Unit-IV: File Operations and Object Oriented Programming

File:

A file is defined to be a collection of related data. Files are stored on secondary storage devices (like hard disk, CD or DVD).

Need of a file:

A program takes some inputs, performs some manipulations over them and produces required outputs. Generally, the inputs are given through the standard input device i.e., keyboard with the help of input() function. The outputs are produced through the standard output devices such as monitor by the use of print() function. This type of I/O operations has the following drawbacks.

1. The entire data is lost when either the program is terminated or the computer is turned off.
2. It is very difficult and time consuming to handle large volumes of data through terminals.

   It is therefore necessary to store data on the disks and read without destroying data. For this, we need files to store and retrieval of data.

Types of FILES

Depending upon the format in which data is stored, files are categorized into two types:

a) Text File
b) Binary File

a) Text File:

The text files are those files that contain textual information like alphabets, digits and special symbols etc. The text files store the ASCII encrypted information. Since data is stored in a storage device in the binary format, the text file contents are converted in the binary form before actually being stored in the storage device.

Ex: python sources code file, C source code files and files with .txt extension etc.,

b) Binary File:

A binary file stores the information in the binary form i.e., in the same format as it is stored in the memory. Thus, the use of binary file eliminates the need of data conversion from text to binary format for storage purpose. But data stored in a binary file is not in human understandable form. All the files with .exe extension are the examples of binary files.

Creating and opening Text files:

All files must be opened first before they can be read from or written to using the Python's built-in *open()* function.

When a file is opened using *open()* function, it returns a file object called a file handler that provides methods for accessing the file.

The syntax of *open()* function is given below.

file_handler = open (filename, mode)

where

- file_handler is the name of the variable that will reference the file object.
- filename is a string specifying the name of the file
- mode is a string specifying the mode (reading, writing, etc.) in which the file will be opened.

Some of the python file modes for text files are given below.

Mode	Description
"r"	Opens the file in read only mode and this is the default mode.
"w"	Opens the file for writing. If a file already exists, then it'll get overwritten. If the file does not exist, then it creates a new file.
"a"	Opens the file for appending data at the end of the file automatically. If the file does not exist it creates a new file.
"r+"	Opens the file for both reading and writing.
"w+"	Opens the file for reading and writing. If the file does not exist it creates a new file. If a file already exists then it will get overwritten.
"a+"	Opens the file for reading and appending. If a file already exists, the data is appended. If the file does not exist it creates a new file.
"x"	Creates a new file. If the file already exists, the operation fails.

Ex1:

the following statement opens a file named customers.txt in the current directory for reading.

```
customer_file = open('customers.txt', 'r')
```

After the execution of above statement, the file named customers.txt will be opened, and the variable customer_file will reference a file object that we can use to read data from the file.

Ex2:

Suppose we want to create a file named sales.txt and write data to it. This can be done using the statement

```
sales_file = open('sales.txt', 'w')
```

After this statement executes, the file named sales.txt will be created, and the variable sales_file will reference a file object that we can use to write data to the file.

Closing a file:

It is important to close the file once the processing is completed. After the file handler object is closed, read or write from the file is not possible. Any attempt to use the file handler object after being closed will result in an error.

The syntax for *close()* function is,

file_handler.close( )

For example, the following statement closes the file that is associated with customer_file.

```
customer_file.close()
```

Testing a File's Existence:

We can test to see if the file exists before opening it for writing. The isfile function in the os.path module can be used to determine whether a file exists.

```
import os.path
if os.path.isfile("mails.txt"):
   print("mails.txt exists")
else:
   print("mails.txt does not exist")
```

Here os.path.isfile("mails.txt") returns True if the file mails.txt exists in the current directory

Reading data from text files:

There are several ways to read the data from the file after it has been opened for reading. The methods that can be called on the file handler object for reading are given below.

Method	Syntax	Description
read()	file_handler.read([size])	This method is used to read the contents of a file up to a size and return it as a string. The argument *size* is optional, and, if it is not specified, then the entire contents of the file will be read and returned.
readline()	file_handler.readline()	This method is used to read a single line in file.
readlines()	file_handler.readlines()	This method is used to read all the lines of a file as list items.

Using read() method:

The read() method is used to read the contents of the file upto a size and return it as a string. The argument size is optional and if it not specified then entire contents of the file will be read as a single string. If the file contains multiple lines of text, the newline characters will be embedded in this string.

Ex:

Write a program to check whether the given file exists or not. If it is available then read the content and print it.

```
import os.path
fname=input("enter the file name with extension:")
if os.path.isfile(fname):
```

```
    print("the file",fname,"exists")
    f = open(fname,"r")
    print("The content of the file is")
    data=f.read()
    print(data)
    f.close()
else:
    print("The file",fname,"does not exist")
```

Output1:

```
enter the file name with extension:mails.txt
the file mails.txt exists
The content of the file is
kumarap.1980@gmail.com
svietexams@gmail.com
abc@gmail.com
pavank_1980@rediffmail.com
```

Output2:

```
enter the file name with extension: presidents.py
The file presidents.py does not exist
```

Ex:

Write a program to check whether the given file exists or not. If it is available then print its first 10 characters.

