UNIT 12 MODULES AND PACKAGES

Structure

12.0 INTRODUCTION

Modules are files that contain various functions, variables or classes which are logically related in some manner. Modules like functions are used to implement modularity feature of OOPs concept. Related operations can be grouped together in a file and can be imported in other files. Package is a collection of modules and other sub-modules. Modules can be well organized and easily accessible if collectively stored in a package.

12.1 OBJECTIVES

Afetr going through this unit, you will be able to :

- Understand usage of Modules
- Create your own Modeules
- Compare Modules and scripts
- Import packages and Create your own packages
- Understand the standard library modules

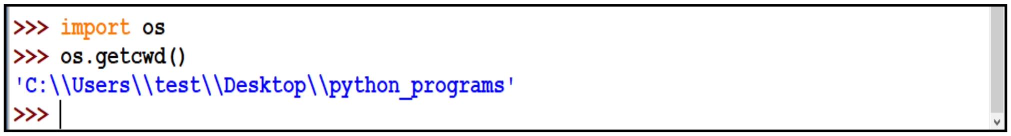
12.2 MODULE CREATION AND USAGE

Module is a logical group of functions , classes, variables in a single python file saved with .pyextension.In other words, we can also say that a python file is a module. We have seen few examples of built-in modules in previous chapters like os, shutil which are used in the program by import statement. Python provides many built-in modules. We can also create our own module.

A major benefit of a module is that functions, variable or objects defined in one module can be easily used by other modules or files, which make the code re-usable. A module can be created like any other python file. Name of the module is the same as the name of a file. Let us create out first module- series.py.

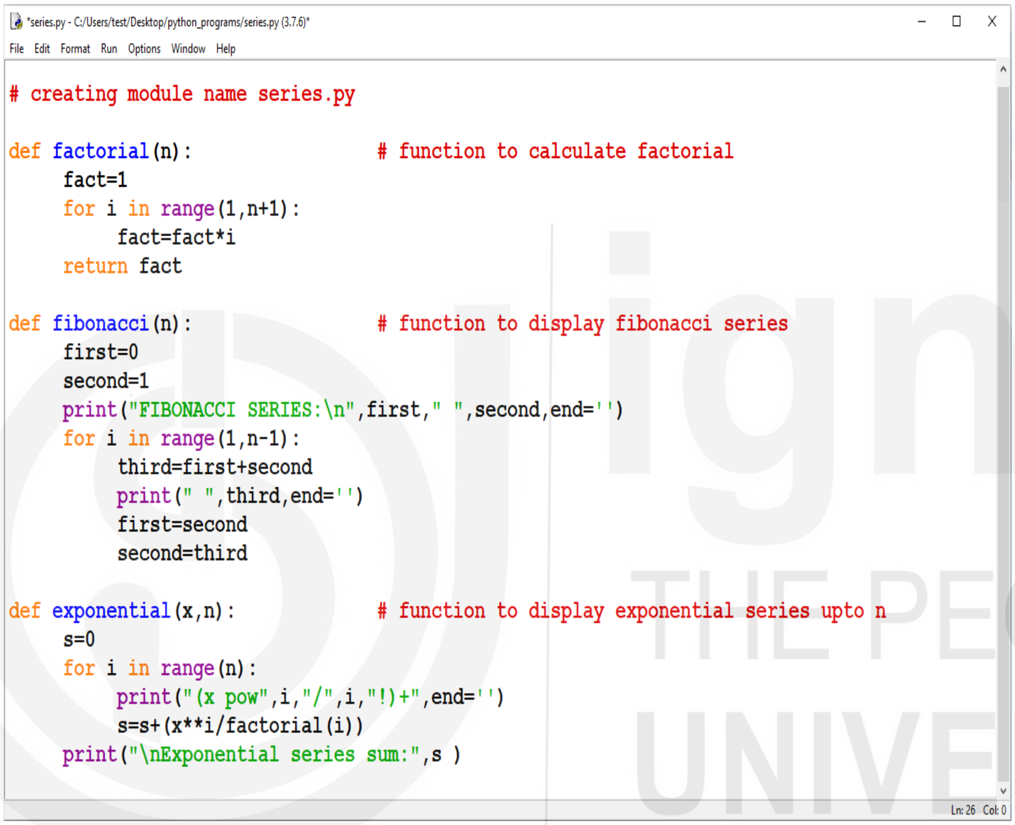
In this module, we have created three functions- to find factorial of a number, fibinacci series upto given number of terms and a function to display exponential series and it sum. After creating a file, save it with name series.py.

Note: it is important to check where we have saved this file or module. Currently, it is saved in my present working directory. You can check current working directory by using built-in function *getcwd()* under os module by using following syntax in console window or python prompt.



```
>>> import os
>>> os.getcwd()
'C:\\Users\\test\\Desktop\\python_programs'
>>> 
```

Example 1: Creating module named series.py



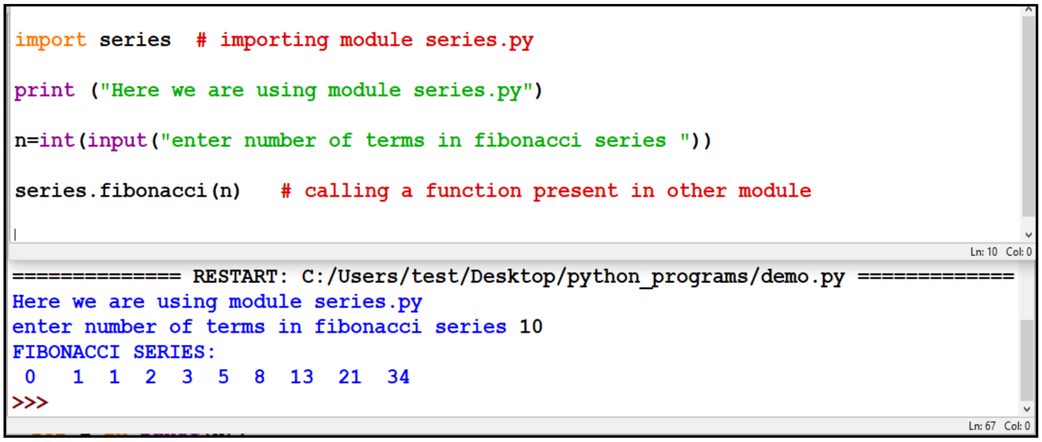
series.py - C:/Users/test/Desktop/python_programs/series.py (3.7.6)
File Edit Format Run Options Window Help
Ln: 26 Col: 0

```
# creating module name series.py

def factorial(n):                # function to calculate factorial
    fact=1
    for i in range(1,n+1):
        fact=fact*i
    return fact

def fibonacci(n):                # function to display fibonacci series
    first=0
    second=1
    print("FIBONACCI SERIES:\n",first," ",second,end='')
    for i in range(1,n-1):
        third=first+second
        print(" ",third,end='')
        first=second
        second=third

def exponential(x,n):            # function to display exponential series upto n
    s=0
    for i in range(n):
        print("(x pow",i,"/",i,"!)+",end='')
        s=s+(x**i/factorial(i))
    print("\nExponential series sum:",s )
```

Our module is successfully created. Now let us test our module by importing in some other file and check whether it is working or not. For verifying that, in a new file, two steps are needed to be done-

1. Import the module we have created to make it accessible
2. Call the functions of that module with module name and a dot (.) symbol.

