PYTHON PROGRAMMING
UNIT –IV

File Operations: Reading config files in python, Writing log files in python, Understanding read functions, read(), readline() and readlines(), Understanding write functions, write() and writelines(), Manipulating file pointer using seek, Programming using file operations
Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance , overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support
Design with Classes: Objects and Classes, Data modeling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism.

File Operations :

- A file is a collection of records. A record is a group of related data items.
- Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character (‘\n’) in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

Configuration file parser in Python (config parser)

The config parser module from Python's standard library defines functionality for reading and writing configuration files as used by Microsoft Windows OS. Such files usually have .INI extension.

The INI file consists of sections, each led by a [section] header. Between square brackets, we can put the section’s name. Section is followed by key/value entries separated by = or : character. It may include comments, prefixed by # or ; symbol. A sample INI file is shown below −

```
[Settings]
# Set detailed log for additional debugging info
DetailedLog=1
RunStatus=1
StatusPort=6090
StatusRefresh=10
Archive=1
# Sets the location of the MV_FTP log file
LogFile=/opt/ecs/mvuser/MV_IPTel/log/MV_IPTel.log
Version=0.9 Build 4
ServerName=Unknown

[FTP]
# set the FTP server active
```

```
RunFTP=1
# defines the FTP control port
FTPPort=21
# Sets the location of the FTP data directory
FTPDir=/opt/ecs/mvuser/MV_IPTel/data/FTPdata
# set the admin Name
UserName=admin
# set the Password
Password=admin
```

The configparser module has ConfigParser class. It is responsible for parsing a list of configuration files, and managing the parsed database.

Object of ConfigParser is created by following statement −

```
parser = configparser.ConfigParser()
```

Following methods are defined in this class −

sections()	Return all the configuration section names.
has_section()	Return whether the given section exists.
options()	Return list of configuration options for the named section.
read()	Read and parse the named configuration file.
read_file()	Read and parse one configuration file, given as a file object.
read_string()	Read configuration from a given string.
get()	Return a string value for the named option.
getint()	Like get(), but convert value to an integer.
getfloat()	Like get(), but convert value to a float.
getboolean()	Like get(), but convert value to a boolean. Returns False or True.
items()	return a list of tuples with (name, value) for each option in the section.

remove_section()	Remove the given file section and all its options.
remove_option()	Remove the given option from the given section.
set()	Set the given option.
write()	Write the configuration state in .ini format.

Following script reads and parses the 'sampleconfig.ini' file

```
import configparser
parser = configparser.ConfigParser()
parser.read('sampleconfig.ini')
for sect in parser.sections():
  print('Section:', sect)
  for k,v in parser.items(sect):
    print(' {} = {}'.format(k,v))
  print()
```

Output

```
Section: Settings
detailedlog = 1
runstatus = 1
statusport = 6090
statusrefresh = 10
archive = 1
logfile = /opt/ecs/mvuser/MV_IPTel/log/MV_IPTel.log
version = 0.9 Build 4
servername = Unknown

Section: FTP
runftp = 1
ftpport = 21
ftpdir = /opt/ecs/mvuser/MV_IPTel/data/FTPdata
username = admin
password = admin
```

The write() method is used to create a configuration file. Following script configures the parser object and writes it to a file object representing 'test.ini'

```
import configparser
parser = configparser.ConfigParser()
```

```
parser.add_section('Manager')
parser.set('Manager', 'Name', 'Ashok Kulkarni')
parser.set('Manager', 'email', 'ashok@gmail.com')
parser.set('Manager', 'password', 'secret')
fp=open('test.ini','w')
parser.write(fp)
fp.close()
```

Python - Print Logs in a File

If you want to print python logs in a **file** rather than on the **console** then we can do so using the basicConfig() method by providing filename and filemode as parameter.The **format of the message** can be specified by using format parameter in basicConfig() method.

```
import logging    # first of all import the module

logging.basicConfig(filename='std.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')

logging.warning('This message will get logged on to a file')
```

```
root - ERROR - This message will get logged on to a file
```

The above output shows **how the message will look like** but keep in mind it will be written to a **file named std.log** instead of the console.

In the above code, the filemode is set to w, which means the log file is opened in "**write mode**" each time basicConfig() is called, and after each run of the program, it will rewrite the file.

The default configuration for filemode is a, that is **append**, which means that logs will be appended to the log file and adding logs to the existing logs.

Python Logging - Store Logs in a File

There are some basic steps and these are given below:

1. First of all, simply import the logging module just by writing import logging.
2. The second step is to create and configure the logger. To configure logger to store logs in a file, it is mandatory to pass the **name of the file** in which you want to record the events.
3. In the third step, the format of the logger can also be set. Note that by default, the file works in **append** mode but we can change that to **write** mode if required.
4. You can also set the level of the logger.

So let's move on to the code now:

```
#importing the module

import logging

#now we will Create and configure logger

logging.basicConfig(filename="std.log", format='%(asctime)s %(message)s', filemode='w')

#Let us Create an object

logger=logging.getLogger()

#Now we are going to Set the threshold of logger to DEBUG

logger.setLevel(logging.DEBUG)

#some messages to test

logger.debug("This is just a harmless debug message")

logger.info("This is just an information for you")

logger.warning("OOPS!!!Its a Warning")

logger.error("Have you try to divide a number by zero")

logger.critical("The Internet is not working....")
```

The above code will write some messages to file named **std.log**. If we will open the file then the messages will be written as follows:

```
2020-06-19 12:48:00,449 - This is just harmless debug message

2020-06-19 12:48:00,449 - This is just an information for you

2020-06-19 12:48:00,449 - OOPS!!!Its a Warning

2020-06-19 12:48:00,449 - Have you try to divide a number by zero

2020-06-19 12:48:00,449 - The Internet is not working...
```

You can change the format of logs, log level or any other attribute of the LogRecord along with setting the filename to store logs in a file along with the mode

Programming using file operations

Various operations carried out on a file are:

a) Creating a file
b) Opening a file
c) Reading from a file
d) Writing to file
e) Closing a file.

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. Read Only ('r') : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
2. Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
3. Write Only ('w') : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
4. Write and Read ('w+') : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. Append Only ('a') : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. Append and Read ('a+') : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Opening a File

It is done using the open() function. No module is required to be imported for this function.

```
File_object = open(r"File_Name","Access_Mode")
```

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The r is placed before filename to prevent the characters in filename string to be treated as special character.
The r can be ignored if the file is in same directory and address is not being placed.
While specifying the exact path, characters prefaced by \ (like \n \r \t etc.) are interpreted as special characters. You can escape them using:
raw strings like r'C:\new\text.txt'
double backslashes like 'C:\\new\\text.txt'

```
# Open function to open the file "MyFile1.txt"
# (same directory) in append mode and
file1 = open("MyFile1.txt","a")
  file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2

Closing a file
close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

```
File_object.close()
```

```
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt","a")
file1.close()
```

Writing to a file
There are two ways to write in a file.

1. write() : Inserts the string str1 in a single line in the text file.
   File_object.write(str1)
2. writelines() : For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.
   File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file
There are three ways to read data from a text file.

1. read() : Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.
   File_object.read([n])
2. readline() : Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.
   File_object.readline([n])
3. readlines() : Reads all the lines and return them as each line a string element in a list.
   File_object.readlines()

Note: '\n' is treated as a special character of two bytes

With Statement

You can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it's good practice to use the with statement where applicable.

