4

Using Python Libraries

In This Chapter

4.1 Introduction

4.2 What is a Library ?

4.3 Importing Modules in a Python Program

4.4 Using Python Standard Library's Functions and Modules

4.5 Creating a Python Library

4.1 INTRODUCTION

You all must have read and enjoyed a lot of books – story books, novels, course-books, reference books etc. And, if you recall, all these book types have one thing in common. Confused ? ☺ Don't be – all these book types are further divided into chapters. Can you tell, why is this done ? Yeah, you are right. Putting all the pages of one book or novel together, with no chapters, will make the book boring and difficult to comprehend. So, dividing a bigger unit into smaller manageable units is a good strategy.

Similarly, in programming, if we create smaller handleable units, these are called *modules*. A related term is *library* here. A library refers to a collection of modules that together cater to specific type of needs or applications *e.g.*, *NumPy* library of Python caters to scientific computing needs. In this chapter, we shall talk about using some Python libraries as well as creating own libraries/modules.

LIBRARY

A **library** refers to a collection of modules that together cater to specific type of needs or applications

4.2 WHAT IS A LIBRARY ?

As mentioned above that a library is a collection of modules (and packages) that together cater to a specific type of applications or requirements. A library can have multiple modules in it. This will become clearer to you when we talk about 'what modules are' in coming lines.[1]

Some commonly used Python libraries are as listed below :

(*i*) **Python standard library.** This library is distributed with Python that contains modules for various types of functionalities. Some commonly used modules of Python standard library are :

- **math module**, which provides mathematical functions to support different types of calculations.
- **cmath module**, which provides mathematical functions for complex numbers.
- **random module**, which provides functions for generating pseudo-random numbers.
- **statistics module**, which provides mathematical statistics functions.
- **Urllib module**, which provides URL handling functions so that you can access websites from within your program.

(*ii*) NumPy **library.** This library provides some advance math functionalities along with tools to create and manipulate numeric arrays.

(*iii*) SciPy **library.** This is another useful library that offers algorithmic and mathematical tools for scientific calculations.

(*iv*) **tkinter library.** This library provides traditional Python user interface toolkit and helps you to create userfriendly GUI interface for different types of applications.

(*v*) Malplotlib **library.** This library offers many functions and tools to produce quality output in variety of formats such as plots, charts, graphs etc.

A library can have multiple modules in it. Let us know what a module means.

4.2.1 What is a Module ?

The act of partitioning a program into individual components (known as *modules*) is called **modularity**. A module is a separate unit in itself. The justification for partitioning a program is that

- it reduces its complexity to some degree and
- it creates a number of well-defined, documented boundaries within the program.

Another useful feature of having modules, is that its contents can be reused in other programs, without having to rewrite or recreate them.

For example, if someone has created a module say **'PlayAudio'** to play different audio formats, coming from different sources, *e.g., mp3player, fm-radio player, dvd player* etc. Now, while writing a different program, if someone wants to incorporate ***fm-radio*** into it, he needs not re-write the code for it. Rather, he can use the *fm-radio functionality* from **PlayAudio** module. Isn't that amazing ? Re-usage without any re-work – well, that's the beauty of modules.

1. Package will become more clear to you in section 4.5.

4.2.1A Structure of a Python Module

A Python module can contain much more than just *functions*. A Python module is a normal Python file (*.py* file) containing *one* or *more* of the following objects related to a particular task :

- *docstrings* — triple quoted comments ; useful for documentation purposes. For documentation, the docstrings should be the first string stored inside a module/function-body/class.
- *variables and constants* — labels for data.
- *classes* — templates/blueprint to create objects of a certain kind.
- *objects* — instances of classes. In general, objects are representation of some real or abstract entity.
- *statements* — instructions.
- *functions* — named group of instructions.

So, we can say that the module '*XYZ*' means it is file '*XYZ. py*'.

Python comes loaded with some predefined modules that you can use and you can even create *your own modules*. The Python modules that come preloaded with Python are called ***standard library modules***. You have worked with one standard library module **math** earlier and in this chapter, you shall also learn to work with another standard library module : **random** module.

PYTHON MODULE

A **Python module** is a file (*.py* file) containing *variables, class definitions, statements* and *functions* related to a particular task.

Figure 4.1 shows the general composition / structure of a Python module.

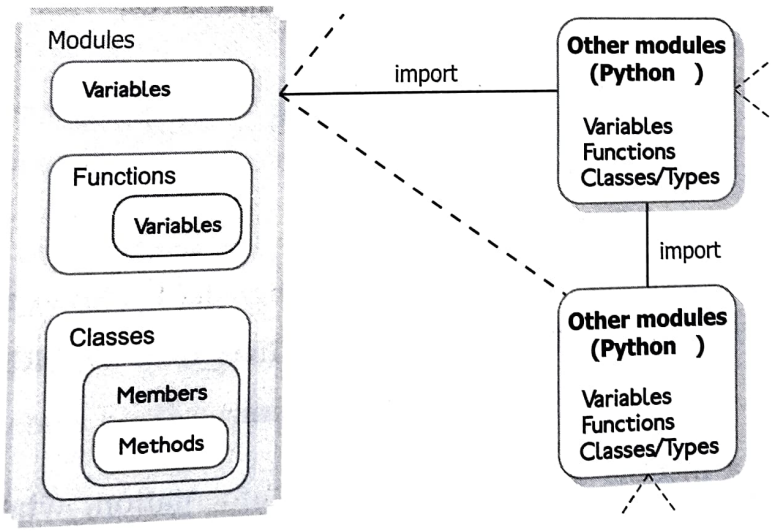


Figure 4.1 Composition/Structure of a Python Module

A module, in general :

- is independent grouping of code and data (variables, definitions, statements and functions).
- can be re-used in other programs.
- can depend on other modules.

Let's have a look at an example module, namely ***tempConversion*** in figure 4.2.

(Please note, it is not from Python's standard library ; it is a user created module, shown here for your understanding purposes).

```
# tempConversion.py
""" "Conversion functions between fahrenheit and centrigrade" "
```

A docstring ; There are two more docstrings in this module – one each inside each function

```
# Functions

def to_centigrade(x):
    """ "Returns: x converted to centigrade" "
    return 5 * (x – 32) / 9.0
```

Two function-definitions inside module tempConversion.py

STYLE-CONVENTION
Two blank lines between two function definitions

```
def to_fahrenheit(x):
    """ "Returns: x converted to fahrenheit" "
    return 9 * x / 5.0 + 32

# Constants

FREEZING_C = 0.0        # water freezing temp.(in celcius)
FREEZING_F = 32.0       # water freezing temp.(in fahrenheit)
```

Two constants defined

Figure 4.2 A sample module (tempConversion.py)

The elements of module shown in figure 4.2 are :

Name of the module	**tempConversion**		
Module file name	**tempConversion.py**		
Contains	*Two Functions*	(*i*) **to_centigrade( )**	(*ii*) **to_fahrenheit( )**
	Two Constants[2]	(*i*) **FREEZING_C**	(*ii*) **FREEZING_F**
	Three docstrings	(triple quotes strings)	

The docstrings of a module are displayed as documentation, when you issue following command on Python's Shell prompt >>>, after importing the module with **import <module-name>** command :

```
help(<module-name>)
```

For example, after importing module given in Fig. 4.2, *i.e.*, module **tempConversion**, if we write

```
help (tempConversion)
```

2. Please note, there is no separate entity like constants in Python ; it is just for understanding purposes to refer to something whose value the programmer wants to keep fixed throughout the program.

Python will display all *docstrings* along with module name, filename, functions' name and constant as shown below :

```
>>> import temConversion
>>> help(tempConversion)
Help on module tempConversion :

NAME
    tempConversion – Conversion functions between fahrenheit and centigrade

FILE
    c:\python37\pythonwork\tempconversion.py

FUNCTIONS
    to_centigrade(x)
        Returns : x converted to centigrade
    to_fahrenheit(x)
        Returns : x converted to fahrenheit

DATA
    FREEZING_C = 0.0
    FREEZING_F = 32.0
```

The docstrings of module displayed as documentation

NOTE

The docstrings are triple quoted strings in a Python module/program which are displayed as document when *help (<module-or-program-name>)* command is issued.

There is one more function **dir( )** when applied to a module, gives you names of all that is defined inside the module. (see below)

```
>>> import tempConversion
>>> dir(tempConversion)

['FREEZING_C', 'FREEZING_F', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', 'to_centigrade', 'to_fahrenheit' ]
```

General docString conventions are :

- First letter of first line is a capital letter.
- Second line is blank line.
- Rest of the details begin from third line.

4.3 IMPORTING MODULES IN A PYTHON PROGRAM

As mentioned before, in Python if you want to use the definitions inside a module, then you need to first import the module in your program. Python provides **import** statement to import modules in a program. The **import** statement can be used in *two* forms :

(*i*) to import entire module : the import <module> command

(*ii*) to import selected objects from a module : the from <module> import<object> command

Following subsections will make the utility of both these *import* commands clear.