Example 2: Accessing function in module created in Example 1.



```
import series  # importing module series.py

print ("Here we are using module series.py")

n=int(input("enter number of terms in fibonacci series "))

series.fibonacci(n)   # calling a function present in other module
```

```
============== RESTART: C:/Users/test/Desktop/python_programs/demo.py =============
Here we are using module series.py
enter number of terms in fibonacci series 10
FIBONACCI SERIES:
 0   1  1  2  3  5  8  13  21  34
>>>
```

Similar to the above example, we can call another function created in the module fibonacci() by following the same process.

```
import series

Series. fibonacci ( 2, 10 )
```

When a module is imported in a file, a folder named __pycache__ folder gets created by interpreter which contains .pyc file of a module imported. This file contains the compiled bytecode of module so that conversion from source code to bytecode can be skipped for subsequent imports and making execution faster.

Importing a module

Importing is the process of loading a module in other modules or files. It is necessary to import a module before using its functions, classes or other objects. It allows users to reference its objects. There are various ways of importing a module.

1. using *import* statement
2. using *from import* statement
3. using *from import* * statement

1. Importing Complete module

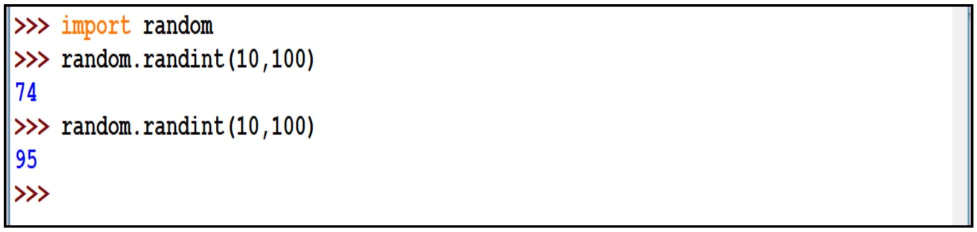
In this method, we can import the whole module all together with a single import statement. In this process, after importing the module, each function (or variable, objects etc.) must be called by the name of the module followed by dot (.) symbol and name of the function.

Syntax of function calling within module

```
import module
module.function_name()
```

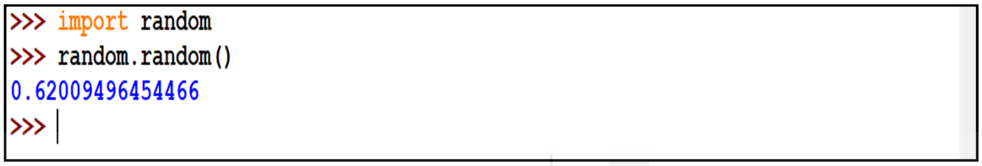
For example, let us import the built-in module random, and call its function randint(), which generates a random integer between a range given by user.

This can be done by running the code below in console window directly or in a python file.



```
>>> import random
>>> random.randint(10,100)
74
>>> random.randint(10,100)
95
>>>
```

In this method, other functions or objects present in module random can be called similarly.



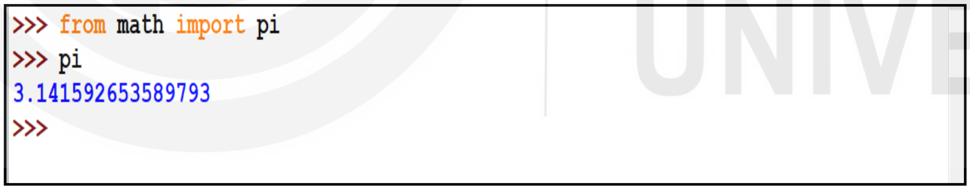
```
>>> import random
>>> random.random()
0.62009496454466
>>> |
```

Here,random () is function present within module random.

2. Importing using *from import* statement

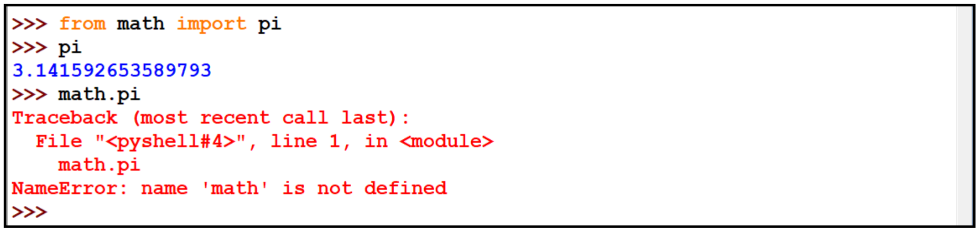
In this method of importing, instead of importing the entire module function or objects, only a particular object needed can be imported. In this method, objects can be directly accessed with its name.

Let us take an example of another module called math. This module contains various functions and variables. One such variable is pi, which contains value of π.



```
>>> from math import pi
>>> pi
3.141592653589793
>>>
```

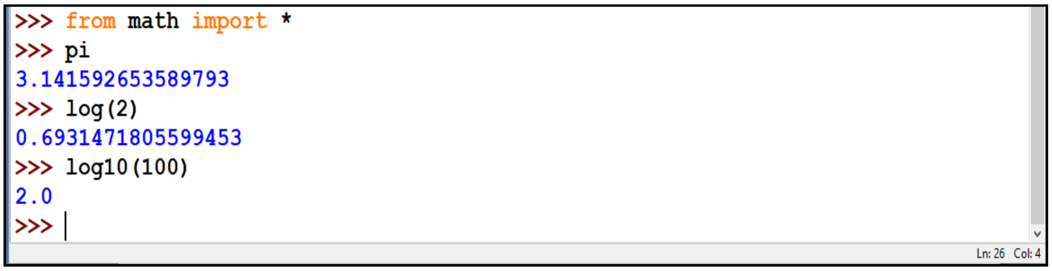
In this example, only a variable called pi is imported from math module, hence it can be directly accessed with its name. In this case, module name cannot be used for calling its objects, doing this will show NameError. Also other functions within the module math cannot be accessed, since only one variable pi is imported. You will be able to access them only after importing them individually with *from import* statement.



```
>>> from math import pi
>>> pi
3.141592653589793
>>> math.pi
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    math.pi
NameError: name 'math' is not defined
>>>
```