Sol:

```
import os.path
fname=input("enter the file name with extension:")
if os.path.isfile(fname):
    print("the file",fname,"exists")
    f = open(fname,"r")
    print("The first 10 characters of the file are")
    data=f.read(10)
```

```
    print(data)
    f.close()
else:
    print("The file",fname,"does not exist")
```

Output:

enter the file name with extension:mails.txt
the file mails.txt exists
The first 10 characters of the file are
kumarap.19

Using readline() method:

the readline() method is used to read a single line from a file. (A line is simply a string of characters that are terminated with a \n.) The method returns the line as a string, including the \n. If it returns an empty string, then the end of the file has been reached.

Ex:

Write a program to check whether the given file exists or not. If it is available then print its first 3 lines using readline() method

```
import os.path
fname=input("enter the file name with extension:")
if os.path.isfile(fname):
    print("the file",fname,"exists")
    print("the content of the file line by line is")
    f = open(fname,"r")
    line1=f.readline()
    print(line1)
    line2=f.readline()
    print(line2)
    line3=f.readline()
    print(line3)
    f.close()
else:
    print("The file",fname,"does not exist")
```

Output:

enter the file name with extension:mails.txt
the file mails.txt exists

```
the content of the file line by line is
kumarap.1980@gmail.com

svietexams@gmail.com

abc@gmail.com
```

Note:

Each of the strings that are returned from the readline( ) method end with a' \n' escape sequence.

Using readlines() method:

readlines() method is used to read all the lines of a file as list items i.e., This method returns a file's contents as a list of strings. Each line in the file will be an item in the list. The items in the list will include their terminating newline character, which in many cases you will want to strip.

Ex:

Write a program to check whether the given file exists or not. If it is available then print all of its lines using readlines() method

```
import os.path
fname=input("enter the file name with extension:")
if os.path.isfile(fname):
    print("the file",fname,"exists")
    print("the content of the file line by line is")
    f = open(fname,"r")
    data=f.readlines()
    #print(data)
    f.close()
    for line in data:
        line = line.strip('\n')
        print(line)
else:
    print("The file",fname,"does not exist")
```

Output:

```
enter the file name with extension:mails.txt
the file mails.txt exists
the content of the file line by line is
kumarap.1980@gmail.com
svietexams@gmail.com
abc@gmail.com
pavank_1980@rediffmail.com
```

Using Python's for Loop to Read Lines:

The Python language also allows you to write a for loop that automatically reads the lines in a file, one after the other, without testing for any special condition that signals the end of the file. The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached. The syntax is

```
for variable in file_handler:
    statement1
    statement 2
    ..........
    ..........
```

In the general format, variable is the name of a variable and file_handler is a variable that references a file object. **The loop will iterate once for each line in the file**. The first time the loop iterates, variable will reference the first line in the file (as a string), the second time the loop iterates, variable will reference the second line, and so forth.

```
import os.path
fname=input("enter the file name with extension:")
if os.path.isfile(fname):
   print("the file",fname,"exists")
   print("the content of the file line by line is")
   f = open(fname,"r")
   for line in f:
      line = line.strip('\n')
      print(line)
   f.close()
```

```
else:
    print("The file",fname,"does not exist")
```

Output:

```
enter the file name with extension:mails.txt
the file mails.txt exists
the content of the file line by line is
kumarap.1980@gmail.com
svietexams@gmail.com
abc@gmail.com
pavank_1980@rediffmail.com
```

Note:
The string method strip removes any whitespace characters from the beginning and end of a string. If we had not used it, each line would contain a newline character at the end of the line.

Reading Numbers from a File:

All of the file input operations return data to the program as strings. If these strings represent other types of data, such as integers or floating-point numbers, the programmer must convert them to the appropriate types before manipulating them further. In Python, the string representations of integers and floating-point numbers can be converted to the numbers themselves by using the functions int and float, respectively.

Ex:
Consider the integers.txt file which is a file of random integers separated by spaces and/or newlines. write a program that reads the integers and print their sum.

```
f = open("integers.txt", 'r')
sum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)        # converting the string to integer
        sum += number
print("The sum is", sum)
```

Ex:
Assume that a file integes.txt contains integers separated by newlines. Write a code segment that opens the file and prints the average value of the integers.

```
f = open("integers1.txt", 'r')
sum = 0
average=0
count=0
lines=f.readlines()
for line in lines:
        number = int(line)        # converting the string to integer
        count=count+1
        sum += number
print("The sum is", sum)
print("the average is",sum/count)
```

The with statement:

The with statement can be used while opening a file. We can use this to group file operation statements within a block. The syntax is

```
with open (file, mode) as file_handler:
        Statement_1
        Statement_2
        ..................
        ..................
        Statement_N
```

The advantage of with statement is that it will take care closing of file, after completing all Operations automatically even in the case of exceptions also. we are not required to close explicitly.

```
import os.path
fname=input("enter the file name with extension:")
if os.path.isfile(fname):
   print("the file",fname,"exists")
   print("The content of the file is")
   with open(fname,"r") as f:
      data=f.read()
      print(data)
```

```
else:
    print("The file",fname,"does not exist")
```

Output:

```
enter the file name with extension:mails.txt
the file mails.txt exists
The content of the file is
kumarap.1980@gmail.com
svietexams@gmail.com
abc@gmail.com
pavank_1980@rediffmail.com
```

Writing data to text files:

There are two ways to write the data to the file after it has been opened for writing. The methods that can be called on the file handler object for writing are given below.

