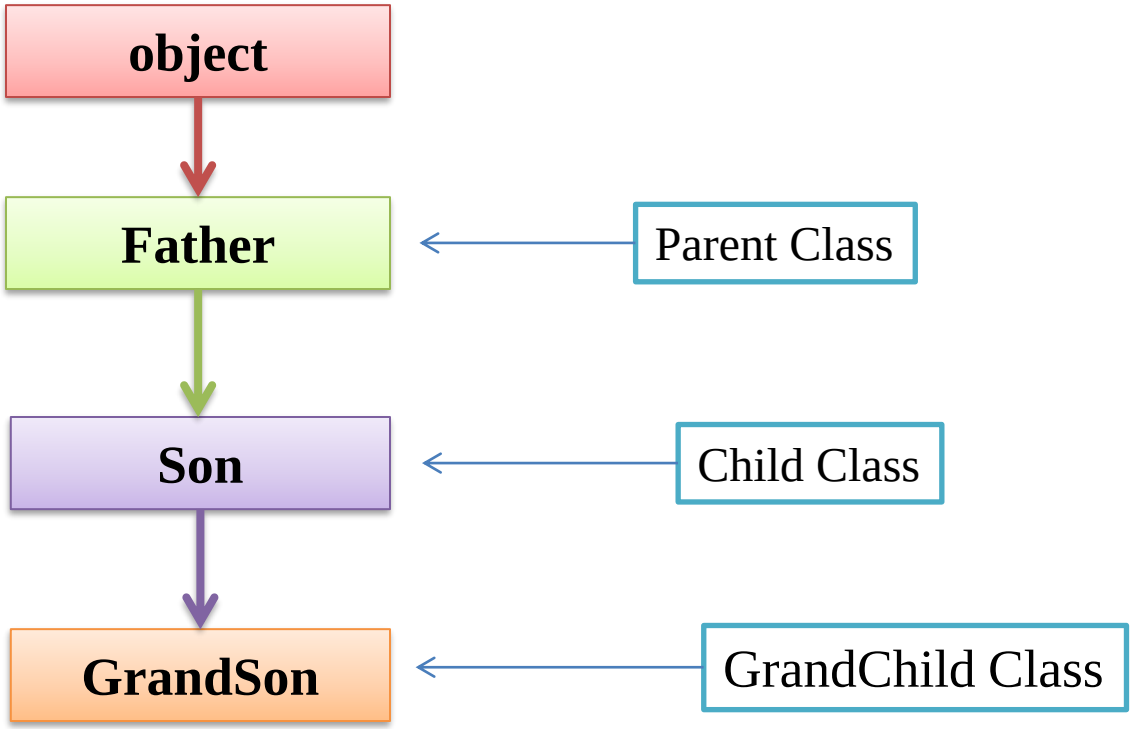
3. _____function that will make the child class inherit all the methods and properties from its parent.

A. self
B. __init__()
C. super
D. pass

Answer: C

Multi-level Inheritance(CO2)

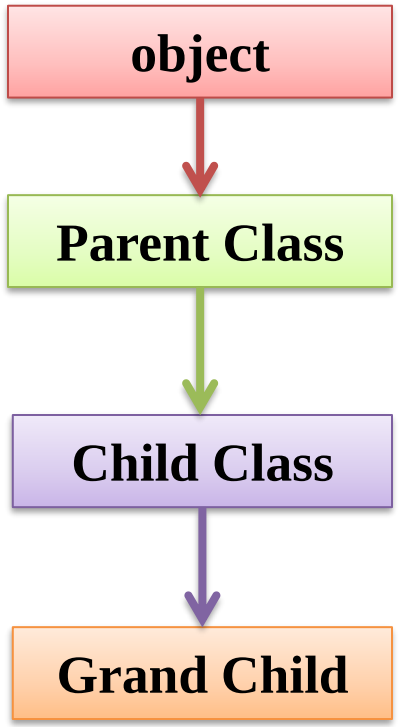
In multi-level inheritance, the class inherits the feature of another derived class (Child Class).



Multi-level Inheritance Syntax(CO2)

Syntax:-

```
class ParentClassName(object):
        members of Parent Class

class ChildClassName(ParentClassName):
        members of Child Class

class GrandChildClassName(ChildClassName):
        members of Grand Child Class
```



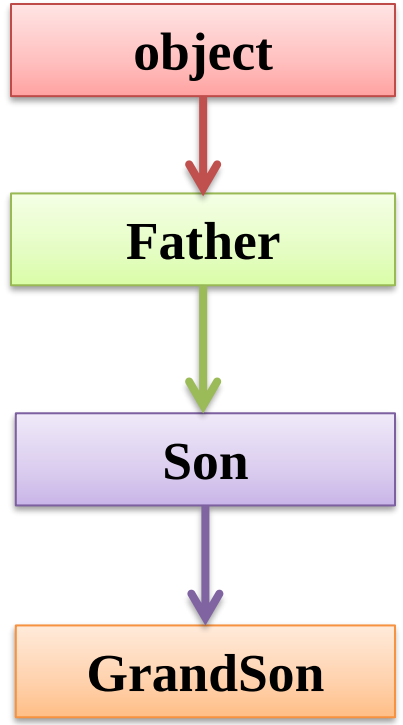
Multi-level Inheritance Example(CO2)

```
class Father (object):
        members of class Father

class Son (Father):
        members of class Son

class GrandSon (Son):
        members of class
GrandSon
```

Parent Class

Child Class

GrandChild Class



Multi-level Inheritance Program(CO2)

```
class Father:
        def __init__(self):
                print("Father Class Constructor")
        def showF(self):
                print("Father Class Method")

class Son(Father):
        def __init__(self):
                print("Son Class Constructor")
        def showS(self):
                print("Son Class Method")

class GrandSon(Son):
        def __init__(self):
                print("GrandSon Class Constructor")
        def showG(self):
                print("GrandSon Class Method")

g = GrandSon()
g.showF()
g.showS()
g.showG()
```

Multi-level Inheritance Program(CO2)

OUTPUT

```
>>>
= RESTART: C:\Users\admin\Desktop
nce\8. MultilevelInheritance.py
GrandSon Class Constructor
Father Class Method
Son Class Method
GrandSon Class Method
>>>
```

Daily MCQs

1. When two or more classes serve as base class for a derived class, the situation is known as _____.

A. multiple inheritance
B. polymorphism
C. encapsulation
D. hierarchical inheritance
E. none of these

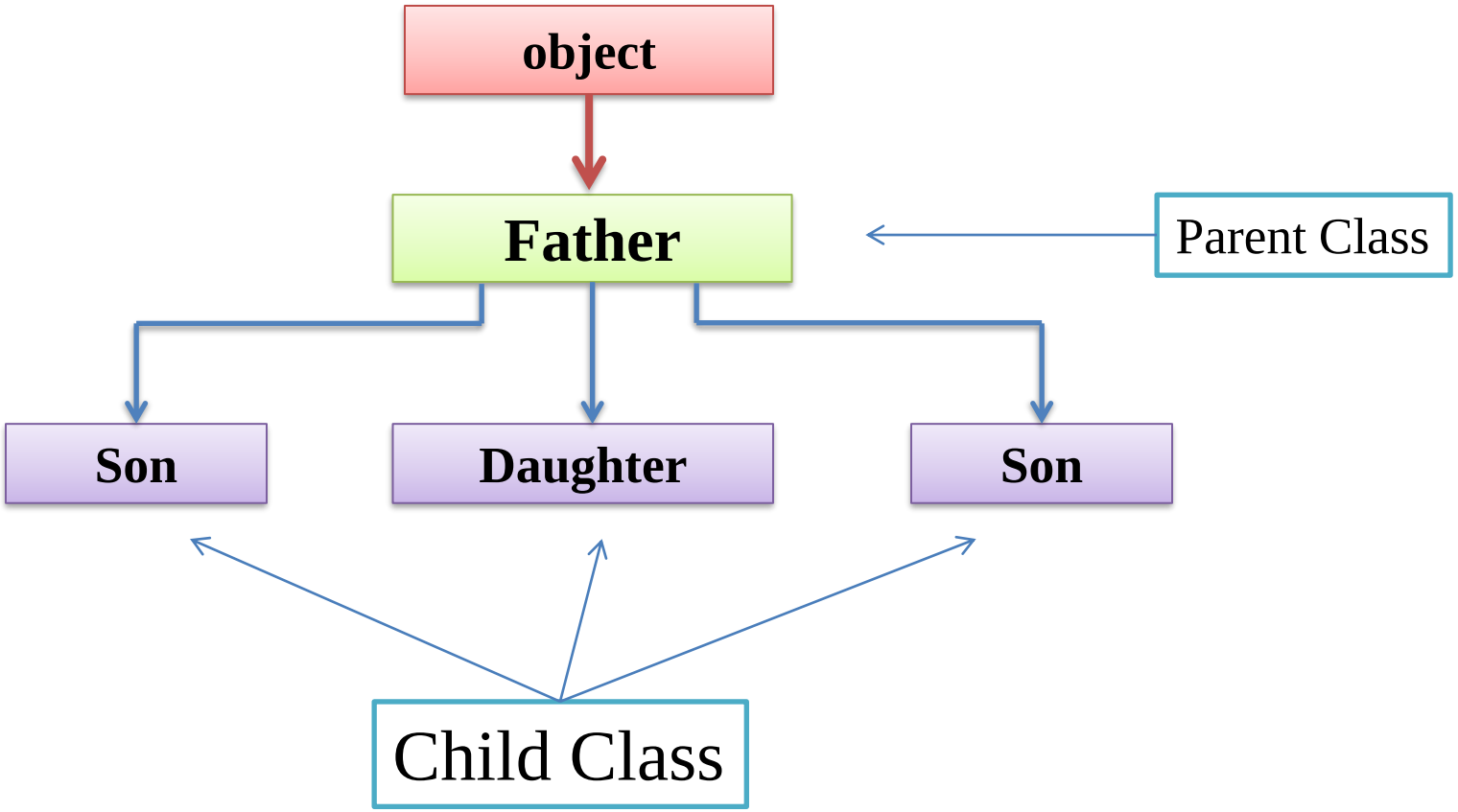
Answer: A

2. Multiple inheritance leaves room for a derived class to have _____ members.

A. dynamic
B. private
C. public
D. ambiguous
E. none of these

Answer: d

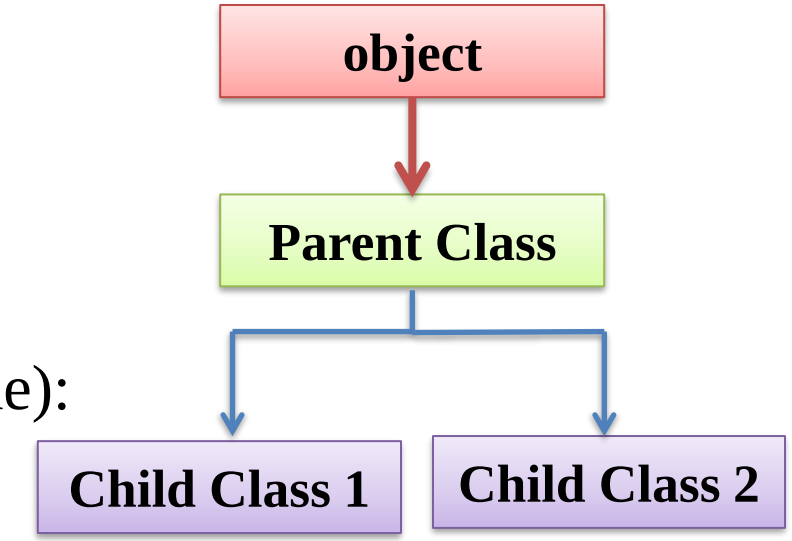
Hierarchical Inheritance(CO2)