4.3.1 Importing Entire Module

The **import** statement can be used to import entire module and even for importing selected items. To **import** entire module, the import statement can be used as per following syntax : (*please remember*, in syntax, [ ] specify optional elements.)

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

For example, to import a module, say time, you'll write :

```
import time
```

Module namely time being imported

To import *two* modules namely decimals and fractions, you'll write :

```
import decimals, fractions
```

Two modules namely decimals and fractions being imported with one import statement.

The **import** statement internally executes the code of module file and then makes it available to your program. The **import** statement like the one shown above, **imports entire module** *i.e.*, everything defined inside the module – *function definitions, variables, constants* etc.

After importing a module, you can use any function/ definition of the imported module as per following syntax :

```
<module-name>.<function-name>()
```

This way of referring to a module's object is called *dot notation*.

For example, consider the module **tempConversion** given in figure 4.2. To use its function **to_centigrade( )**, we'll be writing :

```
import tempConversion
tempConversion.to_centigrade(98.6)
```

calling function to_centrigrade( ) of imported module tempConversion

NOTE

After importing a module, to access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

The name of a module is stored inside a constant **__name__** (prefix & suffix are having two underscores). You can use it like :

```
import time
print(time.__name__)
```

It will print the name of imported module (see below)

```
>>> import time
>>> print time.__name__
time
>>> |
```

NOTE

The name of a module is stored inside a constant **__name__**.[3]

You can given alias name to imported module as :

```
import<module> as <aliasname>
```

e.g.,

```
import tempConversion as tc
```

Now you can use name **tc** for the imported module *e.g.*, **tc.to_centigrade( )**.

Please note, if in a module there is another **import** statement importing an already imported module (from same origin), Python will ignore that import statement. Thus, a module once imported will not be re-imported even another **import** statement for the same module is encountered again.

3. For additional utility of __name__ , refer to Appendix A.

4.3.2 Importing Select Objects from a Module

If you want to import some selected items, not all from a module, then you can use **from <module> import** statement as per following syntax :

```
from <module> import <objectname> [,<objectname> [...]]*
```

To Import Single Object

If you want to import a single object from the module like this so that you don't have to prefix the module's name, you can write the name of object after keyword **import.** For instance, to import just the constant ***pi*** from module **math**, you can write :

```
from math import pi
```

Now, the constant ***pi*** can be used and you need not prefix it with *module name*. That is, to print the value of ***pi***, after importing like above, you'll be writing

```
print(pi)
```

*After **'from <module> import'** command you need not qualify the name of imported item with module name like this*

Not this

```
print(math.pi)
```

Do not use module name with imported object if imported through from <module> import command

Do not use module name with imported object if imported through **from <module> import** command because now the imported object is part of your program's environment.

To Import Multiple Objects

If you want to import multiple objects from the module like this so that you don't have to prefix the module's name, you can write the comma separated list of objects after keyword **import.** For instance, to import just *two* functions *sqrt( )* and *pow( )* from math module, you'll write :

```
from math import sqrt, pow
```

To Import All Objects of a Module

If you want to import all the items from the module like this so that you don't have to prefix the module's name, you can write :

```
from <modulename> import *
```

That is, to import all the items from module **math**, you can write :

```
from math import *
```

Now you can use all the defined functions, variables etc from *math* module, without having to prefix module's name to the imported item name.

4.3.3 Python's Processing of import <module> Command

With every import command issued, Python internally does a series of steps. In this section, we are briefly discussing the same. Before we discuss the internal processing of import statements, let's first discuss namespace – an important and related concept, which plays a role in internal processing of import statement.

Namespace

A namespace, in general, is a space that holds a bunch of names. Before we go on to explain namespaces in Python terms, consider this real life example.

> In an interstate student seminar, there are students from different states having similar names. Say there are *three* **Nalinis**, one from ***Kerala***, one from ***Delhi*** and one from ***Odisha***.
>
> As long they are staying in their state's ward, there is no confusion. Since in ***Delhi***, there is one *Nalini*, calling name Nalini would automatically refer to **Delhi's** student **Nalini**. Same applies to ***Kerala*** and ***Odisha*** wards separately.
>
> But the problem arises when the students from Delhi, Kerala and Odisha states are sitting together. Now calling just Nalini would create confusion – which state's Nalini ? So, one needs to qualify the name as **Odisha's Nalini** or **Kerala's Nalini** and so on.

From the above real-life example, we can say that Kerala ward has its own namespace where there no two names as Nalini ; same holds for Delhi and Odisha.

In Python terms, namespace can be thought of as a named environment holding logical grouping of related objects. You can think of it as named list of some names.

For every module (.py file), Python creates a namespace having its name similar to that of module's name. That is, the name of ***module time's namespace is*** also **time.**

NAMESPACE

Namespace is a named logical environment holding logical grouping of related objects within a namespace, its member object is referred without any prefix.

When two namespaces come together, to resolve any kind of object-name dispute, Python asks you to qualify the names of objects as **<module-name>.<object-name>**, just like in real life example, we did by calling Delhi's Nalini or Odisha's Nalini and so on. *Within a namespace, an object is referred without any prefix.*

Processing of import <module> command

When you issue **import <module>** command, internally following things take place :

- the code of imported module is interpreted and executed[4].
- defined functions and variables created in the module are now available to the program that imported module.
- For imported module, a new ***namespace*** is setup with the same name as that of the module.

For instance, you imported module **myMod** in your program. Now all the objects of module **myMod** would be referred as **myMod.<object-name>**, *e.g.*, if **myMod** has a function defined as **checknum( )**, then it would be referred to as **myMod.checknum( )** in your program.

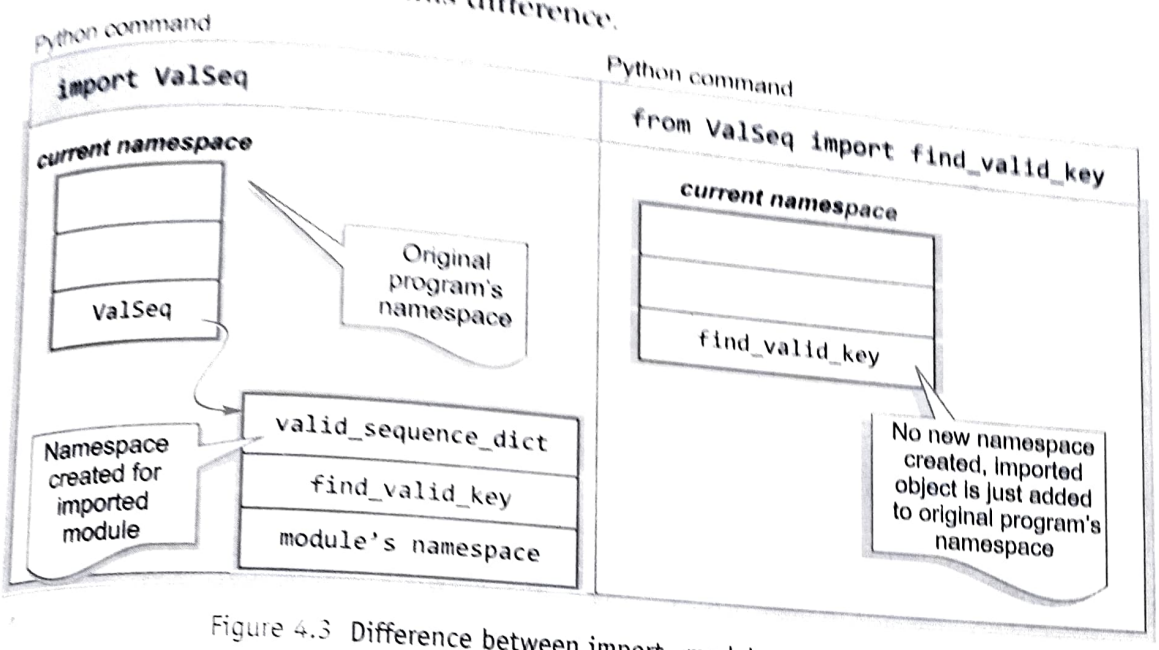
Processing of from <module> import <object> command

When you issue **from <module> import <object>** command, internally following things take place :

- the code of imported module is interpreted and executed[4].
- only the asked functions and variables from the module are made available to the program.
- no new *namespace* is created, the imported definition is just added in the current *namespace*.

4. Refer to *Appendix A* to stop execution of module's main block while importing.

Figure 4.3 illustrates this difference.



To see
Two ways of Import in action

Scan QR Code

Figure 4.3 Difference between import <module> and from <module> import commands.

- That means if your program already has a variable with the same name as the one imported via module, then *the variable in your program* will hide imported member with same name because there cannot be two variables with the same name in one *namespace*.

Following code fragment illustrates this.

