Day-5: Module and File

27/06/2018

Workshop on “Introduction to Python

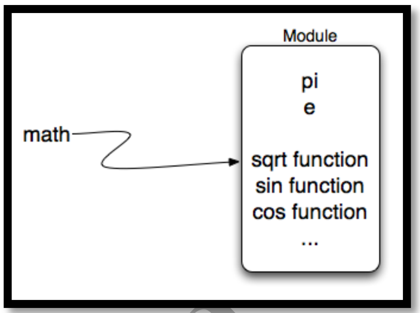
Department Of Computer Science Engineering

SKFGI, Mankundu

1.Modules

Sub Topics –

- Importing Module
- Math Module
- Random Module
- Namespace and DIR()
- Packages
- Composition



1.1.Why Modules?

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

1.2.Module Loading and Execution:

A module imported in a program must be located and loaded into memory before it can be used. When you import a module, the Python interpreter searches for the module in the following sequences −

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default installation specific path. On UNIX, this default path is normally /usr/local/lib/python/.
- If the module is not located even there then an error ImportError exception is generated .

A compiled version of the module with file extension .pyc is generated . next time when the module is imported , this .pyc file is loaded rather than .py file, to save the time of recompiling. A new compiled version of a module is again produced whenever the compiled version is out of date. Even programmer can force the python shell to reload and recompile the .py file to generate a new .pyc file by using reload function.

1.3.The *import* Statement

We can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax −

```
import module1[, module2[,... moduleN]
```

```
import sys
print(sys.path)
```

Output:

```
['', 'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-32\\Lib\\idlelib',
'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-32\\python36.zip',
'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-32\\DLLs',
'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-32\\lib',
'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-32',
```

'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-32\\lib\\site-packages']

1.3.2The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax −

from <module_name> **import** <attribute1,attribute2,attribute3,...attributen>

Example:

1.from math import pi

print(pi)

2. from math import pi,sqrt

print(sqrt(4))

print(pi)

Note1: from **from** <module_name> **import** * uses to import all the identifier in the module except those beginning with an underscore(_)

Example: from math import *

print(sqrt(4))

Note2: using as keyword we can also import a module with a different name. It is help to create alias of module name.

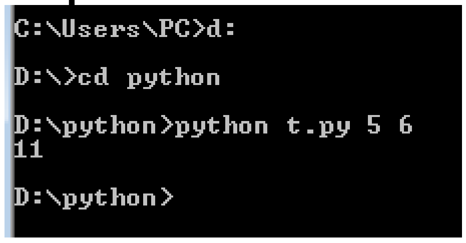
Example: from math import sqrt as square_root

print(square_root (4))

P1. Write a program to add two number that are given using command line argument:

```
import sys
a=int(sys.argv[1])
b=int(sys.argv[2])
sum=a+b
print(sum)
```

Output:



```
C:\Users\PC>d:

D:\>cd python

D:\python>python t.py 5 6
11

D:\python>
```

1.4.Calendar import

- **calendar(year, w, l, c)** :- This function displays the year, width of characters, no. of lines per week and column separations.
  print (calendar.calendar(2012,2,1,6))
- **firstweekday()** :- This function returns the first week day number. By default 0 (Monday).
  print (calendar.firstweekday())
- **isleap (year)** :- This function checks if year mentioned in argument is leap or not.
  ```
  if (calendar.isleap(2008)):

          print ("The year is leap")
    else : print ("The year is not leap")
  ```
- **month (year, month, w, l)** :- This function prints the month of a specific year mentioned in arguments. It takes 4 arguments, year, month, width of characters and no. of lines taken by a week.
  print (calendar.month(2016,5,2,1))

1.5. Making your own Module:

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__. For instance, use text editor to create a file called fibo.py in the current directory with the following contents:

```
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print(__name__)
```

```
def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
import fibo
fibo.fib(1000)
fibo.fib2(100)

Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
'fibo'
```

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module’s global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module’s symbol table. For example:

```
from fibo import fib, fib2
```

```
 fib(500)
```
Output:
```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

There is even a variant to import all names that a module defines:

```
 from fibo import *
fib(500)
```
output:
```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

If the module name is followed by as, then the name following as is bound directly to the imported module.

```
 import fibo as fib
 fib.fib(500)
```
Output:
```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that import fibo will do, with the only difference of it being available as fib.

It can also be used when utilising from with similar effects:

```
from fibo import fib as fibonacci
 fibonacci(500)
```
Output:
```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

1.6.Math Modules:

It provides access to the mathematical functions defined by the C standard.
These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers.