```
with open("testfile.txt") as file:
    data = file.read()
    print(data)
```

EXAMPLE:

```
# Program to show various ways to read and
# write data in a file.
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes
file1 = open("myfile.txt","r+")
print( "Output of Read function is ")
print (file1.read())
# seek(n) takes the file handle to the nth
# bite from the beginning.
file1.seek(0)
print ("Output of Readline function is ")
print (file1.readline())
file1.seek(0)
# To show difference between read and readline
print ("Output of Read(9) function is ")
print (file1.read(9))

file1.seek(0)
print ("Output of Readline(9) function is ")
print (file1.readline(9))

file1.seek(0)
# readlines function
print ("Output of Readlines function is ")
print (file1.readlines())
file1.close()
```

Output:

```
Output of Read function is
Hello
This is Delhi
This is Paris
```

```
This is London
Output of Readline function is
Hello
Output of Read(9) function is
Hello
Th
Output of Readline(9) function is
Hello
Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

Appending to a file

```
# Python program to illustrate

# Append vs write mode

file1 = open("myfile.txt","w")

L = ["This is Delhi \n","This is Paris \n","This is London \n"]

file1.close()

# Append-adds at last

file1 = open("myfile.txt","a")#append mode

file1.write("Today \n")

file1.close()

file1 = open("myfile.txt","r")

print ("Output of Readlines after appending")

print (file1.readlines())

file1.close()

# Write-Overwrites

file1 = open("myfile.txt","w")#write mode

file1.write("Tomorrow \n")
```

```
file1.close()

file1 = open("myfile.txt","r")

print ("Output of Readlines after writing")

print (file1.readlines())

print

file1.close()
```

Output:

Output of Readlines after appending

['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']

Output of Readlines after writing

['Tomorrow \n']

seek() method

In Python, seek() function is used to change the position of the File Handle to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax: f.seek(offset, from _what),
where
f is file pointer
Parameters:
Offset: Number of positions to move forward
from_what: It defines point of reference.
Returns: Does not return any value

The reference point is selected by the from_what argument. It accepts three values:
0: sets the reference point at the beginning of the file
1: sets the reference point at the current file position
2: sets the reference point at the end of the file

By default from_what argument is set to 0.
Note: Reference point at current position / end of file cannot be set in text mode except when offset is equal to 0.

Example 1: Let's suppose we have to read a file named "GfG.txt" which contains the following text:

"Code is like humor. When you have to explain it, it's bad."

```
# Python program to demonstrate
# seek() method
# Opening "GfG.txt" text file
f = open("GfG.txt", "r")
# Second parameter is by default 0
# sets Reference point to twentieth
# index position from the beginning
f.seek(20)
# prints current position
print(f.tell())
print(f.readline())
f.close()
```

Output:

```
20
When you have to explain it, it's bad.
```

Example 2: Seek() function with negative offset only works when file is opened in binary mode. Let's suppose the binary file contains the following text.

b'Code is like humor. When you have to explain it, its bad.'

```
# Python code to demonstrate
# use of seek() function

# Opening "GfG.txt" text file
# in binary mode
f = open("data.txt", "rb")
# sets Reference point to tenth
# position to the left from end
f.seek(-10, 2)
# prints current position
print(f.tell())
# Converting binary to string and
# printing
print(f.readline().decode('utf-8'))
f.close()
```

Output:

```
47
, its bad.
```

Python OOPs Concepts

Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming.

In OOP languages it is mandatory to create a class for representing data.

It allows us to develop applications using an Object-Oriented approach.

In Python, we can easily create and use classes and objects.

The object is related to real-word entities such as book, house,pencil,etc .

The oops concept focuses on writing the reusable code.

Features of OOP

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

Object

- An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated
- 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. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code..
- For example, a car can be an object. If we consider the car as an object then its properties would be – its color, its model, its price, its brand, etc. And its behavior/function would be acceleration, slowing down, gear change.
- Another example- If we consider a dog as an object then its properties would be- his color, his breed, his name, his weight, etc. And his behavior/function would be walking, barking, playing, etc.

Class

- The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods.
- The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- A class is a blueprint for the object
- 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..

The example for class of parrot can be :

```
class Parrot:
   pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Method

- Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.
- The method is a function that is associated with an object.

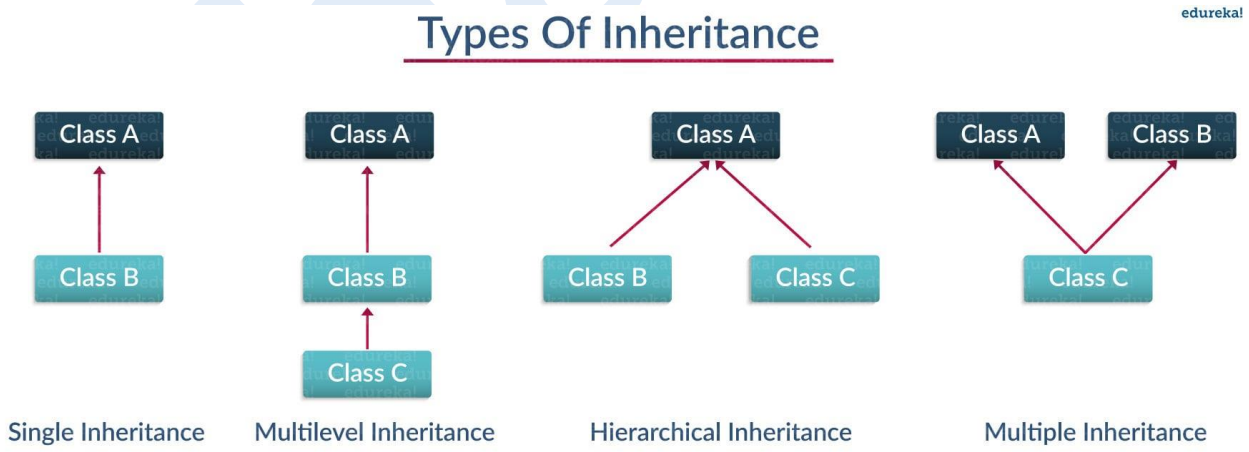
Ex:

```
def method1 (self):
  print "Guru99"
```

- The self-argument refers to the object itself. Hence the use of the word self. So inside this method, self will refer to the specific instance of this object that's being operated on.

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 the re-usability of the code.



Polymorphism

- Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types..
- By polymorphism, we understand that one task can be performed in different ways.

- For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

- Binding of data and methods into a single unit is called encapsulation. Encapsulation is accomplished when each object inside the class keeps its state private. The data inside this unit is not accessible by outside objects and only those functions inside this unit are able to access it. Thus, the object manages its state with the help of its methods, and to communicate with this object, we will require the help of the public methods of this class.
- Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, is a process of binding data members and member functions into a single unit, where data members are variables or properties and member functions are methods.

Data Abstraction

- Abstraction is an extension of encapsulation. It means providing only the necessary information to the outside world while hiding the internal details of implementation. It reveals only the appropriate operations for other objects. The advantage of this is that we can change the implementation without affecting the class, as the method interface remains the same.
- .We use Abstraction for hiding the internal details or implementations of a function and showing its functionalities only. This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch but you don't know what is happening behind the socket.

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-oriented uses objects, classes, messages	Procedural uses procedures, modules, procedure calls.
It is easy to maintain.	It is not easy to maintain.
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
Data hiding is possible, hence more secure than procedural.	Data hiding is not possible.
In OOP, operator overloading is allowed.	Operator overloading is not allowed.

Advantages & Disadvantages of Object-Oriented Programming

Advantages of OOP

Re-usability:

"Write once and use it multiple times" you can achieve this by using class.

Redundancy:

Inheritance is the good feature for data redundancy. If you need a same functionality in multiple class you can write a common class for the same functionality and inherit that class to sub class.

Easy Maintenance:

It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.

Security:

Using data hiding and abstraction only necessary data will be provided thus maintains the security of data.

Disadvantages of OOP

Size:

Object Oriented Programs are much larger than other programs.

Effort:

Object Oriented Programs require a lot of work to create.

Python Class

- Python is a completely object-oriented language
- Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.
- An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object
- Every element in a Python program is an object of a class. A number, string, list, dictionary, etc., used in a program is an object of a corresponding built-in class. You can retrieve the class name of variables or objects using the type() method, as shown below.

Example: Python Built-in Classes

```
>>> num=20
>>> type(num)
<class 'int'>
>>> s="Python"
>>> type(s)
<class 'str'>
```

Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with **a class** keyword.

A class in Python can be defined using the class keyword.

```
class <ClassName>:
    <statement1>
    <statement2>    .
    .
    <statementN>
```

As per the syntax above, a class is defined using the class keyword followed by the class name and : operator after the class name, which allows you to continue in the next indented line to define class members.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.
Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
There are also special attributes in it that begins with double underscores __. For example, __doc__ gives us the docstring of that class.
As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
    "This is a person class"
    age = 10
    def greet(self):
        print('Hello')
print(Person.age)
print(Person.greet)
print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>
This is a person class
```

A class can also be defined without any members. The following example defines an empty class using the pass keyword.
Example: Define Python Class

```
class Student:
    pass
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.
It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> harry = Person()
```

This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.
Attributes may be data or method. Methods of an object are corresponding functions of that class.
This means to say, since Person.greet is a function object (attribute of class), Person.greet will be a method object.

```
class Person:
    "This is a person class"
    age = 10
    def greet(self):
        print('Hello')
# create a new object of Person class
harry = Person()
print(Person.greet)
print(harry.greet)
# Calling object's greet() method
# Output: Hello
harry.greet()
```

Output

```
<function Person.greet at 0x7fd288e4e160>
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>
Hello
```

You may have noticed the self parameter in function definition inside the class but we called the method simply as harry.greet() without any arguments. It still worked.

Class Attributes

Class attributes are the defined directly in the class that are shared by all objects of the class. Class attributes can be accessed using the class name as well as using the objects.

Example: Define Python Class

```
class Student:
    schoolName = 'XYZ School'
```

Above, the schoolName is a class attribute defined inside a class. The value of the schoolName will remain the same for all the objects unless modified explicitly.

Example: Define Python Class

```
>>> Student.schoolName
'XYZ School'
>>> std = Student()
>>> std.schoolName
'XYZ School'
```

A class attribute is accessed by Student.schoolName as well as std.schoolName. Changing the value of class attribute using the class name would change it across all instances. However, changing class attribute value using instance will not reflect to other instances or class.

Example: Define Python Class

```
>>> Student.schoolName = 'ABC School' # change attribute value using class name
>>> std = Student()
>>> std.schoolName
'ABC School'   # value changed for all instances
>>> std.schoolName = 'My School'  # changing instance's attribute
>>> std.schoolName
'My School'
>>> Student.schoolName # instance level change not reflectd to class attribute
'ABC School'
>>> std2 = Student()
>>> std2.schoolName
'ABC School'
```

Constructor

In Python, the constructor method is invoked automatically whenever a new object of a class is instantiated, same as constructors in C# or Java. The constructor must have a special name __init__() and a special parameter called self.

The first parameter of each method in a class must be the self , which refers to the calling object. However, you can give any name to the first parameter, not necessarily self.

The following example defines a constructor.

Example: Constructor

```
class Student:
    def __init__(self): # constructor method
        print('Constructor invoked')
```

Now, whenever you create an object of the Student class, the __init__() constructor method will be called, as shown below.

Example: Constructor Call on Creating Object

```
>>>s1 = Student()
Constructor invoked
>>>s2 = Student()
Constructor invoked
```

The constructor in Python is used to define the attributes of an instance and assign values to them.

Instance Attributes

Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

The following example defines instance attributes name and age in the constructor.

Example: Instance Attributes

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self): # constructor
        self.name = '' # instance attribute
        self.age = 0 # instance attribute
```

An instance attribute can be accessed using dot notation: [instance name].[attribute name], as shown below.

Example:

```
>>> std = Student()
>>> std.name
''
>>> std.age
0
```

You can set the value of attributes using the dot notation, as shown below.

Example:

```
>>> std = Student()
>>> std.name = "Bill" # assign value to instance attribute
>>> std.age=25        # assign value to instance attribute
>>> std.name          # access instance attribute value
Bill
>>> std.age           # access value to instance attribute
25
```

You can specify the values of instance attributes through the constructor. The following constructor includes the name and age parameters, other than the self parameter.

Example: Setting Attribute Values

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

Now, you can specify the values while creating an instance, as shown below.

Example: Passing Instance Attribute Values in Constructor

```
>>> std = Student('Bill',25)
>>> std.name
'Bill'
>>> std.age
25
```

You don't have to specify the value of the self parameter. It will be assigned internally in Python.

You can also set default values to the instance attributes. The following code sets the default values of the constructor parameters. So, if the values are not provided when creating an object, the values will be assigned latter.

Example: Setting Default Values of Attributes