Let's consider the module given in Fig 4.2

```
from tempConversion import *

FREEZING_C = -17.5      # it will hide FREEZING_C of tempConversion module
print(FREEZING_C)
```

The above code will print

```
-17.5
```

If you change above code to the following (we made FREEZING_C = – 17.5 as a COMMENT, see below :)

```
from tempConversion import *

# FREEZING_C = -17.5        # it is just a comment now
print(FREEZING_C)
```

Now the above code will give result as :

```
0.0
```

as no variable from the program shares its name, hence it is not hidden.

NOTE

Avoid using the **from <module> import *** form of the import statement, it may lead to name clashes. If you use plain **import <module>**, no problems occur.

Sometimes a module is stored inside another module. Such a submodule can also be imported as :

```
from <parent module> import<submodule> [as <alias name>]
```

e.g.,

```
from products import views
from products import views as PV
```

*this is alias name for **product.views** submodule*

CREATING AND USING MODULES — Progress In Python 4.1

This Prip session is based on practice for creating and using modules.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 4.1 under Chapter 4 after practically doing it on the computer.

>>>❖<<<

4.4 USING PYTHON STANDARD LIBRARY'S FUNCTIONS AND MODULES

Python's standard library is very extensive that offers many built-in functions that you can use without having to import any library. Python's standard library is by default available, so you don't need to import it separately.

Python's standard library also offers, other that the built-in functions, some modules for specialized type of functionality, such as ***math*** module for mathematical functions ; ***random*** module for pseudo-random number generation ; ***urllib*** for functionality for adding and using web sites' address from within program ; etc.

In this section, we shall discuss how you can use Python's built-in functions and import and use various modules and Python's standard library.

4.4.1 Using Python' Built-in Functions

The Python interpreter has a number of functions built into it that are always available ; you need not import any module for them. In other words, the built in functions are part of current namespace of Python interpreter. So you use built-in functions of Python directly as :

```
<function-name>()
```

For example, the functions that you have worked with uptill now such as *input( ), int( ), float( ), type( ), len( )* etc. are all built in functions, that is why you never prefixed them with any module name.

4.4.1A Python's built-in Mathematical Functions

Python provides many mathematical built-in functions. You have worked with many built-in mathematical functions of Python in your previous class. Let us quickly recall those and then we shall talk about some more such functions.

Function name	Description
len( )	Returns the length of a sequence or iterable *e.g.*, len("abc") gives 3.
pow( )	Returns a^b when a and b are given as arguments, *e.g.*, pow(3, 4) gives 81.
str( )	Converts a number to a string, *e.g.*, str(12) will give '12' and str(12.4) will give '12.4'.
int( )	Converts an integer-convertible string to integer, *e.g.*, int('12') will give 12.
float( )	Converts a float-convertible string to integer, *e.g.*, float('12.2') will give 12.2.
range( )	Returns an immutable sequence type, *e.g.*, range(3) will give sequence 0, 1, 2.
type( )	Returns the data type of passed argument, *e.g.*, type(12) will give <class 'int'>.

Let us now talk about some more built-in mathematical functions.

Table 4.1 Some useful built-in mathematical functions

Function name	Description	Examples

abs(x)¶	Takes an integer or a floating point number as argument and returns the absolute value of a number.	>>> abs(-12)
		12
		>>> abs(-12.4)
		12.4
		>>> abs(12.4)
		12.4

divmod(a, b)	Takes two (non-complex) numbers as arguments and returns a pair of numbers consisting of their quotient and remainder.
	For integers, the result is the same as (*a* // *b*, *a* % *b*). For floating point numbers the result is (*q*, *a* % *b*)
		>>> divmod(7, 2)
		(3, 1)
		Pair of quotient and remainder returned
		>>> divmod(7.25, 2.5)
		(2.0, 2.25)
		>>>

sum(iterable)
sum(iterable,arg)
	Returns sum of the items of an ***iterable*** from left to right and returns the total.
	With two arguments iterable and arg, it returns the sum of the items of an ***iterable*** and the ***arg's*** value.
	The ***iterable's*** items are normally numbers, and the ***arg*** value should be a number.
	(Recall that all sequence types are iterable.)
		>>> sum([2, 3, 4])
		9
		Sum of the elements of the iterable returned
		>>> sum((2.5, 6, 4.2))
		12.7
		>>> sum([2, 3, 4], 5)
		14
		Sum of the elements of the iterable along with the argument value returned
		>>> sum([2, 3, 4], 8.3)
		17.3

max(iterable)
max(arg1, arg2,....)
	Returns the largest item in an iterable / sequence or the largest of two or more arguments.
	If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned.
		>>> max(3, 5)
		5
		Maximum of two integer values returned
		>>> max([3, 5, 9,], [7])
		[7]
		Maximum of two list arguments returned – list that begins with a higher value is returned
		>>> max((3, 5, 9, 10), (7,))
		(7,)
		Maximum of two tuple arguments returned – tuple that begins with a higher value is returned
		>>> max((3, 5, 9, 10))
		10
		Maximum of one iterable argument returned – highest element from the tuple iterable
		>>> max([13, 15, 27, 10])
		27
		Maximum of one iterable argument returned – highest element from the list iterable

Function name	Description	Examples

min(iterable)
min(arg1, arg2,....)
	Returns the smallest item in an iterable / sequence or the smallest of two or more arguments.
	If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned.
		>>> min(3, 5)
		3 *Minimum of two integer values returned*
		>>> min([3, 5, 9,], [7])
		[3, 5, 9] *Minimum of two list arguments returned – list that begins with a higher value is returned*
		>>> min((3, 5, 9, 10), (7,))
		(3, 5, 9, 10) *Minimum of two tuple arguments returned – tuple that begins with a higher value is returned*
		>>> min([13, 15, 27, 10])
		10 *Minimum of one iterable argument returned – smallest element from the tuple iterable*
		>>> min([13, 15, 27, 10])
		10 *Minimum of one iterable argument returned – smallest element from the list iterable*

oct(<integer>)
hex(<integer>)
	returns octal string for given numbers *i.e.*, 00 + octal equivalent of number.
	returns hex string for given numbers *i.e.*, 0x + hexadecimal equivalent of number.
	Please note that *oct( ), hex( )* and *bin( )* do not return a number ; they return a string representation of converted number.
		>>> n = 24
		>>> oct(n)
		'0o30'
		>>> hex(n)
		'0x18'

Consider following program that uses two more built-in functions :

- **int (<number>)**[5] — truncates the fractional part of given number and returns only the integer or whole part.
- **round(<number>, [<ndigits>])** — returns number rounded to *ndigits* after the decimal points. If *ndigits* is not given, it returns nearest integer to its input.

Program 4.1 Write a program that inputs a real number and converts it to nearest integer using two different built-in functions. It also displays the given number rounded off to 3 places after decimal.

```
num = float(input("Enter a real number:"))
tnum = int(num)
rnum = round(num)
print("Number", num, "converted to integer in 2 ways as", tnum, "and", rnum )
rnum2 = round(num, 3)
print(num, "rounded off to 3 places after decimal is", rnum2)
```

5. The int( ) can also convert a number string into an equivalent number *e.g.*, "1235" can be converted to number 1235 using int("1235") ; works with integer strings only.

Sample run of above program is :

```
Enter a real number: 5.555682
Number 5.555682 converted to integer in 2 ways as 5 and 6
5.555682 rounded off to 3 places after decimal is 5.556
```

**The behaviour of round( ) can be surprising for floats, e.g., round(0.5) and round(–0.5) are 0, and round(1.5) is 2.

4.4.1B Python's built-in String Functions

Let us now use some string functions. Although you have worked with many string functions in your previous class, let us use three new string based functions. These are :

- **<Str>.join(<string iterable>)** – joins a string or character (*i.e.*, <str>) after each member of the *string iterator i.e., a string based sequence.*
- **<Str>.split(<string /char>)** – splits a string (*i.e.*, <str>) based on given string or character (*i.e.*, <string/char>) and returns a list containing split strings as members.
- **<Str>.replace(<word to be replaced>, <replace word>)** – replaces a word or part of the string with another in the given string <str>.

Let us understand and use these string functions practically. Carefully go through the examples of these as given below :

<str>.join( )

(*i*) If the string **based** iterator is a string then the <str> is inserted after every character of the string, *e.g.*,

```
>>>"*".join("Hello")
'H*e*l*l*o'
```

See, a character is joined with each member of the given string to form the new string

```
>>>"***".join("TRIAL")
'T***R***I***A***L'
```

*See, a string("***" here) is joined with each member of the given string to form the new string*

(*ii*) If the string based iterator is a *list* or *tuple* of strings then, the given string/character is joined with each member of the list or tuple, BUT the tuple or list must have all member as strings otherwise Python will raise an error.

```
>>>"$$".join(["trial", "hello"])
Out[7]: 'trial$$hello'
```

Given string ("$$") joined between the individual items of a string based list

```
>>>"###".join(("trial", "hello", "new"))
Out[8]: 'trial###hello###new'
```

Given string ("$$") joined between the individual items of a string based tuple

```
>>>"###".join((123, "hello", "new"))
Traceback (most recent call last):
  File "<ipython-input-11-a0be3b94faec>", line 1, in <module>
    "###".join((123, "hello", "new"))
TypeError: sequence item 0: expected str instance, int found
```

The sequence must contain all strings, else Python will raise an error.