Method	Explanation	Example	Output
math.ceil(*x*)	Return the ceiling of *x*, the smallest integer greater than or equal to *x*.	math.ceil(4.2)	5
math.copysign(*x*, *y*)	Return a float with the magnitude (absolute value) of *x* but the sign of *y*	math.copysign(5, -1)	-5.0
math.fabs(*x*)	Return the absolute value of *x*.	math.fabs(-5)	5.0
math.floor(*x*)	Return the floor of *x*, the largest integer less than or equal to *x*.	math.floor(*4.2*)	4
math.factorial(*x*)	Return *x* factorial. Raises ValueError if *x* is not integral or is negative.	math.factorial(4)	24

math.fmod(*x*, *y*)	Return x%y	math.fmod(*4*,2)
		math.fmod(*5,2*)
		math.fmod(*5.2*, *2*)
			0
			1
			1.5

math.gcd(*a*, *b*)	Return the greatest common divisor of the integers *a* and *b*.	math.gcd(5, 21)	1

math.fsum([values])	Return sum value	math.fsum([1,2,3]	6
math.trunc(*x*)	Return the Real value *x*	math.trunc(*4.29*)	4
math.modf(*x*)	Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.	math.modf(*4.2*)	(0.20000000000000018, 4.0)
math.exp(*x*)	Return e^x	math.exp(*0*)	1.0

math.log(*x*[, *base*])	With one argument, return the natural logarithm of *x* (to base *e*).With two arguments, return the logarithm of *x* to the given *base*	math.log(2,10)
		math.log(2,2)
		math.log(2)
			0.30103
			1.0
			.69315

math.log2(*x*)	Return the base-2 logarithm of *x*. This is usually more accurate than log(x, 2).	math.log2(2*)*	1.0
math.log10(*x*)	Return the base-10 logarithm of *x*. This is usually more accurate than log(x, 10).	math.log10(2)	0.301029
math.pow(*x*, *y*)	Return x raised to the power y.	math.pow(*2*, 3)	8
math.sqrt(*x*)	Return the square root of *x*.	math.sqrt(81)	9
math.cos(*x*)	Return the cosine of *x*, in radians.	math.cos(0)	1
math.sin(*x*)	Return the sine of *x*, in radians.	math.sin(0)	0
math.tan(*x*)	Return the tan of *x*, in	math.tan(0)	0

	radians.		
math.degrees(*x*)	Convert angle *x* from radians to degrees.	math.degrees(1)	57.295779
math.radians(*x*)	Convert angle *x* from degrees to radians	math.radians(57.29577951308232)	1.0
math.pi	The mathematical constant π = 3.141592…, to available precision.	print(math.pi)	3.14159
math.tau	The mathematical constant τ = 6.283185	print(math.tau)	6.283185
math.e	The mathematical constant e = 2.718281…	print(math.e)	2.718281

1.7. Random Module: This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

Import random

Method	Explanation	Example	Output
random.randrange (*start*, *stop*, *step*)	Return a randomly selected element from range(start, stop, step).	random.randrange(5, 15, 3)	11
random.randint(*a*, *b)*	Return a random integer *N* such that a <= N <= b.	random.randint(5,15)	13
random.choice(*seq*)	Return a random element from the non-empty sequence *seq*. I	random.choice([1,2,3,4,5])	3

random.shuffle(sequence)	Shuffle the sequence	l=[1,21,3,4,5]
		random.shuffle(l)
		print(l)
			[4, 3, 1, 5, 21]

random.random()	Return the next random floating point number in the range [0.0, 1.0).	random.random()	0.268373
random.uniform(*a*, *b*)	Return a random floating point number *N* such that a <= N <= b for a <= b and b <= N <= a for b < a.	random.uniform(*2*,*5*)	3.42618

1.8. NAMESPACES AND SCOPING

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
Money = 2000
def AddMoney():     # Uncomment the following line to fix the code:
#global Money
   Money = Money + 1
print (Money)
AddMoney()
print (Money)
```

1.9. The **dir( )** Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

```
#!/usr/bin/python

# Import built-in module math

import math

content = dir(math)

print (content)
```

When the above code is executed, it produces the following result −

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',

'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',

'sqrt', 'tan', 'tanh']
```

Here, the special string variable __*name*__ is the module's name, and __*file*__is the filename from which the module was loaded.

