Advanced OOP Concepts

UNIT -1

Contents

- Decorators
- Generators
- Context Managers
- Function and Attribute Decorators
- Multiple Inheritance
- Abstract classes and interfaces

Iterators

- Iterators are methods that iterate collections like lists, tuples, etc. Using an iterator method, we can loop through an object and return its elements.
- Technically, a Python **iterator object** must implement two special methods, __iter__() and __next__(), collectively called the **iterator protocol.**

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

- The for loop actually creates an iterator object and executes the next() method for each loop.

Iterate the characters of a string:

```
mystr = "banana"
for x in mystr:
  print(x)
```

Create an Iterator

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

```
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)
```

Generators

- In Python, a generator is a function that returns an iterator that produces a sequence of values when iterated over.
- Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.
- In Python, similar to defining a normal function, we can define a generator function using the def keyword, but instead of the return statement we use yield statement.

```
def function_name():
        yield statement
```

- Here, the yield keyword is used to produce a value from the generator.
- When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

```
def gen_fun():
    yield 10
    yield 20
    yield 30
#using for loop
for i in gen_fun():
    print(i)
#using next function
x=gen_fun()
print(next(x))
print(next(x))
print(next(x))
```

Generators Examples

- In Python, there are some ways to create a generator function.

1. Yield Statement

- Yield keyword in Python that is used to return from the function without destroying the state of a local variable.

2. Python Generator Expression

- Generator Expression is a short-hand form of a generator function. They are similar to lambda function. They are implemented similarly to list comprehension does, and the only difference in implementation is, **instead of using square brackets('[]'), it uses round brackets('()')**.
- The main and important difference between list comprehension and generator expression is list comprehension returns a list of items, whereas generator expression returns an iterable object.

```
def simple():
for no in range(10):
   if(no%2==0):
       yield no
#Successive Function call using for loop
for i in simple():
   print(i)
```

```
x = 10
#generator expression
gen = (i for i in range(x) if i % 2 == 0)
#list comprehension
list_ = [i for i in range(x) if i % 2 == 0]
print(gen)
print(list_)
for j in gen:
   print(j)
```

Decorators

- Decorators are functions (or classes) that provide enhanced functionality to the original function (or class) without the programmer having to modify their structure.
- Accepting functions as arguments, defining functions within other functions, and returning them.
- Python allows to **use decorator in easy way with @symbol**.

```
def upper_case_decorator(original_function):
    def wrapper():
        result = original_function().upper()          #result = result.upper()
        return result
    return wrapper
@upper_case_decorator
def greeting():
    return 'hello Dharma'
print(greeting())
```

Class Decorators

- Python provides two ways to decorate a class. Firstly, we can decorate the method inside a class; there are built-in decorators like **@classmethod, @staticmethod** and **@property** in Python.
- The **@classmethod** and **@staticmethod** define methods inside class that is not connected to any other instance of a class.
- The @property is generally used to modify the getters and setters of a class attributes.

Decorator	Description
@property	Declares a method as a property's setter or getter methods.
@classmethod	Declares a method as a class's method that can be called using the class name.
@staticmethod	Declares a method as a static method.

Attribute Decorators

- The @property is generally used to modify the getters and setters of a class attributes.
- Python's property() built-in function is an interface for accessing instance variables of a class. The @property decorator turns an instance method into a "getter" for a read-only attribute with the same name, and it sets the docstring for the property to "Get the current value of the instance variable."
- Use @property decorator on any method in the class to use the method as a property.
- You can use the following three decorators to define a property:
  - @property: Declares the method as a property.
  - @<property-name>.setter: Specifies the setter method for a property that sets the value to a property.
  - @<property-name>.deleter: Specifies the delete method as a property that deletes a property.