`<str>.split( )`

(*i*) If you do not provide any argument to split then by default it will split the given string considering whitespace as a separator, *e.g.*,

```
>>>"I Love Python".split()
['I', 'Love', 'Python']

>>>"I Love Python".split(" ")
['I', 'Love', 'Python']
```

With or without whitespace, the output is just the same i.e., the list containing individual words

(*ii*) If you provide a string or a character as an argument to split( ), then the given string is divided into parts considering the given string/character as separator and separator character is not included in the split strings *e.g.*,

```
>>>"I Love Python".split("o")
['I L', 've Pyth', 'n']
```

The given string is divided from positions containing "o"

`<str>.replace( )`

```
In[ ]:"I Love Python".replace("Python", "Programming")
'I Love Programming'
```

Word 'Python' has been replaced with 'Programming' in given string

Program 4.2 Write a program that inputs a main string and then creates an encrypted string by embedding a short symbol based string after each character. The program should also be able to produce the decrypted string from encrypted string.

```
def encrypt(sttr, enkey):
    return enkey.join(sttr)
def decrypt(sttr, enkey):
    return sttr.split(enkey)
#–main–
mainString = input("Enter main string :")
encryptStr = input("Enter encryption key :")
enStr = encrypt(mainString, encryptStr)
deLst = decrypt(enStr, encryptStr)
# deLst is in the form of a list, converting it to string below
deStr="".join(deLst)
print("The encrypted string is", enStr)
print("String after decryption is :", deStr)
```

The sample run of the above program is as shown below :

```
Enter main string : My main string
Enter encryption key : @$
The encrypted string is M@$y@$ @$m@$a@$i@$n@$ @$s@$t@$r@$i@$n@$g
String after decryption is :  My main string
```

Clearing Variables from Previous Program Run

When you are testing your code or program, you need to run it multiple times with various types of inputs etc. By default, IPython maintains the variables created by previous run of your program, which may hamper your testing. For instance, if you have changed a variable's name in one line of the code and a later line is still using the earlier name; then chances are that this may go unnoticed if you have run the program earlier. The reason behind this is that the variables created by previous run of your program still reside in memory and IPython used its value from memory and did not report to you.

The best way to avoid this is that you clear all the variables in memory before running your program. For this you can do one of the following *two* things :

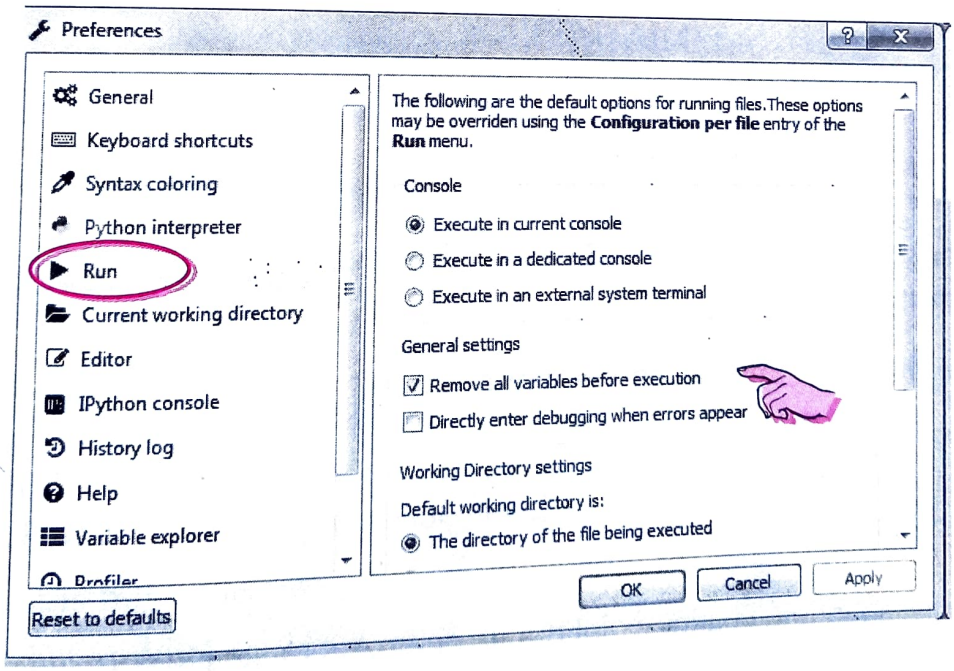
(*i*) In the IPython console, run **%reset** magic command :

```
In [6]: %reset

Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

The command **%reset** clears everything from memory – it's like you have restarted the shell.

(*ii*) Set preferences for **Spyder** by running command **Tools** → **Preference** → **Run** → (*under* ***General Settings***) **Remove all variables before execution** (see below)



You have already learnt to use math module in previous class. First chapter's section 1.8.5 also revises the same. So let us use some other useful modules of Python (as recommended by practical suggestions of the syllabus).

4.4.2 Working with Some Standard Library Modulus

Other than built-in functions, standard library also provides some modules having functionality for specialized actions. Let us learn to use some such modules. In the following lines we shall talk about how to use some useful functions of *random* and *string* modules[6] of Python's standard library.

4.4.2A Using Random Module

Python has a module namely **random** that provides random-number[7] generators. A random number in simple words means – *a number generated by chance, i.e.,* randomly.

To use random number generators in your Python program, you first need to import module ***random*** using any import command, *e.g.,*

```
import random
```

Some most common random number generator functions in ***random module*** are :

random( )	it returns a random floating point number N in the range [0.0, 1.0), *i.e.,* $\mathbf{0.0 \le N < 1.0}$. Notice that the number generated with *random( )* will always be less than 1.0. (only lower range-limit is inclusive). Remember, it generates a floating point number.
randint(*a, b*)	it returns a random integer N in the range (a, b), *i.e.,* $a \le N \le b$ (both range-limits are inclusive). Remember, it generates an integer.
random.uniform(*a, b*)	it returns a random floating point number N such that a <= N <= b for a <= b and b <= N <= a for b < a.
random.randrange(stop) random.randrange(start, stop[, step])	it returns a randomly selected element from range(start, stop, step).

Let us consider some examples. In the following lines we are giving some sample codes along with their output.

1. To generate a random floating-point number between 0.0 to 1.0, simply use **random()**:

```
>>> import random
>>> print(random.random())
0.022353193431
```

The output generated is between range [0.0, 1.0)

2. To generate a random floating-point number between range ***lower to upper*** using **random()**:

   (*a*) multiply **random( )** with difference of upper limit with lower limit, *i.e.,* (upper – lower)

   (*b*) add to it lower limit

6. Please note that learning how to use modules is important for learning how to use libraries.
7. In fact, pseudo-random numbers because it is generated via some algorithm or procedure and hence deterministic somewhere.

For example, to generate between 15 to 35 , you may write :

```
>>> import random        # need not re-write this command, if random
                         # module already imported
>>> print(random.random()* (35 -15 ) + 15)
28.3071872734
```

The output generated is floating point number between range 15 to 35

3. To generate a random integer number in range 15 to 35 using *randint( )*, write :

```
>>> print(random.randint(15, 35))
16
```

The output generated is integer between range 15 to 35

4. To generate a floating point random number in the ranges 11...55 or 111...55, after importing random module using import statement, you may write :

```
>>> random.uniform(11, 55)
41.34518981317353
>>> random.uniform(111, 55)
60.03906551659219
```

5. To generate a random integer in the ranges 23..47 with a step 3 or 0..235, after importing random module using import statement, you may write :

```
>>> random.randrange(23, 47, 3)
38
>>> random.randrange(235)
126
```

EXAMPLE 4.1 *Given the following Python code, which is repeated four times. What could be the possible set of outputs out of given four sets (dddd represent any combination of digits) ?*

```
import random
print(15 + random.random() * 5)
```

(*i*) 17.dddd, 19.dddd, 20.dddd, 15.dddd
(*ii*) 15.dddd, 17.dddd. 19.dddd, 18.dddd
(*iii*) 14.dddd, 16.dddd, 18.dddd, 20.dddd
(*iv*) 15.dddd, 15.dddd, 15.dddd, 15.dddd

Solution. Option (*ii*) and (*iv*) are the correct possible outputs because :

(*a*) *random( )* generates number N between range $0.0 <= N < 1.0$.
(*b*) when it is multiplied with 5, the range becomes 0.0 to < 5
(*c*) when 15 is added to it, the range becomes 15 to < 20

Only option (*ii*) and (*iv*) fulfill the condition of range 15 to < 20.

EXAMPLE 4.2 *What could be the minimum possible and maximum possible numbers by following code ?*

```
import random
print(random.randint(3, 10) - 3)
```

Solution. minimum possible number = 0
maximum possible number = 7

Because,

- randint(3, 10) would generate a random integer in the range 3 to 10
- subtracting 3 from it would change the range to 0 to 7 (because if ***randint(3,10)*** generates 10 then –3 would make it 7 ; similarly, for lowest generated value 3, it will make it 0)

EXAMPLE 4.3 *In a school fest, three randomly chosen students out of 100 students (having roll numbers 1-100) have to present bouquets to the guests. Help the school authorities choose three students randomly.*

Solution.