1.10.The *globals()* and *locals()* Functions

The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

- If locals() is called from within a function, it will return all the names that can be accessed locally from that function.
- If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

```
b=2
c=5
l=globals()
def localsPresent():
    present = True
    a=5
    return locals()
print('globalPresent:',l)
print('localsPresent:', localsPresent())
Output:
```

```
globalPresent: {'__name__': '__main__', '__doc__': None,
'__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'__file__':
'C:\\Users\\PC\\AppData\\Local\\Programs\\Python\\Python36-
```

32\\tr.py', 'b': 2, 'c': 5, 'l': {...}, 'localsPresent': <function localsPresent at 0x013D3300>}

localsPresent: {'a': 5, 'present': True}

1.11. Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on. Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a submodule named B in a package named A.

1.11.1User Defined package:

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code −

```
def Pots():

  print ("I'm Pots Phone")
```

Similar way, we have another two files having different functions with the same name as above −

- *Phone/Isdn.py* file having function Isdn()
- *Phone/G3.py* file having function G3()

Now, create one more file __init__.py in *Phone* directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows −

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
# Now import your Phone Package.

import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

When the above code is executed, it produces the following result −

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

Question 1

Create a module named example.py. It should have a function add() inside the module named example. The function takes in two numbers and returns their sum.
Using the module name apply the dot (.) operation. For example, add two numbers 4 and 5

sum.py	__init__	test.py

def sum(a,b):
 	
    return a+b
	from sum import sum	import example
		
		print(example.sum(4,5))

Note: Create a folder "example" and store the above three file in "example" folder

1.11.2Package installation:

1. If you're using **Python 2.7.9 (or greater)** or **Python 3.4 (or greater)**, then PIP comes installed with Python by default. You have to make sure Python is properly installed on your system. On Windows, open up the Command Prompt . then type

```
python –version
pip –version   # pip stands for "preferred installer program"
```

2. Update pip version

```
python -m pip install --upgrade pip
```

3. Install package :

```
pip install <packagename>
 Example:
 pip install gmplot         #for google map
 pip install numpy          #for scientific computing
```

P1: To create a Base Map using google map

```
# GoogleMapPlotter return Map object Pass the center latitude and center longitude and zoom

import  gmplot
gmap1 = gmplot.GoogleMapPlotter(22.849962,  88.344772,100 )
 # Pass the absolute path
gmap1.draw( "D:map11.html" )
```

p2. Scatter points on the google map and draw a line in between them .

```
# import gmplot package
import gmplot
 latitude_list = [ 30.3358376, 30.307977, 30.3216419 ]
longitude_list = [ 77.8701919, 78.048457, 78.0413095 ]
gmap3 = gmplot.GoogleMapPlotter(30.3164945,78.03219179999,13)
# scatter method of map object  scatter points on the google map
gmap3.scatter( latitude_list, longitude_list, 'red', size = 40, marker = False )
 # Plot method Draw a line in between given coordinates
gmap3.plot(latitude_list, longitude_list,
          'cornflowerblue', edge_width = 2.5)

gmap3.draw( "D:map11.html" )
```

P3: Create Two dimensional array using Numpy

```
 import numpy as np
a = np.arange(15).reshape(3, 5)
print (a)
print(a.shape)
```

P4: Plot a graph

```
import matplotlib.pyplot as plt
C = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]
plt.plot(C)
plt.show()
```

1.12. Function Composition in Python

Function composition is a way of combining functions such that the result of each function is passed as the argument of the next function. For example, the composition of two functions f and g is denoted f(g(x)). x is the argument of g, the result of g is passed as the argument of f and the result of the composition is the result of f.