Hierarchical Inheritance Syntax(CO2)

Syntax:-

```
class ParentClassName(object):
        members of Parent Class

class ChildClassName1(ParentClassName):
        members of Child Class 2

class ChildClassName2(ParentClassName):
        members of Child Class 2
```



Hierarchical Inheritance(CO2)

```
class Father (object):
    members of class Father
```

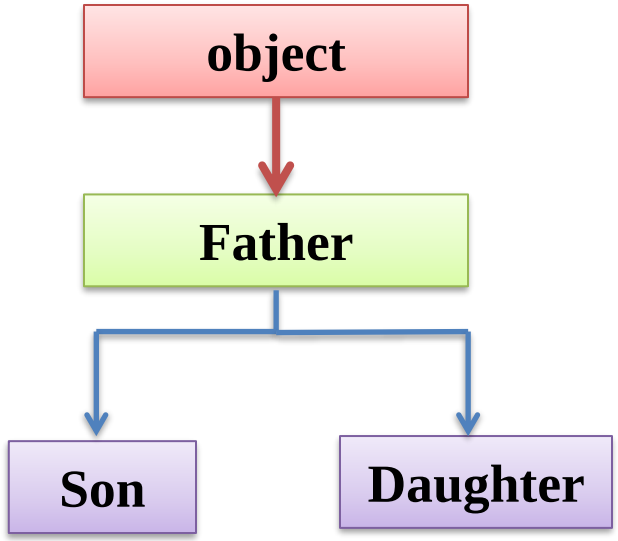
Parent Class

```
class Son (Father):
    members of class Son
```

Child Class

```
class Daughter (Father):
    members of class Daughter
```

Child Class



Hierarchical Inheritance Program(CO2)

```
class Father:
        def __init__(self):
                print("Father Class Constructor")
        def showF(self):
                print("Father Class Method")

class Son(Father):
        def __init__(self):
                print("Son Class Constructor")
        def showS(self):
                print("Son Class Method")

class Daughter(Father):
        def __init__(self):
                print("Daughter Class Constructor")
        def showD(self):
                print("Daughter Class Method")

d = Daughter()
d.showF()
d.showD()
s = Son()
s.showF()
s.showS()
```

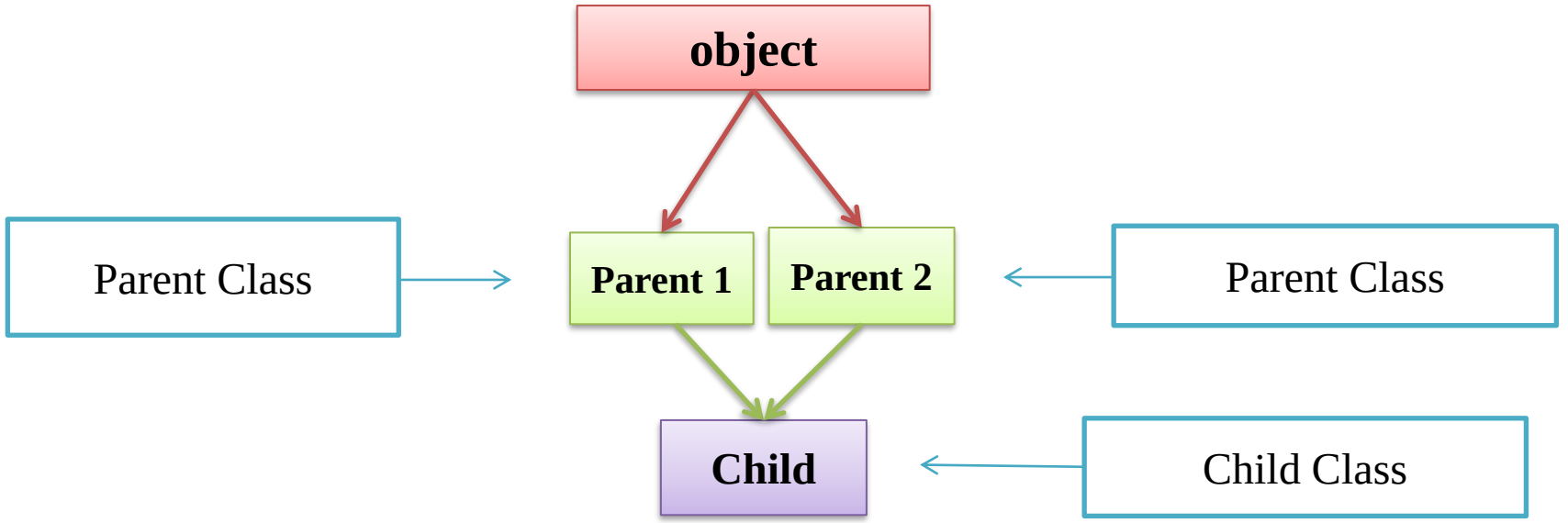
Hierarchical Inheritance Program(CO2)

OUTPUT

```
= RESTART: C:\Users\admin\Desktop\AI
nce\10. HierarchicalInheritance.py
Daughter Class Constructor
Father Class Method
Daughter Class Method
Son Class Constructor
Father Class Method
Son Class Method
>>>
```

Multiple Inheritance(CO2)

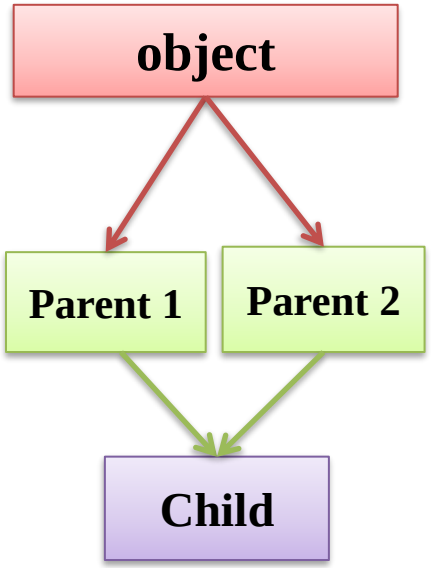
If a class is derived from more than one parent class, then it is called multiple inheritance.



Multiple Inheritance Syntax(CO2)

Syntax:-

```
class ParentClassName1(object):
        members of Parent Class

class ParentClassName2(object):
        members of Parent Class

class ChildClassName(ParentClassName1, ParentClassName2):
        members of Child Class
```



Multiple Inheritance(CO2)

```
class Father (object):
    members of class Father
```

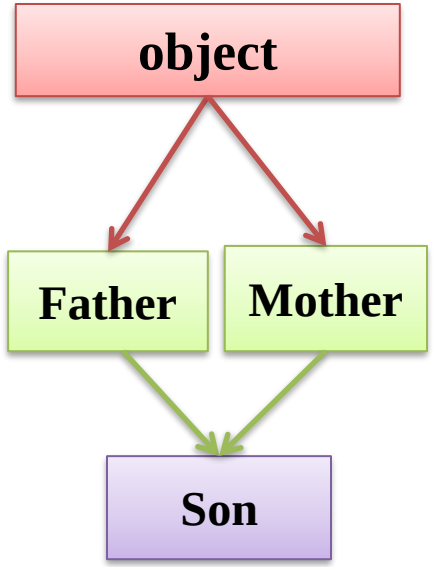
Parent Class

```
class Mother (object):
    members of class Mother
```

Parent Class

```
class Son (Father, Mother):
    members of class Son
```

Child Class



Multiple Inheritance Program(CO2)

```
class Father:
        def __init__(self):
                print("Father Class Constructor")
        def showF(self):
                print("Father Class Method")

class Mother:
        def __init__(self):
                print("Mother Class Constructor")
        def showM(self):
                print("Mother Class Method")

class Son(Father, Mother):
        def __init__(self):
                print("Son Class Constructor")
        def showS(self):
                print("Son Class Method")


s = Son()
s.showF()
s.showM()
s.showS()
```

Multiple Inheritance Program(CO2)

OUTPUT

```
>>>
= RESTART: C:\Users\admir
nce\12. MultipleInheritar
Son Class Constructor
Father Class Method
Mother Class Method
Son Class Method
>>>
```

Daily Quiz

1.Which of the following best describes inheritance?

a) Ability of a class to derive members of another class as a part of its own definition.

b) Means of bundling instance variables and methods in order to restrict access to certain class members

c) Focuses on variables and passing of variables to functions

d) Allows for implementation of elegant software that is well designed and easily modified.

Answer: a

2. All subclasses are a subtype in object-oriented programming.

a) True

b) False

Answer: b

3. When defining a subclass in Python that is meant to serve as a subtype, the subtype Python keyword is used.

a) True

b) False

Answer: b

Daily Quiz

4. What will be the output of the following Python code?

```
class Test:
        def __init__(self):
            self.x = 0
class Derived_Test(Test):
        def __init__(self):
            self.y = 1
def main():
            b = Derived_Test()
     print(b.x,b.y)
 main()
```

Answer:c

a) 0 1
b) 0 0
c) Error because class B inherits A but variable x isn't inherited
d) Error because when object is created, argument must be passed like Derived_Test(1)

Daily Quiz

5.**What will be the output of the following Python code?**

```
 class A():
        def disp(self):
print("A disp()")
            class B(A):
        pass
obj = B()
obj.disp()
```

Answer: d

a) Invalid syntax for inheritance
b) Error because when object is created, argument must be passed
c) Nothing is printed
d) A disp()

Weekly Assignment

Q1. A) Create child class Bus that will inherit all of the variables and methods of the Vehicle class

Given:

class Vehicle:

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

B) Create a Bus object that will inherit all of the variables and methods of the Vehicle class and display it.

Q2. How to check if a class is subclass of another?

Q3. How to access parent members in a subclass?