3. Importing entire module using *from import **

This method can be used to import the entire module using from import * statement. Here,*represents all the functions of a module. Like previous method, an object can be accessed directly with its name.



```
>>> from math import *
>>> pi
3.141592653589793
>>> log(2)
0.6931471805599453
>>> log10(100)
2.0
>>> |
```

Check your Progress 1

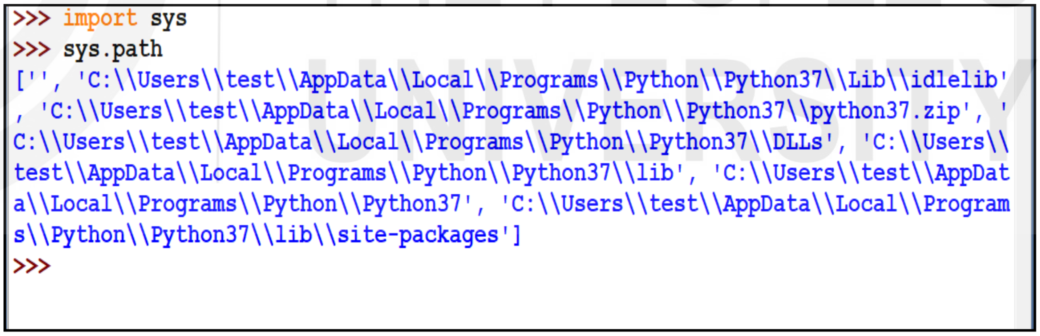
Ex 1. What are modules in Python and how can we create modules ?

Ex 2. What are the various ways of importing modules ?

Ex 3. Name any 3 built-in modules in python.

12.3 MODULE SEARCH PATH

When we use import statements to import a module, it is searched in a list of directories or search paths stored by the environment variable PYTHONPATH.This list of directories can be checked using sys.path variable.



```
>>> import sys
>>> sys.path
['', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib'
, 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip', '
C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'C:\\Users\\
test\\AppData\\Local\\Programs\\Python\\Python37\\lib', 'C:\\Users\\test\\AppDat
a\\Local\\Programs\\Python\\Python37', 'C:\\Users\\test\\AppData\\Local\\Program
s\\Python\\Python37\\lib\\site-packages']
>>>
```

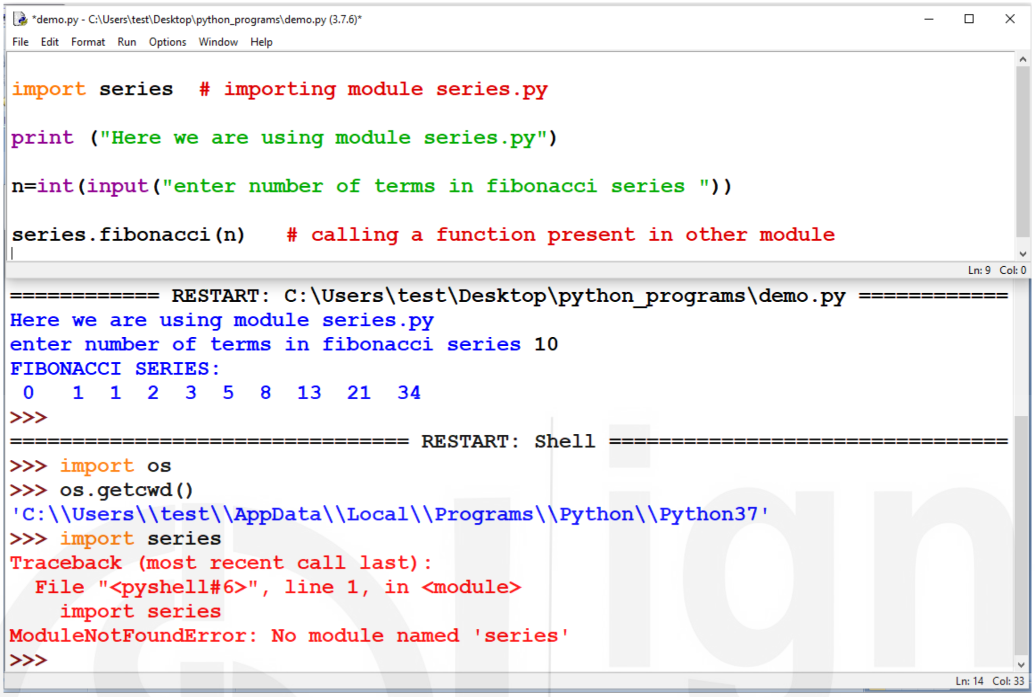
These directories include:

1. The current directory in which user is working [‘ ’]
2. Installation dependent paths
3. Directories stored in variable PYTHONPATH

Upto now, we were able to import our modules without doing anything special because they were all created in the same current directory. But if we move to some other directory, and try to import modules located in previous directories, we will not be able to use it.

In example 1 of this unit, we have created a module named series.py and used this module in a file named demo.py in example2. We were able to import modulessince both of themwere in the same directory. But when we re-start shell, we move to python's default location. In this location, we will not be able to import of series.py module. Shown in example 3 below.