Method	Syntax	Description
write()	file_handler. write(string)	This method will write the contents of the string to the file, returning the number of characters written. If you want to start a new line, you must include the new line character.
writelines()	file_handler. writelines(sequence)	This method will write a sequence of strings to the file.

Using write() method:

File handler objects have a method named write that can be used to write data to a file. Here is the general format of how you call the write method:

file_handler.write(string)

In the format, file_handler is a variable that references a file object, and string is a string that will be written to the file. **The file must be opened for writing (using the 'w' or 'a' mode) or an error will occur.**

The write method expects a single string argument. If you want the output text to end with a newline, you must include the escape character '\n' in the string

Ex:

Assume that customer_file references a file object, and the file was opened for writing with the 'w' mode. To write the string 'pavan kumar' to the file the following statement can be used

```
customer_file.write ('pavan kumar')
```

(or)

```
name = 'pavan kumar'
customer_file.write (name)
```

When we are done writing, we should close the file to make sure all of our changes take. Failure to close an output file can cause a loss of data.

Ex:

Write a program that creates a file and writes data to the file using write method

```
outputfile=open('sviet.txt','w')
outputfile.write("this file is created by pavan kumar.\n")
outputfile.write("He is working in sri vasavi Institute of engieering ad technology")
outputfile.write("Nandamuru,Pedana(M)")
outputfile.close()
print("writing to a file done successfully. you can open the file and see it")
```

Output:

```
writing to a file done successfully. you can open the file and see it
```

Note:

while writing data by using write() method, compulsory we have to provide line seperator(\n),otherwise total data should be written to a single line.

Using writelines() method:

The writelines() method writes a sequence of strings to the file. Each item contained in the sequence should be a string.

Ex:

Write a program that creates a file and writes data to the file using writelines method

```
f=open("sviet2.txt","w")
cities=["machilipatnam","bantumilli","pedana"]
f.writelines(cities)
f.close()
print("writing to a file done successfully. you can open the file and see it")
```

Output:

writing to a file done successfully. you can open the file and see it

Note:

while writing data by using writelines() method, compulsory we have to provide line seperator(\n),otherwise total data should be written to a single line. For ex, the content of the file sviet2.txt after writing the cities data is as follows.

machilipatnambantumillipedana

Manipulating file pointer using seek:

Every file maintains a file pointer which tells the current position in the file where writing or reading will take place. Python provides methods that help to manipulate the position of file pointer and thus read and write operations takes place from desired position in the file. The methods that can be called on the file handler object for this purpose are given below.

Method	Syntax	Description
tell()	file_handler.tell()	This method returns an integer giving the file handler's current position within the file, measured in bytes from the beginning of the file.
seek()	file_handler. seek(offset, from_what)	This method is used to change the file handler's position. The position is computed from adding *offset* to a reference point. The reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. If the *from_what* argument is omitted, then a default value of 0 is used, indicating that the beginning of the file itself is the reference point.

Ex:

```
data="All Students are STUPIDS"
f=open("abc.txt","w")
f.write(data)
f.close()
with open("abc.txt","r+") as f:
    text=f.read()
    print("The Initial content before modification:")
    print(text)
    print("The Current Cursor Position: ",f.tell())
    f.seek(17)
    print("The Current Cursor Position: ",f.tell())
    f.write("GEMS!!!")
    f.seek(0)
    text=f.read()
    print("The content After Modification:")
    print(text)
```

Output:

```
The Initial content before modification:
All Students are STUPIDS
The Current Cursor Position:  24
The Current Cursor Position:  17
The content After Modification:
All Students are GEMS!!!
```

```
#Program to print the number of lines,words and characters present in the given file
import os,sys
fname=input("Enter File Name: ")
if os.path.isfile(fname):
    print("The File",fname,"exists")
    f=open(fname,"r")
else:
    print("The File",fname," does not exists")
    sys.exit(0)
lcount=wcount=ccount=0
for line in f:
    lcount=lcount+1
    ccount=ccount+len(line)
    words=line.split()
    wcount=wcount+len(words)
print("The number of Lines:",lcount)
print("The number of Words:",wcount)
print("The number of Characters:",ccount)
```

Output:

```
Enter File Name: cars.txt
The File cars.txt exists
The number of Lines: 3
The number of Words: 3
The number of Characters: 20
```

Ex:

Assume that there is a wordlist text file called wordlist.txt where each line contains a different word. Write a python program to print all the 3 letter words from wordlist.txt file

```
wordlist = [line.strip() for line in open('wordlist.txt')]
for word in wordlist:
    if len(word)==3:
        print(word)
```

