COMPUTER SCIENCE With python

Textbook for
Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

Preface

Quality of education depends a lot on the curriculum imparted. Since *Computer Science* is a rapidly evolving field, CBSE constituted a committee to look into existing Computer Science curriculum and recommend a curriculum that is modern, relatively light, teaches timeless concepts like computational thinking, is upto international standards and focuses on problem solving skills than just the syntax.

The new syllabus of Computer Science (083) is the outcome of the recommendations by the curriculum review committee. The objectives of this new syllabus are laudable — focus on clear understanding of concepts, applications of concepts, problem solving skills, develop computational thinking etc.

This book has been written keeping all this in mind. This book adheres to the CBSE curriculum for Computer Science (083) for **Class XII**. Based of the syllabus, the book has been divided into four units.

Unit 1 : Programming and Computational Thinking (PCT-2)

Unit 1 has been divided into ***ten*** chapters **(chapters 1-10)**. It covers Python programming through revision tour, Python functions, Libraries, File handling in Python, Recursion in Python functions, Data visualization using PyPlot, Data Structures in Python. This unit also covers basics of determining algorithm efficiency.

Unit 2 : Computer Networks (CN)

This unit has been divided into ***two*** chapters **(chapters 11-12)**. It covers computer networks' basics such as types of networks, cloud technology, IoT, wired and wireless networks, client, server networks, network devices, modulation techniques, collision in wireless networks, IPv4, IPv6, TCP, other protocols and network tools.

Unit 3 : Data Management (DM-2)

This unit has been divided into ***four*** chapters **(chapters 13-16)**. It covers SQL revision, more on SQL commands, developing a minimal web application using Django web framework, and Python database connectivity with MySQL databases.

Unit 4 : Society, Law and Ethics (SLE-2)

This unit has been covered in ***one*** chapter **(*chapter 17*)**. It covers many topics and issues related to society, law and ethics, such as IPR, plagiarism, digital rights and licencing, privacy laws and cyber crime, IT act, e-waste management, challenges online and gender and disability issues.

Apart from the text book, we have also provided a practical book *'Progress In Python'* that contains additional practice exercises. Although the text book contains sufficient number of practice questions and exercises, yet the additional practice of exercises given in the practical book will make the foundation of programming and other concepts even stronger.

[redacted]

Syllabus

Distribution of Marks

Unit No.	Unit Name	Marks
1.	Programming and Computational Thinking-2	30
2.	Computer Networks	15
3.	Data Management—2	15
4.	Society, Law and Ethics—2	10
5.	Practicals	30
	Total	100

Unit 1 : Programming and Computational Thinking (PCT-2) (80 Theory + 70 Practical)

- Revision of the basics of Python
- Functions: scope, parameter passing, mutable/immutable properties of data objects, pass arrays to functions, return values, functions using libraries : mathematical, and string functions.
- File handling: open and close a file, read, write, and append to a file, standard input, output and error streams, relative and absolute paths.
- Using Python libraries: create and import Python libraries.
- Recursion: simple algorithms with recursion: factorial, Fibonacci numbers; recursion on arrays: binary search.
- Idea of efficiency: performance defined as inversely proportional to the wall clock time, count the number of operations a piece of code is performing, and measure the time taken by a program. Example: take two different programs for the same problem, and understand how the efficient one takes less time.
- Data visualization using Pyplot: line chart, pie chart, and bar chart.
- Data-structures: lists, stacks, queues.

Unit 2: Computer Networks (CN) (30 Theory + 10 Practical)

- Structure of a network : Types of networks: local area and wide area (web and internet), new technologies such as cloud and IoT, public vs private cloud, wired and wireless networks; concept of a client and server.
- Network devices such as a NIC, switch, hub, router, and access point.
- Network stack: amplitude and frequency modulation, collision in wireless networks, error checking, and the notion of a MAC address, main idea of routing. IP addresses: (v4 and v6), routing table, router, DNS, and web URLs, TCP : basic idea of retransmission, and rate modulation when there is congestion (analogy to a road network), Protocols: 2G, 3G, 4G, Wi-Fi. What makes a protocol have a higher bandwidth ?
- Basic network tools: traceroute, ping, ipconfig, nslookup, whois, speed-test.
- Application layer: HTTP (basic idea), working of email, secure communication : encryption and certificates (HTTPS), network applications: remote desktop, remote login, HTTP, FTP, SCP, SSH, POP/IMAP, SMTP, VoIP, NFC.

Unit 3: Data Management (DM-2)

(20 Theory + 20 Practical)

- Write a minimal Django based web application that parses a GET and POST request, and writes the fields to a file - flat file and CSV file.
- Interface Python with an SQL database.
- SQL commands: aggregation functions – having, group by, order by.