Attribute Decorators

```
class Student:
def __init__(self, name):
                self.__name = name
@property
def name(self):
                return self.__name
@name.setter
def name(self, value):
                self.__name=value
@name.deleter
#property-name.deleter decorator
def name(self):
                print('Deleting..’)
                del self.__name
std = Student('Steve’)
print(s.name)
s.name=“Bill”
Print(s.name)
del std.name
print(std.name) #AttributeError
```

Class Method Decorators

- In Python, the @classmethod decorator is used to declare a method in the class as a class method that can be called using ClassName.MethodName().
- The method received an implicit object referred to by self. A class method on the other hand implicitly receives the class itself as first argument.

```
class Student:
name = 'unknown'   # class attribute
def __init__(self):
        self.age = 20 # instance attribute
@classmethod
def tostring(cls):
        print('Student Class Attributes: name=',cls.name)
Student.tostring() #Student Class Attributes:
name=unknown
```

Static Method Decorators

- The @staticmethod is a built-in decorator that defines a static method in the class in Python. A static method doesn't receive any reference argument whether it is called by an instance of a class or by the class itself.
- The static method cannot access the class attributes or the instance attributes.
- The static method can be called using ClassName.MethodName() and also using object.MethodName().
- It can return an object of the class.

```
class Student:
name = 'unknown'   # class attribute
def __init__(self):
        self.age = 20 # instance attribute
@staticmethod
def tostring(cls):
        print('Student Class Attributes: name=',cls.name)
Student.tostring() #Student Class Attributes:
name=unknown
```

Context Managers

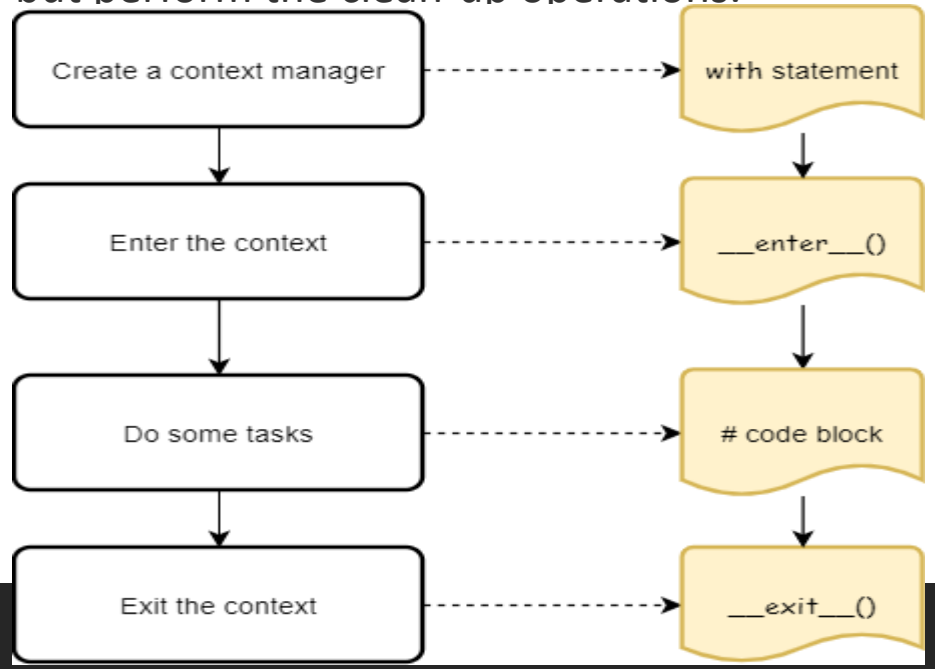
- **Context manager** in python programming language works as the automatic resource releaser.
- Most of the resources of python programming language are limited in supply such as files or databases and if they are not released then it may lead to resource leakage which will bring the problem of slow downing the program or may crash the program.
- Context manager python can be implemented using **classes** and will automatically release the resource after the use.

```
file_descriptors = []
for x in range(100000):
        file_descriptors.append(open('test.txt', 'w'))
```

- The above example is the case of the file descriptors leakage. An error occurred saying, "Too many open files". This has happened because there are so many open files, and they are not closed.

How to create Context Managers

- When the user creates a context manager, they need to make sure that the class has the following methods: __enter__() and __exit__().
- The **__enter__()** method will return the resource that has to be managed.
- The **__exit__()** method will not return anything but perform the clean-up operations.