Ex:

Assume that there is a wordlist text file called wordlist.txt where each line contains a different word. Write a python program to print all the words that start with pe or pa.

```
list1 = [line.strip() for line in open('wordlist.txt')]
for word in list1:
    if word[:2]=='pe' or word[:2]=='pa':
        print(word)
```

Ex:

Write a python program to copy the contents of one file to another file.

```
import os.path
sfname=input("enter the source file name with extension:")
dfname=input("enter the destination file name with extension:")
if os.path.isfname(sfname):
   print("the file",sfname,"exists")
   fs = open(sfname,"r")
   fd = open(dfname,"w")
   data=fs.read()
   fd.write(data)
   fs.close()
   fd.close()
   print("the content of the file",sfname,"is copied to",dfname)
else:
   print("The file",sfname,"does not exist.copying cannot be done")
```

Output:

```
enter the source file name with extension:note1.txt
enter the destination file name with extension:sviet2.txt
the file note1.txt exists
the content of the file note1.txt is copied to sviet2.txt
```

Reading config files in python:

In computing, configuration files (commonly known simply as config files) are files used to configure the parameters and initial settings for some computer programs. They are used for user applications, server processes and operating system settings.

In linux, Some examples of configuration files include **rc.conf** for the system startup, **syslog.conf** for system logging and **httpd.conf** for the Apache Web server.

Windows desktops still have a desktop.ini file to configure desktop appearances, icons and other visual characteristics.

The configparser 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 configuaration 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.

Ex: A sample configuaration file called db.ini is given below.

```
[mysql]
host = localhost
user = user7
passwd = s$cret
db = ydb

[postgresql]
host = localhost
user = user8
passwd = mypwd$7
db = testdb
```

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.
has_option()	Return whether the given option exists in the given section.
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.
read_dict()	Read configuration from a dictionary. Keys are section names, values are dictionaries with keys and values that should be present in the section.
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.

Ex:
The python script that reads and parses the 'db.ini' file is given below.

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

Output:

```
Section: mysql
 host = localhost
 user = user7
 passwd = s$cret
 db = ydb

Section: postgresql
 host = localhost
 user = user8
 passwd = mypwd$7
 db = testdb
```

Writing log files in python:

In computing, a log file is a file that keeps a registry of events, processes, messages and communication between various communicating software applications and the operating system. Log files are present in executable software, operating systems and programs where by all the messages and process details are recorded. Every executable file produces a log file where all activities are noted.

The phenomenon of keeping a log is called logging where as a record file itself is called a log file. It usually depends on the developers on how they would like to generate logs.

Ex:

On a web server, an access log lists al the individual files that people have requested from a website. These files will include a HTML files and their imbedded graphic images and any other associated files that get transmitted. From the server's log files, an administrator can identify the number of visitors, the domains from which they are visiting , the number of requests for each page and usage patterns according to the variables such as times of the day, week, month or year.

Ex:
Every database has a transaction log that basically records all database modifications.

Ex:
The system log file contains events that are logged by the operating system components. System log files may contain information about device changes, device drivers, system changes, events, operations and more.

Python has a built-in module called logging which allows writing status messages to a file or any other output streams. These messages contain information on which parts of your code have executed, and what problems may have arisen.
Each log message has a level. The six built in levels are NOTSET,DEBUG,INFO, WARNING, ERROR AND CRITICAL. If you want, you can create additional levels, but these are sufficient for most cases.

Each level is associated with an integer that indicates the log severity.

level	Value	Severity
CRITICAL	50	Represents a very serious problem that needs high attention
ERROR	40	Represents a serious error
WARNING	30	Represents a warning message, some caution needed. It is alert to the programmer
. INFO	20	Represents a message with some important information
DEBUG	10	Represents a message with debugging information
NOTSET	0	Represents that the level is not set.

Implementing Logging:

To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store. We can do this by using basicConfig() function of logging module.

For example, consider the statement

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
```

The above statement will create a file log.txt and we can store either WARNING level or higher level messages to that file.

After creating log file, we can write messages to that file by using the following methods.

```
logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)
```

Python program to create a log file and write WARNING and higher level messages is given below.

```
import logging
logging.basicConfig(filename='log.txt',level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

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

```
WARNING:root:This is warning message
ERROR:root:This is error message
CRITICAL:root:This is critical message
WARNING:root:This is warning message
ERROR:root:This is error message
CRITICAL:root:This is critical message
```

Writing python exceptions to a log file:

By using the following function we can write exceptions information to the log file.

logging.exception(msg)

```
import logging
logging.basicConfig(filename='mylog.txt',level=logging.INFO)
logging.info("A New request Came:")
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError as msg:
    print("cannot divide with zero")
    logging.exception(msg)
except ValueError as msg:
    print("Enter only int values")
    logging.exception(msg)
logging.info("Request Processing Completed")
```

Object Oriented Programming:

Python is an object oriented programming language. OOP uses the concept of objects and classes.

class:

A class can be thought of as a 'blueprint' for objects. These can have their own attributes (characteristics they possess), and methods (actions they perform).