Example 3:



demo.py - C:\Users\test\Desktop\python_programs\demo.py (3.7.6)
File Edit Format Run Options Window Help
Ln: 9 Col: 0
Ln: 14 Col: 33

```
import series  # importing module series.py

print ("Here we are using module series.py")

n=int(input("enter number of terms in fibonacci series "))

series.fibonacci(n)   # calling a function present in other module
```

```
============ RESTART: C:\Users\test\Desktop\python_programs\demo.py ============
Here we are using module series.py
enter number of terms in fibonacci series 10
FIBONACCI SERIES:
 0   1  1  2  3  5  8  13  21  34
>>>
================================ RESTART: Shell ================================
>>> import os
>>> os.getcwd()
'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37'
>>> import series
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    import series
ModuleNotFoundError: No module named 'series'
>>>
```

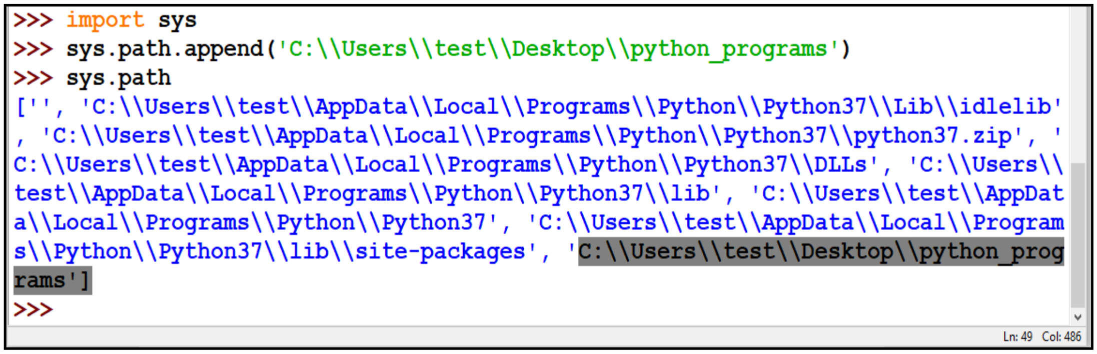
Therefore, any modules created must be located inpython's search path for its global identification. This can be done in either of the ways-

1. Creating module in one of the locations already present in search path
2. Adding your module path in the search path using sys.path.
3. Updating PYTHONPATH environment variable.
4.

Adding module location to search path

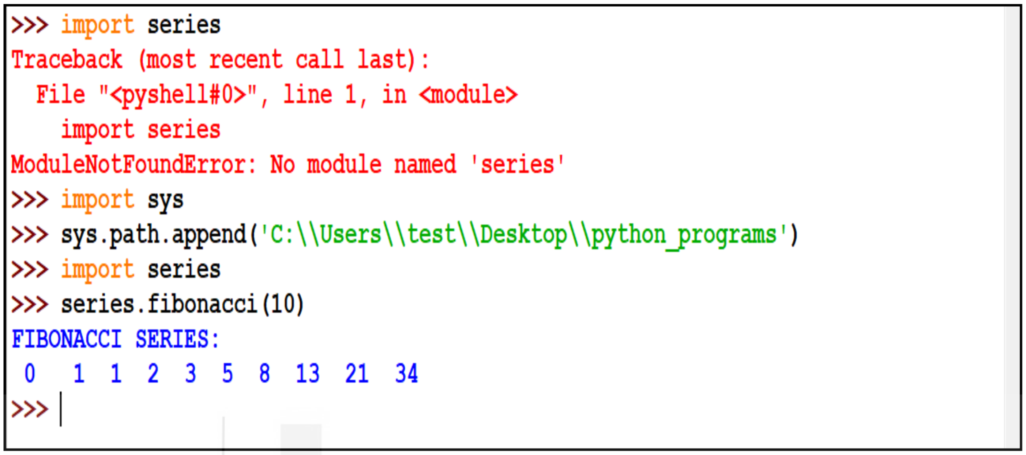
A module path can be added to python's module search path by appending the sys.path variable. This can be done by using the append( ) function of sys.path. Directory in which your module is located should be appended as shown below example 4.

Example 4: Adding module path to search path



```
>>> import sys
>>> sys.path.append('C:\\Users\\test\\Desktop\\python_programs')
>>> sys.path
['', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib'
, 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip', '
C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'C:\\Users\\
test\\AppData\\Local\\Programs\\Python\\Python37\\lib', 'C:\\Users\\test\\AppDat
a\\Local\\Programs\\Python\\Python37', 'C:\\Users\\test\\AppData\\Local\\Program
s\\Python\\Python37\\lib\\site-packages', 'C:\\Users\\test\\Desktop\\python_prog
rams']
>>>
```

As we can see in above example, our directory is now present in the list of search directories. Hence, now we can import series module from any location. This method is not robust since it adds modules only for current session. For each new session, path needs to be added again.



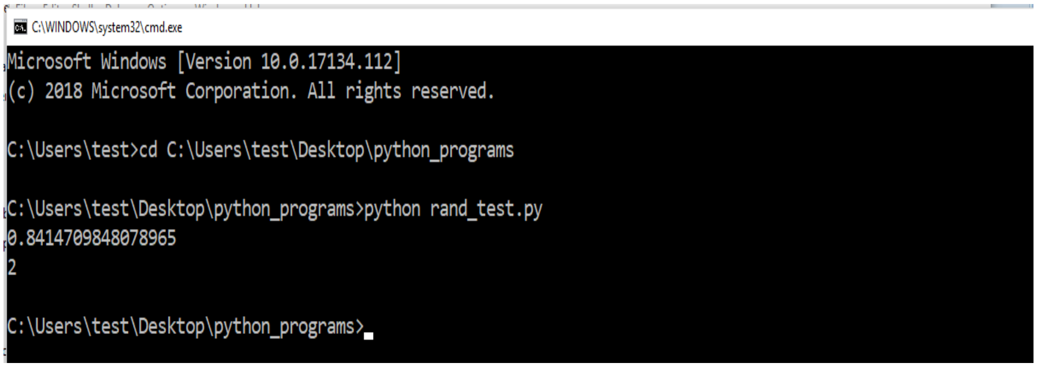
```
>>> import series
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import series
ModuleNotFoundError: No module named 'series'
>>> import sys
>>> sys.path.append('C:\\Users\\test\\Desktop\\python_programs')
>>> import series
>>> series.fibonacci(10)
FIBONACCI SERIES:
 0   1  1  2  3  5  8  13  21  34
>>> |
```

12.4 MODULE VS SCRIPT

For large number of instructions to be used together instead of directly running in shell or console, we used to write the code text files. These files can be modules or scripts. Extension of both the files is .py. Though there are several similarities, there are few differences as well.