Let's define compose2, a function that takes two functions as arguments (f and g) and returns a function representing their composition:

```
def compose2(f, g):
    return lambda x: f(g(x))
def double(x):
    return x * 2
def inc(x):
    return x + 1
inc_and_double = compose2(double, inc)
x=inc_and_double(10)
print(x)
```

Output:

```
22
```

2. Topic-File

Sub Topics –

- What is file
- Text and Binary File
- Opening and closing file
- Reading and writing files
- Functions

2.1What is a file?

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

2.2 What is Text file?

A text file is a stream of characters that can be sequentially processed by a computer in forward direction. For this reason , a text file is usually opened only one kind of operation(reading, writing, appending). It is Stream-oriented data files i.e. the data is stored in same manner as it appear on the screen. The I/O operation like buffering, data conversions etc. take place automatically.

2.3What is Binary file?

A file stored in binary format. A binary file is computer -readable but not human-readable. All executable programs are stored in binary files, If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used. It is System oriented data file i.e. system oriented data file are more closely associated with the OS and data stored in memory without converting into text format.

2.3 How to open a file?

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("D:\\README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Python File Modes	
Mode	Description
'r'	Open a file for reading. (default)
‘rb’	This mode opens a file for reading only in binary mode
'w'	Open a file for writing. Creates a new file if it does not exist or the content are overwritten in the file if it exists.
’wb’	Opens a file in binary format for writing only. Creates a new file if it does not exist or the content are overwritten in the file if it

	exists.
‘r+’	This mode opens a file for both reading and writing the file pointer is placed at the beginning of the file
‘rb+’	This mode opens a file for both reading and writing in the binary format . the file pointer is placed at the beginning of the file
‘w+’	Open a files for both writing and reading
‘wb+’	Open a files for both writing and reading in binary foemat
'a'	Open for appending at the end of the file without overwriting it. Creates a new file if it does not exist. the file pointer is placed at the end of the file
‘ab’	Open for appending at the end of the file in binary format without overwriting it. Creates a new file if it does not exist. the file pointer is placed at the end of the file
't'	Open in text mode. (default)
‘a+’	Opens a file for reading and appending . the file pointer is placed at the end of the file
'ab+'	Opens a file for reading and appending in binary format . the file pointer is placed at the end of the file

Example:

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
```

Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

2.4The file Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all the attributes related to a file object −

S.No.	Attribute & Description
1	**file.closed** Returns true if file is closed, false otherwise.
2	**file.mode** Returns access mode with which file was opened.
3	**file.name** Returns name of the file.

Example:

```
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
fo.close()
```

2.5How to close a file Using Python?

When we are done with operations to the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file and is done using Python close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

```
f = open("test.txt",encoding = 'utf-8')
# perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

2.6 How to write to File Using Python?

In order to write into a file in Python, there are two method a) write() b) writelines().

2.6.1 write() method: The write() method is used to write a string to an already opened file. Write() method does not add a new line character to the end of the string. The write method returns none
f=open("d:\\test.txt",'w',encoding = 'utf-8')
f.write("my first file\n")

```
f.write("This file\n\n")
f.write("contains three lines\n")
f.close()
```

This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten.

2.6.2writelines() method:

it is used to write a list of string or sequence.

```
f=open("d:\\test.txt",'w',encoding = 'utf-8')

line=["hello world","welcome to theworld of  python","enjoy learning python"]

f.writelines(line)

f.close()
```

2.7How to read files in Python?

There are three method a)read() b)readline() c) list()

2.7.1 read() method:

it is used to read a string from an already opened file. The string can include alphabets, number, character , symbol

sysntax: fileobj.read([count]). Count is an optional parameter which if passed to the read() specifies number of byte to be read from opened file. Read method start reading from the beginning of the file, if count is missing or has a negative value then it read the entire content

```
f=open("d:\\test.txt",'r',encoding = 'utf-8')

print(f.read(10))

print(f.read())

f.close()
```

2.7.2 readline() method :

it is used to read a single line from the file . It returns an empty string when the end of the file has been reached. Note that a blank line is represented by \n and the readline() method returns a string containing only a single newline character when a blank line is encountered in the file

test.text file

```
hello world

welcome to the world of  python
enjoy learning python
```

```
f=open("d:\\test.txt",'r',encoding = 'utf-8')

print(f.readline())

print(f.readline())

print(f.readline())

print(f.readline())

f.close()
```