Defining a class:

To define a class in Python, you can use the class keyword, followed by the class name and a colon. The syntax to define a class is given as follows

```
class ClassName:
    " " " Optional class docstring" " "
    class_suite
```

- The class has a documentation string which has brief description of the class. This can be accessed via ***ClassName.__doc__***.
- The ***class_suite*** consists of all the component statements defining class members, data attributes and functions.

The variables defined inside a class are called class variables and the functions defined in the classes are called as methods. These class variables and class methods together called as class members. The class members can be accessed through class objects.

When a class definition is entered, a new namespace is created, and used as the local scope. This means that a class creates a new local namespace where all its attributes are defined. Attributes may be data or methods.

```
class Employee:

  " " "Common base class for all employees" " "

  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

 def displayCount(self):
   print ("Total Employee %d" % Employee.empCount)


  def displayEmployee(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

Creating objects:

Once a class is defined, we can create objects of the class. Creating objects of the class is called instantiation. The syntax for creating an object is

Objectname=classname( )

Ex:

```
emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
```

After an object is created, the class variables and methods can be accessed through objects using dot(.) operator.The syntax is

Objectname . class_variable_name
Objectname . class_method_name( )

Ex:

```
emp1.displayEmployee()
emp2.displayEmployee()
```

Methods:

Class methods are similar to the ordinary functions with just one small difference. **class methods must have the first argument named "self". This is the first argument that needs to be added to the beginning of the parameter list**.

But , When we call the method we don't pass any value for this parameter. Python provides it automatically. **The self argument refers to the object itself.**

Ex:

```
class Employee:

   """Common base class for all employees"""

   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1


   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)


   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)


emp1 = Employee("pavan", 2000)
emp2 = Employee("kumar", 5000)

emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

Output:

```
Name :  pavan , Salary:  2000
Name :  kumar , Salary:  5000
Total Employee 2
```

Here the variable empCount is a class variable whose value is shared among all instances of a class. This can be accessed as Employee.empCount from inside the class or outside the class.
Also the first method _ _init_ _ is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

Constructor:

A constructor is a special type of method which is used to initialize the instance members of the class. Constructor definition is executed when we create the object of this class.
In python, the method **__init__** ( ) simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon **__init__**( ) definition. It is mostly used to initialize the class attributes.

The _ _ init_ _( ) method can be declared as follows

```
def _ _ init _ _(self, argumentlist):

    statement-1
    statement-2
    ..............
    ..............
    statement-n
```

Ex:

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

    def show(self):
        print("Hello",self.name)

s1 = Student("John")
s1.show()
```

Output:

```
Hello John
```

Ex:

```
"""
File: student.py
Resources to manage a student's name and test scores.
"""
class Student(object):
 """Represents a student."""

 def __init__(self, name, number):
 """Constructor creates a Student with the given
 name and number of scores and sets all scores
 to 0."""
    self.name = name
    self.scores = []
    for count in range(number):
        self.scores.append(0)

 def getName(self):
 """Returns the student's name."""
    return self.name


 def setScore(self, i, score):
 """Resets the ith score, counting from 1."""
```

```
    self.scores[i - 1] = score


def getScore(self, i):
"""Returns the ith score, counting from 1."""
    return self.scores[i - 1]


def getAverage(self):
"""Returns the average score."""
    return sum(self.scores) / len(self.scores)


def getHighScore(self):
"""Returns the highest score."""
    return max(self.scores)


def __str__(self):
"""Returns the string representation of the
student."""
    return "Name: " + self.name + "\nScores: " + \
    " ".join(map(str, self.scores))
```

Destroying Objects: GarbageCollection: (Lifetime of objects)

Python deletes unneeded objects built – in types or class instances automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes. An object's reference count increases when it is assigned a new name or placed in a container list, tuple, ordictionary. The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope. **When an object's reference count reaches zero, Python collects it automatically.**

A class can implement the special method __del__( ), called a destructor, that is invoked when the instance is about to be destroyed.

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

    def show(self):
       print("Hello",self.name)

    def __del__(self):
        print("object of Student class is destroyed")
```

Output:

```
Hello John
object of Student class is destroyed
```

s1 = Student("John") s1.show() del s1	

The __str__ Method :

Many built-in Python classes usually include an __str__ method. This **method builds and returns a string representation of an object's state.** An object's state is simply the values of the object's attributes at any given moment. When the str function is called with an object, that object's __str__ method is automatically invoked to obtain the string that str returns. For example, the function call str(s) is equivalent to the method call s.__str__()

The programmer can return any information that would be relevant to the users of a class. Perhaps the most important use of __str__ is in debugging, when you often need to observe the state of an object after running another method.

Accessors and Mutators:

Methods that allow a user to observe but not change the state of an object are called accessors i.e., Accessor method returns a value from a class's attribute but does not change it.