```
import random
student1 = random.randint(1, 100)
student2 = random.randint(1, 100)
student3 = random.randint(1, 100)
print("3 chosen students are",)
print(student1, student2, student3)
```

4.4.2B Using String Module

Python has a module by the name **string** that comes with many constants and classes. It also offers a utility function **capwords( )**. Let us talk about some useful constants defined in the string module.

Please note that like other modules, before you can use any of the constants/functions defined in the **string** module, you must import it using an import statement :

```
import string
```

Some useful **constants** defined in the string module are being listed below :

string.ascii_letters	it returns a string containing all the collection of ASCII letters.
string.ascii_lowercase	it returns a string containing all the lowercase ASCII letters, *i.e.*, 'abcdefghijklmnopqrstuvwxyz'.
string.ascii_uppercase	it returns all the uppercase ASCII letters, *i.e.*, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
string.digits	it returns a string containing all the digits Python allows, *i.e.*, the string '0123456789'.
string.hexdigits	it returns a string containing all the hexadecimal digits Python allows, *i.e.*, the string '0123456789abcdefABCDEF'.
string.octdigits	it returns a string containing all the octal digits Python allows, *i.e.*, the string '01234567'.
string.punctuation	it returns a string of ASCII characters which are considered punctuation characters, *i.e.*, the string '!"#$%&'()*+,-./:;?@[\]^_'{|}~'

The string module also offers a utility function capwords( ) :

capwords(<str>, [sep=None])	it splits the specified string *<Str>* into words using *<Str>.split( )*. Then it capitalizes each word using *<Str>.capitalize( )* function. Finally, it joins the capitalized words using *<Str>.join( )*. If the optional second argument *sep* is absent or is *None*, it will remove leading and trailing whitespaces and all inside whitespace characters are replaced by a single space.

You can obtain the value of the constants defined in **string** module by simply giving their name with the string module name after importing string module, *e.g.*,

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.punctuation
'!"#$%&\'()*+,-./:;?@[\\]^_'{|}~'
```

You can use ***capwords( )*** using the string module name and passing the string name as its argument, *e.g.*,

```
>>> import string
>>> line = "this is a simple line\n New line"
>>> string.capwords(line)
'This Is A Simple Line New Line'
```

See it has capitalized each word separately and all leading, trailing whitespaces have been removed and inside whitespace characters (eg '\n' and spaces) have been replaced with a single space.

WORKING WITH math AND random MODULES

PriP — Progress In Python 4.2

This PriP session is based on using Python Standard Library Modules – *math* and *random*.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 4.2 under Chapter 4 after practically doing it on the computer.

>>>❖<<<

4.5 CREATING A PYTHON LIBRARY

Other than standard library, there are numerous libraries available which you can install and use in your programs. Some such libraries are *NumPy, SciPy, tkinter* etc.

One of these libraries, *NumPy*, has been covered briefly in chapter 8. **Appendix B** discusses another Python library – **tkinter**. You can also create your own libraries.

You have been using terms *modules* and *libraries* so far. Let us talk about a related word, **package**. In fact, most of the times **library** and **package** terms are used interchangeably.

A **package** is collection of Python modules under a common namespace, created by placing different modules on a single directory along with some special files (such as __init__.py)

In a directory structure, in order for a folder (containing different modules *i.e.*, .py files) to be recognized as a package, a special file namely *__init__.py* must also be stored in the folder, even if the file __init__.py is empty.

A ***library*** can have one or more packages and *subpackages*.

Let us now learn how you can create your own library in Python.

Now on, we shall be using word *package* as we shall be creating simple libraries that can be called **package** interchangeably.

4.5.1 Structure of a Package

As you know that *Python packages* are basically collections of modules under common namespace. This common namespace is created via a directory that contains all related modules. But here you should know one thing that NOT ALL folders having multiple .py files (*i.e.*, modules) are packages. In order for a folder containing Python files to be recognized as a package, an **__init__.py** file (even if empty) must be part of the folder. Following figure (Fig 4.4) explains it.

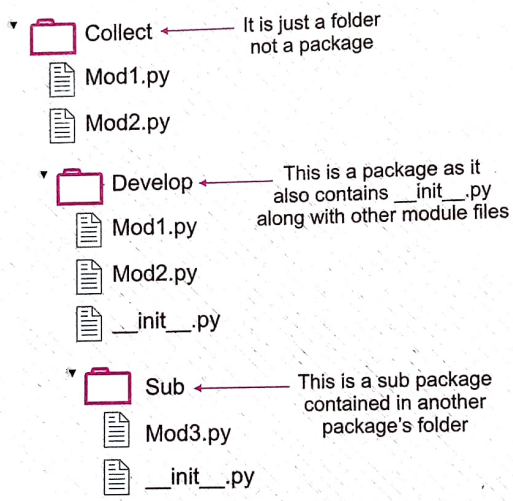


Figure 4.4 A package *vs.* folder.

NOTE

The file __init__.py in a folder, indicates it is an importable Python package, Without __init__.py, a folder is not considered a Python package.

You can even add an empty file having name as __init__.py in a folder to make it a Python package.

4.5.2 Procedure For Creating Packages

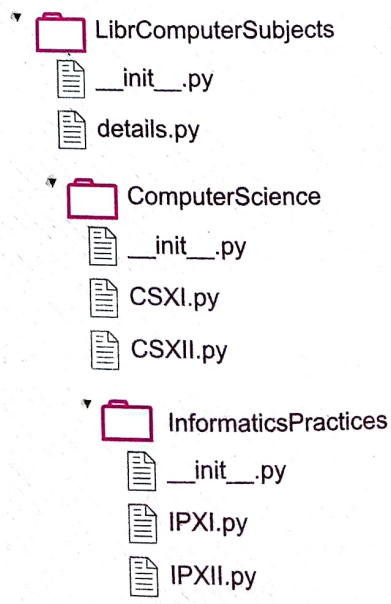
In order to create a package, you need to follow a certain procedure as discussed below.

Major steps to create a package are :

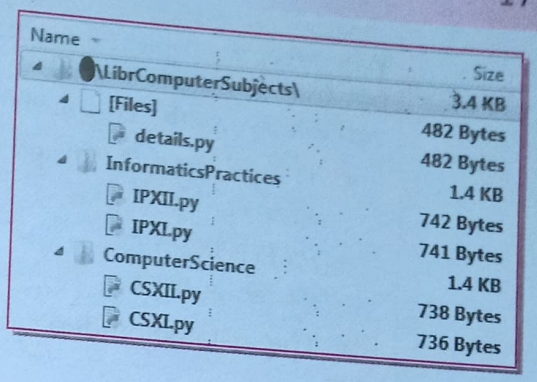
1. ***Decide about the basic structure of your package.*** It means that you should have a clear idea about what will be your package's name (*i.e.*, a directory/folder with that name will be created) and what all modules and subfolders (to serve as sub packages) will be part of it.

(*Just keep in mind that while naming the folders/subfolders, keep the words in the name underscore-separated; don't use any other word separators, at all (not even hyphens)*)

For instance, we are going to create the package shown on the right.



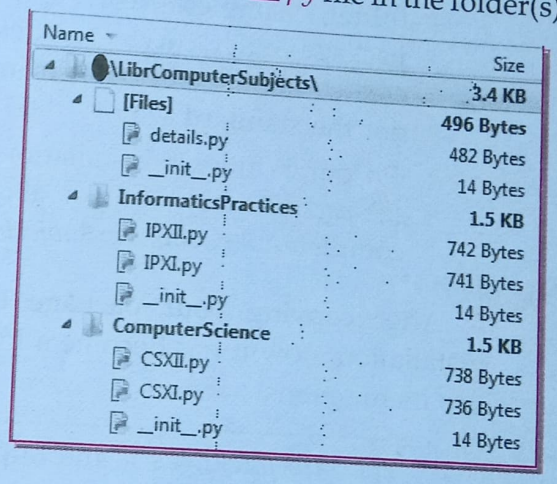
2. ***Create the directory structure having folders with names of package and subpackages.*** In our example shown above, we created a folder by the name **LibrComputerSubjects**.



Inside this folder, we created files/modules, *i.e.*, the .py files and subfolders with their own *.py* files. Now our topmost folder has the package name as per above figure and subfolders have names as the subpackages (as per adjacent figure).

But this is not yet eligible to be a package in Python as there is no *__init__.py* file in the folder(s).