SCRIPTS

Scripts are the files with sequence of instructions, which are executed each time the script is executed. There are various ways to execute a script, provided by different IDEs. It can also be executed in the console (shell in Unix/Linux and cmd in windows) using the command given below.



```
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\test>cd C:\Users\test\Desktop\python_programs

C:\Users\test\Desktop\python_programs>python rand_test.py
0.8414709848078965
2

C:\Users\test\Desktop\python_programs>_
```

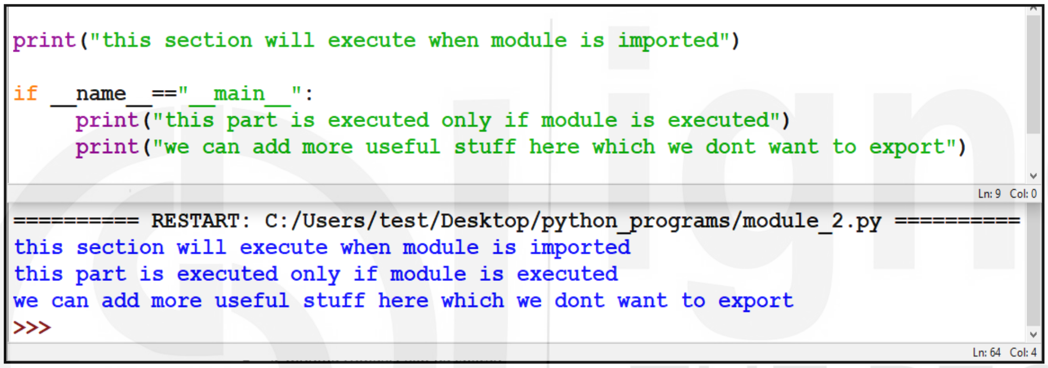
It should be noted that this command should be run in the directory where your python script exists otherwise *no file or directory exists* error will be shown.

MODULES

Functions which can be called from multiple scripts should be created within a module or we can say that a module is a file which is created for the purpose of importing. They are used to organize code in hierarchy. Module after creation should be added to search path.

When a module is imported, it runs the file from top to bottom. But when a module is executed, it runs the entire file and set the __name__ attribute to the value "__main__". This allows us to put a special code in a particular section which we want and we execute only when the module is executed directly. This section will not be executed during import.

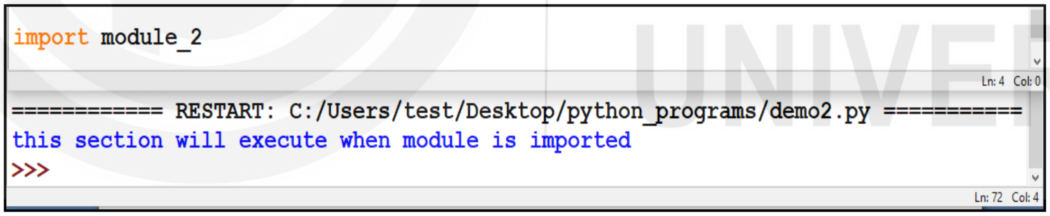
Here, a module named module_2.py is created. If this module is executed, output received is given in the below screenshot. The whole file will be executed.



```
print("this section will execute when module is imported")

if __name__=="__main__":
     print("this part is executed only if module is executed")
     print("we can add more useful stuff here which we dont want to export")
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/module_2.py ==========
this section will execute when module is imported
this part is executed only if module is executed
we can add more useful stuff here which we dont want to export
>>>
```

But when the above module is imported, the section under if __name__="__main__": will not be executed as show below.



```
import module_2
```

```
============ RESTART: C:/Users/test/Desktop/python_programs/demo2.py ===========
this section will execute when module is imported
>>>
```

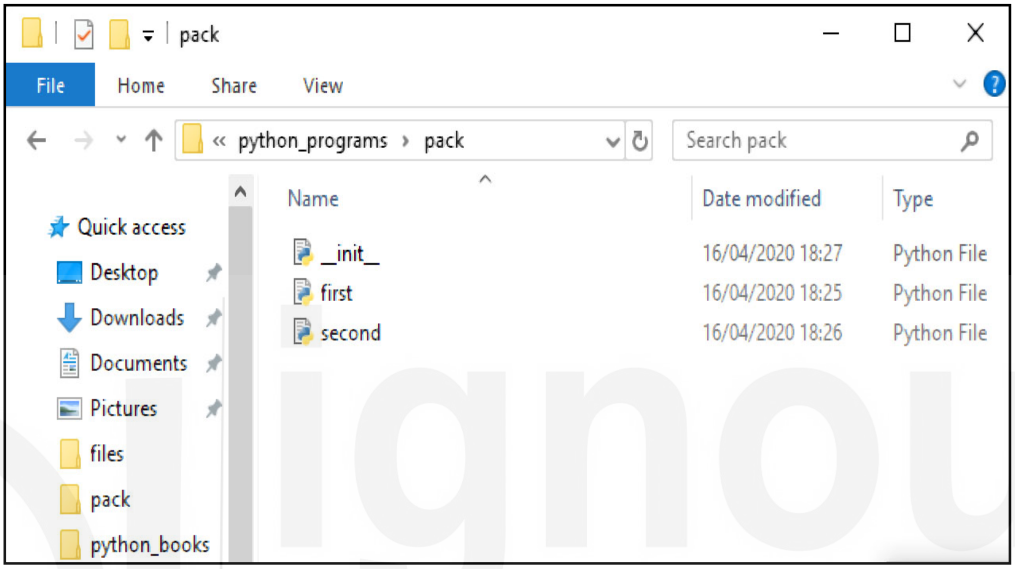
12.5 PACKAGE CREATION AND IMPORTING

Packages like modules are also used to organize the code in a better way. A package is a directory which contains multiple python modules. It is used to group multiple related python modules together. A python package in addition to modules must contain a file called __init__.py. This file may be empty or contains data like other modules of package.

File __init__.py

It is a file that makes the package importable. When a package is imported in a script, this file is automatically executed. It initializes variables, objects and makes the functions in the package accessible.

Let us create a package named*pack* and within this package create two modules *first.py* and *second.py* .



Name	Date modified	Type
__init__	16/04/2020 18:27	Python File
first	16/04/2020 18:25	Python File
second	16/04/2020 18:26	Python File

The __init__.py file created is empty. Module one contains function abc() and module two contains function xyz().

Packages can be imported in the same way as we import modules.