Ex:

```
def getName(self):
 """Returns the student's name."""
    return self.name
```

Methods that allow a user to modify an object's state are called mutators i.e., A mutator method stores a value in a data attribute or changes the value of a data attribute.

Ex:

```
def setScore(self, i, score):
 """Resets the ith score, counting from 1."""
    self.scores[i - 1] = score
```

Built in functions to class:

1. **getattr(object, name[, default]) :**

The getattr( ) function returns the value of the named attribute of an object. If not found, it returns the default value provided to the function. If named attribute is not found and default is not defined then AttributeError exception raised. This is same as
object.name

```
class Person:
    age = 23
    name = "pavan"

p1 = Person()
print('The name of the person is:', getattr(p1,"name"),'and age is:', getattr(p1, "age"))
print('The name of the person is:', p1.name,'and age is:', p1.age)
```

The output of the above code is

```
The name of the person is: pavan and age is: 23
The name of the person is: pavan and age is: 23
```

2. hasattr(object, name):

The hasattr() function is used to check whether the object has named argument or not . this function returns True if an object has the given named attribute and False if it does not.

```
class Person:
    age = 23
    name = 'pavan'

p1 = Person()

print('Person has age?:', hasattr(p1, 'age'))
print('Person has salary?:', hasattr(p1, 'salary'))
```

Output:

```
Person has age?: True
Person has salary?: False
```

3. **setattr(object, name, value):**

The setattr( ) method sets the value of given attribute of an object.

```
class Person:
    name = 'pavan'

p = Person()
print('Before modification:', p.name)

# setting name to 'pavan kumar'

setattr(p, 'name', 'pavan kumar')

print('After modification:', p.name)
```

Output:

```
Before modification: pavan
After modification: pavan kumar
```

4. **delattr(object, name):**

The delattr() deletes an attribute from the object (if the object allows it). You can also delete attribute of an object using del operator.

```
class Person:
    name = 'pavan'

p = Person()
print('Before deletion:', p.name)

# deleting the attribute name
delattr(p, 'name')

print('After deletion:', p.name)
```

Output:

```
Before deletion: pavan

Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
    delattr(p, 'name')
AttributeError: name
```

Ex:

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

    def display(self):
        print("name is",self.name,'and age is',self.age)


p1 = Person('pavan',32)
p1.display()

print("person has age attribute?",hasattr(p1,'age'))
print("value for the age attribute is",getattr(p1,'age'))
setattr(p1,'age',39)
print("value for the age attribute is",getattr(p1,'age'))
delattr(p1,'age')
print("person has age attribute?",hasattr(p1,'age'))
```

Output:

```
name is pavan and age is 32
person has age attribute? True
value for the age attribute is 32
value for the age attribute is 39
person has age attribute? False
```

Built-in class attributes:

Built-in class attributes gives us information about the class. We can access the built-in class attributes using the dot(.) operator.Following are the built-in class attributes.

Attribute	Description
_ _dict_ _	This is a dictionary holding the class namespace.
_ _doc_ _	This gives us the class documentation if documentation is present. None otherwise.
_ _name_ _	This gives us the class name.
_ _module_ _	This gives us the name of the module in which the class is defined. In an interactive mode it will give us __main__.
_ _bases_ _	A possibly empty tuple containing the base classes in the order of their occurrence.

Ex:

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

    def display(self):
        print("name is",self.name,'and age is',self.age)


p1 = Person('pavan',32)
p1.display()

print('namespace of the class:',Person.__dict__)
print('name of the class:',Person.__name__)
print('name of the module:',Person.__module__)
print('name of the bases:',Person.__bases__)
```

The output of the above program is

```
name is pavan and age is 32
namespace of the class: {'display': <function Person.display at 0x7fb9c5917488>, '__dict__': <attribute '__dict__' of 'Person' objects>, '__init__': <function Person.__init__ at 0x7fb9c59176a8>, '__doc__': None, '__weakref__': <attribute '__weakref__' of 'Person' objects>, '__module__': '__main__'}
name of the class: Person
name of the module: __main__
name of the bases: (<class 'object'>,)
```

Rules of Thumb for Defining a Simple Class:

The list of several rules of thumb for designing and implementing a simple class are given below:

1. Before writing a line of code, think about the behavior and attributes of the objects of the new class. What actions does an object perform, and how, from the external perspective of a user, do these actions access or modify the object's state?

2. Choose an appropriate class name, and develop a short list of the methods available to users. This interface should include appropriate method names and parameter names, as well as brief descriptions of what the methods do. Avoid describing how the methods perform their tasks.

3. Write a short script that appears to use the new class in an appropriate way. The script should instantiate the class and run all of its methods.

4. Choose the appropriate data structures to represent the attributes of the class. These will be either built-in types such as integers, strings, and lists, or other programmer-defined classes.

5. Fill in the class template with a constructor (an __init__ method) and an __str__ method. Remember that the constructor initializes an object's instance variables, whereas __str__ builds a string from this information. As soon as you have defined these two methods, you can test your class by instantiating it and printing the resulting object.

6. Complete and test the remaining methods incrementally, working in a bottom-up manner. If one method depends on another, complete the second method first.