3. ***Create __init__.py files in package and subpackage folders.*** We created an empty file and saved it as "__init__.py" and then copied this empty *__init__.py* file to the package and subpackage folders.



Now our directory structure looked like the one shown here.

Without the __init__.py file, Python will not look for submodules inside that directory, so attempts to import the module will fail.

4. ***Associate it with Python installation.*** Once you have your package directory ready, you can associate it with Python by attaching it to Python's **site-packages** folder of current Python distribution in your computer. You can import a library and package in Python only if it is attached to its **site-packages** folder.

(*i*) In order to check the path of the site-packages folder of Python, on the Python prompt, type the following two commands, one after another and try to locate the path of site-packages folder :

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

```
In [11]: import sys

In [12]: print(sys.path)
['', 'C:\\ProgramData\\Anaconda3\\python36.zip',
\Anaconda3\\DLLs'
\Anaconda3', 'C:\\ProgramData\\Anaconda3\\lib\\site-packages'
\ProgramData\\Anaconda3\\lib\\site-packages\\win32\\lib', 'C:\\ProgramData\
\Anaconda3\\lib\\site-packages\\win32\\lib', 'C:\\ProgramData\\Anaconda3\
\lib\\site-packages\\Pythonwin', 'C:\\ProgramData\\Anaconda3\\lib\\site-
packages\\IPython\\extensions', 'C:\\Users\\Edup\\.ipython']
```

This is the path of site-packages folder on your computer

The **sys.path** attribute gives an important information about **PYTHONPATH**, which specifies the directories that the Python interpreter will look in when importing modules.

If you carefully look at above result of **print(sys.path)** command, you will find that the first entry in PYTHONPATH is ' '. It means that the Python interpreter will first look in the current directory when trying to do an import. If the asked module/package is not found in current directory, then it check the directory listed in PYTHONPATH.

Whenever we import a module, say ***info***, the interpreter searches a built-in version. If not found, it searches for a file named ***info.py*** under a list of directories given by variable **sys.path**. This variable is initialized from the following locations :

- The directory holding the input script (or the current directory, in case no file is specified).
- PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
- The installation-dependent default.

(*ii*) Once you have figured out the path of **site-packages folder**, simplest way is to copy your own package-folder (*e.g., LibrComputerSubjects* that we created above) and paste it in this folder. (*But this is not the standard way.)

On our Windows installation, we simply opened the *site-packages folder* as per above path and pasted the complete *LibrComputerSubjects* folder there.

NOTE

The standard way of attaching a libray/package to site-packages folder uses a command like python setup.py install, BUT we are not going in details of that as we are keeping this discussion as simple as possible).

5. After copying your package folder in **site-packages** folder of your current Python installation, now it has become a Python library so that now you can import its modules and use its functions.

The ***site-packages*** is the target directory in which all installed Python packages are placed by default.

4.5.3 Using/Importing Python Libraries

Once you have created your own package (and subpackages) and attached it to **site-packages** folder of current Python installation on your computer, you can use them just as the way you use other Python libraries.

To use an installed Python library you need to do the following :

1. Import the library using import command :

```
import <full name of library>
```

2. Use the functions, attributes etc. defined in the library by giving their full name.

For instance, to use package *LibrComputerPackages's* two module files, which are shown here, you can import them and then use them as shown below :

```
#details.py
"""Details about computer subjects in CBSE"""
def SubjectsList( ) :
    print("There are two computer subjects in CBSE")
    print("'Computer Science' and 'Informatics Practices'")
```

```
#CSXI.py
"""Details about CBSE CS XI"""
def Syllabus() :
    print("Unit 1 : Programming and Computational Thinking-1")
    print("      : Pythom basics, basic sorting techniques etc.")
    print("Unit 2 : Computer Systems and Organisation")
    print("      : Computer organisation, execution, cloud computing etc.")
    print("Unit 3 : Data Management - 1")
    print("      : SQL, basics of NoSQL databases etc.")
    print("Unit 4 : Society, Law and ethics - 1")
    print("      : Cyber Safety, safe data communication etc.")
    print("Unit 5 : Practical Unit")
def About( ) :
    print("Computer Science XI")
    print("Contains 4 units and a Practical unit")
```

As you can see that the **details** module has a function namely ***SubjectsList( )*** and **CSXI** module has *two* functions namely ***Syllabus( )*** and ***About( )***.

(*i*) To import module ***details***, you can write import command as :

```
import LibrComputerSubjects.details
```

*As **details** module is inside package folder **LibraComputerDetails**, its full name is as shown here*

and use its function(s) by giving its full name *i.e.*,

LibrComputerSubjects.details.<function>( )

```
In [13]: import LibrComputerSubjects.details

In [14]: LibrComputerSubjects.details.SubjectsList()
There are two computer subjects in CBSE
'Computer Science' and 'Informatics Practices'
```

See how function **SubjectsList()** of **details** module in **LibrComputerSubjects** package is being invoked

(*ii*) To import module ***CSXI***, you can write import command as :

```
import LibrComputerSubjects.ComputerScience.CSXI
```

*As **CSXI** module is inside **ComputerScience** subfolder of package **LibraComputerDetails**, its full name is as shown here*

and use its function(s) by giving its full name *i.e.*,

LibrComputerSubjects.ComputerScience.CSXI.<function>( )

```
In [15]: import LibrComputerSubjects.ComputerScience.CSXI

In [16]: LibrComputerSubjects.ComputerScience.CSXI.Syllabus()
Unit 1 : Programming and Computational Thinking-1
       : Pythom basics, basic sorting techniques etc.
Unit 2 : Computer Systems and Organisation
       : Computer organisation, execution, cloud computing etc.
Unit 3 : Data Management - 1
       : SQL, basics of NoSQL databases etc.
Unit 4 : Society, Law and ethics - 1
       : Cyber Safety, safe data communication etc.
Unit 5 : Practical Unit
```

See how function **Syllabus()** of CSXI module in subfolder **ComputerScience** of package folder **LibrComputerSubjects** is being invoked

We have kept this discussion very simple and basic. You can even create installable libraries in Python. The installable libraries also have a *setup.py* file. But we are not going into those details as this is beyond the scope of this book.

However, we shall recommend one thing: While naming a library, try to use all lowercase letters although we used capital letters in above example (LibrComputerSubjects). We did this to make it more readable so that you could understand it easily. But if you are creating an actual library and you want it to be used by any python user then try to give it a unique name (*i.e.*, check in PyPI[8] – *Python Package Index* if your library name is unique) all in lowercase so that everyone can use it without bothering about the case of the letters.

Appendix B. '**Working with some useful Python Libraries – tkinter**' briefly talks about how you can use a useful Python Library : *tkinter* (your suggested practical exercises expect you to use this library alongwith *NumPy* and *SciPy*. *NumPy* is covered in chapter 8)

Check Point 4.1

1. Which operator is used in Python to import all modules from packages?
   (*a*) . operator
   (*b*) * operator
   (*c*) -> symbol
   (*d*) , operator
2. Which file must be part of the folder containing Python module files to make it importable python package?
   (*a*) init.py
   (*b*) __ setup__.py
   (*c*) __init__.py
   (*d*) setup.py
3. In python which is the correct method to load a module math?
   (*a*) include math
   (*b*) import math
   (*c*) #include<math.h>
   (*d*) using math
4. Which is the correct command to load just the tempc method from a module called usable?
   (*a*) import usable, tempc
   (*b*) import tempc from usable
   (*c*) from usable import tempc
   (*d*) import tempc
5. What is the extension of Python library modules?
   (*a*) .mod (*b*) .lib (*c*) .code (*d*) .py

CREATING AND USING PACKAGES — Progress In Python 4.3

This PriP session is based on practice questions for creating and using Python packages.

⋮

>>>❖<<<

With this, we have come to the end of this chapter. Let us quickly revise what we have covered in this chapter.

LET US REVISE

- *A library refers to a collection of modules that together cater to specific type of needs or applications.*
- *A module is a separately saved unit whose functionality can be reused at will.*
- *A function is a named block of statements that can be invoked by its name.*
- *Python can have three types of functions : built-in functions, functions in modules and user-defined functions.*
- *A Python module can contain objects like docstrings, variables, constants, classes, objects, statements, functions.*
- *A Python module has the .py extension.*

8. The *Python Package Index* (PyPI) is a repository of software for the Python programming language. PyPI helps you find and install software developed and shared by the Python community. Package authors use PyPI to distribute their software.