Unit 4: Society, Law and Ethics (SLE-2)

(10 Theory)

- Intellectual property rights, plagiarism, digital rights management, and licensing (Creative Commons, GPL and Apache), open source, open data, privacy.
- Privacy laws, fraud; cyber-crime, phishing, illegal downloads, child pornography, scams; cyber forensics, IT Act, 2000.
- Technology and society: understanding of societal issues and cultural changes induced by technology.
- E-waste management : proper disposal of used electronic gadgets.
- Identity theft, unique ids, and biometrics.
- Gender and disability issues while teaching and using computers.

PRACTICAL

Unit No.	Unit Name	Marks
1.	**Lab Test (10 marks)**	
	Python program (60% logic + 20% documentation + 20% code quality)	7
	Small Python program that sends a SQL query to a database and displays the result. A stub program can be provided.	3
2.	**Report File + Viva (9 marks)**	
	Report file : Minimum 21 Python programs. Out of this at least 4 programs should send SQL commands to a database and retrieve the result; at least 1 program should implement the web server to write user data to a CSV file.	7
	Viva voce : (based on the report file)	2
3.	**Project + Viva (11 marks)**	
	Project (that uses most of the concepts that have been learnt)	8
	Project viva voce	3

Programming in Python

- Recursively find the factorial of a natural number.
- Read a file line by line and print it.
- Remove all the lines that contain the character '*a*' in a file and write it to another file.
- Write a Python function sin (*x*, *n*) to calculate the value of sin (*x*) using its Taylor series expansion up to *n* terms. Compare the values of sin (*x*) for different values of *n* with the correct value.
- Write a random number generator that generates random numbers between 1 and 6 (simulates a dice).
- Write a recursive code to find the sum of all elements of a list.
- Write a recursive code to compute the n^{th} Fibonacci number.
- Write a Python program to implement a stack and queue using a list data-structure.
- Write a recursive Python program to test if a string is a palindrome or not.

- Write a Python program to plot the function $y = x^2$ using the pyplot or matplotlib libraries.
- Create a graphical application that accepts user inputs, performs some operation on them, and then writes the output on the screen. For example, write a small calculator. Use the tkinter library.
- Open a webpage using the urllib library.
- Compute EMIs for a loan using the numpy or scipy libraries.
- Take a sample of 10 phishing e-mails and find the most common words.

Data Management : SQL and Web-Server

- Find the min, max, sum, and average of the marks in a student marks table.
- Find the total number of customers from each country in the table (customer ID, customer name, country) using group by.
- Write a SQL query to order the (student ID, marks) table in descending order of the marks.
- Integrate SQL with Python by importing the MySQL module.
- Write a Django based web server to parse a user request (POST), and write it to a CSV file.

Project

- The aim of the class project is to create something that is tangible and useful. This should be done in groups of 2 to 3 students, and should be started by students at least 6 months before the submission deadline. The aim here is to find a real world problem that is worthwhile to solve. Students are encouraged to visit local businesses and ask them about the problems that they are facing. For example, if a business is finding it hard to create invoices for filing GST claims, then students can do a project that takes the raw data (list of transactions), groups the transactions by category, accounts for the GST tax rates and creates invoices in the appropriate format. Students can be extremely creative here. They can use a wide variety of Python libraries to create user friendly applications such as games, software for their school, software for their disabled fellow students, and mobile applications. Of course to do some of this projects some additional learning is required; this should be encouraged. Students should know how to teach themselves.
- If three people work on a project for 6 months, at least 500 lines of code is expected. The committee has also been made aware about the degree of plagiarism in such projects. Teachers should take a very strict look at this situation, and take very strict disciplinary action against students who are cheating on lab assignments, or projects, or using pirated software to do the same. Everything that is proposed can be achieved using absolutely free, and legitimate open source software.

Contents

1

Python Revision Tour

In This Chapter

1.1 INTRODUCTION

You must have enjoyed learning Python in class XI. Python programming language, developed by *Guido Van Rossum* in early 1990s, has become a very popular programming language among beginners as well as developers. The journey of Python that you started in class XI will continue in class XII as well. In class XII, you shall learn more about Python and some advanced concepts. Before we start with newer topics and concepts in Python, let us revise all that you have learnt in class XI. And this chapter will be doing just the same, *i.e.*, take you to the revision tour of Python that you learnt in your previous class.

