Unit 4

Python OOPs Concepts

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc.

Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	Procedural-Oriented Programming (Pop)
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of *Access modifiers* 'public', private', protected'	Doesn't use *Access modifiers*
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

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

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

Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

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.

The syntax to create the instance of the class is given below.

1. <object-name> = <**class**-name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

Example

```
1. class Employee:
2.     id = 10
3.     name = "John"
4.     def display (self):
```

```
5.         print("ID: %d \nName: %s"%(self.id,self.name))
6.  # Creating a emp instance of Employee class
7.  emp = Employee()
8.  emp.display()
```

Output:

```
ID: 10
Name: John
```

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

Delete the Object

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

Example

```
class Employee:
    id = 10
    name = "John"

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))
    # Creating a emp instance of Employee class

emp = Employee()

# Deleting the property of object
del emp.id
# Deleting the object itself
```

del emp

emp.display()

It will through the Attribute error because we have deleted the object **emp**.

Python Constructor

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

Constructor is a special type of function that is called automatically whenever an object of that class is created.

Types of Constructors:

- Parameterized Constructor
- Non- Parameterized Constructor

Features of Python Constructors:

- In Python, a Constructor begins with double underscore (_) and is always named as __init__().
- In python Constructors, arguments can also be passed.
- In Python, every class must necessarily have a Constructor.
- If there is a Python class without a Constructor, a default Constructor is automatically created without any arguments and parameters.

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

Example

```
class Student:

    # Constructor - non parameterized

    def __init__(self):

        print("This is non parametrized constructor")
```

```
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("John")
```

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

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()
```

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

Example

```
class Student:
    roll_num = 101
    name = "Joseph"

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

st = Student()
```

1. st.display()

Note: The constructor overloading is not allowed in Python.

- **Example:**

```
class Employees():
  def __init__(self, Name, Salary):
     self.Name = Name
     self.Salary = Salary

  def details(self):
     print "Employee Name : ", self.Name
     print "Employee Salary: ", self.Salary
     print "\n"

first = Employees("Khush", 10000)
second = Employees("Raam", 20000)
third = Employees("Lav", 10000)
fourth = Employees("Sita", 30000)
fifth = Employees("Lucky", 50000)

first.details()
second.details()
third.details()
fourth.details()
fifth.details()
```

Inheritance in Python

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.

The class which inherits another class is generally known as the Child class, while the class which is inherited by other classes is called as the Parent class.

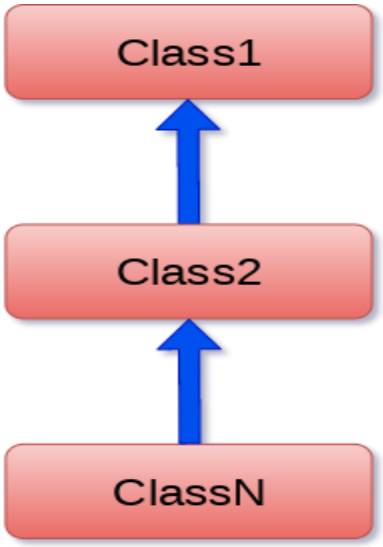
Types of Inheritance in Python

In Python, there are two types of Inheritance:

1. Multiple Inheritance
2. Multilevel Inheritance

Python Multi-Level inheritance

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.
In multilevel inheritance, we inherit the classes at multiple separate levels. We have three classes A, B and C, where A is the super class, B is its sub(child) class and C is the sub class of B.



Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
```

d.bark()

d.speak()

d.eat()

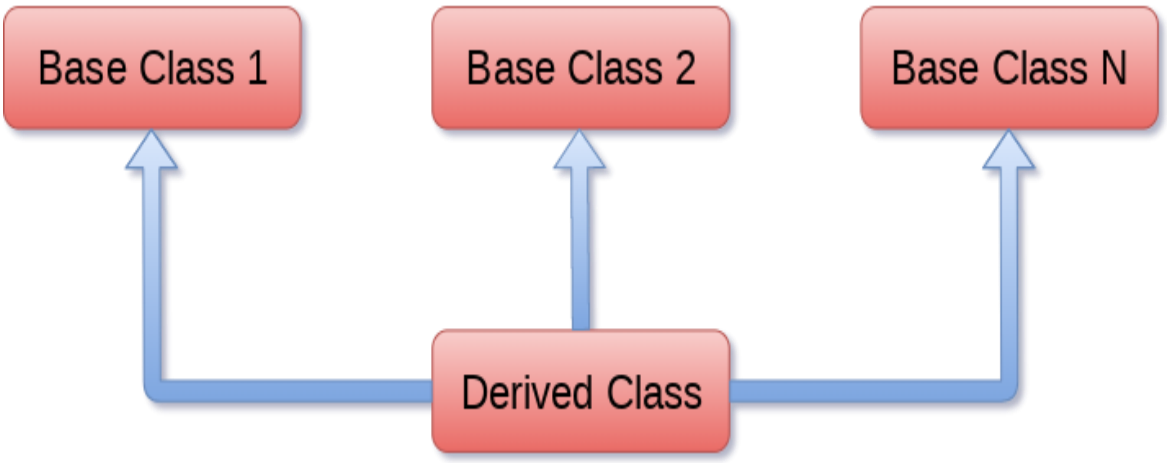
Python - Multiple Inheritance

Multiple Inheritance means that you're inheriting the property of multiple classes into one. In case you have two classes, say A and B, and you want to create a new class which inherits the properties of both A and B, then:

```
class A:

    # variable of class A

    # functions of class A



class B:

    # variable of class A

    # functions of class A



class C(A, B):

    # class C inheriting property of both class A and B
```

```
# add more properties to class C
```

Python provides us the flexibility to inherit multiple base classes in the child class.