```
class Student:
    def __init__(self, name="Guest", age=25)
        self.name=name
        self.age=age
```

Class Properties

In Python, a property in the class can be defined using the property() function.

The property() method in Python provides an interface to instance attributes.

The property() method takes the get, set and delete methods as arguments and returns an object of the property class.

Example: property()

```
class Student:
    def __init__(self):
        self.__name=''
    def setname(self, name):
        print('setname() called')
        self.__name=name
    def getname(self):
        print('getname() called')
        return self.__name
    name=property(getname, setname)
```

In the above example, property(getname, setname) returns the property object and assigns it to name. Thus, the name property hides the private instance attribute __name. The name property is accessed directly, but internally it will invoke the getname() or setname() method, as shown below.

Example: property()

```
>>> std = Student()
>>> std.name="Steve"
setname() called
>>> std.name
getname() called
'Steve'
```

Class Methods

You can define as many methods as you want in a class using the **def** keyword. Each method must have the first parameter, generally named as self, which refers to the calling instance.

Example: Class Method

```
class Student:
    def displayInfo(self): # class method
        print('Student Information')
```

Self is just a conventional name for the first argument of a method in the class. A method defined as mymethod(self, a, b) should be called as x.mymethod(a, b) for the object x of the class.

The above class method can be called as a normal function, as shown below.

Example: Class Method

```
>>> std = Student()
>>> std.displayInfo()
'Student Information'
```

The first parameter of the method need not be named self. You can give any name that refers to the instance of the calling method. The following displayInfo() method names the first parameter as obj instead of self and that works perfectly fine.

Example: Class Method

```
class Student:
    def displayInfo(obj): # class method
        print('Student Information')
```

Defining a method in the class without the self parameter would raise an exception when calling a method.

Example: Class Method

```
class Student:
    def displayInfo(): # method without self parameter
        print('Student Information')
>>> std = Student()
>>> std.displayInfo()
```

```
Traceback (most recent call last):
std.displayInfo()
TypeError: displayInfo() takes 0 positional arguments but 1 was given
```

The method can access instance attributes using the self parameter.

Example: Class Method

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def displayInfo(self): # class method
        print('Student Name: ', self.name,', Age: ', self.age)
```

You can now invoke the method, as shown below.

Example: Calling a Method

```
>>> std = Student('Steve', 25)
>>> std.displayInfo()
Student Name: Steve , Age: 25
```

Deleting Attribute, Object, Class

You can delete attributes, objects, or the class itself, using the del keyword, as shown below.

Example: Delete Attribute, Object, Class

```
>>> std = Student('Steve', 25)
>>> del std.name # deleting attribute
>>> del std  # deleting object
>>> del Student  # deleting class
>>> std = Student('Steve', 25)
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
std = Student()
NameError: name 'Student' is not defined
```

Destructors

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.

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

Syntax of destructor declaration :

```
def __del__(self):

  # body of destructor
```

Note : A reference to objects is also deleted when the object goes out of reference or when the program ends.

Example 1 : Here is the simple example of destructor. By using del keyword we deleted the all references of object 'obj', therefore destructor invoked automatically.

```
# Python program to illustrate destructor
class Employee:

    # Initializing
    def __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Output:

```
Employee created.
Destructor called, Employee deleted.
```

Note : The destructor was called after the program ended or when all the references to object are deleted i.e when the reference count becomes zero, not when object went out of scope.

Example 2 :This example gives the explanation of above mentioned note. Here, notice that the destructor is called after the 'Program End…' printed.

```
# Python program to illustrate destructor

class Employee:
```

```
    # Initializing
    def __init__(self):
        print('Employee created')

    # Calling destructor
    def __del__(self):
        print("Destructor called")

def Create_obj():
    print('Making Object...')
    obj = Employee()
    print('function end...')
    return obj

print('Calling Create_obj() function...')
obj = Create_obj()
print('Program End...')
```

Output:

```
Calling Create_obj() function...
Making Object...
Employee created
function end...
Program End...
Destructor called
```

Real time use of class in live projects

when working on data science projects or any Python programming project, you will most likely find yourself utilizing plenty of self-made functions and variables. You may have even create an entire script filled with functions you created in order to streamline the process of your project.

The purpose of these functions can be for numerous things within your code. From cleaning your DataFrame to training a machine learning model. It's useful to create a ton of functions in order to

organize your Python code but there is another way to make your code look and act more presentable — By using a Python ***Class***!

What is a Python Class?

A Python class is like an outline for creating a new object. An object is anything that you wish to manipulate or change while working through the code. Every time a class object is instantiated, which is when we declare a variable, a new object is initiated from scratch. Class objects can be used over and over again whenever needed.

This data preprocessing step is achieved through the use of numerous functions. It is not organized in the best way but it gets the job done. However, we can improve the process by utilizing a Python Class.

Class for Dating Profiles

In order to learn more about class objects and utilizing them, we will be implementing a class object for our AI Dating Algorithm. Let's organize and clean up the code we used into a Class object.

Class Object Purpose

First, we must ask ourselves — What do we want this class object to do? We want it to:

1. Create a new dating profile.
2. Add that new profile to a larger pool of dating profiles.
3. Scale or vectorize that profile in order for it to be machine learning model friendly.

Basically, we will want it to condense the entire data preprocessing step into a Class object which we can then use for every new dating profile we wish to add. We can use this class object whenever we need to create a new dating profile.

Constructing a Class

Within the class, tab over to start defining our first function. Usually, when creating a class, you will have to define a function called __init__ with self as the initial argument.

```
class CreateProfile:
    def __init__(self):
```

What is an "__init__" function?

An __init__ function is called when a class is *instantiated*. By instantiate, we mean when you declare the class, which can happen either by itself or by assigning it to a variable. Here are some quick examples of instantiating a class object:

```
CreateProfile()# ORprofile = CreateProfile()
```

Here we are instantiating the class object and by doing so, we are implicitly calling the __init__ function. Any arguments within the __init__ function will also be the same arguments when instantiating the class object. These initial arguments can be the data we wish to manipulate throughout the class object. But in regards to the self argument, it will not be necessary to replace when instantiating the class object.

Python Inheritance

- Inheritance is the process of creating a new Class, called the **Derived Class** , from the existing class, called the **Base Class** .
- When a new class inherits from an existing one, the existing one is called the **parent class(super-class)** and the new class is called the **child class(sub-class).**
- 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.
- Child class is the class that **inherits from another class**, also called derived class.
- It represents real-world relationships well.
- It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- **It offers faster development time, easier maintenance and easy to extend.**

Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named Person, with firstname and lastname properties, and a printname method:

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

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named Student, which will inherit the properties and methods from the Person class:

Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

```
class Student(Person):
  pass

x = Student("Mike", "Olsen")
x.printname()
```

Example2:

```
class Parent: # define parent class
   parentAttr = 100
   def __init__(self):
      print ("Calling parent constructor")
   def parentMethod(self):
      print ('Calling parent method')
class Child(Parent): # define child class
   def __init__(self):
      print ("Calling child constructor")
   def childMethod(self):
      print ('Calling child method')
c = Child()       # instance of child
c.childMethod()   # child calls its method
c.parentMethod()  # calls parent's method
```

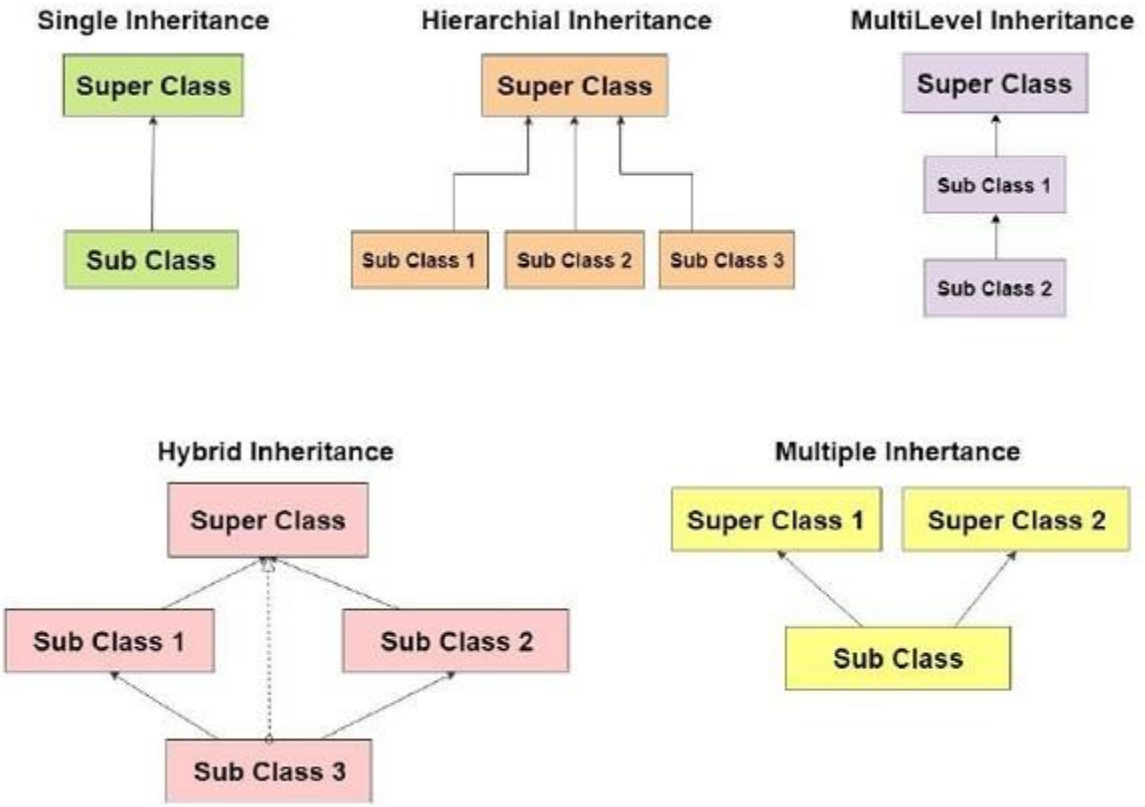
Output

```
Calling child constructor
Calling child method
Calling parent method
```

Different Types of Inheritance

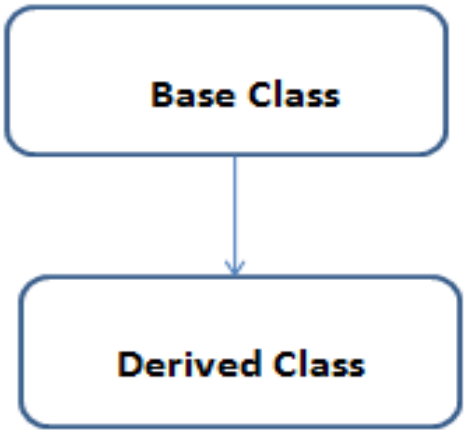
inheritance may be implemented in different combinations in **Object-Oriented Programming languages** as illustrated in figure and they include:

- **Single Inheritance**
- **Multi Level Inheritance**
- **Multipath inheritance**
- **Multiple Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**



Single Inheritence:

When a **Derived Class** to inherit properties and behavior from a single **Base Class** , it is called as single inheritance.



Example:

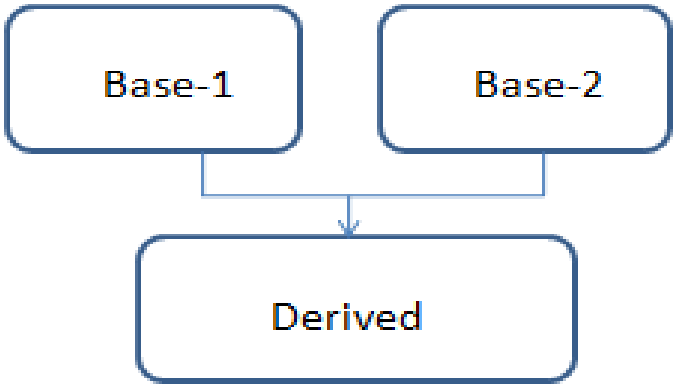
```
# Single inheritence
class Apple:
    manufacturer = 'Apple Inc'
    contact_website = 'www.apple.com/contact'
    name = 'Apple'

    def contact_details(self):
        print('Contact us at ', self.contact_website)


class MacBook(Apple):
    def __init__(self):
        self.year_of_manufacture = 2018

    def manufacture_details(self):
        print('This MacBook was manufactured in {0}, by {1}.'
              .format(self.year_of_manufacture, self.manufacturer))

macbook = MacBook()
macbook.manufacture_details()
```

output:

This MacBook was manufactured in 2018, by Apple Inc.

Multiple inheritance

In multiple inheritance one child class can inherit multiple parent classes.



In multiple inheritance, a class *inherits from two or more super classes*.
It inherits the *methods and variables* from all **super classes**.

If you create an object, it has all methods and variables from the classes.

Example, where a class inherits from three classes

```
class Parent1:
    pass
class Parent2:
    pass
class Parent3:
    pass
class Kid1(Parent1, Parent2, Parent3):
    pass
```

Example2:

```
# Father class created
class Father:
    fathername = ""

    def show_father(self):
        print(self.fathername)


# Mother class created
class Mother:
```

```
    mothername = ""

    def show_mother(self):
        print(self.mothername)