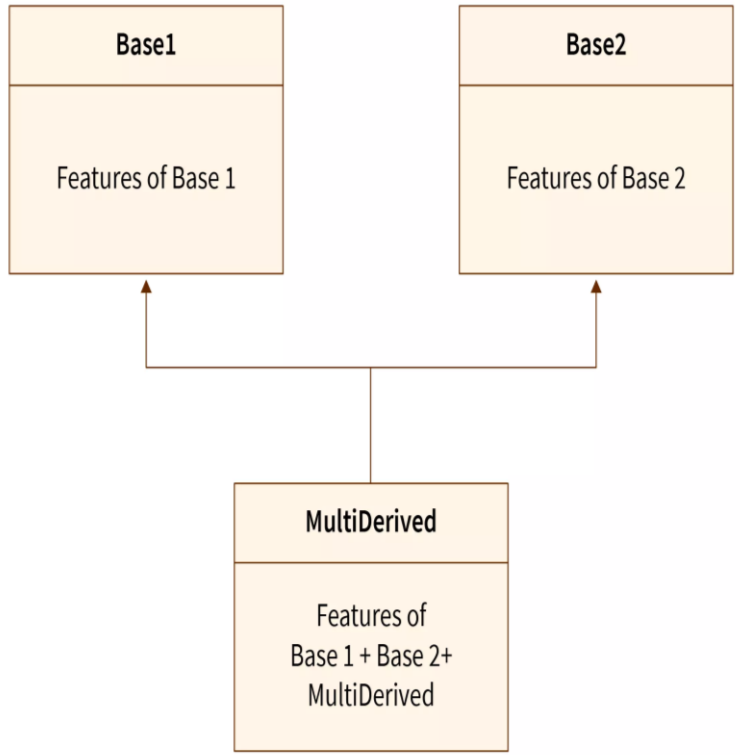

```
# Python program showing  file management using  context manager
class FileManager():
        def __init__(self, filename, mode):
                self.filename = filename
                self.mode = mode
                self.file = None
        def __enter__(self):
                self.file = open(self.filename, self.mode)
                return self.file
        def __exit__(self, exc_type, exc_value, exc_traceback):
                self.file.close()
# loading a file
with FileManager('test.txt', 'w') as f:
        f.write('Test')
print(f.closed)
```

Multiple Inheritance

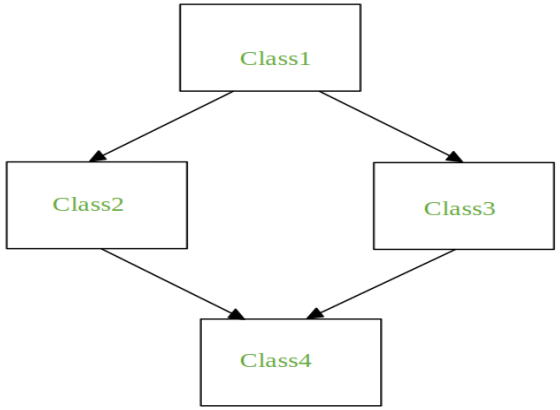
- When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all features of the base classes are inherited into the derived class.

```
# Python program to demonstrate multiple inheritance
class Mother:
        mothername = "“
        def mother(self):
                print(self.mothername)
class Father:
        fathername = ""
        def father(self):
                print(self.fathername)
class Son(Mother, Father):
        def parents(self):
                print("Father :", self.fathername)
                print("Mother :", self.mothername)
```

```
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```



Problems in Multiple Inheritance



- **The Diamond Problem:** It refers to ambiguity that arises when two classes Class2 & Class3 inherit from superclass Class1 & class Class4 inherits from Class2 & Class3.

```
# Python Program to depict multiple inheritance when every class defines the same method
class Class1:
        def m(self):
                print("In Class1")
class Class2(Class1):
        def m(self):
                print("In Class2")
class Class3(Class1):
        def m(self):
                print("In Class3")
class Class4(Class2, Class3):
        def m(self):
                print("In Class4")
obj = Class4()
obj.m()
```

Output :

In Class4

Object of derived class is created, and the method name is same in each class. So, it will call derived class method.

Super() in Multiple Inheritance

- Python provides a solution to the above problem with the help of the super() function.
- Using the super() function, we can get a *temporary* object of the parent class / superclass, and this way, we can get access to all of the methods of the superclass to its child class.

