Python Programming
Unit 4
(KNC-302)

Prepared By

Abhishek Kesharwani

Assistant Professor,UCER Naini,Allahabad

UNIT IV

- Modules: Introduction,
- Importing Modules,
- Abstract Data Types : Abstract data types and ADT interface in Python Programming.
- Classes : Class definition and other operations in the classes
- Special Methods (such as _init_,_str_, comparison methods and Arithmetic methods etc.)
- Class Example , Inheritance , Inheritance and OOP.

Python Modules

- A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module.
- We may have a runnable code inside the python module.
- Modules in Python provides us the flexibility to organize the code in a logical way.
- To use the functionality of one module into another, we must have to import the specific module.

Loading the module in our python code

Python provides two types of statements as defined below.

- The import statement
- The from-import statement

The import statement

- The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.
- We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

- The syntax to use the import statement is given below.

```
import module1,module2,..... module n
```

Let's create the module named as **file.py.**

#displayMsg prints a message to the name being passed.

```
def displayMsg(name)
    print("Hi "+name);
```

```
import file;
name = input("Enter the name?")
file.displayMsg(name)
```

Output:

```
Enter the name John
Hi John
```

The from-import statement

- Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module.
- This can be done by using from? import statement. The syntax to use the from-import statement is given below.

from < module-name> import <name 1>, <name 2>..,<name n>

calculation.py

```
def summation(a,b):
    return a+b
def multiplication(a,b):
    return a*b;
def divide(a,b):
    return a/b;
```

Main.py

```
from calculation import summation
#it will import only the summation() from calculatio
n.py
a = int(input("Enter the first number"))
b = int(input("Enter the second number"))
print("Sum = ",summation(a,b)) #we do not need
to specify the module name while accessing summa
tion()
```

Output

Enter the first number 10

Enter the second number 20

Sum = 30

NOTE: We can also import all the attributes from a module by using *.

from <module> **import** *

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

syntax

import <module-name> as <specific-name>

dir() function

- The dir() function returns a sorted list of names defined in the passed module.
- This list contains all the sub-modules, variables and functions defined in this module.

Example

```
import json
List = dir(json)
print(List)
```

Output

['JSONDecoder', 'JSONEncoder', '__all__',
'__author__', '__builtins__', '__cached__',
'__doc__','__file__', '__loader__', '__name__',
'__package__', '__path__', '__spec__',
'__version__', '_default_decoder',
'_default_encoder', 'decoder', 'dump', 'dumps',
'encoder', 'load', 'loads', 'scanner']

Python OOPs Concepts

- Python is an object-oriented programming language.
- It allows us to develop applications using an Object Oriented approach.
- In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below.

- Object
- Class
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Object

- The object is an entity that has state and behavior.
- It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
- Everything in Python is an object, and almost everything has attributes and methods.
- All functions have a built-in attribute __doc__ which returns the doc string defined in the function source code.

Class

- It is a logical entity that has some specific attributes and methods.

For example:

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

Method

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

Inheritance

- Inheritance is the most important aspect of object-oriented programming which simulates the real world concept of inheritance.
- It specifies that the child object acquires all the properties and behaviors of the parent object.
- By using inheritance, we can create a class which uses all the properties and behavior of another class.

- The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.
- It provides re-usability of the code.

Polymorphism

- Polymorphism contains two words "poly" and "morphs". Poly means many and Morphs means form, shape.
- By polymorphism, we understand that one task can be performed in different ways.

Encapsulation

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

Data Abstraction

- Data abstraction and encapsulation both are often used as synonyms.
- Both are nearly synonym because data abstraction is achieved through encapsulation.
- Abstraction is used to hide internal details and show only functionalities.
- Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

Creating classes in python

- In python, a class can be created by using the keyword class followed by the class name.

Syntax

```
class ClassName:
    #statement_suite
```

A class contains a statement suite including fields, constructor, function, etc. definition.

Example

```
class Employee:
    id = 10;
    name = "ayush"
    def display (self):
        print(self.id,self.name)
```

NOTE: self is used as a reference variable which refers to the current class object. It is always the first argument in the function definition

Creating an instance of the class

- A class needs to be instantiated if we want to use the class attributes in another class or method.
- A class can be instantiated by calling the class using the class name.

Example

```
class Employee:
    id = 10;
    name = "John"
    def display (self):
        print("ID: %d \nName:
%s"%(self.id,self.name))
emp = Employee()
emp.display()
```

Output

ID: 10

Name: ayush

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

- Parameterized Constructor
- Non-parameterized Constructor

Creating the constructor in python

- Constructor definition is executed when we create the object of this class.
- Constructors also verify that there are enough resources for the object to perform any start-up task.
- **__init__** simulates the constructor of the class. This method is called when the class is instantiated.
- It is mostly used to initialize the class attributes.
- Every class must have a constructor, even if it simply relies on the default constructor.