7. Remember to document your code. Include a docstring for the module, the class, and each method. Do not add docstrings as an afterthought. Write them as soon as you write a class header or a method header. Be sure to examine the results by running help with the class name.

Structuring Classes with Inheritance and Polymorphism:

The three most important features of object-oriented programming are encapsulation, inheritance, and polymorphism. All three features simplify programs and make them more maintainable.

Although Python is considered an object-oriented language, its syntax does not enforce data encapsulation.

Inheritance:

Inheritance refers to defining a new class with little or no modification to an existing class. Inheritance allows a new class to extend an existing class. The new class inherits the members of

the class it extends.The new class is called **derived (or child or sub) class** and the old class or existing class is called the **base (or parent or super) class**.

Python Inheritance Syntax:

```
class BaseClass:
  class_suite

class DerivedClass(BaseClass):
  class_suite
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code

```
# definition of the superclass starts here
class Person:
    #initializing the variables
    name = ""
    age = 0

    #defining constructor
    def __init__(self, personName, personAge):
        self.name = personName
        self.age = personAge

    #defining class methods
    def showName(self):
        print(self.name)

    def showAge(self):
        print(self.age)

    # end of superclass definition
#definition of subclass starts here

class Student(Person):              # Person is the  superclass and Student is the subclass
    studentId = ""

    def __init__(self, studentName, studentAge, studentId):
        Person.__init__(self, studentName, studentAge)    # Calling the superclass constructor
                                                            #and sending values of attributes.

        self.studentId = studentId

    def getId(self):
        return self.studentId                          #returns the value of student id

#end of subclass definition
```

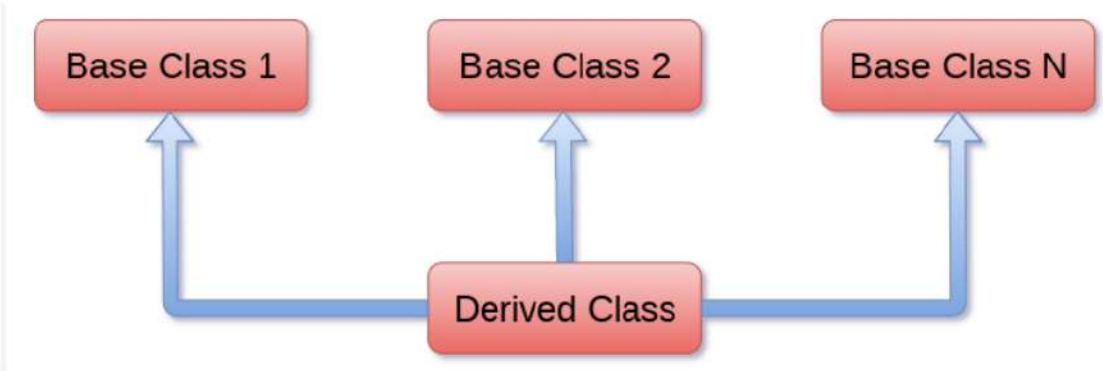
```
# Create an object of the superclass
person1 = Person("pavan", 23)
#call member methods of the objects
person1.showAge()
# Create an object of the subclass
student1 = Student("kumar", 22, 102)
print(student1.getId())
student1.showName()
```

Output:

23
102
kumar

multiple Inheritance:

A class can be derived from more than one base classes in Python. This is called multiple inheritance. In multiple inheritance, the features of all the base classes are inherited into the derived class. This is illustrated in the following figure.



Syntax:

```
class Base1:
    class-suite

class Base2:
    class-suite
    ..........
    ..........
    ..........
class BaseN:
    class-suite

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

```
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))
```

Output:

```
30
200
0.5
```

Note:
In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.

Ex:

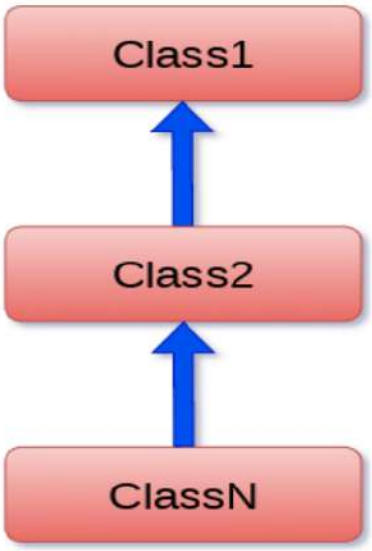
```
class B:
   def rk(self):
      print(" In class B")
class C:
   def rk(self):
      print("In class C")

# classes ordering
class D(B, C):
   pass

r = D()
r.rk()
```

Output:

In class B

Multilevel 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. This is illustrated in the following figure



Syntax:

```
class class1:
    class-suite
class class2(class1):
    class suite
class class3(class2):
    class suite
.....
.....
.....
```

```
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()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Polymorphism:

Polymorphism refers to the ability of the method with the same name to carry different functionality altogether.

Python can implement **polymorphism** in many different ways.

Polymorphism with methods and Objects:

The following script shows the use of polymorphism between two different classes. Here we have **two classes 'Rectangle'** and **'Square'**.

Both these classes have a method definition **'area'**, that calculates the area of the **corresponding shapes**.