2.7.3 list() method: it is used to display entire content of the file as a list. we just need to pass the file object as an argument to the lsit() method.

```
f=open("d:\\test1.txt",'r')

k=list(f)

print(k)
```

2.8 Some other useful file method:

2.8.1 next() : The method **next()** is used when a file is used as an iterator, typically in a loop, the next() method is called repeatedly. This method returns the next input line, or raises *StopIteration* when EOF is hit.

```
f=open("d:\\test.txt",'r',encoding = 'utf-8')

for index in range(3):

   line = next(f)

   print ("Line No %d - %s" % (index, line))

fo.close()
```

2.8.2 truncate(): Resize the file to n byte

```
f=open("d:\\test1.txt",'w',encoding = 'utf-8')

f.write("hello student........")
```

```
f.truncate(10)

f=open("d:\\test1.txt",'r',encoding = 'utf-8')

print(f.read())

f.close()
```

2.8.3 tell(): It tells the current position within the file at which the next read and write operation will occur. It is specified as number of bytes from the beginning of the file. When you just open a file for reading, the file pointer is positioned at location 0, which is beginning of the file **.**

```
f=open("d:\\test.txt",'rb')

print(f.tell())

f.read(3)

print(f.tell())
```

2.8.4 seek(): The seek (offset,whence) is used to set the position of the file pointer or in simpler term move the file pointer to a new location. The offset argument indicates the number of bytes to be moved and the whence argument is optional and defaults to 0, which means from the beginning of the file, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Note: to execute the seek method it is mandatory to open a file 'rb' mode. If open a file r or w mode then whence argument is not used.

```
f=open("d:\\test.txt",'rb')
print(f.tell())
f.seek(3,0)
print(f.tell())
f.seek(5,1)
print(f.tell())
f.seek(-6,1)
print(f.tell())
```

output:

```
0
3
```

```
8
2
```

Example:

```
f=open("d:\\test.txt",'rb')

print(f.tell())

print(f.read(3).decode('utf-8'))

f.seek(3)

print(f.read(2).decode('utf-8'))

f.seek(2,1)

print(f.read(2).decode('utf-8'))

f.seek(-5,1)

print(f.read(2).decode('utf-8'))

f.close()
```

Note: decode () is used to convert from binary file to utf-8 text file

2.9 Opening files using with keyword

It is good programming habit to use the with keyword when working with file object. This ahs advantage that the file is properly closed after it is used even if an error occur during read or write operation or even when you forget to explicitly close the file.

Example:

```
with open("d:\\test.txt",'r',encoding = 'utf-8') as f:
    l=f.read(12)
    print(l)
```

2.10 Renaming File:

The rename() method takes two argument, the current file name and new file name

```
import os

os.rename('d:\\test.txt','d:\\new.txt')
```

2.11 remove () method:

This method can be used to delete files. This method takes a filename as an argument and deletes that file.

```
import os
os.remove('d:\\new.txt')
```

2.12 mkdir() method:

It is used to create a directories. it takes the name of the directory as an argument.

```
import os

os.mkdir('d:\\skfgi')
```

2.13 getcwd() method:

It is used to display the current working directory

```
import os

print(os.getcwd())
```

2.14 rmdir() method:

It is used to remove the directory

```
import os

os.rmdir('d:\\skfgi')
```

2.15 chdir() method:

It is used to change directory . The method takes the name of the directory as an argument which you want to make current directory .

```
import os

print(os.getcwd())

os.chdir('d:\\')

print(os.getcwd())
```

Example :Write a program that accept filename as an input from the user. Open the file and count the number of times a character appears in the file

```
fname=input("enter the file name")

with open(fname) as file:

    text=file.read()

    letter =input('enter the character to be searched ')

    c=0

    for char in text:

        if char==letter:

            c=c+1

print(letter,"appear", c, "times in file")
```

Note: store .py file and text file at same location.

Example2: WAP that reads data from a file and count vowel and consonant.

```
fname=input("enter the file name")
with open(fname) as file:
    text=file.read()
    c=0
    v=0
    for char in text:
        if char in "aeiou":
            v=v+1
        else:
            c=c+1
print("total consonants are",c)
print("total vowels are",v)
```