- *The docstrings are useful for documentation purposes.*
- *A Python module can be imported in a program using import statement.*
- *There are two forms of import statements*
  (i) import <modulename> [as <aliasname>]
  (ii) from <module> import <object>
- *The built-in functions of Python are always available ; one needs not import any module for them.*
- *The random module of Python provides random-number-generation functionality.*
- *The string module of Python offers many useful constant strings and a utility function capwords( ).*
- *In order to create own package, it should be created so that it has a folder having its name and it also has a special file __init__.py in it. After this, it must be attached to site-packages folder of current Python installation.*
- *A package installed or attached to **site-packages** folder of Python installation can easily be imported using import command.*

Objective Type Questions

OTQs

Multiple Choice Questions

1. A .py file containing constants/variables, classes, functions etc. related to a particular task and can be used in other programs is called
   (*a*) module (*b*) library (*c*) classes (*d*) documentation
2. The collection of modules and packages that together cater to a specific type of applications or requirements, is called _____ .
   (*a*) module (*b*) library (*c*) classes (*d*) documentation
3. An independent triple quoted string given inside a module, containing documentation related information is a _____ .
   (*a*) Documentation string (*b*) docstring
   (*c*) dstring (*d*) stringdoc
4. The help <module> statement displays _____ from a module.
   (*a*) constants (*b*) functions (*c*) classes (*d*) docstrings
5. Which command(s) modifies the current namespace with the imported object name ?
   (*a*) import <module> (*b*) import <module1>, <module2>
   (*c*) from <module> import <object> (*d*) from <module> import *
6. Which command(s) creates a separate namespace for each of the imported module ?
   (*a*) import <module> (*b*) import <module1>, <module2>
   (*c*) from <module> import <object> (*d*) from <module> import *
7. Which of the following random module functions generates a floating point number ?
   (*a*) random( ) (*b*) randint( )
   (*c*) uniform( ) (*d*) all of these
8. Which of the following random module functions generates an integer ?
   (*a*) random( ) (*b*) randint( )
   (*c*) uniform( ) (*d*) all of these

9. Which file must be a part of a folder to be used as a Python package ?
   (*a*) package.py (*b*) __init__.py (*c*) __package__.py (*d*) __module.py__
10. A Python module has _____ extension.
   (*a*) .mod (*b*) .imp (*c*) .py (*d*) .mpy

Fill in the Blanks

1. The file _____ must be part of the folder holding library files and other definitions in order to be treated as importable package.
2. A _____ refers to a collection of modules that together cater to specific type of needs or applications.
3. A Python _____ is a file (.py file) containing variables, class definitions, statements and functions related to a particular task.
4. The uniform( ) function is the part of _____ module.
5. The capwords( ) function is the part of _____ module.

True/False Questions

1. A python program and a Python module means the same.
2. A python program and a python module have the same .py file extension.
3. The import <module> statement imports everything in the current namespace of the Python program.
4. Any folder having .py files is a Python package.
5. A folder having .py files along with a special file *i.e.*, __init__.py in it is an importable Python package.

NOTE : Answers for OTQs are given at the end of the book.

Solved Problems

1. *What is a module, package and a library ?*

Solution.**Module.** A module is a file with some Python code and is saved with a .py extension.

Package. A package is a directory that contains subpackages and modules in it along with some special files such as __init__.py.

Library. A Python library is a reusable chunk of code that is used in program/script using import command. A package is a library if it is installable or gets attached to ***site-packages*** folder of Python installation.

The line between a package and a Python library is quite blurred and both these terms are often used interchangeably.

2. *What is a Python module ? What is its significance ?*

Solution. A "module" is a chunk of Python code that exists in its own (.py) file and is intended to be used by Python code outside itself.

Modules allow one to bundle together code in a form in which it can easily be used later.

The Modules can be "imported" in other programs so the functions and other definitions in imported modules become available to code that imports them.

3. *What is the utility of built-in function help( ) ?*

Solution. Python's built-in function help( ) is very useful. When it is provided with a program-name or a module-name or a function-name as an argument, it displays the documentation of the argument as help. It also displays the docstrings within its passed-argument's definition.

For example,

```
help(math)
```

will display the documentation related to module *math*.

It can even take function name as argument, *e.g.*,

```
help(math.sqrt)
```

The above code will list the documentation of ***math.sqrt( )*** function only.

4. *What are docstrings ? How are they useful ?*

Solution. A *docstring* is just a regular Python triple-quoted string that is the first thing in a function body / a module / a class.

When executing a function body (or a module / class), the *docstring* doesn't do anything like comments, but Python stores it as part of the function documentation. This documentation can later be displayed using *help( )* function. So, even though *docstrings* appear like comments (no execution) but these are different from comments.

5. *What happens when Python encounters an* **import** *statement in a program ? What would happen, if there is one more import statement for the same module, already imported in the same program ?*

Solution. When Python encounters an **import** statement, it does the following :

- the code of imported module is interpreted and executed.
- defined functions and variables created in the module are now available to the program that imported module.
- For imported module, a new **namespace** is setup with the same name as that of the module.

Any duplicate import statement for the same module in the same program is ignored by Python.

6. *The random( ) function generates a random floating point number in the range 0.0 to 1.0 and randint(a, b) function generates random integer between range a to b. To generate random numbers between range a to b using random( ), following formula is used :*

```
math.random()*(b - a) + a
```

Now if we have following two statements (carefully have a look)

(*i*) `int( ( math.random() * (b - a) + a) )`

(*ii*) `math.randint(a, b)`

Can we say above two statements are now producing random integers from the same range ? Why ?

Solution. No, their range is not the same.

The first statement will be able to produce random integers (say *N*) in the range $a <= N < b$ whereas the second statement will be producing random integers in the range $a <= N <= b$.

The reason being ***random( )*** function excludes the upper limit while generating random numbers whereas ***radint( )*** includes both upper limit and lower limit.

7. *Name the Python Library modules which need to be imported to invoke the following functions :* [CBSE Sample Paper 2019-20]

(*i*) *sin( )* (*ii*) *randint ( )*

Solution. (*i*) math module (*ii*) random module

8. *Consider a module 'simple' given below :*

```
# module simple.py
""" "Greets or scolds on call" """

def greet():
    """ "Greet anyone you like :-)" """
    print("Helloz")

def scold():
    """ "Use me for scolding, but scolding is not good :-(" """
    print("Get lost")

count = 10
print("greeting or scolding - is it simple ?")
```

Another program 'test.py' imports this module. The code inside test.py is :

```
# test.py
import simple
print(simple.count)
```

What would be the output produced, if we run the program test.py ? Justify your answer.

Solution. The output produced would be :

```
greeting or scolding - is it simple ?
10
```

The reason being, import module's main block[9] is executed upon *import*, so its import statement causes it to print :

```
greeting or scolding - is it simple ?
```

And ***print(simple.count)*** statement causes output's next line, *i.e.*,

```
10
```

9. *What would be the output produced by the following code :*

```
import math
import random
print(math.ceil(random.random()))
```

Justify your answer.

Solution. The output produced would be : `1.0`

Reason being that ***random.random()*** would generate a number in the range [0.0, 1.0) but ***math.ceil()*** will return ceiling number for this range, which is 1.0 for all the numbers in this range. Thus the output produced will always be **1.0**

10. *Consider the following code :*

```
import math
import random
print(str( int( math.pow( random.randint(2, 4), 2))), end = ' ')
print(str( int( math.pow( random.randint(2, 4), 2))), end = ' ')
print(str( int( math.pow( random.randint(2, 4), 2))))
```

9. Refer to Appendix A for more details.

What could be the possible outputs out of the given four choices?

(*i*) 2 3 4 (*ii*) 9 4 4 (*iii*) 16 16 16 (*iv*) 2 4 9 (*v*) 4 9 4 (*vi*) 4 4 4

Solution. The possible outputs could be (*ii*), (*iii*) (*v*) and (*vi*).

The reason being that ***randint( )*** would generate an integer between range 2...4, which is then raised to power 2, so possible outcomes can be any one of these *three* : 4, 9 or 16.

11. *What possible outputs(s) are expected to be displayed on screen at the time of execution of the program from the following code ? Also specify the maximum values that can be assigned to each of the variables FROM and TO.*

```
import random
AR = [20,30,40,50,60,70];
FROM = random.randint(1,3)
TO = random.randint(2,4)
for K in range(FROM,TO+1):
    print (AR[K],end = "#")
```

(i) 10#40#70# *(ii) 30#40#50#* *(iii) 50#60#70#* *(iv) 40#50#70#* [CBSE Sample Paper 2019-20]

Solution. (*ii*) 30#40#50# Maximum value FROM, TO is 3,4

12. *What is the procedure to create own library/package in Python?*

Solution. To create own library or package in Python, we should do the following :

(*i*) Create a package folder having the name of the package/library

(*ii*) Add module files (.py files containing actual code functions) to this package folder

(*iii*) Add a special file __init__.py to it (even if the file is empty)

(*iv*) Attach this package folder to site-packages folder of Python installation.

13. *Why is a package attached to site-packages folder of Python installation? Can't we use it without doing so?*

Solution. In order to import a package using **import** command, the package and its contents must be attached to ***site-packages*** folder of Python installation as this is the default place from where Python interpreter imports Python library and packages.

So in order to import our package with **import** command in our programs, we must attach it to ***site-packages*** folder of Python installation.