```
class Rectangle:
    def __init__(self, length, breadth):
        self.l = length
        self.b = breadth
    def area(self):
        return self.l * self.b
class Square:
    def __init__(self, side):
        self.s = side
    def area(self):
        return self.s ** 2
rec = Rectangle(10, 20)
squ = Square(10)
print("Area of rectangle is: ", rec.area())
print("Area of square is: ", squ.area())
```

Output:

```
Area of rectangle is:  200
Area of square is:  100
```

Here, we implemented polymorphism using the method area(). This method works on objects of the types – Rectangle and Square. And it operates differently on objects of different classes.

Polymorphism with Inheritance in python

We can also achieve **polymorphism** with **inheritance**.Polymorphism allows subclasses to have methods with the same names as methods in their superclasses. It gives the ability for a program to call the correct method depending on the type of object that is used to call it.

Method overriding:

Method overriding is a concept of object oriented programming that allows us to change the implementation of a method in the **child class** that is defined in the **parent class**. It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

In Python, method overriding occurs by simply defining a method in the child class with the same name of a method in the parent class, but gives a new meaning.

```
class Robot:
    def action(self):
        print('Robot action')

class HelloRobot(Robot):
    def action(self):
        print('Hello world')

class DummyRobot(Robot):
    def start(self):
        print('Started.')

r = HelloRobot()
d = DummyRobot()

r.action()
d.action()
```

Output:

```
Hello world
Robot action
```

Note:

To call the base class method from the derived class method , we have to use super( ) built-in function. when we invoke the method using super( ) the parent version method is called.

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

class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal

    def display(self):
        print('Employee Name:',self.name)
        print('Employee Age:',self.age)
        print('Employee Number:',self.eno)
        print('Employee Salary:',self.esal)

e1=Employee('pavan',48,872425,26000)
e1.display()
print(".....")
e2=Employee('kumar',39,872426,36000)
e2.display()
```

Output:

```
Employee Name: pavan
Employee Age: 48
Employee Number: 872425
Employee Salary: 26000
.....
Employee Name: kumar
Employee Age: 39
Employee Number: 872426
Employee Salary: 36000
```

Data Hiding:

An object's attributes may or may not be visible outside the class definition .Python uses a special naming scheme for attributes **to control the accessibility of the attributes**. The attribute names, which can be freely used inside or outside of a class definition, correspond to public attributes.

There **are two ways to restrict the access** to class attributes:

- First, we can prefix an attribute name with a leading underscore "_". This marks the attribute as protected. It tells users of the class not to use this attribute unless, somebody writes a subclass.

- Second, we can prefix an attribute name with two leading underscores "__". The **attribute is now inaccessible and invisible from outside**. It's neither possible to read nor write to those attributes except inside of the class definition itself.

But in some situations, we need to access these attributes outside the class. This can be done with the following syntax:

```
Objectname . _className__attrName
```

class A:
 	
def __init__(self):
self.__priv = "I am private"
self._prot = "I am protected"
self.pub = "I am public"
 	
x = A()
print(x.pub)
print(x._prot)
print(x.__priv) #results in AttributeError
	Output:
	
	I am public
	I am protected
	Traceback (most recent call last):
	File "jdoodle.py", line 13, in <module>
	print(x.__priv)
	AttributeError: 'A' object has no attribute '__priv'

Note:

1.Private data fields and methods can be accessed within a class, but they cannot be accessed outside the class. To make a data field accessible for the client, provide a accessor(or getter) method to return its value. To enable a data field to be modified, provide a mutator(or setter) method to set a new value.

2.If a class is designed for other programs to use, to prevent data from being tampered with and to make the class easy to maintain, define data fields as private. If a class is only used internally by your own program, there is no need to hide the data fields.

Operator overloading:

Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings as can be seen below.

This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

Overaloading built-in operators:

By default, most of the built-in operators will not work with objects of your classes. But **In python, each built-in function or operator has a special method corresponding to it.**
You must **add the corresponding special methods in your class** definition to make your object compatible with built-in operators.

When you do this, the behavior of the operator associated with it changes according to that defined in the method thus providing the operator overloading.

For example, to overload the + operator to work with objects of our class, we will need to implement the _ _ add_ _( ) method in the class. we can write own code but an object of the class must be returned.

Ex:

```
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(2,3)
p2 = Point(-1,2)
print(p1 + p2)
```

Output:

(1,5)

When we call p1+p2, Python will call p1.__add__(p2) which in turn is Point.__add__(p1,p2)

Similarly, we can overload other operators as well. The special methods for some of the operators are given below.

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	__add__(self, other)	Addition
*	__mul__(self, other)	Multiplication
-	__sub__(self, other)	Subtraction
%	__mod__(self, other)	Remainder
/	__truediv__(self, other)	Division
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__ge__(self, other)	Greater than or equal to