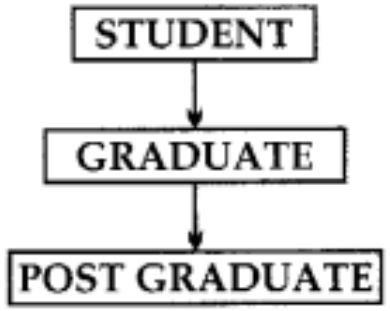
Weekly Assignment

Q4. What will be the output of the following Python code?

```
class A:
def __init__(self):
self.__i = 1
self.j = 5
def display(self):
print(self.__i, self.j)
class B(A):
def __init__(self):
super().__init__()
self.__i = 2
self.j = 7
c = B()c.display()
```

Weekly Assignment

Q5. Consider the figure given below and answer the questions that follows:



- Name the base class and the derived class.
- Which concept of OOP is implemented in the figure given above?

Old Question Papers

Q1. Predict the output of the following program. Also state which concept of OOP is being implemented?

```
def sum(x,y,z):
print "sum= ", x+y+z
def sum(a,b):
print "sum= ", a+b
sum(10,20)
sum(10,20,30)
```

Q2. Write a program that uses an area() function for the calculation of area of a triangle or a rectangle or a square. Number of sides (3, 2 or 1) suggest the shape for which the area is to be calculated.

Old Question Papers

Q3. Give a suitable example using Python code to illustrate single level inheritance considering COUNTRY to be BASE class and STATE to be derived class.

Q4. What is the difference between Multilevel inheritance and multiple inheritance? Give suitable examples to illustrate.

Q5. What are the different ways of overriding function call in derived class of python? Illustrate with example.

Expected Questions for University Exam

Q1.What is the need of object-oriented systems? Explain with the help of classes and objects.

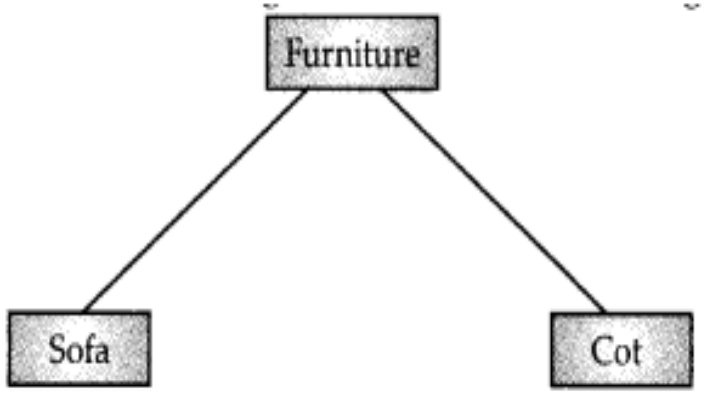
Q2. What is Inheritance? How code reusability is achieved using inheritance? Explain with the help of a program.

Q3. Write short note on hybrid inheritance.

Q4. What is the difference between Multilevel inheritance and multiple inheritance? Give suitable examples to illustrate.

Expected Questions for University Exam

Q5. Based on the diagram, answer the following:



- Write the name of the base class and the derived classes.
- Write the type of inheritance depicted in the above diagram.

SUMMARY

Inheritance allows us to define a class that **inherits** all the methods and properties from another class. Parent class is the class being **inherited** from, also called base class.

Class, methods and polymorphism

CONTENTS

- Class Method
- Concrete method
- Abstract Class
- Instance Method
- MRO & Super()
- Polymorphism
- Method Overriding

Prerequisite and Recap(CO2)

- Operators
- Loop
- Method

- Variables
- Class , Objects
- Constructor

Topic Objectives(CO2)

After you have read and studied this topic, you should be able to

- Understand the class types and method.
- Understand how Polymorphism is done.
- Understand how method overriding is done.

Type of Methods(CO2)

- Instance Methods
  - Accessor Methods
  - Mutator Methods

- **Class Methods**

- Static Methods Unit-1 Read already

Instance Method(CO2)

- Instance methods are the methods which act upon the instance variables of the class.
- Instance method need to know the memory address of the instance which is provided through *self* variable by default as first parameter for the instance method.

Syntax:-

```
def method_name(self):
        function body
```

Instance Method without Parameter/Formal Arguments

```
def method_name(self, f1, f2):
        function body
```

Instance Method with Parameter/Formal Arguments

Calling Instance Method w/o Argument(CO2)

Instance methods are bound to object of the class so we call instance method with object name.

Syntax:- object_name.method_name()

Ex:- realme.show_model()

```
class Mobile:
    def show_model(self):
        print("RealMe X")
realme = Mobile( )
realme.show_model()
```

Calling Instance Method w/o Argument

Calling Instance Method with Argument(CO2)

Syntax:- object_name.method_name(Actual_argument)

Ex:- realme.show_model(1000)

```
class Mobile:
    def __init__(self):
        self.model = 'RealMe X'
    def show_model(self, p):
        self.price = p
        print(self.model, self.price)

realme = Mobile( )
realme.show_model(1000)
```

Calling Method with argument

Accessor Method(CO2)

This method is used to access or read data of the variables. This method do not modify the data in the variable. This is also called as getter method.

Ex:-

```
def get_value(self):
def get_result(self):
def get_name(self):
def get_id(self):
```

```
class Mobile:
    def __init__(self):
        self.model = 'RealMe X'
    def get_model(self):
        return self.model

realme = Mobile( )
m = realme.get_model()
print(m)
```

Mutator Method(CO2)

This method is used to access or read and modify data of the variables. This method modify the data in the variable. This is also called as setter method.

Ex:-

```
def set_value(self):
def set_result(self):
def set_name(self):
def set_id(self):
```

```
class Mobile:
    def __init__(self):
        self.model = 'RealMe X'

    def set_model(self):
        self.model = 'RealMe 2'
realme = Mobile( )
realme.set_model()
```

```
class Mobile:

    def set_model(self, m):
        self.model = m

realme = Mobile( )
realme.set_model('Real Me X')
```

Class Methods(CO2)

Class methods are the methods which act upon the class variables or static variable of the class.

Decorator @classmethod need to write above the class method.

By default, the first parameter of class method is cls which refers to the class itself.

Syntax:-

```
@classmethod
def method_name(cls):
        method body
```

Decorator

Class Method without Parameter/Formal Arguments

```
@classmethod
def method_name(cls, f1, f2):
        method body
```

Decorator

Class Method with Parameter/Formal Arguments

Class Method without Parameter(CO2)

```
class Mobile:
    @classmethod
    def show_model(cls):
        print("RealMe X")

realme = Mobile( )
```

Decorator

Class Method

```
class Mobile:
    fp = 'Yes'
    @classmethod
    def show_model(cls):
        print(cls.fp)

realme = Mobile( )
```

Class Variable

Decorator

Class Method

Accessing Class variable Inside Class Method

Calling Class Method without Argument(CO2)

Syntax:- Classname.method_name()

```
class Mobile:
    @classmethod
    def show_model(cls):
            print("RealMe X")

realme = Mobile( )
Mobile.show_model()
```

Calling Class Method w/o Argument

Class Method with Parameter(CO2)

```
class Mobile:
    fp = 'Yes'
    @classmethod
    def show_model(cls, r):
        cls.ram = r
        print(cls.fp, cls.ram)

realme = Mobile( )
```

Class Variable

Decorator

Defining Method with parameter

Class Method with Parameter Program(CO2)

```
class Mobile:
        fp = 'Yes'      # Class Variable

        @classmethod
        def show_model(cls, r): # Class Method
                cls.ram = r
                # Accessing Class Variable
                print("Fingerprint Scaner:", cls.fp, "RAM:", cls.ram)

realme = Mobile()
Mobile.show_model('4GB')# Calling Class Method
```

Calling Class Method with Argument(CO2)

Syntax:- Classname.method_name(Actual_argument)

Ex:- Mobile.show_model('4GB')

```
class Mobile:
    fp = 'Yes'
    @classmethod
    def show_model(cls, r):
        cls.ram = r
        print(cls.fp, cls.ram)
realme = Mobile( )
Mobile.show_model(101)
```

Calling Method with argument

Static Methods(CO2)

- Static Methods are used when some processing is related to the class but does not need the class or its instances to perform any work.
- We use static method when we want to pass some values from outside and perform some action in the method.

Decorator @staticmethod need to write above the static method.

Syntax:-

```
@staticmethod
def method_name():
        method body
```

Decorator

Static Method without Parameter/Formal Arguments

```
@staticmethod
def method_name(f1, f2):
        method body
```

Decorator

Static Method with Parameter/Formal Arguments

Static Method without Parameter(CO2)

```
class Mobile:
    @staticmethod
    def show_model():
        print("RealMe X")

realme = Mobile( )
```

Decorator

Static Method

```
class Mobile:
    fp = 'Yes'
    @staticmethod
    def show_model():
        print(Mobile.fp)

realme = Mobile( )
```

Static Method

Calling Static Method without Argument(CO2)

Syntax:- Classname.method_name()

```
class Mobile:
    @staticmethod
    def show_model():
        print("RealMe X")

realme = Mobile( )
Mobile.show_model()
```

Calling Static Method w/o Argument

Static Method with Parameter(CO2)

```
class Mobile:
    @staticmethod
    def show_model(m, p):
        model = m
        price = p
        print(model, price)
realme = Mobile( )
```

Decorator

Defining Method with parameter

Calling Static Method with Argument(CO2)

Syntax:- Classname.method_name(Actual_argument)

Ex:- Mobile.show_model(1000)

```
class Mobile:
    @staticmethod
    def show_model(m, p):
        model = m
        price = p
        print(model, price)
realme = Mobile( )
Mobile.show_model('RealMe X', 1000)
```

Calling Method with argument

Daily MCQs

1. To create a class, use the keyword?

A. new
B. except
C. class
D. Object

Answer: C

2. ___is used to create an object.

A. class
B. constructor
C. user-defined functions
D. In-built functions

Answer: B

3. **_____ represents an entity in the real world with its identity and behavior.**

A. A method
B. An object
C. A class
D. An operator

Answer:B

Daily MCQs

4. Special methods need to be explicitly called during object creation.

A. TRUE

B. FALSE

Answer: B

5. Overriding means changing behaviour of methods of derived class methods in the base class.

A. TRUE

B. FALSE

Answer: B

6. Which of the following is not a class method?

A. Non-static

B. Static

C. Bounded

D. Unbounded

Answer:A

Abstract Class(CO2)

1. A class derived from ABC class which belongs to abc module, is known as abstract class in Python.
2. ABC Class is known as Meta Class which means a class that defines the behavior of other classes. So we can say, Meta Class ABC defines that the class which is derived from it becomes an abstract class.
3. Abstract Class can have abstract method and concrete methods.
4. Abstract Class needs to be extended and its method implemented.
5. PVM can not create objects of an abstract class.