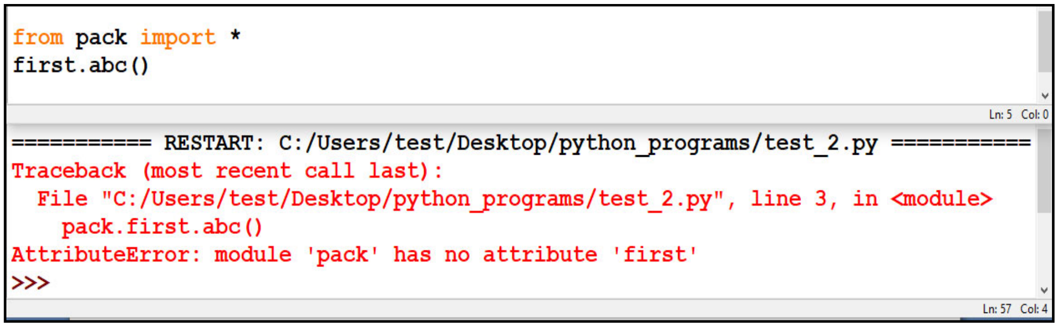
The various ways in which we can import from package are-

```
importpack.first
pack.first.abc()
```

```
from  pack  import first
first.abc()
```

```
from  pack.first  import abc
abc()
```

There are more methods to import. We have used * to import all the functions from a module in the previous section. This method can also be used here. But by default importing package modules using * will show error.



```
from pack import *
first.abc()
```

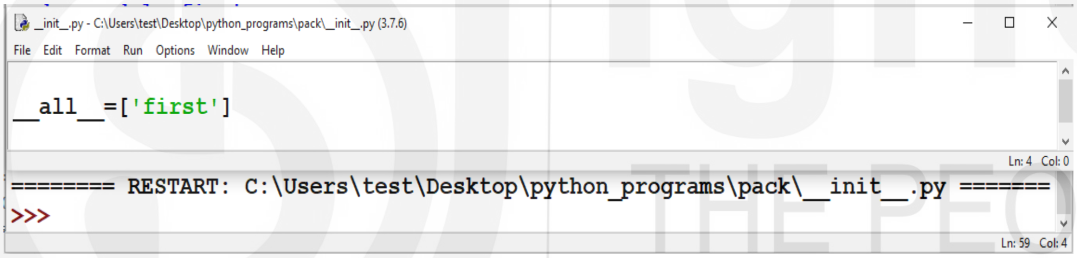
```
=========== RESTART: C:/Users/test/Desktop/python_programs/test_2.py ===========
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test_2.py", line 3, in <module>
    pack.first.abc()
AttributeError: module 'pack' has no attribute 'first'
>>>
```

This can be made possible using __all__ variable. This variable when added to __init__.py file, can make modules within package accessible outside using *from import* * statement.
Hence, we need to add __all__ statement in __init__.py

__all__= [‘ first ’]

The above statement makes module first.py accessible using *from import* * statement.

Adding statement __all__ to __init__.py file.



init.py - C:\Users\test\Desktop\python_programs\pack_init_.py (3.7.6)
File
Edit
Format
Run
Options
Window
Help

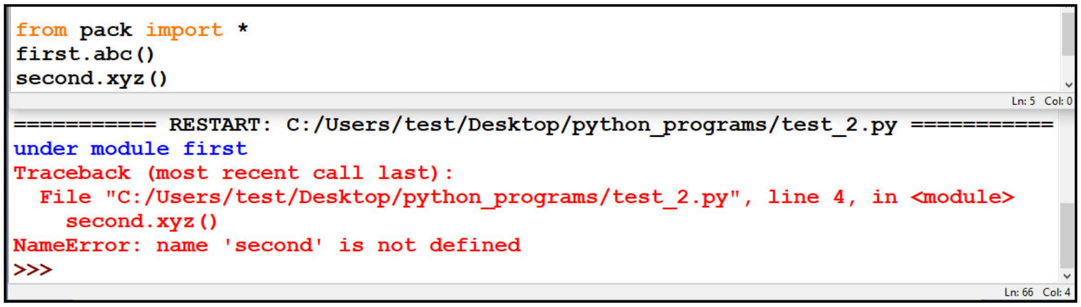
```
__all__=['first']
```

```
======== RESTART: C:\Users\test\Desktop\python_programs\pack\__init__.py ========
>>>
```

Now another way to import the module is given below:

Syntax to import using *from import* *

```
from  pack import *
first.abc()
```



```
from pack import *
first.abc()
second.xyz()
```

```
=========== RESTART: C:/Users/test/Desktop/python_programs/test_2.py ===========
under module first
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test_2.py", line 4, in <module>
    second.xyz()
NameError: name 'second' is not defined
>>>
```

Here, we can clearly see that first.py module is now accessible, since we have added it to __all__ variable. But second.py module is not accessiblesimultaneously, since it was not added to __all__ attribute in __init__.py.

Check your Progress 2

Ex. 1 What are packages ?How are they different from modules ?

Ex. 2 What is module search path ?How can we check it ?State the ways of adding a user defined module to search path.

Ex. 3 Create a package named Area and create 3 module in it named – square, circle and rectangle each having a function to calculate area of square, circle and rectangle respectively. Import the module in separate location and use the functions.

12.6 STANDARD LIBRARY MODULES

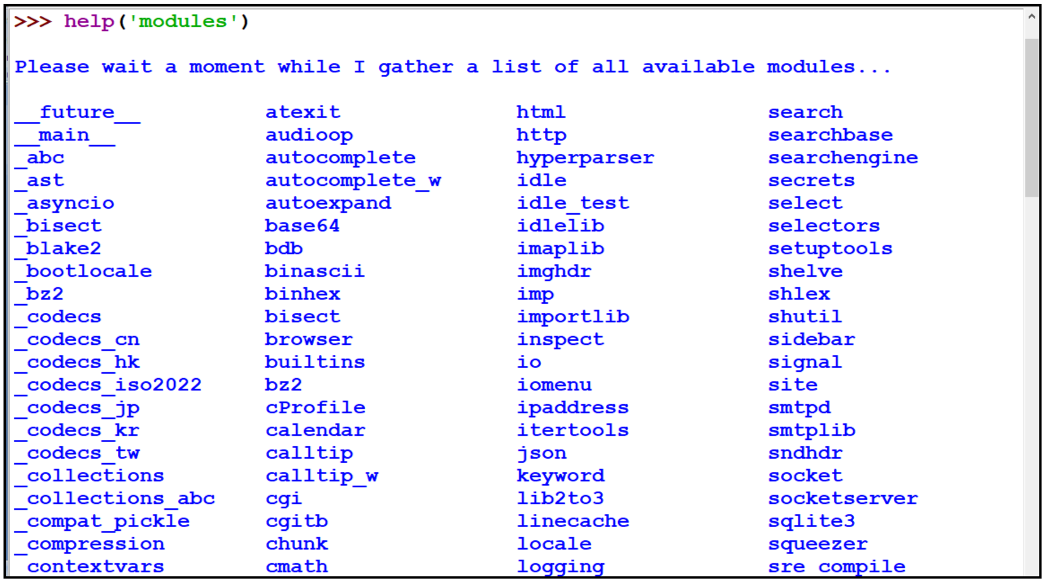
Python standard library provides number of built-in modules. They are automatically loaded when an interpreter starts. We have already used few of them in previous chapters. It should be noted that before using any module, it should be imported first. Some of the commonly used library modules are-