14. *What is the output of the following piece of code ?*

```
#mod1
def change(a):
    b = [x*2 for x in a]
    print(b)
#mod2
def change(a):
    b = [x*x for x in a]
    print(b)
from mod1 import change
from mod2 import change
#main
s = [1,2,3]
change(s)
```

Solution. There is a name-clash. A name clash is a situation when two different entities with the same name become part of the same scope. Since both the modules have the same function name, there is a name clash, which is an error.

15. *What is the problem in the following piece of code?*

```
from math import factorial
print(math.factorial(5))
```

Solution. In the "from-import" form of import, the imported identifiers (in this case factorial()) become part of the current local namespace and hence their module's names aren't specified along with the module name. Thus, the statement should be :

```
print(factorial(5))
```

GLOSSARY

Module	Named independent grouping of code and data.
Namespace	Named logical environment holding logical grouping of related objects.
Package	A directory containing modules and subpackages and some special files.
Library	A reusable chunk of code that can be included and used in other programs.

For Selective Assignment Solutions

Scan QR Code

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is the significance of Modules ?
2. What are docstrings ? What is their significance ? Give example to support your answer.
3. What is a package ? How is a package different from module ?
4. What is a library ? Write procedure to create own library in Python.
5. What is the use of file __init__.py in a package even when it is empty ?
6. What is the importance of *site-packages* folder of Python installation ?
7. How are following import statements different ?
   (*a*) import X (*b*) from X import * (*c*) from X import a, b, c
8. What is PYTHON PATH variable ? What is its significance ?
9. In which order Python looks for the function/module names used by you.
10. What is the usage of help( ) and dir( ) functions.
11. Name the Python Library modules which need to be imported to invoke the following functions :
    (*i*) log( ) (*ii*) pow( ) [CBSE D 2016]
12. What is dot notation of referring to objects inside a module ?
13. Why should the **from <module> import <object>** statement be avoided to import objects ?
14. What do you understand by standard library of Python ?
15. Explain the difference between *import <module>* and *from <module> import* statements, with examples.

Type B : Application Based Questions

1. Create module tempConversion.py as given in Fig. 8.2 in the chapter.
   If you invoke the module with two different types of import statements, how would the function call statement for imported module's functions be affected ?
2. A function checkMain( ) defined in module Allchecks.py is being used in two different programs. In program 1 as

   ```
   Allchecks.checkMain(3, 'A')
   ```

   and in program 2 as

   ```
   checkMain(4, 'Z')
   ```

   Why are these two function-call statements different from one another when the function being invoked is just the same ?
3. Given below is semi-complete code of a module *basic.py* :

   ```
   #
   """_____"""

   def square(x) :
     """_____"""
     return mul(x, x)
   ____ mul(x, y) :
     """_____"""
     return x * y
   def div(x, y) :
     """_____"""
     return float(x)/y

   ____ fdiv(x, y) _____
     """_____"""
     _____ x//y

   def   floordiv(x, y)_____
     _____ fdiv(x, y)
   ```

   Complete the code. Save it as a module.
4. After importing the above module, some of its functions are executed as per following statements. Find errors, if any :

   (*a*) `square(print 3)` (*b*) `basic.div()` (*c*) `basic.floordiv(7.0, 7)`
   (*d*) `div(100, 0)` (*e*) `basic.mul(3, 5)` (*f*) `print(basic.square(3.5))`
   (*g*) `z = basic.div(13, 3)`
5. Import the above module *basics.py* and write statements for the following :

   (*a*) Compute square of 19.23 (*b*) Compute floor division of 1000.01 with 100.23
   (*c*) Compute product of 3, 4 and 5. (Hint. use a function multiple times).
   (*d*) What is the difference between
   `basics.div(100, 0)` and `basics.div(0, 100)` ?

6. Suppose that after we import the random module, we define the following function called *diff* in a Python session :

```
def diff():
    x = random.random() - random.random()
    return(x)
```

What would be the result if you now evaluate

```
y = diff()
print(y)
```

at the Python prompt ? Give reasons for your answer.

7. What are the possible outcome(s) executed from the following code ? Also specify the maximum and minimum values that can be assigned to variable NUMBER. [CBSE D 2015]

```
STRING = "CBSEONLINE"
NUMBER = random.randint(0, 3)
N = 9
while STRING[N] != 'L' :
    print (STRING[N] + STRING[NUMBER] + '#', end = ' ')
    NUMBER = NUMBER + 1
    N = N - 1
```

(*i*) ES#NE#IO# (*ii*) LE#NO#ON# (*iii*) NS#IE#LO# (*iv*) EC#NB#IS#

8. Consider the following code :

```
import random
print(int( 20 + random.random() * 5), end = ' ' )
print(int( 20 + random.random() * 5), end = ' ' )
print(int( 20 + random.random() * 5), end = ' ' )
print(int( 20 + random.random() * 5))
```

Find the suggested output options (*i*) to (*iv*). Also, write the least value and highest value that can be generated.

(*i*) 20 22 24 25 (*ii*) 22 23 24 25
(*iii*) 23 24 23 24 (*iv*) 21 21 21 21

9. Consider the following code :

```
import random
print(100 + random.randint(5, 10), end = ' ' )
print(100 + random.randint(5, 10), end = ' ' )
print(100 + random.randint(5, 10), end = ' ' )
print(100 + random.randint(5, 10))
```

Find the suggested output options (*i*) to (*iv*). Also, write the least value and highest value that can be generated.

(*i*) 102 105 104 105 (*ii*) 110 103 104 105
(*iii*) 105 107 105 110 (*iv*) 110 105 105 110

10. Consider the following package

```
music/
├── __init__.py                    Top-level package
├── formats/                       Subpackage for file format
│   ├── __init__.py
│   ├── wavread.py
│   └── wavwrite.py
├── effects/                       Subpackage for sound effects
│   ├── __init__.py
│   ├── echo.py
│   ├── surround.py
│   └── reverse.py
└── filters/                       Subpackage for filters
    ├── __init__.py
    ├── equalizer.py
    ├── vocoder.py
    └── karaoke.py
```

Each of the above modules contain functions play( ), writefile( ) and readfile( ).

(*a*) If the module **wavwrite** is imported using command **import music.formats.wavwrite.** How will you invoke its writefile( ) function? Write command for it.

(*b*) If the module **wavwrite** is imported using command **from music.formats import wavwrite.** How will you invoke its writefile( ) function? Write command for it.

11. What are the possible outcome(s) executed from the following code ? Also specify the maximum and minimum values that can be assigned to variable PICKER. [CBSE D 2016]

```
import random
PICK = random.randint(0, 3)
CITY = ["DELHI", "MUMBAI", "CHENNAI", "KOLKATA"] ;
for I in CITY :
    for J in range(1, PICK) :
        print(I, end = " ")
    print( )
```

(*i*)
```
DELHIDELHI
MUMBAIMUMBAI
CHENNAICHENNAI
KOLKATAKOLKATA
```

(*ii*)
```
DELHI
DELHIMUMBAI
DELHIMUMBAICHENNAI
```

(*iii*)
```
DELHI
MUMBAI
CHENNAI
KOLKATA
```

(*iv*)
```
DELHI
MUMBAIMUMBAI
KOLKATAKOLKATAKOLKATA
```

Type C : Programming Practice/Knowledge based Questions

1. Write a Python program having following functions :

   (*i*) A function with the following signature :

   `remove_letter(sentence, letter)`

   This function should take a string and a letter (as a single-character string) as arguments, returning a copy of that string with every instance of the indicated letter removed. For example, **remove_letter("Hello there!", "e")** should return the string "Hllo thr!".

   Try implementing it using **<str>.split( )** function.

   (*ii*) Write a function to do the following :

   Try implementing the **capwords( )** functionality using other functions, *i.e.*, split( ), capitalize( ) and join( ). Compare the result with the capwords( ) function's result.

2. Create a module lengthconversion.py that stores functions for various lengths conversion *e.g.*,

   - miletokm( ) to convert miles to kilometer
   - kmtomile( ) to convert kilometers to miles
   - feettoinches( )
   - inchestofeet( )

   It should also store constant values such as value of (mile in kilometers and vice versa).

   [1 mile = 1.609344 kilometer ; 1 feet = 12 inches]

   Help( ) function should display proper information.

3. Create a module MassConversion.py that stores function for mass conversion *e.g.*,

   - kgtotonne( ) to convert kg to tonnes
   - tonnetokg( ) to convert tonne to kg
   - kgtopound( ) to convert kg to pound
   - poundtokg( ) to convert pound to kg

   (Also store constants 1 kg = 0.001 tonne, 1 kg = 2.20462 pound)

   Help( ) function should give proper information about the module.

4. Create a package from above two modules as this :

```
Conversion
├── Length
│   └── Lengthconversion.py
└── Mass
    └── Massconversion.py
```

Make sure that above package meets the requirements of being a Python package. Also, you should be able to import above package and/or its modules using import command.