Abstract Class Program(CO2)

```
from abc import ABC, abstractmethod
class Father(ABC):

        @abstractmethod
        def disp(self): # Abstract Method
                pass

        def show(self):          # Concrete Method
                print('Concrete Method')

#my = Father()           # Not possible to create object of a abstract class

class Child(Father):
        def disp(self):
                print("Defining Abstract Method")


c = Child()
c.disp()
c.show()
```

Abstract Class Program(CO2)

OUTPUT

```
>>>
= RESTART: C:\Users\admin\De
t Class\1. Example1.py
Defining Abstract Method
Concrete Method
>>>
```

Abstract Method(CO2)

An abstract method is a method whose action is redefined in the child classes as per the requirement of the object.

We can declare a method as abstract method by using @abstractmethod decorator.

Ex:-

```
from abc import ABC, abstractmethod
Class Father(ABC):
    @abstractmethod
     def disp(self):
         pass
```

Concrete Method(CO2)

A Concrete method is a method whose action is defined in the abstract class itself.

Ex:-

```
from abc import ABC, abstractmethod
Class Father(ABC):
    @abstractmethod
     def disp(self):
          pass
     def show(self):
          print("Concrete Method")
```

Abstract Method / Method Without Body

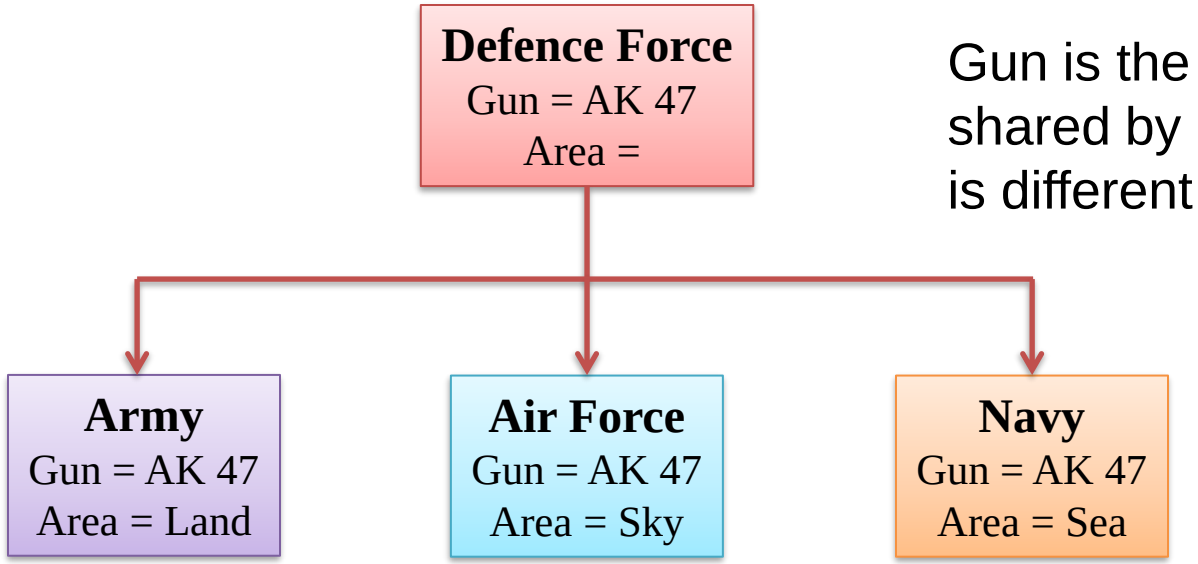
Concrete Method / Method with Body

Rules of Abstract Class (CO2)

- PVM cannot create objects of an abstract class.
- It is not necessary to declare all methods abstract in an abstract class.
- Abstract Class can have abstract method and concrete methods.
- If there is any abstract method in a class, that class must be abstract.
- The abstract methods of an abstract class must be defined in its child class/subclass.
- If you are inheriting any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

When use Abstract Class(CO2)

We use abstract class when there are some common feature shared by all the objects as they are.



Gun is the common feature shared by all Forces but area is different for them.

Daily MCQs

1. What is an abstract class?

A. An abstract class is one without any child classes.

B. An abstract class is any parent class with more than one child class.

C. An abstract class is class which cannot be instantiated, but can be a base class.

D. abstract class is another name for "base class."

Answer: C

2. Can an abstract parent class have non-abstract children?

A. No—an abstract parent must have only abstract children.

B. No—an abstract parent must have no children at all.

C. Yes—all children of an abstract parent must be non-abstract.

D. Yes—an abstract parent can have both abstract and non-abstract children.

Answer: D

Method Resolution Order (MRO)(CO2)

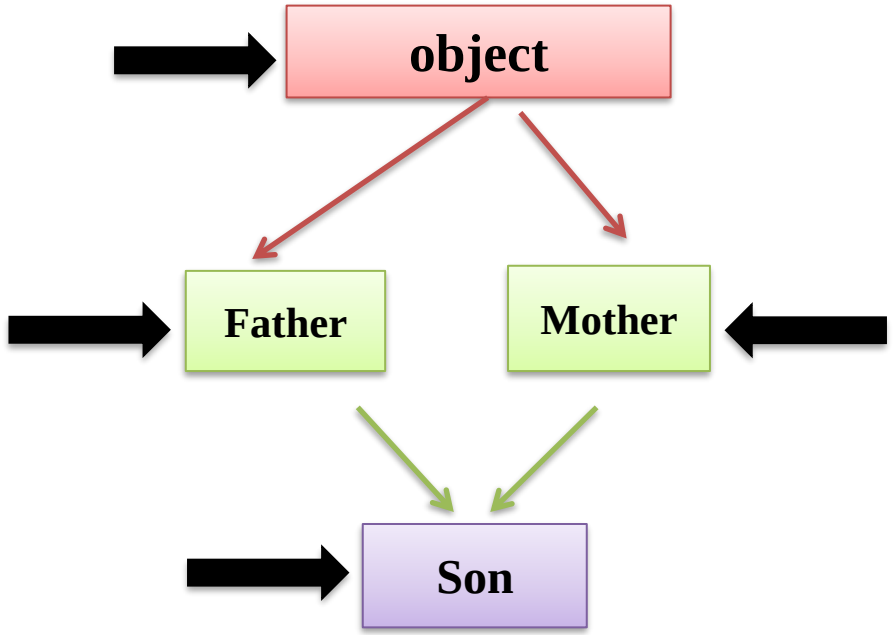
In the multiple inheritance scenario members of class are searched first in the current class. If not found, the search continues into parent classes in depth-first, left to right manner without searching the same class twice.

- Search for the child class before going to its parent class.
- When a class is inherited from several classes, it searches in the order from left to right in the parent classes.
- It will not visit any class more than once which means a class in the inheritance hierarchy is traversed only once exactly.

Method Resolution Order (MRO)(CO2)

s = Son()

- The search will start from Son. As the object of Son is created, the constructor of Son is called.
- Son has super().__init__() inside his constructor so its parent class, the one in the left side 'Father' class's constructor is called.
- Father class also has super().__init__() inside his constructor so its parent 'object' class's constructor is called.
- Object does not have any constructor so the search will continue down to right hand side class (Mother) of object class so Mother class's constructor is called.
- As Mother class also has super().__inti__() so its parent class 'object' constructor is called but as object class already visited, the search will stop here.



Use the super() Function(CO2)

Python also has a super() function that will make the child class inherit all the methods and properties from its parent.

Example

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
```

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Use the super() Function(CO2)

1. _____function that will make the child class inherit all the methods and properties from its parent.

A. self
B. __init__()
C. super
D. pass
Answer: C

2. What will be output for the folllowing code?

```
class A:
    def __init__(self, x= 1):
        self.x = x
class der(A):
    def __init__(self,y = 2):
        super().__init__()
        self.y = y
def main():
    obj = der()
    print(obj.x, obj.y)
main()
```

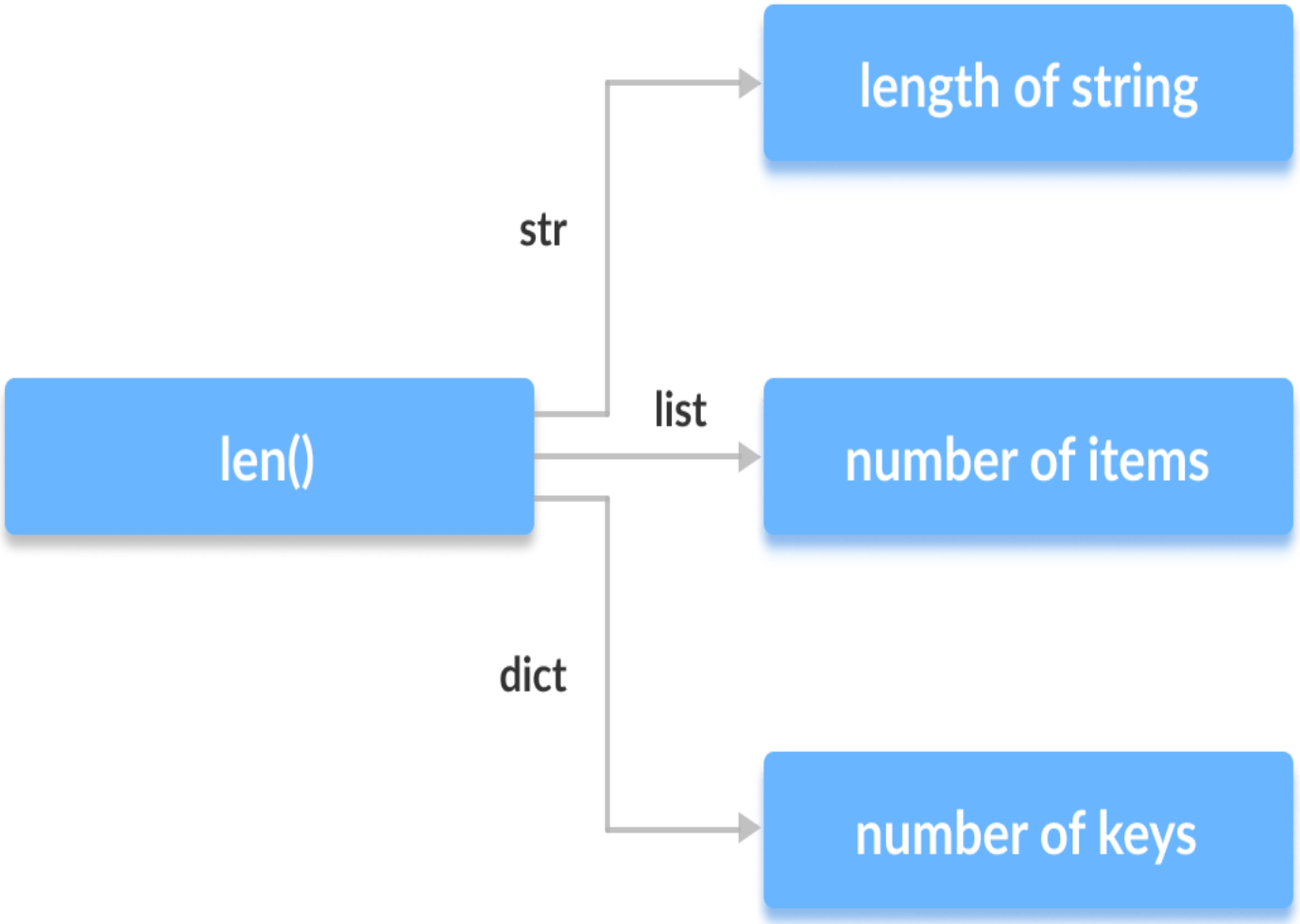
A. Error, the syntax of the invoking method is wrong
B. The program runs fine but nothing is printed
C. 1 0
D. 1 2
Answer: D

Polymorphism(CO2)

- Polymorphism is a word that came from two greek words, poly means many and morphos means forms.
- If a variable, object or method perform different behavior according to situation, it is called **polymorphism**.

- **Method Overriding**
- Method Overloading
- Operator Overloading

Polymorphic len() function(CO2)



Method overriding (CO2)

- The method overriding in Python means **creating two methods with the same name but differ in the programming logic.**
- The concept of Method overriding allows us to change or override the Parent Class function in the Child Class.
- You can't override a method within the same class. It means you have to do it in the child class using the **Inheritance** concept.
- To override the Parent **Class** method, you have to create a method in the Child class with the same name and the same number of parameters.

Method overriding Syntax(CO2)