- sys
- os
- math
- random
- statistics

Module Attributes

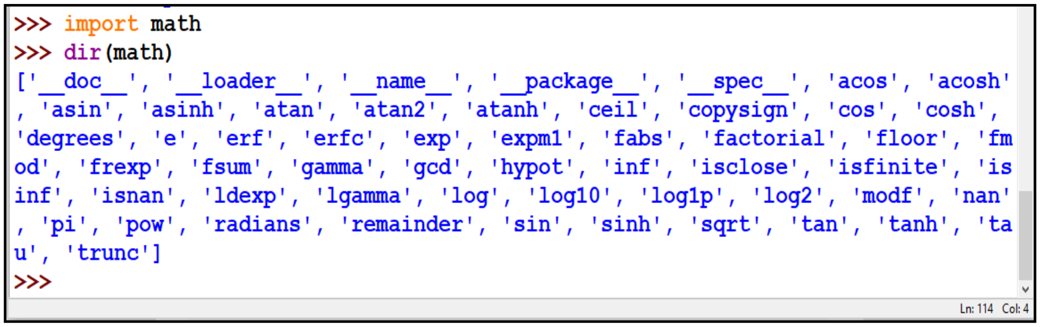
There are some attributes or functions that work for every module whether it is built-in library module or custom module. These attributes help in smooth operations of these modules. Some of them are explained below:

1. help () – it is a function used to display modules available for use in python or to get help on specific module.



```
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

__future__          atexit              html                search
__main__            audioop             http                searchbase
_abc                autocomplete        hyperparser         searchengine
_ast                autocomplete_w      idle                secrets
_asyncio            autoexpand          idle_test           select
_bisect             base64              idlelib             selectors
_blake2             bdb                 imaplib             setuptools
_bootlocale         binascii            imghdr              shelve
_bz2                binhex              imp                 shlex
_codecs             bisect              importlib           shutil
_codecs_cn          browser             inspect             sidebar
_codecs_hk          builtins            io                  signal
_codecs_iso2022     bz2                 iomenu              site
_codecs_jp          cProfile            ipaddress           smtpd
_codecs_kr          calendar            itertools           smtplib
_codecs_tw          calltip             json                sndhdr
_collections        calltip_w           keyword             socket
_collections_abc    cgi                 lib2to3             socketserver
_compat_pickle      cgitb               linecache           sqlite3
_compression        chunk               locale              squeezer
_contextvars        cmath               logging             sre_compile
```

2. dir ()- it is a function which is used to display objects or functions present in a specific module.Before using dir() function, module should be first imported.

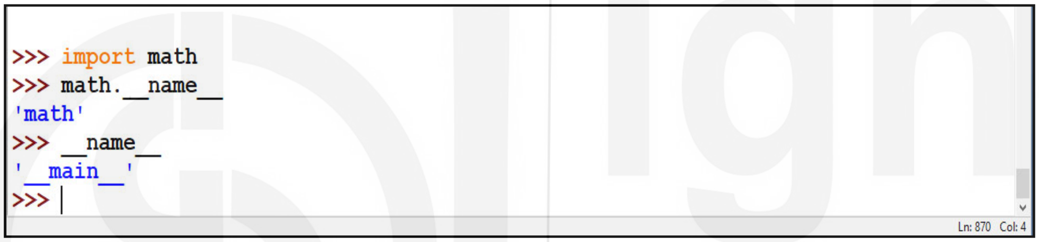


```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh'
, 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fm
od', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'is
inf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan'
, 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'ta
u', 'trunc']
>>>
```

3. __name__ attribute

This attribute returns name of the module. By default its value is the same as the name of the module.

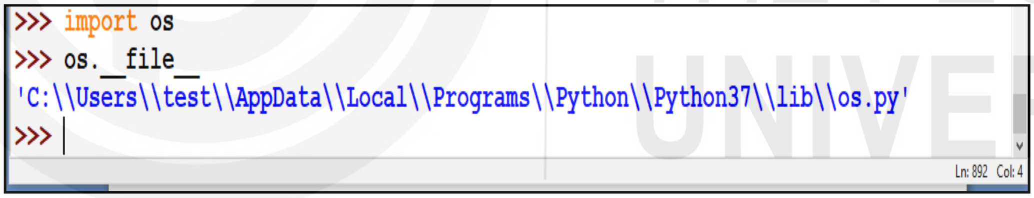
When a module or script is executed, its value becomes ‘__main__’. Also, when called without module name, it returns ‘__main__’.



```
>>> import math
>>> math.__name__
'math'
>>> __name__
'__main__'
>>> |
```

4. __file__ attribute

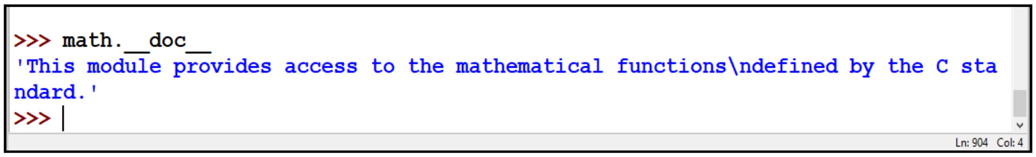
This attribute returns the location or path of the module.



```
>>> import os
>>> os.__file__
'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\lib\\os.py'
>>> |
```

5. __doc__ attribute

This attribute displays documentation given at the beginning of the module file.



```
>>> math.__doc__
'This module provides access to the mathematical functions\ndefined by the C sta
ndard.'
>>> |
```

OS MODULE

This is a module responsible for performing many operating system tasks. It provides functionality like- creating directory, removing directory, changing directory, etc.Some of the functions in os modules are given in the table below. It should be noted that before using these functions, the module should be imported.

importos

function	Description
os.mkdir("location")	Creates new directory in a given location.
os.rmdir("location")	Removes directory given by user. It should be taken care that current working directory cannot be removed and the directory to be removed should be empty.
os.getcwd()	Displays the current working directory.
os.chdir("location")	Changes current working directory to a given location.
os.listdir("location")	Displays list of files and directories in a given location. If location is not given, files of current directory will be displayed.