```
# Python program to demonstrate super()
class Class1:
        def m(self):
                print("In Class1")
class Class2(Class1):
        def m(self):
                print("In Class2")
                super().m()
class Class3(Class1):
        def m(self):
                print("In Class3")
                super().m()
```

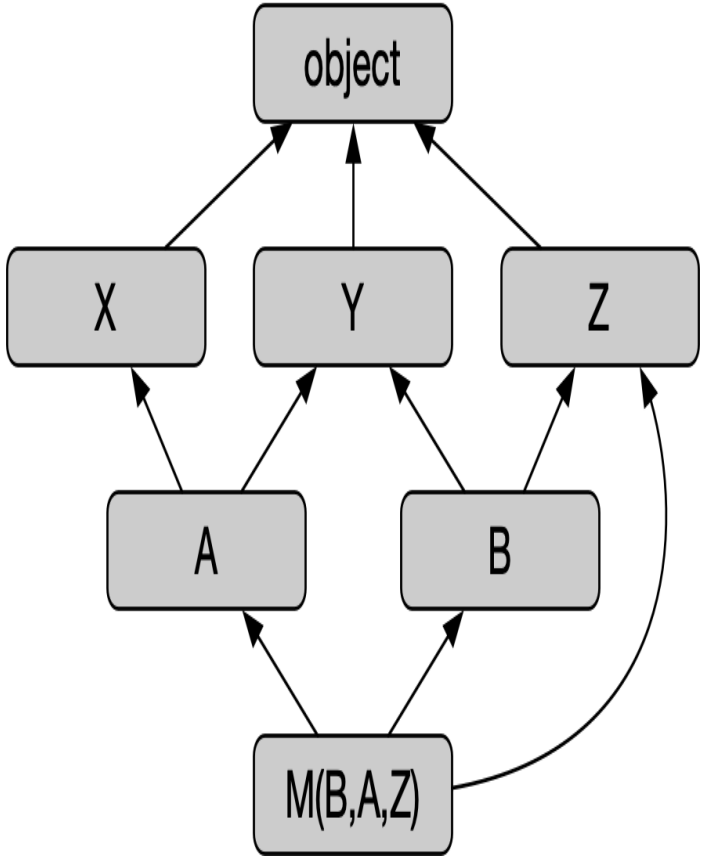
```
class Class4(Class2, Class3):
        def m(self):
                print("In Class4")
                super().m()
obj = Class4()
obj.m()
```

Output :

```
In Class4
In Class2
In Class3
In Class1
```

Method Resolution Order(MRO)

- In Python, every class whether built-in or user-defined is derived from the object class and all the objects are instances of the class object. Hence, **the object class is the base class for all the other classes.**
- In the case of multiple inheritance, a given attribute is first searched in the current class if it's not found then it's searched in the parent classes. The parent classes are searched in a **left-right fashion** and each class is searched once.
- The order that is followed is known as a linearization of the class Derived and this order is found out using a set of rules **called Method Resolution Order (MRO).**



MRO Example

```
#Method resolution order
class X(object):
    def method(self):
        print("X class")
        super().method()
class Y(object):
    def method(self):
        print("Y class")
        super().method()
class Z(object):
    def method(self):
        print("Z class")
```

```
class A(X,Y):
    def method(self):
        print("A class")
        super().method()
class B(Y,Z):
    def method(self):
        print("B class")
        super().method()
class M(B,A,Z):
    def method(self):
        print("M class")
        super().method()
m1=M()
m1.method()
print(M.mro())
```

Output :

```
M class
B Class
A Class
X Class
Y Class
Z Class
[<class '__main__.M'>, <class
'__main__.B'>, <class
'__main__.A'>, <class
'__main__.X'>, <class
'__main__.Y'>, <class
'__main__.Z'>, <class
'object'>]
```

Abstract Class (1)

- Abstract classes are classes that contain one or more abstract methods.
- Abstract Class is a class contains abstract methods whose implementation is later defined in the sub classes. Abstract Class may also contain concrete method which is a method with body.
- Object of Abstract Class cannot be created.
- Abstract classes may not be instantiated, and required to create sub classes and all the abstract methods should be implemented in the sub classes. Then its possible to create objects to the sub classes.
- An abstract method is a method that is declared but contains no implementation.
- Abstract method is redefined in the sub classes as per the requirement of the objects.
- To mark a method as abstract, we should use the decoration @abstractmethod.

Abstract Class (2)

- The way to create an abstract class is to derive it from a meta class ABC that belongs to abc (abstract base class) module, so we should import this module into the program.