# Son class inherits Father and Mother classes
class Son(Father, Mother):
    def show_parent(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
s1 = Son()  # Object of Son class
s1.fathername = "Mark"
                                        s1.mothername = "Sonia"
s1.show_parent()
```

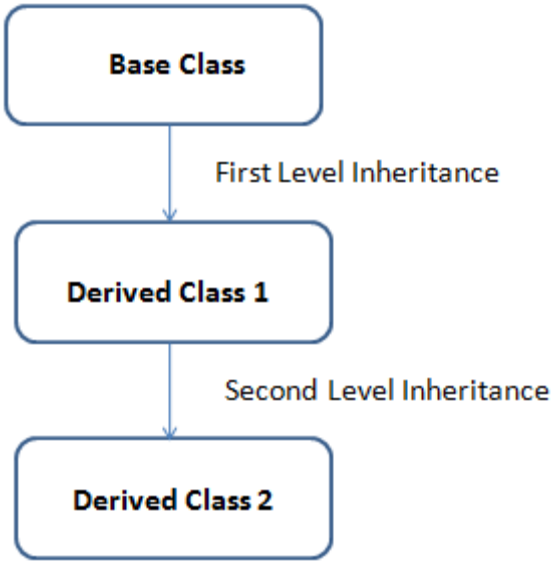
output:

Father : Mark
Mother : Sonia

Multilevel Inheritance in Python

A **derived class** is created from another derived class is called **Multi Level Inheritance** .

In this type of inheritance, a class can inherit from a child class or derived class.



In multi-level there are several levels, which create an inheritance relationship.

This is similar to the relationship between grandpa, father and child.

Here each class only inherits once, at most, but they inherit as a series.

Example:

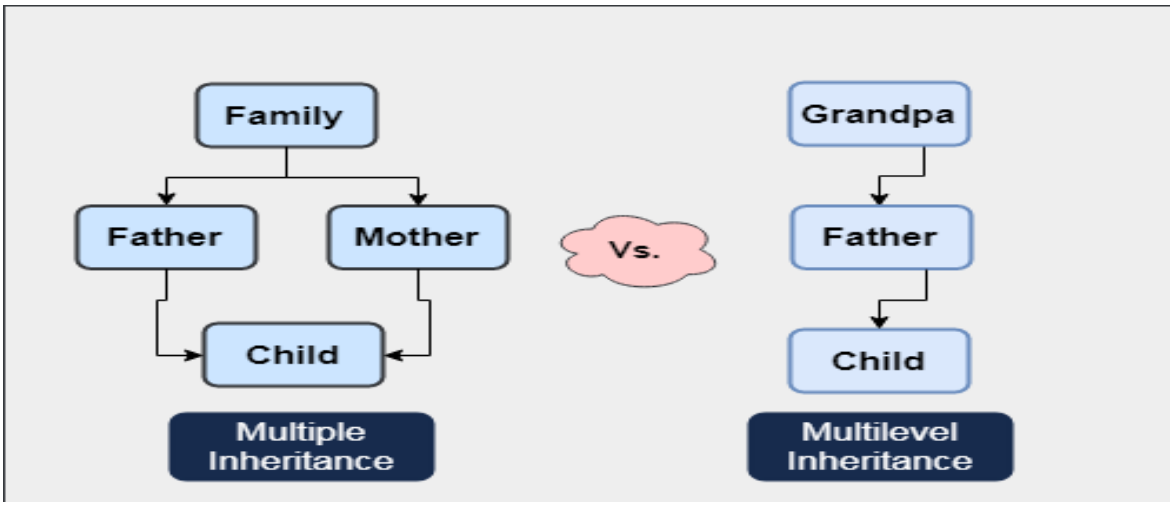
```
class Family:
    def show_family(self):
        print("This is our family:")
# Father class inherited from Family
class Father(Family):
    fathername = ""

    def show_father(self):
        print(self.fathername)
# Mother class inherited from Family
class Mother(Family):
    mothername = ""
    def show_mother(self):
        print(self.mothername)
# Son class inherited from Father and Mother classes
class Son(Father, Mother):
    def show_parent(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
s1 = Son()  # Object of Son class
s1.fathername = "Mark"
s1.mothername = "Sonia"
```

```
s1.show_family()

s1.show_parent()
```

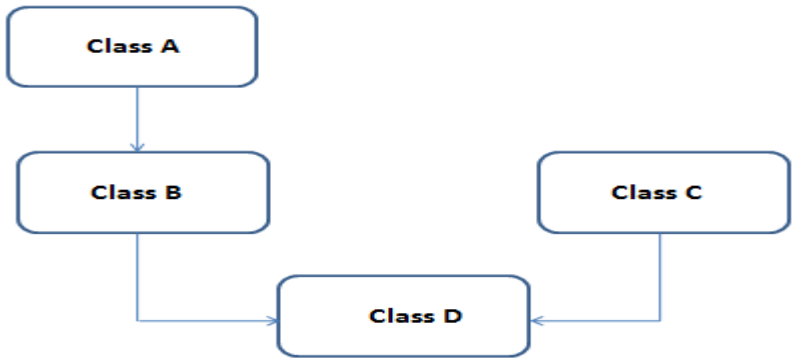
Output:
This is our family:
Father : Mark
Mother : Sonia

Multiple Inheritance vs Multi-Level inheritance

- Multiple Inheritance denotes a scenario when a class derives from more than one base classes.
- Multilevel Inheritance means a class derives from a subclass making that subclass a parent for the new class.
- Multiple Inheritance has **two classes** in the hierarchy, i.e., a base class and its subclass.
- Multilevel Inheritance requires **three levels** of classes, i.e., a base class, an intermediate class, and the subclass.



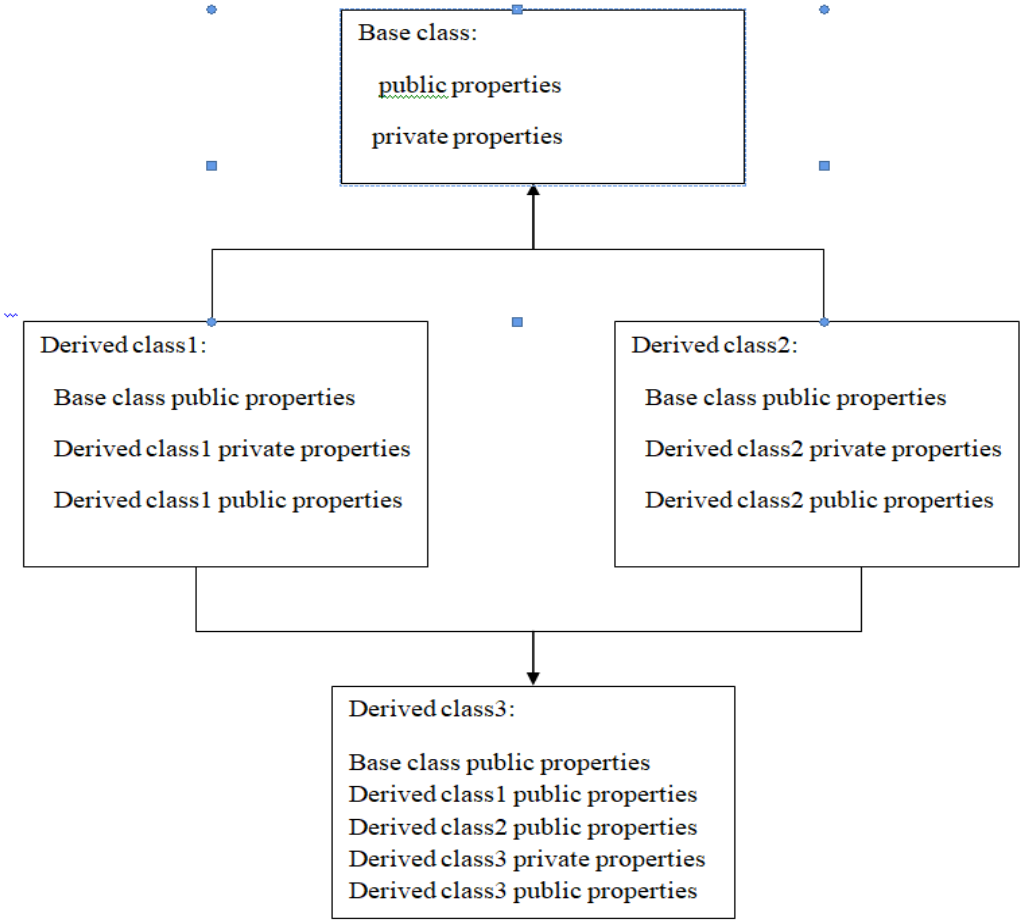
Multipath inheritance



Multiple inheritance is a method of inheritance in which one **derived class** can inherit properties of base class in different paths.

Multipath inheritance is a two level inheritance. Derived class in a Multipath Inheritance inherits the properties of two or more parents classes and their parents classes inherits the properties from single base class.

The multipath inheritance is a combination of multi level and multiple inheritances. Thus, in this type of inheritance, the public properties of base class are derived into various sub classes. All the public properties of sub classes are further derived into the final sub class. The multi path inheritance is shown in Fig. 10.7. The syntax of the multiple inheritance is given below