```
# Python Method Overriding
class Employee:
        def message(self):
              print('This message is from Employee Class')
class Department(Employee):
        def message(self):
             print('This Department class is inherited from Employee')
emp = Employee()
emp.message()
print('------------')
dept = Department()
dept.message()
```

Method overriding(CO2)

Output:

```
This message is from Employee Class
-----------
This Department class is inherited from Employee
```

Method Overriding with arguments(CO2)

```
class Employee:
    def add(self, a, b):
        print('The Sum of Two = ', a + b)
class Department(Employee):
    def add(self, a, b, c):
        print('The Sum of Three = ', a + b + c)
emp = Employee()
emp.add(10, 20)
print('------------')
dept = Department()
dept.add(50, 130, 90)
```

Method Overriding with arguments(CO2)

Output:

The Sum of Two = 30

'The Sum of Three = 270

Method Overriding(CO2)

```
class Add:
        def result(self, a, b):
            print("Addition:", a+b)

class Multi(Add):
        def result(self, a, b):
            print("Multiplication:", a*b)

m = Multi()
m.result(10, 20)
```

Method with super( ) Method(CO2)

- If we write method in the both classes, parent class and child class then the parent class's method is not available to the child class.
- In this case only child class's method is accessible which means child class's method is replacing parent class's method.

- **super ( )** method is used to call parent class's constructor or methods from the child class.

  Syntax:- super().methodName()

Weekly Assignment

Q1.Write short notes on:

1) MRO

2) Super()

3) Abstract Class

Q2.What are the different ways of overriding function call in derived class of python? Illustrate with example.

Q3.Why use Abstract Base Classes?

Q4.How Abstract Base classes work?

Weekly Assignment

Q5. What is the output of following program:

```
class A:
    def rk(self):
        print(" In class A")
class B(A):
    def rk(self):
        print(" In class B")

r = B()
r.rk()
```

Daily MCQs

1. **Which of the following represents a template, blueprint, or contract that defines objects of the same type?**

a. A class
b. An object
c. A method
d. A data field

Answer: a

2. **Which of the following represents a distinctly identifiable entity in the real world?**

a. A class
b. An object
c. A method
d. A data field

Answer: b

Daily MCQs

1. Overriding means changing behaviour of methods of derived class methods in the base class.

A. True

B. False

Answer: B

2. What will be the output of the following Python code?

```
class A:
    def __repr__(self):
        return "1"
class B(A):
    def __repr__(self):
        return "2"
class C(B):
    def __repr__(self):
        return "3"
o1 = A()
o2 = B()
o3 = C()
print(obj1, obj2, obj3)
```

A. 1 1 1

B. 1 2 3

C. ‘1’ ‘1’ ‘1’

D. An exception is thrown

Daily Quiz

1.**Which of the following best describes polymorphism?**

a) Ability of a class to derive members of another class as a part of its own definition

b) Means of bundling instance variables and methods in order to restrict access to certain class members

c) Focuses on variables and passing of variables to functions

d) Allows for objects of different types and behaviour to be treated as the same general type

Answer: d

2.**A class in which one or more methods are only implemented to raise an exception is called an abstract class.**

a) True

b) False

Answer: a

3.**Overriding means changing behaviour of methods of derived class methods in the base class.**

a) True

b) False

Answer: b

Daily Quiz

4.**What will be the output of the following Python code?**

```
class A:
    def __repr__(self):
        return "1"
class B(A):
    def __repr__(self):
        return "2”
class C(B):
    def __repr__(self):
        return "3"
o1 = A()
o2 = B()
o3 = C()
print(obj1, obj2, obj3)
```

a) 1 1 1
b) 1 2 3
c) ‘1’ ‘1’ ‘1’
d) An exception is thrown

Answer: b

Old Question Papers

Q1. What is the output of following python code?

```
class Parent():
    def __init__(self):
        self.value = "Inside Parent"
    def show(self):
        print(self.value)
class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"
    def show(self):
        print(self.value)
obj1 = Parent()
obj2 = Child()
obj1.show()
obj2.show()
```

Old Question Papers

Q2. What should be the output of calling the parent's class method inside the overridden method?

```
class Parent():
    def show(self):
        print("Inside Parent")
class Child(Parent):
    def show(self):
        Parent.show(self)
        print("Inside Child")
obj = Child()
obj.show()
```

Old Question Papers

Q3.What is super method? Explain with example.

Q4. Explain method overriding with suitable example.

Q5.What is method resolution operator? Support your answer with suitable example.

Expected Questions for University Exam

Q1. Write short note on- (a) Method overriding

(b) Polymorphism

Q2. Explain method overriding with suitable example.

Q3. What is method resolution operator? Support your answer with suitable example.

Q4. Explain the concept of polymorphism by giving suitable examples.

Q5. Explain calling the parent's class method inside the overridden method .

SUMMARY

Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class.

Introspection

NIET Greater Noida
GET FUTURE READY

CONTENTS

- Introspection: Introspection types , Introspection objects
- Introspection scopes, Inspect modules, Introspect tools

Prerequisite and Recap(CO2)

- Loop
- Method
- Variables
- Class , Objects
- Constructor

Topic Objectives(CO2)

After you have read and studied this topic, you should be able to:

- Understand Introspection using python.
- Understand Introspection tools using python.

Introspection(CO2)

- **Introspection** is an ability to determine the type of an object at runtime.
- Everything in **python** is an object.
- Every object in **Python** may have attributes and methods.
- Introspection reveals useful information about your program's objects.

Code Introspection(CO2)

- Code **Introspection** is used for examining the classes, methods, objects, modules, keywords and get information about them so that we can utilize it.

- Python, being a dynamic, object-oriented programming language, provides tremendous introspection support. Python's support for introspection runs deep and wide throughout the language.

Introspection Types(CO2)

Python provides some built-in functions that are used for code introspection. **These are following:**

- **type() :** This function returns the type of an object
- **dir() :**This function return list of methods and attributes associated with that object.
- **str() :**This function converts everything into a string
- **id() :**This function returns a special id of an object.
- **isinstance():** Using this function, we can determine if a certain object is an instance of the specified class.
- **hasattr():** to check if it has the attribute

Code Introspection(CO2)

type() : This function returns the type of an object

```
import math

# print type of math
print(type(math))

# print type of 1
print(type(1))

# print type of "1"
print(type("1"))

# print type of rk
rk =[1, 2, 3, 4, 5, "radha"]

print(type(rk))
print(type(rk[1]))
print(type(rk[5]))
```

Output:

```
<class 'module'>
<class 'int'>
<class 'str'>
<class 'list'>
<class 'int'>
<class 'str'>
```

Code Introspection(CO2)

dir() : This function return list of methods and attributes associated with that object.

```
import math
rk =[1, 2, 3, 4, 5]

# print methods and attributes of rk
print(dir(rk))
rk =(1, 2, 3, 4, 5)

# print methods and attributes of rk
print(dir(rk))
rk ={1, 2, 3, 4, 5}

print(dir(rk))
print(dir(math))
```

Output:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem_
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'a
```

Code Introspection(CO2)

str() : This function converts everything into a string

```
# Python program showing
# a use of str() function

a = 1
print(type(a))

# converting integer
# into string
a = str(a)
print(type(a))

s =[1, 2, 3, 4, 5]
print(type(s))

# converting list
# into string
s = str(s)
print(type(s))
```

Output:

```
<class 'int'>
<class 'str'>
<class 'list'>
<class 'str'>
```

Code Introspection(CO2)

id() : This function returns a special id of an object.

```
import math
a =[1, 2, 3, 4, 5]

# print id of a
print(id(a))
b =(1, 2, 3, 4, 5)

# print id of b
print(id(b))
c ={1, 2, 3, 4, 5}

# print id of c
print(id(c))
print(id(math))
```

Output:

```
139787756828232
139787757942656
139787757391432
139787756815768
```

Method Of Code Introspection(CO2)

Function	Description
help()	It is used to find what other functions do.
hasattr()	Checks if an object has an attribute.
getattr()	Returns the contents of an attribute if there are some.
repr()	Return the string representation of object.
callable()	Checks if an object has a callable object or not.
Issubclass()	Checks if a specific class is a derived class of another class.
Isinstance()	Checks if an object is an instance of a specific class.
sys()	Give access to system specific variables and functions.

Introspection Objects(CO2)

1. In computer programming, **introspection** is the ability to determine the type of an **object** at runtime.

2. It is one of Python's strengths. Everything in Python is an **object** and we can examine those **objects:**

- type()
- dir()
- id()
- getattr()
- hasattr()
- globals()
- locals()

Introspection Scopes(CO2)

- Python contains two built-in functions for examining the content of scopes.
- The first function is globals().
- This returns a dictionary which represents the global namespace.
- Let’s define a variable a = 42 and call globals() again, and we can see that the binding of the name 'a' to the value of 42 has been added to the namespace.
- In fact, the dictionary returned by globals() is the global namespace. Let's create a variable tau and assign value 6.283185.
- We can now use this variable just like any other variables. The second function is locals().