SYS MODULE

This module contains various variables and functions that can manipulate python runtime environment. Some of them are listed intable given below:

Function	Description
sys.path	Shows list of directories used to search python modules
sys.argv	Displays list of values passed as command line arguments to python program
sys.maxsize	Returns the largest integer value a variable can store
sys.version	Returns string representing python version
sys.getsizeof(object)	Returns size of an object in bytes
sys.exit	Used to exit from a program in case of exception

MATH MODULE

This module provides various mathematical functions and constant variables. It includes logarithmic, trigonometric functions etc. Some of the functions are listed in table below:

Function	Description
math.pow(x,y)	Returns x to the power y i.e. x**y
math.sqrt(x)	Returns square root of x i.e.$\sqrt{x}$
math.pi	Returns value of π
math.e	Returns value of e, Euler's number
math.radians(x)	Converts angle x from degree to radians
maths.degree(x)	Converts angle x from radians to degree
math.sin(x)	Returns sin() of angle x in radians
math.log(x)	Returns natural log of x
math.log10(x)	Returns log base 10 of x
math.floor(x)	Returns largest integer <=x
math.ceil(x)	Returns smallest integer >=x

STATISTICS MODULE

This module contains various functions used in statistics. These functions are widely used for data analysis or data science.

Function	Description
Statictics.mean(list)	Returns arithmetic mean of list or data given by user
Statistics.median(list)	Returns median value of list given by user
Statictics.mode(list)	Returns mode (highest frequency) value given by user
Statistics.stdev(list)	Returns standard deviation of list given by user
Statistics.variance(list)	Returns variance of list given by user

12.7 SUMMARY

In this unit, we have discussed modules and package creations in details. Modules are python files which can be imported in other files. A package is a folder which can store multiple modules and sub-packages within. Moreover, built-in modules are also discussed in details that add real power to python programming.

SOLUTION TO CHECK YOUR PROGRESS

check your Progress 1

Ex.1A**Module** is a logical group of functions , classes, variables in a single python file. A major benefit of a module is thatfunctions, variable or objects defined in one module can be easily used by other modules or files, which make the code re-usable.

A module can be created like any other python file i.e. with .py extension. Name of the module is the same as the name of a file.

Ex. 2 The various methods of importing a module are

1. using *import* statement
2. using *from import* statement
3. using *from import* * statement

1. import module
module.function()

This method is used to import the entire module. Individual functions can be used with module name and .symbol.

2. from module import function
function()

This method is used to import individual function from a module. In this method, function can be directly called with its name.

*3. From module import **
function()

This method can be used to import the entire module using from import * statement. Here, * represents all the functions of a module. Like previous method, an object can be accessed directly with its name.

Ex. 3 The 3 built-in modules in python are –os, math, random.

check your Progress 2

Ex. 1 A package is a directory which contains multiple python modules. It is used to group multiple related python modules together. A python package in addition to modules must contain a file called __init__.py. This file may be empty or may contain data like other modules of package.

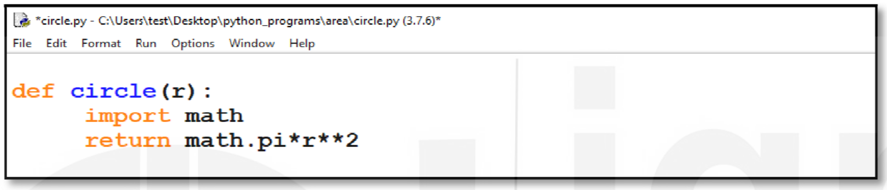
Ex. 2 When we use import statements to import a module, it is searched in a list of directories or search paths stored by the environment variable PYTHONPATH. This is called **module search path**. This list of directories can be checked using **sys.path** variable. Any modules created must be located in python's search path for its global identification.

Modules can be added to search path by either of the ways-

1. Creating module in one of the locations already present in search path
2. Adding module path in the search path using sys.path.
3. Updating PYTHONPATH environment variable.

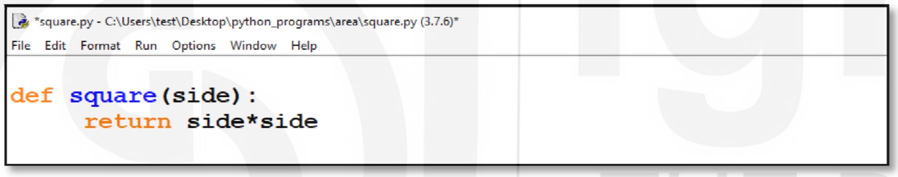
Ex. 3 First of all,ceate a folder named area and place 4 file named – circle.py, square.py, rectangle.py and __init__.py in folder.

circle.py


circle.py - C:\Users\test\Desktop\python_programs\area\circle.py (3.7.6)
File Edit Format Run Options Window Help

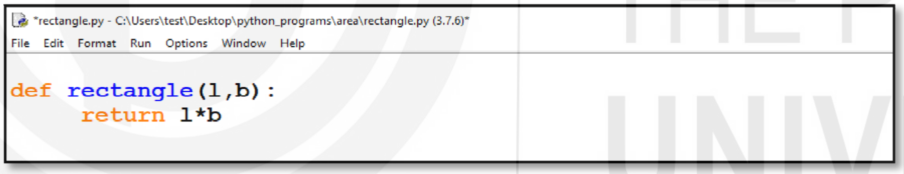
```
def circle(r):
     import math
     return math.pi*r**2
```

square.py


square.py - C:\Users\test\Desktop\python_programs\area\square.py (3.7.6)
File Edit Format Run Options Window Help

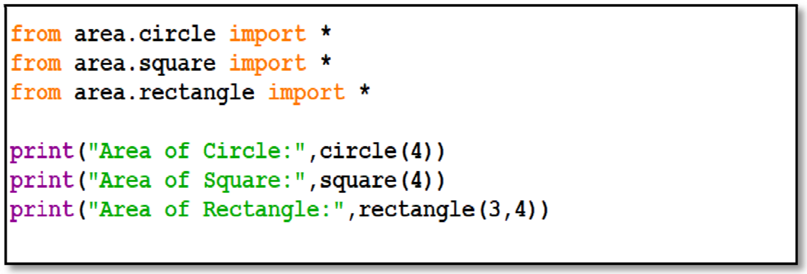
```
def square(side):
     return side*side
```

rectangle.py


rectangle.py - C:\Users\test\Desktop\python_programs\area\rectangle.py (3.7.6)
File Edit Format Run Options Window Help

```
def rectangle(l,b):
     return l*b
```

Now, we can import the package along with all the modules in any file.



```
from area.circle import *
from area.square import *
from area.rectangle import *

print("Area of Circle:",circle(4))
print("Area of Square:",square(4))
print("Area of Rectangle:",rectangle(3,4))
```