The syntax to perform multiple inheritance is given below.

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))
```

Using issubclass() method

In python, there is a function which helps us to verify whether a particular class is a sub class of another class, that built-in function is issubclass(paramOne, paramTwo), where paramOne and paramTwo can be either class names or class's object name.

```
class Parent:

        var1 = 1

        def func1(self):

            # do something



class Child(Parent):

        var2 = 2

        def func2(self):

            # do something else
```

In order to check if Child class is a child class of Parent class.

```
>>> issubclass(Child, Parent)
```

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))
```

Method Overriding

We can provide some specific implementation of the parent class method in our child class. 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.

Consider the following example to perform method overriding in python.

Example

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

Output:

```
Barking
```

Real Life Example of method overriding

```
class Bank:

    def getroi(self):

        return 10;

class SBI(Bank):

    def getroi(self):

        return 7;


class ICICI(Bank):

    def getroi(self):

        return 8;

b1 = Bank()

b2 = SBI()

b3 = ICICI()

print("Bank Rate of interest:",b1.getroi());

print("SBI Rate of interest:",b2.getroi());

print("ICICI Rate of interest:",b3.getroi());
```

Python Error and In-built Exception in Python

A small typing mistake can lead to an error in any programming language because we must follow the syntax rules while coding in any programming language.

Python: Syntax Error

This is the most common and basic error situation where you break any syntax rule like if you are working with Python 3.x version and you write the following code for printing any statement,

```
print "I love Python!"
```

```
SyntaxError: Missing parentheses in call to 'print'.
```

Because, Python 3 onwards the syntax for using the print statement has changed. Similarly if you forget to add colon(:) at the end of the if condition, you will get a **SyntaxError**:

```
if 7 > 5

    print("Yo Yo!")
```

```
SyntaxError: invalid syntax
```

Hence syntax errors are the most basic type of errors that you will encounter while coding in python and these can easily be fixed by seeing the error message and correcting the code as per python syntax.

Python: What is an Exception?

Contrary to syntax error, exception is a type of error which is caused as a result of malfunctioning of the code during execution.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed.

Therefore, an exception is the run-time errors that are unable to handle to Python script.

An exception is a Python object that represents an error.

Your code might not have any syntax error, still it can lead to exceptions when it is executed.

Let's take the most basic example of **dividing a number by zero**:

```
a = 10

b = 0

print("Result of Division: " + str(a/b))
```

```
Traceback (most recent call last):

  File "main.py", line 3, in <module>
```

```
    print("Result of Division: " + str(a/b))

ZeroDivisionError: division by zero
```

As we can see in the output, we got **ZeroDivisionError** while the syntax of our python code was absolutely correct, because in this case the error or we should say the exception was generated while code execution.

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

Python Exception Handling

Exception handling is a concept used in Python to handle the exceptions and errors that occur during the execution of any program. Exceptions are unexpected errors that can occur during code execution.

Handling Exceptions using try and except

For handling exceptions in Python we use two types of blocks, namely, try block and except block.

In the try block, we write the code in which there are chances of any error or exception to occur.

Whereas the except block is responsible for catching the exception and executing the statements specified inside it.

Below we have the code performing **division by zero**:

```
a = 10

b = 0

print("Result of Division: " + str(a/b))
```

Copy

```
Traceback (most recent call last):
File "main.py", line 3, in <module>
print("Result of Division: " + str(a/b))
ZeroDivisionError: division by zero
```

The above code leads to exception and the exception message is printed as output on the console.

If we use the try and except block, we can handle this exception gracefully.

```
# try block

try:

    a = 10

    b = 0

    print("Result of Division: " + str(a/b))

except:

    print("You have divided a number by zero, which is not allowed.")
```

```
You have divided a number by zero, which is not allowed.
```

Example:

```
# try block
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
    print("You have divided a number by zero, which is not allowed.")
# except block handling wrong value type
except(ValueError):
    print("You must enter integer value")
```

Python File Handling

In Python, there is no need for importing external library to read and write files.

Python provides an inbuilt function for creating, writing, and reading files.

File Handling

The key function for working with files in Python is the open() function.

The open() function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

```
"r" - Read - Default value. Opens a file for reading, error if the file does not exist
"a" - Append - Opens a file for appending, creates the file if it does not exist
"w" - Write - Opens a file for writing, creates the file if it does not exist
```

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the readline() method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling readline() two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()
```

```
#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Python Delete File

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the os.rmdir() method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.