Introspection Scopes(CO2)

- To really see locals() in action, we're are going to create another local scope, which we can do by defining a function that accepts a single argument, defines a variable X to have a value of 496, and then prints the locals() dictionary with a width of 10 characters.
- When run, we see that this function has the expected three entries in its local namespace. By using locals() to provide the dictionary, we can easily refer to local variables in format strings.

Inspect Module (CO2)

- The inspect module helps in checking the objects present in the code that we have written.

- As Python is an OOP language and all the code that is written is basically an interaction between these objects, hence the inspect module becomes very useful in inspecting certain modules or certain objects.

- We can also use it to get a detailed analysis of certain function calls or tracebacks so that debugging can be easier.

Methods to verify the type of token(CO2)

- The inspect module provides a lot of methods, these methods can be classified into two categories i.e. methods to verify the type of token and methods to retrieve the source of token.
- **The most commonly used methods of both categories are mentioned below.**
- **isclass():** The isclass() method returns True if that object is a class or false otherwise.
- When it combined with the getmembers() functions it shows the class and its type. It is used to inspect live classes.
- **ismodule():** This returns true if the given argument is an imported module.

Methods to verify the type of token(CO2)

The most commonly used methods of both categories are mentioned below.

- **isfunction():** This method returns *true* if the given argument is an inbuilt function name.
- **ismethod():** This method is used to check if the argument passed is the name of a method or not.

```
# import required modules
import inspect
import numpy

# use ismodule()
print(inspect.ismodule(numpy))
```

o/p---True

Methods to retrieve the source of token(CO2)

- **getclasstree()**: The getclasstree() method will help in getting and inspecting class hierarchy. It returns a tuple of that class and that of its preceding base classes. That combined with the getmro() method which returns the base classes helps in understanding the class hierarchy.
- **getmembers():** This method returns the member functions present in the module passed as an argument of this method.
- **stack():** This method helps in examining the interpreter stack or the order in which functions were called.

Methods to retrieve the source of token(CO2)

- **getsource():** This method returns the source code of a module, class, method, or a function passes as an argument of getsource() method.
- **getmodule():** This method returns the module name of a particular object pass as an argument in this method.
- **getdoc():** The getdoc() method returns the documentation of the argument in this method as a string.
- **signature():** The signature() method helps the user in understanding the attributes which are to be passed on to a function.

Program of Inspect(CO2)

```
# import required modules
import inspect
import collections

# use signature()
print(inspect.signature(collections.Counter))
```

```
# import required modules
import inspect

def fun(a,b):
    # product of
    # two numbers
    return a*b

# use getsource()
print(inspect.getsource(fun))
```

```
# import required modules
import inspect
import collections

# use getmodule()
print(inspect.getmodule(collections))
```

Introspect Tools(CO2)

- We'll now build a tool to introspect objects by leveraging the interesting techniques we've discussed and bring them together into a useful program.
- Our objective is to create a function, which when passed a single object prints out a nicely formatted dump of that object's attributes with rather more clarity.
- This is a small tool that we are going to create and the main use case of this tool is to help us identify a different aspect of the objects with respect to its type, methods, attributes, and documentation.

Weekly Assignment

Q1. Python provides some built-in functions that are used for code introspection. Explain them with examples.

Q2. Write short note on following functions:

Type()

Dir()

Str()

Q3. What are methods of code introspection? Explain.

Q4. What does isinstance() and issubclass() describes?

Q5. What does getattr() and hasattr() method are used for?

MCQs

1. Which is not introspection method?
a) help()
b) sys()
c) repr()
d) remove()

Answer: d

2.Which of the following is not an introspection object?
a) type()
b) id()
c) dir()
d) Unbounded()

Answer: d

3.Special methods need to be explicitly called during object creation.
a) True
b) False

Answer: b

Daily Quiz

1. Is the following Python code valid?

```
class B(object):
def first(self):
print("First method called")
def second():
print("Second method called")
 ob = B()
 B.first(ob)
```

a) It isn't as the object declaration isn't right
b) It isn't as there isn't any __init__ method for initializing class members
c) Yes, this method of calling is called unbounded method call
d) Yes, this method of calling is called bounded method call

Answer: c

Old Question Papers

Q1. In the following program find out which objects are callable.

Callable.py

```
class Car(object):
      def setName(self, name):
        self.name = name
def fun():
    pass
c = Car()
print(callable(fun))
print(callable(c.setName))
print(callable([]))
print(callable(1))
```

Q2. Explain various methods used for introspection in python.

Q3.What does callable() describes?

Old Question Papers

Q4. Explain various ways of inspecting Python objects.

Q5. What should be the output of the following program?

```
# subclass.py
class Object(object):
    def __init__(self):
      pass
class Wall(Object):
    def __init__(self):
      pass
print(issubclass(Object, Object))
print(issubclass(Object, Wall))
print(issubclass(Wall, Object))
print(issubclass(Wall, Wall))
```

Expected Questions for University Exam

Q1.What does callable() describes?

Q2. Explain various ways of inspecting Python objects.

Q3. What are introspection tools? Explain them briefly.

Q4. Explain with example issubclass() and getattr() functions.

Q5. Explain inspect modules with example.

Q6. What should be the output of the following program?

```
class Object(object):
    def __init__(self):
      pass
class Wall(Object):
    def __init__(self):
      pass
print(issubclass(Object, Object))
print(issubclass(Object, Wall))
print(issubclass(Wall, Object))
print(issubclass(Wall, Wall))
```

Video Links/You Tube Links

- https://www.python-course.eu/python3_inheritance.php
- https://www.youtube.com/watch?v=u9x475OGj_U
- https://www.youtube.com/watch?v=byHcYRpMgI4
- https://www.youtube.com/watch?v=FsAPt_9Bf3U

TEXT & REFERENCE BOOKS

Text books:

(1) Magnus Lie Hetland, "Beginning Python-From Novice to Professional"—Third Edition, Apress

(2) Peter Morgan, Data Analysis from Scratch with Python, AI Sciences

(3) Allen B. Downey, “Think Python: How to Think Like a Computer Scientist”, 2nd edition, Updated for Python 3, Shroff/O‘Reilly Publishers, 2016

(4) Miguel Grinberg, Developing Web applications with python, OREILLY

Reference Books:

(1) Dusty Phillips, Python 3 Object-oriented Programming - Second Edition, O’Reilly

(2) Burkhard Meier, Python GUI Programming Cookbook - Third , Packt

(3) DOUG HELLMANN, THE PYTHON 3 STANDARD LIBRARY BY EXAMPLE, :Pyth 3 Stan Libr Exam _2 (Developer's Library) 1st Edition, Kindle Edition.

(4) Kenneth A. Lambert, —Fundamentals of Python: First Programs‖, CENGAGE Learning, 2012.

SUMMARY

Introspection is an ability to determine the type of an object at runtime. Everything in **python** is an object. Every object in **Python** may have attributes and methods.

Functional Programming

Unit: 3

Problem Solving Using Advanced Python

Course Details
(B Tech 2nd Sem /1st Year)

NIET Greater Noida

GET FUTURE READY

Contents

- Map
- Filter
- Reduce
- Comprehensions
- Immutability
- Closures and Decorators
- Generators
- Co-routines, iterators
- Declarative programming

Functional Programming(CO3)

- Functional Programming is a ***programming*** paradigm with software primarily composed of functions processing data throughout its execution.

- **Python** allows us to code in a ***functional, declarative style.***

- It even has support for many common ***functional features*** like Lambda Expressions and the map and filter functions.

Functional Programming Characteristics (CO3)

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports ***higher-order functions*** and ***lazy evaluation*** features.
- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism

Advantages of Functional Programming (CO3)

- ***Bugs-Free Code*** − Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.
- ***Efficient Parallel Programming*** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- ***Efficiency*** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- ***Supports Nested Functions*** − Functional programming supports Nested Functions.
- ***Lazy Evaluation*** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Comparison between FP & OOPS(CO3)

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.

Anonymous or Lambda Function

A function without a name is called as Anonymous Function. It is also known as Lambda Function.

Anonymous Function are not defined using def keyword rather they are defined using lambda keyword.

Syntax:-

lambda argument_list : expression

Ex:-

lambda x : print(x)

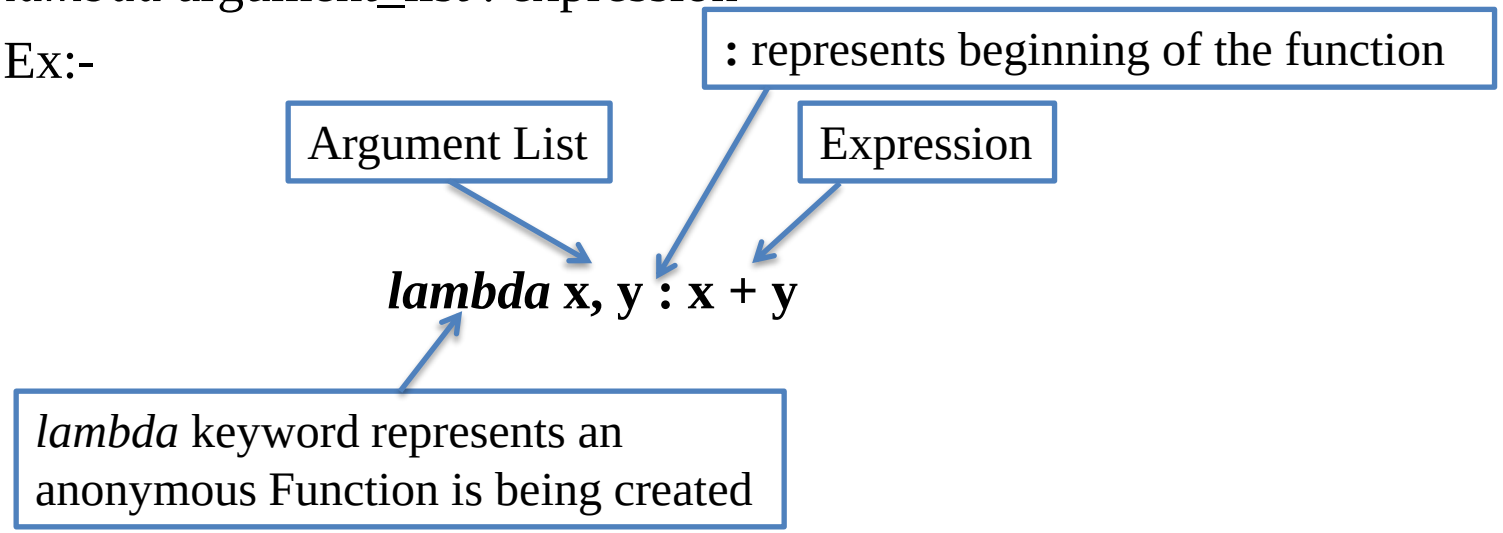
lambda x, y : x + y

Creating a Lambda Function

Syntax:-

lambda argument_list : expression

Ex:-



Calling Lambda Function

```
sum = lambda x : x + 1
sum(5)

add = lambda x, y : x + y
add(5, 2)
```

Anonymous Function or Lambdas

- Lambda Function doesn't have any Name

  Ex:- *lambda* x : print(x)

- Lambda function returns a function

  Ex:- show = *lambda* x : print(x)

- Lambda function can take zero or any number of argument but contains only one expression

  Ex:- *lambda* x, y : x + y

- In lambda Function there is no need to write return statement
- It can only contain expressions and can't include statements in its body
- You can use all the type of Actual Arguments

Function- Map()(CO2)

- Lambda functions are mainly used in combination with the functions **filter(), map() and reduce().**

- The lambda feature was added to Python due to the demand from Lisp programmers.