```
class base_class1
  #body of base class1
class sub_class1(base_class1):
 #body of subclass1
class sub_class2(base_class1):
```

```
  #body of the sub class2
 class sub_class3(sub_class1,sub_class2):
 #body of the sub class
```

example:

```
class a:
 x=100

class b(a):
 y=200

class c(a):
 z=300

class d(b,c):
 p=500

 def f1(self):
    print("Properties of sub class: ",self.x,self.y,self.z,self.p)

obj1=d()
obj1.f1()
```

Output:

```
D:\PythonPrograms>python b2.py Properties of
sub class: 100 200 300 500
```

Python Operator Overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```
# Python program to show use of + operator for different purposes.
print(1 + 2)
# concatenate two strings
print("Learn"+"For")
# Product two numbers
print(3 * 4)
# Repeat the String
print("Learn"*4)
```

Output:

```
3
LearnFor
12
LearnLearnLearnLearn
```

Operator overloading is nothing but same name but differerent type of arguments or return type

Ex:print(1*7)

print(“hii"+”hello”)

Overlapping means ex:the elements of tuple1 is at least one element equal to tuple2

Then this is called overlapping

Ex:q=(1,2,7)

W=(1,9,7)

in operator : The ‘in’ operator is used to check if a value exists in a sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise.

```
# Python program to illustrate
# Finding common member in list
# using 'in' operator
list_1=[1,2,3,4,5]
list2=[6,7,8,9]
for item in list_1:
if item in list2:
print("overlapping")
else:
print("not overlapping")
```

Output:

not overlapping

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```
# Python program to show use of
# + operator for different purposes.

print(1 + 2)

# concatenate two strings
print("Geeks"+"For")
```

```
# Product two numbers
print(3 * 4)

# Repeat the String
print("Geeks"*4)
```

Output:

3

GeeksFor

12

GeeksGeeksGeeksGeeks

To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

Overloading binary + operator in Python :

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined. There by changing this magic method's code, we can give extra meaning to the + operator.

Code 1:

- Python3

```
# Python Program illustrate how
# to overload an binary + operator

class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)
```

Output :

3

GeeksFor

Code 2:

- **Python3**

```
# Python Program to perform addition
# of two complex numbers using binary
# + operator overloading.

class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

     # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)Output :
```

(3, 5)

To overload the + operator, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

Now let's try the addition operation again:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
```

```
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

Output

(3,5)

What actually happens is that, when you use `p1 + p2`, Python calls `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. After this, the addition operation is carried out the way we specified. Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`

Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Adding and retrieving dynamic attributes of classes:

Dynamic attributes in Python are terminologies for attributes that are **defined at runtime**, after creating the objects or instances. In Python we call all functions, methods also as an object. So you can define a dynamic instance attribute for nearly anything in Python. Consider the below example for better understanding about the topic.

Example 1:

```
class GFG:
    None

def value():
    return 10

# Driver Code
g = GFG()

# Dynamic attribute of a
# class object
g.d1 = value
```

```
# Dynamic attribute of a
# function
value.d1 = "Geeks"

print(value.d1)
print(g.d1() == value())
```

Output:

```
Geeks
True
```

Now, the above program seems to be confusing, but let's try to understand it. Firstly let's see the objects, g and value(functions are also considered as objects in Python) are the two objects. Here the dynamic attribute for both the objects is "d1". This is defined at runtime and not at compile time like static attributes.

Note: The class "GFG" and all other objects or instances of this class do not know the attribute "d1". It is only defined for the instance "g".

Example 2:

```
class GFG:

    employee = True

# Driver Code
e1 = GFG()
e2 = GFG()

e1.employee = False
e2.name = "Nikhil"

print(e1.employee)
print(e2.employee)
print(e2.name)

# this will raise an error
# as name is a dynamic attribute
# created only for the e2 object
print(e1.name)
```

Output:

```
False
True
Nikhil
Traceback (most recent call last):
  File "/home/fbcfcf668619b24bb8ace68e3c400bc6.py", line 19, in
    print(e1.name)
AttributeError: 'GFG' object has no attribute 'name'
```

Programming using Oops support

OOP in Python

Python is a great programming language that supports OOP. You will use it to define a class with attributes and methods, which you will then call. Python offers a number of benefits compared to other programming languages like Java, C++ or R. It's a dynamic language, with high-level data types. This means that development happens much faster than with Java or C++. It does not require the programmer to declare types of variables and arguments.

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

A parrot is an object, as it has the following properties:

- name, age, color as attributes
- singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code

In Python, the concept of OOP follows some basic principles:

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example 1: Creating Class and Object in Python

```
class Parrot:
    # class attribute
    species = "bird"
    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))
```

```
# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the __init__ method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

We can access the class attribute using __class__.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

```
class Parrot:
    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
```

```
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Output

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e sing() and dance(). These are called instance methods because they are called on an instance object i.e blu.

Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
```

```
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the __init__() method. This allows us to run the __init__() method of the parent class inside the child class.

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 __.

Example 4: 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()
# 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
```

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.

Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

Example 5: Using Polymorphism in Python

```
class Parrot:
    def fly(self):
        print("Parrot can fly")
    def swim(self):
        print("Parrot can't swim")
```

```
class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

Output

Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying_test() function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying_test() function, it ran effectively.

Python Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

- 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
```

Example:

```
class Dog:
    pass
```

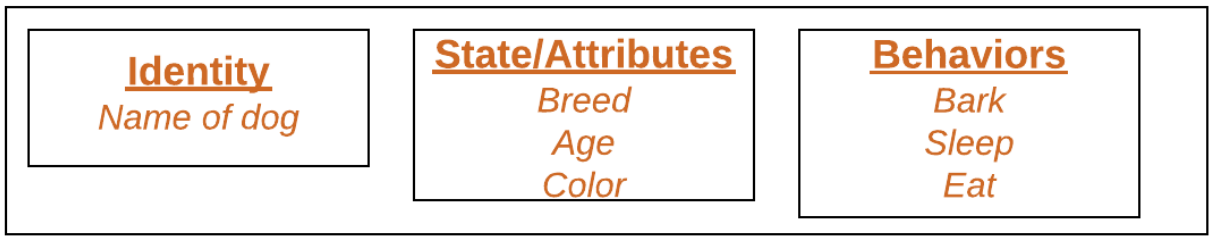
In the above example, the class keyword indicates that you are creating a class followed by the name of the class (Dog in this case).

Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of :

State: It is represented by the attributes of an object. It also reflects the properties of an object.
Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
Identity: It gives a unique name to an object and enables one object to interact with other objects.
python class



Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example:

```
class Dog:

    # A simple class
    # attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr1)
Rodger.fun()
```

Output:

```
mammal
I'm a mammal
I'm a dog
```

Basic ATM Function Requirements

In the previous ATM program in Python we completed a list a of basic ATM requirements as shown below:

- Input user pin for authentication
- Check account balance
- Deposit funds
- Withdraw funds
- Create random generated transaction id
- Account interest rate and monthly accrued interest rate

Additional ATM Function Requirements

In this module, the ATM program will add the below additional requirements:

- Separate balance into a checking account balance and a savings account balance
- Create a transfer of funds between checking account and savings account
- Create a transfer of funds between savings account and checking account

By completing the above, multiple new class objects and functions will need to be completed. Also, the formatting of print statements will be different to accommodate the balance of funds to two decimal places.

Import Python Module Random and Time

In order to complete the above ATM, we will need to import two Python modules; random and time.

```
import random
import time

class Account:
# Construct an Account object
def __init__(self, id, checkingBalance = 0, savingsBalance = 0, annualInterestRateSavings = 3.4):
self.id = id
self.checkingBalance = checkingBalance
self.savingsBalance = savingsBalance
self.annualInterestRateSavings = annualInterestRateSavings
def getId(self):
return self.id

def checkingAccountBalance(self):
return self.checkingBalance

def withdrawCheckingAccount(self, amount):
self.checkingBalance -= amount

def depositCheckingAccount(self, amount):
self.checkingBalance += amount

def transferCheckingAccount(self, amount):
self.checkingBalance += amount
self.savingsBalance -= amount

def savingsAccountBalance(self):
return self.savingsBalance

def withdrawSavingsAccount(self, amount):
self.savingsBalance -= amount
```

```
def depositSavingsAccount(self, amount):
self.savingsBalance += amount

def transferSavingsAccount(self, amount):
self.savingsBalance += amount
self.checkingBalance -= amount

def savingsAccountMonthlyInterest(self):
return self.savingsBalance * self.savingsAccountMonthlyInterest()

def savingsAccountAnnualInterestRate(self):
return self.annualInterestRateSavings

def savingsAccountMonthlyInterestRate(self):
return self.annualInterestRateSavings / 12
```

Structuring Classes with Inheritance

Inheritance is one of the most important aspects of Object Oriented Programming, as per object oriented programming, we can take out the common part and put it in a separate class, and make all the other classes inherit this class, to use its methods and variables, hence reducing re-writing the common features in every class, again and again.

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**.

If we have a class Parent and another class Child and we want the class Child to inherit the class Parent, then

```
# Parent class

class Parent:
    # class variable
    a = 10;
    b = 100;
    # some class methods
    def doThis();
    def doThat();

# Child class inheriting Parent class
class Child(Parent):
    # child class variable
    x = 1000;
    y = -1;
    # some child class method
```

```
    def doWhat();
    def doNotDoThat();
```

By specifying another class's name in parentheses, while declaring a class, we can specify inheritance. In the example above, all the properties of Parent will be inherited to the Child class. With this, all the methods and variables defined in the class Parent becomes part of Child class too.

Benefits of using Inheritance

Here are a few main advantages of using Inheritance in your programs.

1. Less code repeatition, as the code which is common can be placed in the parent class, hence making it available to all the child classes.
2. **Structured Code**: By dividing the code into classes, we can structure our software better by dividing functionality into classes.
3. Make the code more scalable.

Accessing Parent Class Element in Child Class

While working in a child class, at some point you may have to use parent class's properties or functions. In order to access parent class's elements you can use the dot . operator.

Parent.variableName

Mentioned above is how you can access the variable, or in case you need to call parent class's function then,

Parent.functionName()

Where Parent is the name of our parent class, and variableName and functionName() are its variable and function respectively.

Below is an example, we have a simple example to demonstrate this:

```
class Parent:
        var1 = 1
        def func1(self):
           # do something here

class Child(Parent):
        var2 = 2
        def func2(self):
      # do something here too
                # time to use var1 from 'Parent'
           myVar = Parent.var1 + 10
```

```
    return myVar
```

Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Example of inbuilt polymorphic functions :

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list


print(len([10, 20, 30]))
```

Output:

5

3

Examples of user-defined polymorphic functions :

```
# A simple Python function to demonstrate
# Polymorphism

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

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output:

5

9

Polymorphism with class methods:

The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
 class India():

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

    def language(self):
```

```
        print("Hindi is the most widely spoken language of India.")

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

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

    def language(self):
        print("English is the primary language of USA.")

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

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

Output:

New Delhi is the capital of India.

Hindi is the most widely spoken language of India.

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.

**********END**********