```
# import abc module's ABC class and abstract method
  from abc import ABC, abstractmethod
                    OR
  from abc import *
#abstract class should be derived from the ABC class
Class Myclass(ABC):
#used @abstractmethod decorator to specify that this is an abstract method
      @abstractmethod
       def calculate(self):
            pass       #empty body no code
```

Abstract Class Example

```
# Python program showing abstract base class work
from abc import ABC, abstractmethod
class Polygon(ABC):
        @abstractmethod
        def noofsides(self):
                pass
class Triangle(Polygon):
        # overriding abstract method
        def noofsides(self):
                print("I have 3 sides")
class Pentagon(Polygon):
        # overriding abstract method
        def noofsides(self):
                print("I have 5 sides")
```

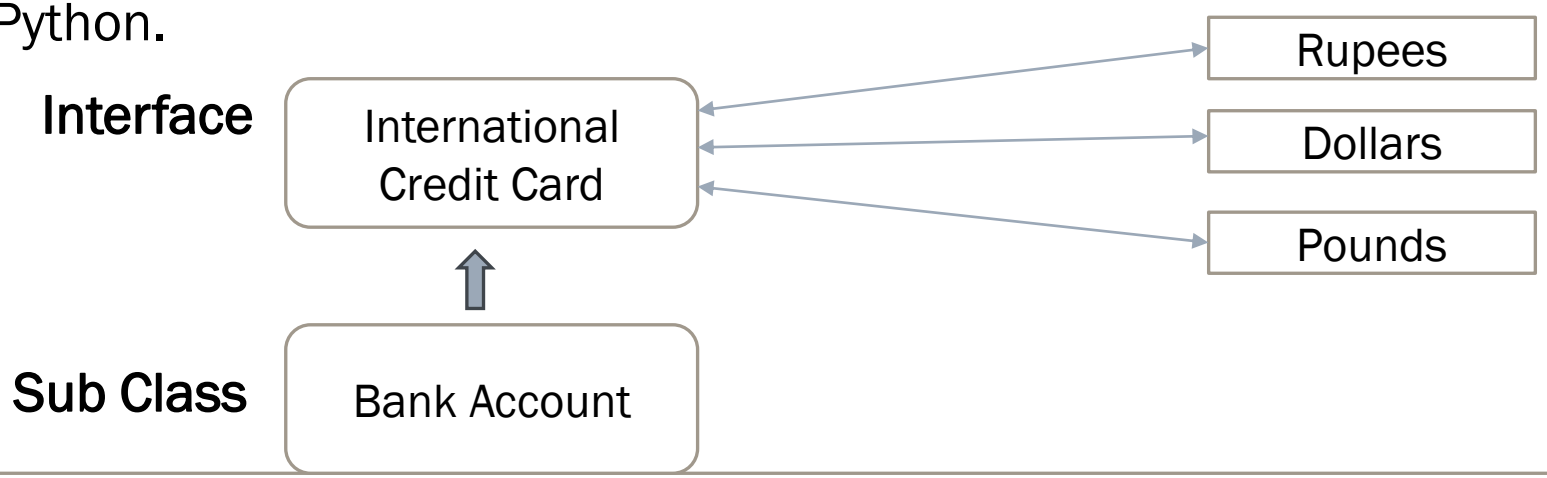
```
# Driver code
R = Triangle()
R.noofsides()

R = Pentagon()
R.noofsides()
```

Output

```
I have 3 sides
I have 5 sides
```

Interface

- Interface is an abstract class, but it contains only abstract methods.
- Since None of the methods in the interface will have body, it is not possible to create objects to an interface.
- In this case, we can create sub classes where we can implement all the methods of the interface.
- The interface concept is not explicitly available in python, we must use abstract classes as interface in Python.



Interface Example

```
from abc import ABC, abstractmethod

class Shape(ABC):
@abstractmethod
def area(self):
pass

class Rectangle(Shape):
def __init__(self, width, height):
self.width = width
self.height = height

def area(self):
return self.width * self.height
```

```
R1= Rectangle(5,10)

print(R1.area())
```

Output:

50