Function- Map()(CO2)

- The advantage of the lambda operator can be seen when it is used in combination with the map() function.
  map() is a function with two arguments:

  r = map(func, Seq)
- The first argument func is the name of a function and the second a sequence (e.g. a list) seq.
- map() applies the function func to all the elements of the sequence seq.
- It returns a new list with the elements changed by func.

Function- Map() Program(CO2)

```
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(map(lambda x: x*2, li))
print(final_list)
```

OUTPUT
25,49,484,9409,2916,3844,5929,529,5329,3721

Function- reduce()(CO2)

- The reduce() function in Python takes in a function and a list as an argument.
- The function is called with a lambda function and an iterable and a new reduced result is returned.
- This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the ***functools*** module.

Function- reduce() Program(CO2)

```
# reduce() with lambda()
# to get sum of a list

from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

OUTPUT
193

Function- filter() (CO2)

- The filter() function in Python takes in a function and a list as arguments.

- This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.

Function- filter() Program(CO2)

```
# filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

OUTPUT
5,7,97,77,23,73,61

Comprehension

- Python provides compact syntax for deriving one list from another. These expressions are called ***list comprehensions***.

- List comprehensions are one of the most powerful tools in Python. Python's list comprehension is an example of the language's support for functional programming concepts.

- The Python list comprehensions are a very easy way to apply a function or filter to a list of items. List comprehensions can be very useful if used correctly but very unreadable if you're not careful

Comprehension

Syntax

The general syntax of list comprehensions is −

```
[expr for element in iterable if condition]
```

Above is equivalent to −

```
for element in iterable:
    if condition:
        expr
```

List Comprehension vs For Loop

```
#USING FOR LOOP
evens = []
for i in range(10):
    if i % 2 == 0:
        evens. append(i)
    print(evens)
```

Output
[0, 2, 4, 6, 8]

A better and faster way to write the above code is through list comprehension.

```
>>> [i for i in range(10) if i % 2 ==0]
```

Output
[0, 2, 4, 6, 8]

List comprehensions vs lambda function

List comprehensions are clearer than the map built-in function for simple cases. the map requires creating a lambda function for the computation, which is visually noisy.

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 >>> list(map(lambda x: x**2, lst))
```

Output

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

List comprehensions with conditional expression

Example:

when you only want to compute the squares of the numbers that are divisible by 2.

```
–>>> lst=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> even_squares = [x**2 for x in lst if x % 2 == 0]
>>> print(even_squares)
```

Output :
[4, 16, 36, 64, 100]

List comprehensions with conditional expression

```
# a list of even numbers between 1 and 100

>>> evens = [i for i in range(1,100) if not i % 2]
> >>print(evens)
```

Output:

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24,
26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46,
48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90,
92, 94, 96, 98]

Nested IF with List Comprehension

```
>>> num_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
>>> print(num_list)
```

Output:
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

if...else With List Comprehension

```
>>> obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
>>> print(obj)
```

Output:
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']

Python for-loop to list comprehension?

- List comprehensions offer a concise way to create lists based on existing lists.
- When using list comprehensions, lists can be built by leveraging any iterable, including strings and tuples.
- List comprehensions consist of an iterable containing an expression followed by a for the clause. This can be followed by additional for or if clauses.
- Let's look at an ***example*** that creates a list based on a string:

```
>>> hello_letters = [letter for letter in 'hello']
 >>> print(hello_letters)
```

Output:
['h', 'e', 'l', 'l', 'o']

Python for-loop to list comprehension? (cont.)

String hello is iterable and the letter is assigned a new value every time this loop iterates. This list comprehension is equivalent to:

```
>>> hello_letters = []
>>> for letter in 'hello':
>>>    hello_letters.append(letter)
```

Nested Loops in List Comprehension

We need to compute the transpose of a matrix that requires nested for loop. Let's see how it is done using normal for loop first.

Transpose of Matrix using Nested Loops

```
transposed = []
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
for i in range(len(matrix[0])):
        transposed_row = []
        for row in matrix:
                transposed_row.append(row[i])
        transposed.append(transposed_row)
print(transposed)
```

Output:
[[1, 4], [2, 5], [3, 6], [4, 8]]

Nested Loops in List Comprehension

- We can also perform nested iteration inside a list comprehension. In this section, we will find transpose of a matrix using nested loop inside list comprehension.'
- Transpose of a Matrix using List Comprehension

```
>>> matrix = [[1, 2], [3,4], [5,6], [7,8]]
>>> transpose = [[row[i] for row in matrix] for i in range(2)]
>>> print (transpose)
```

Output:
[[1, 3, 5, 7], [2, 4, 6, 8]]

Python Dictionary Comprehension

- We need the key: value pairs to create a dictionary. To get these key-value pairs using dictionary comprehension the general statement of dictionary comprehension is as follows:

  {key: value for ___ in iterable}

Let's see how to generate numbers as keys and their squares as values within the range of 10. Our result should look like

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}.

For this result the below code is as follows:-

Python Dictionary Comprehension

```
# creating the dictionary
>>> squares = {i: i ** 2 for i in range(10)}
>>> print(squares)
```

Output

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

Let's see how to create a dictionary from **[1, 2, 3, 4, 5]** and **[a, b, c, d, e].**

```
# keys
>>> key=['a','b','c','d']
# value=[3,5,6,7]
# creating a dict from the above lists
>>> dictionary = {key: value for (key, value) in zip(keys, values)}
>>> print(dictionary)
{'a': 3, 'b': 5, 'c': 6, 'd': 7}
```

IMMUTABILITY

- Every variable in python holds an instance of an object. There are two types of objects in python i.e. **Mutable** and **Immutable objects**. Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can't be changed afterwards. However, it's state can be changed if it is a mutable object.

- To summarize the difference, mutable objects can change their state or contents and immutable objects can't change their state or content.

- In python, the string data types are immutable. Which means a string value cannot be updated. We can verify this by trying to update a part of the string which will led us to an error.

IMMUTABILITY

Example:

```
# Can not reassign
 >>> t= “Mumbai"
>>> print type(t)
 >>> t[0] = “P”.
```

When we run the above program, we get the following output −

t[0] = "M" TypeError: 'str' object does not support item assignment

IMMUTABILITY

- We can further verify this by checking the memory location address of the position of the letters of the string.

```
>>> x = 'banana' for idx in range (0,5):
>>> print x[idx], "=", id(x[idx])
```

When we run the above program we get to see that N and N also point to the same location

CLOSURE

- A *closure* is a nested function which has access to a free variable from an enclosing function that has finished its execution. Three characteristics of a Python closure are:
  - it is a nested function
  - it has access to a free variable in outer scope
  - it is returned from the enclosing functio

- A *free variable* is a variable that is not bound in the local scope
- Python closures help avoiding the usage of global values and provide some form of data hiding.

CLOSURE

When we define a function inside of another, the inner function is said to be nested inside the outer one. Let's take an *example.*

```
def outerfunc(x):
        def innerfunc():
                print(x)
        innerfunc()
outerfunc(7)
```

OUTPUT
7

CLOSURE

Innerfunc() could read the variable ‘x’, which is nonlocal to it. And if it must modify ‘x’, we declare that it’s nonlocal to innerfunc()

Let’s *define a closure*:-

```
def outerfunc(x):
        def innerfunc():
            print(x)
            return innerfunc        #Return the object instead of calling the function

myfunc=outerfunc(7)
 myfunc()
```

OUTPUT
7

CLOSURE

- The point to note here is that instead of calling innerfunc() here, we returned it (the object).
- Once we've defined outerfunc(), we call it with the argument 7 and store it in variable myfunc()
- when we call myfunc next, how does it remember that 'x' is 7?
- ***A Python3 closure is when some data gets attached to the code.***
- So, this value is remembered even when the variable goes out of scope, or the function is removed from the namespace.

CLOSURE

- Python closure is used when a nested function references a value in its enclosing scope.

- These *three* conditions must be met:

  1. We must have a nested function.
  2. This nested function must refer to a variable nonlocal to it(a variable in the scope enclosing it).
  3. The enclosing scope must return this function.

Benefits of Python Closure

While it seems like a very simple concept, a closure in python3 helps us in the following ways:

- With Python closure, we don’t need to use global values. This is because they let us refer to nonlocal variables. A closure then provides some form of data hiding.
- When we have only a few Python methods (usually, only one), we may use a Python3 closure instead of implementing a class for that. This makes it easier on the programmer.
- A closure, lets us implement a Python decorator.
- A closure lets us invoke Python function outside its scope.

Decorators(CO3)

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.
- Decorators are usually called before the definition of a function.

```
def uppercase_decorator(function):
    def wrapper():
            func = function()
            make_uppercase=func.upper()
            return make_uppercase
return wrapper
```

Note: Our decorator function takes a function as an argument, therefore, define a function and pass it to our decorator.

Call of Decorators(CO3)

```
def say_hi():
    return 'hello there'
decorate = uppercase_decorator(say_hi)


decorate()
```

Output:
'HELLO THERE'

Call of Decorators(CO3)

```
# call  of decoratots
@uppercase_decorator
def say_hi():
return 'hello there‘
```

```
say_hi()
```

Output:
‘HELLO THERE’

Applying Multiple Decorators to a Single Function(CO3)

```
def split_string(function):
    def wrapper():
            func = function()
            splitted_string = func.split()
            return splitted_string
return wrapper
@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'
say_hi()
```

Output:
['HELLO ','THERE']

Accepting Arguments in Decorator Functions(CO3)

```
def decorator_with_arguments(function):
    def wrapper_accepting_arguments (arg1, arg2):
      print("My arguments are: {0}, {1}".format(arg1,arg2))
      function(arg1, arg2)
      return wrapper_accepting_arguments
@decorator_with_arguments
def cities(city_one, city_two):
     print("Cities I love are {0} and {1}".format(city_one,
    city_two))
# call of Decorators
 cities("Nairobi", "Accra")
```

Output:

My arguments are: Nairobi, Accra Cities I love are Nairobi and Accra

Generator Functions

- A Python generator is a kind of an iterable, like a Python list or a python tuple. It generates for us a sequence of values that we can iterate on.
- You can use it to iterate on a for-loop in python, but you can't index it. Let's take a look at how to create one with python generator example.

- To create a python generator, we use the yield statement, inside a function, instead of the return statement.