Example

```
class Employee:
    def __init__(self,name,id):
        self.id = id;
        self.name = name;
    def display (self):
        print("ID: %d \nName: %s"%(self.
id,self.name))
emp1 = Employee("John",101)
emp2 = Employee("David",102)
emp1.display();
emp2.display();
```

Output

ID: 101

Name: John

ID: 102

Name: David

Python Non-Parameterized Constructor

```
class Student:
    def __init__(self):
        print("This is non parametrized constructor")

    def show(self,name):
        print("Hello",name)
st1 = Student()

st1.show("John")
```

Output

This is non parametrized constructor

Hello John

Parameterized Constructor Example

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("John")
student.show()
```

Output

This is parametrized constructor

Hello John

Python In-built class functions

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Example

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
s = Student("John",101,22)
print(getattr(s,'name'))
setattr(s,"age",23)
print(getattr(s,'age'))
print(hasattr(s,'id'))
delattr(s,'age')
print(s.age)
```

Output

John

23

True

AttributeError: 'Student' object has no attribute 'age'

```
class Student:
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
        self.marks=marks
s1=Student("Rahul",1,65)
print(getattr(s1,'name'))
print(hasattr(s1,'roll'))
setattr(s1,'roll',3)
print(s1.roll)
delattr(s1,'roll')
print(s1.roll)
```

Built-in class attributes

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

Example

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(se
lf.name,self.id))
s = Student("John",101,22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

Output

None

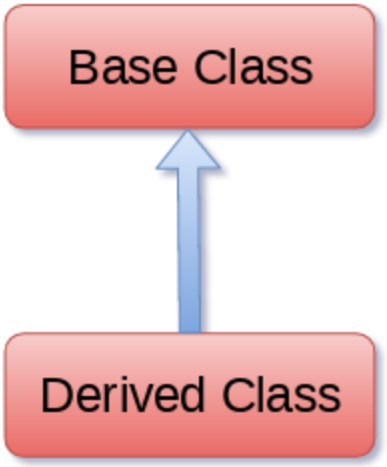
{'name': 'John', 'id': 101, 'age': 22}

__main__

Python Inheritance

- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.

Syntax



class derived-**class**(base **class**):
    <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket.

Syntax

```
class derive-class
(<base class 1>, <base class 2>, ..... <base class n>):
    <class - suite>
```

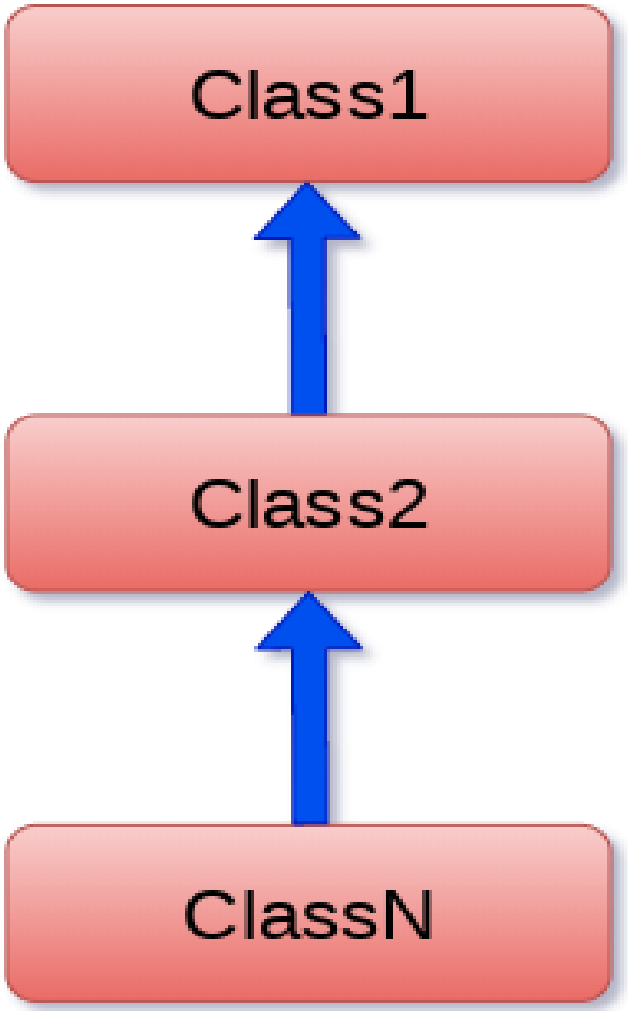
Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class A
nimal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Python Multi-Level inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Multi-Level inheritance



Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

Example

```
1. class Animal:
2.     def speak(self):
3.         print("Animal Speaking")
4. #The child class Dog inherits the base class Animal
5. class Dog(Animal):
6.     def bark(self):
7.         print("dog barking")
8. #The child class Dogchild inherits another child class Dog
9. class DogChild(Dog):
10.    def eat(self):
11.        print("Eating bread...")
12.d = DogChild()
13.d.bark()
14.d.speak()
15.d.eat()
```

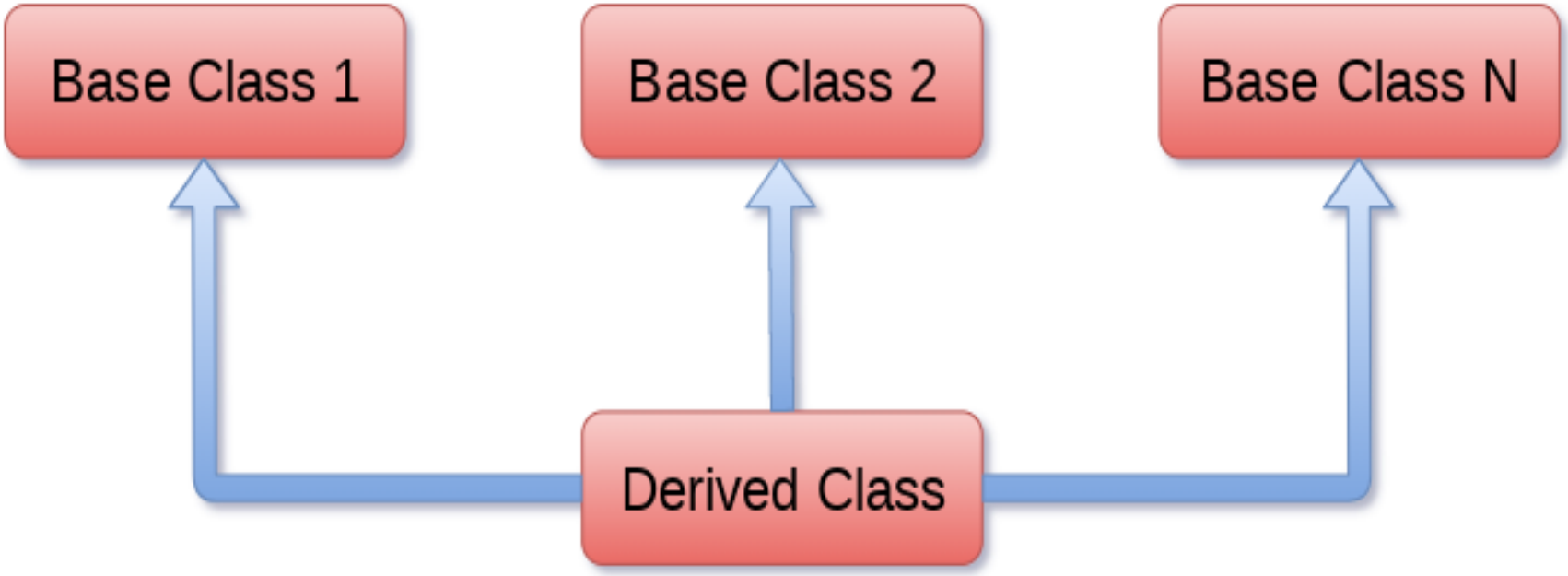
Output

dog barking

Animal Speaking

Eating bread..

Python Multiple inheritance



Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b
class Calculation2:
    def Multiplication(self,a,b):
        return a*b
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

issubclass(sub,sup) method

- The issubclass(sub, sup) method is used to check the relationships between the specified classes.
- It returns true if the first class is the subclass of the second class, and false otherwise.

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

Output

- True
- False

Method Overriding

- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Example

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

Output

- Barking

Data abstraction in python

- Abstraction is an important aspect of object-oriented programming.
- In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden.
- After this, the attribute will not be visible outside of the class through the object.

```
class Employee:
    __count = 0;
    def __init__(self):
        Employee.__count = Employee.__count+1
    def display(self):
        print("The number of employees",Employee
.__count)
emp = Employee()
emp2 = Employee()
try:
    print(emp.__count)
finally:
    emp.display()
```

Output

- The number of employees 2
- AttributeError: 'Employee' object has no attribute '__count'