```
>>> def counter():
        i=1
        while(i<=10):
            yield i
            i+=1
```

Generator Functions

- With this, we define a Python generator called counter() and assign 1 to the local variable i. As long as i is less than or equal to 5, the loop will execute.
- Inside the loop, we yield the value of i, and then increment it.

- Then, we iterate on this generator using the for-loop.

```
>>> for i in counter():
        print(i)
```

Output:

```
1
2
3
4
5
```

Working of Python Generator

- To understand how this code works, we'll start with the for-loop. For each item in the Python generator (each item that it yields), it prints it, here.
- We begin with i=1. So, the first item that it yields is 1. The for-loop prints this because of our print statement. Then, we increment I to 2.
- And the process follows until i is incremented to 11. Then, the while loop's condition becomes False.
- However, if you forget the statement to increment I, it results in an infinite generator. This is because a Python generator needs to hold only one value at a time.
- So, there are no memory restrictions.

Working of Python Generator

```
def even(x):
        while x%2==0:
                yield 'Even'
for i in even(2):
print(i)
```

since 2 is even, 2%2 is always 0. Hence, the condition for while is always true.

Output:

```
Even
Even
Even
Even
Even
EvenEvenTraceback (most recent call last):File "<pyshell#24>", line 2, in <module>print(i)
KeyboardInterrupt
```

Working of Python Generator

Generator may contain more than one Python yield statement. This is comparable to how a Python generator function may contain more than one return statement.

```
def my_gen(x):
        while(x>0):
                if x%2==0:
                        yield 'Even'
                else:
                        yield 'Odd'
        x-=1
for i in my_gen(7):
print(i)
```

Output: Odd
Even
Odd
Even
Even
Odd

Yielding into a Python List

- This one's a no-brainer. If you apply the list() function to the call to the Python generator, it will return a list of the yielded values, in the order in which they are yielded.
- Here, we take an example that creates a list of squares of numbers, on the condition that the squares are even.

```
def even_squares(x):
        for i in range(x):
            if i**2%2==0:
                yield i**2


print(list(even_squares(10)))
```

Output:
[0, 4, 16, 36, 64]

Python List vs Generator in Python

- A list holds a number of values at once. But a Python generator holds only one value at a time, the value to yield.

- This is why it needs much less space compared to a list. With a generator, we also don’t need to wait until all the values are rendered.

Introduction to Sub routine...

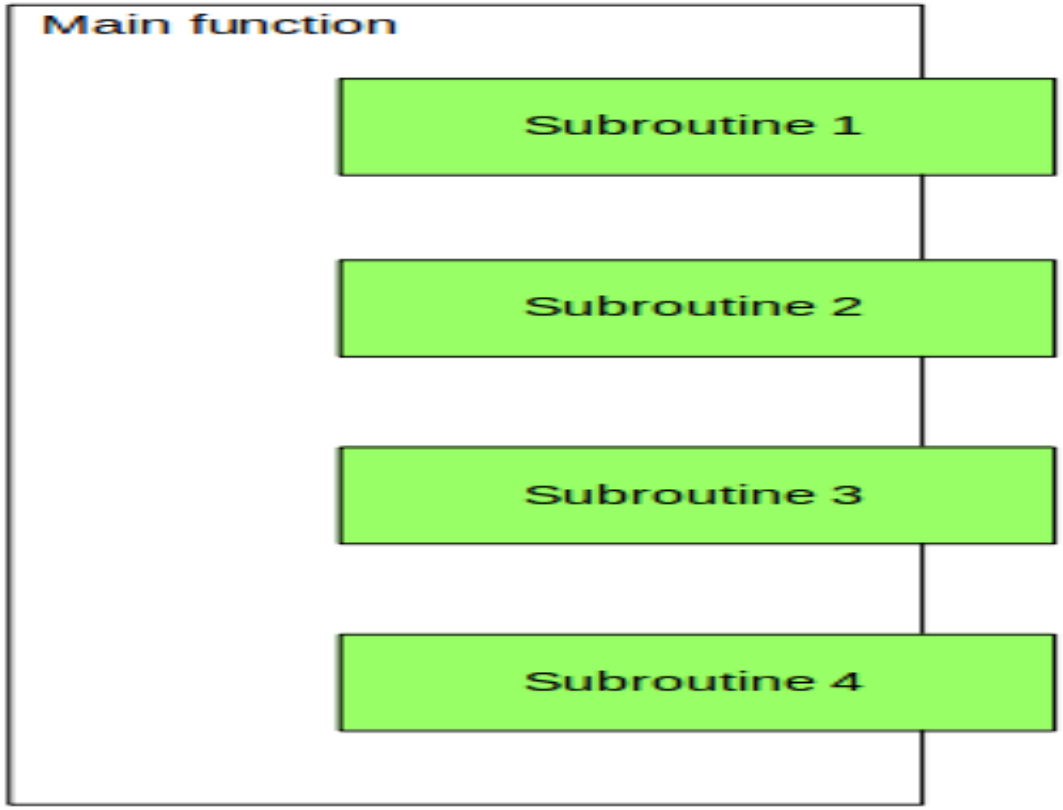
Function which is also known
as **subroutine**, **procedure**, **subprocess** etc."

A function is a sequence of instructions packed as a unit to perform a certain task.

When the logic of a complex function is divided into several self-contained steps that are themselves functions, then these functions are called helper functions or **subroutines**.

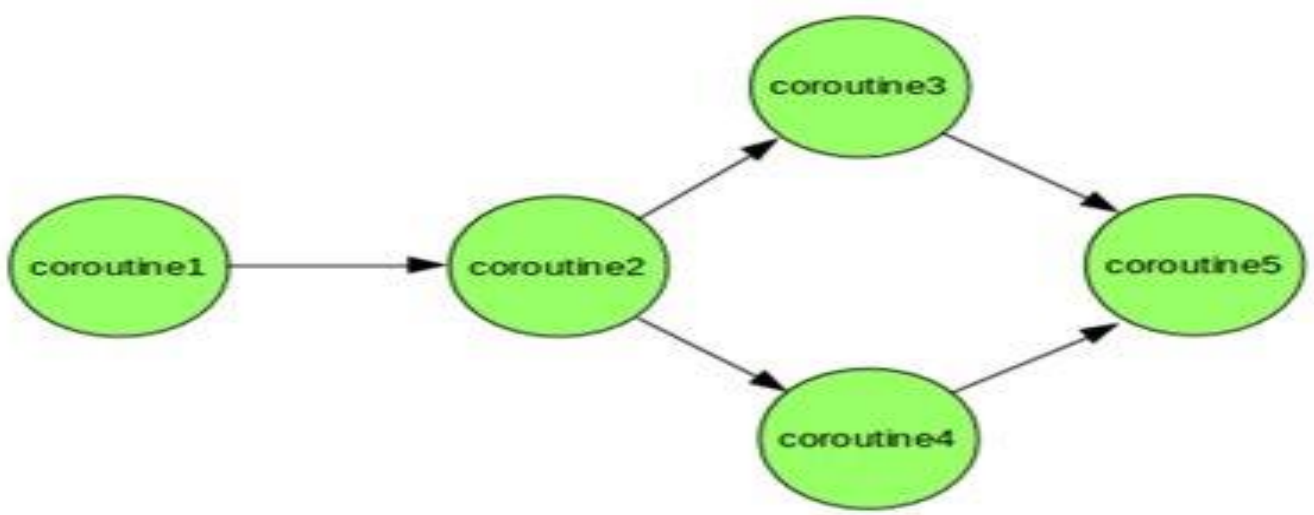
Introduction to Sub routine...

Subroutines in Python are called by **main function** which is responsible for coordination the use of these subroutines. Subroutines have single entry point.



Co routines ...

"Coroutines are generalization of subroutines." They are used for cooperative multitasking where a process voluntarily **yield** (give away) control periodically or when idle in order to enable multiple applications to be run simultaneously.



Python program for demonstrating # coroutine execution

```
def print name(prefix):
    print("Searching prefix:{}".format(prefix))
    while True:
        name = (yield)
        if prefix in name:
            print(name)

# calling coroutine, nothing will happen
corou = print_name("Dear")

# This will start execution of coroutine and
# Prints first line "Searchig prefix..."
# and advance execution to the first yield expression
```

Python program for demonstrating # coroutine execution ...

```
corou.__next__()

# sending inputs
corou.send(“XYZ")
corou.send("Dear XYZ")
```

Output:
Searching prefix:Dear Dear XYZ

Closing a Coroutine

```
def print_name(prefix):
    print("Searching prefix:{}".format(prefix))
    try :
        while True:
            name = (yield)
            if prefix in name:
                print(name)
    except GeneratorExit:
        print("Closing coroutine!!")
```

```
corou = print_name("Dear")
corou.__next__()
corou.send("Atul")
corou.send("Dear Atul")
corou.close()
```

OUTPUT
Searching prefix:Dear Dear XYZ Closing coroutine!!

Iterator

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().

Iterator vs Iterable

- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.
- All these objects have a iter() method which is used to get an iterator:

Example:

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

OUTPUT

Apple
Banana
cherry

Iterator vs Iterable

“Even strings are iterable objects, and can return an iterator”

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example:

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
    print(x)
```

Output:

```
apple
banana
cherry
```

Create an Iterator

- To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.
- All classes have a function called __init__(), which allows to do some initializing when the object is being created.
- The __iter__() method acts similar, and can do operations (initializing etc.), but must always return the iterator object itself.
- The __next__() method also allows to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will

increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

def __next__(self):
    x = self.a
    self.a += 1
    return x.
```

Example

```
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:-

```
1
2
3
4
5
```

Stop Iteration

- The example above would continue forever if you had enough next() statements, or if it was used in a for loop.
- To prevent the iteration to go on forever, we can use the StopIteration statement.
- In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

StopIteration

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
  print(x)
```

Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Declarative Programming Language

Declarative Programming Languages focus on :

- ***describing what should be computed*** - and avoid mentioning how that computation should be performed. This means avoiding expressions of control flow: loops and conditional statements are removed and replaced with higher level constructs that describe the logic of what needs to be computed.
- The usual ***example*** of a declarative programming language is ***SQL***. It lets you define what data you want computed - and translates that efficiently onto the database schema.
- ***Python isn't a pure Declarative Language*** - but the same flexibility that contributes to its sluggish speed can be be leveraged to create Domain Specific API's that use the same principles.