1.2 TOKENS IN PYTHON

The smallest individual unit in a program is known as a *Token* or a *lexical unit.*

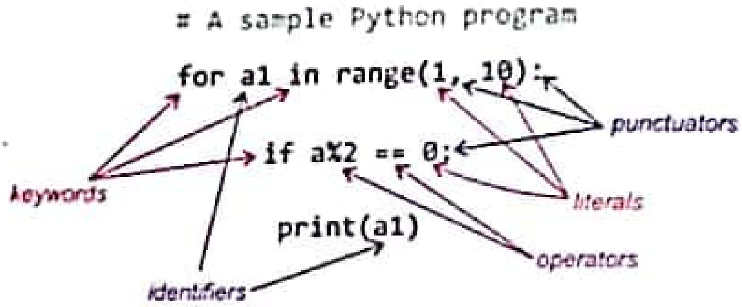
TOKENS

The smallest individual unit in a program is known as a ***Token*** or a ***lexical unit.***

Python has following tokens :

(*i*) Keywords (*ii*) Identifiers (Names) (*iii*) Literals

(*iv*) Operators (*v*) Punctuators



```
# A sample Python program
for a1 in range(1, 10):
    if a%2 == 0:
        print(a1)
```

Figure 1.1 Some tokens in a Python program.

Let us revise our understanding of tokens.

1.2.1 Keywords

Keywords are predefined words with special meaning to the language compiler or interpreter. These are reserved for special purpose and must not be used as normal identifier names.

KEYWORD

A ***keyword*** is a word having special meaning reserved by programming language.

Python programming language contains the following keywords :

False	assert	del	for	in	or	while
None	break	elif	from	is	pass	with
True	class	else	global	lambda	raise	yield
and	continue	except	if	nonlocal	return	
as	def	finally	import	not	try	

1.2.2 Identifiers (Names)

Identifiers are the names given to different parts of the program *viz. variables, objects, classes, functions, lists, dictionaries* and so forth.

The naming rules for Python identifiers can be summarized as follows :

- Variable names must only be a *non-keyword word* with no spaces in between.
- Variable names must be made up of only letters, numbers, and underscore (_).
- Variable names cannot begin with a number, although they can contain numbers.

NOTE

Python is case sensitive as it treats upper and lower-case characters differently.

The following are some *valid* identifiers :

`Myfile`	`DATE9_7_77`
`MYFILE`	`_DS`
`_CHK`	`FILE13`
`Z2T0Z9`	`_HJI3_JK`

The following are some *invalid* identifiers :

`DATA-REC`	contains special character - (hyphen) (other than A - Z, a - z and _ (underscore))
`29CLCT`	Starting with a digit
`break`	reserved keyword
`My.file`	contains special character dot (.)

1.2.3 Literals/Values

Literals are data items that have a fixed/constant value.

Python allows several kinds of literals, which are being given below.

(i) String Literals

A *string literal* is a sequence of characters surrounded by quotes (single or double or triple quotes). String literals can either be ***single line strings*** or ***multi-line strings***.

- **Single line strings** must terminate in one line *i.e.*, the closing quotes should be on the same line as that of the opening quotes. (See below)
- **Multiline strings** are strings spread across multiple lines. With single and double quotes, each line other that the concluding line has an end character as \ (backslash) but with triple quotes, no backslash is needed at the end of intermediate lines. (see below) :

```
>>> Text1 = "Hello World"  ←———  Single line string

>>> Text2 = "Hello\  ←———  Multi-line string
    World "

Text3 = '''Hello  ←———  No backslash needed
    World'''
```

In strings, you can include non-graphic characters through escape sequences. Escape sequences are given in following table :

Escape sequence	What it does [Non-graphic character]	Escape sequence	What it does [Non-graphic character]
\\\\	Backslash (\\)	\r	Carriage Return (CR)
\\'	Single quote (')	\t	Horizontal Tab (TAB)
\\"	Double quote (")	\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\a	ASCII Bell (BEL)	\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (Unicode only)
\b	ASCII Backspace (BS)	\v	ASCII Vertical Tab (VT)
\f	ASCII Formfeed (FF)	\ooo	Character with octal value ooo
\n	New line character	\xhh	Character with hex value hh
\N{name}	Character named name in the Unicode database (Unicode only)		

(ii) Numeric Literals

Numeric literals are numeric values and these can be one of the following types :

(*a*) int **(signed integers)** often called just **integers** or **ints**, are positive or negative whole numbers with no decimal point.

The integer literals can be written in :

- **Decimal form** : an integer beginning with digits 1-9. *e.g.*, 1234, 4100 etc.
- **Octal form** : an integer beginning with 0o (zero followed by letter o) *e.g.*, 0o35, 0o77 etc. Here do remember that for Octal, 8 and 9 are invalid digits.
- **Hexadecimal form** : an integer beginning with 0x (zero followed by letter X) *e.g.*, 0x73, 0xAF etc. Here remember that valid digits/letters for hexadecimal numbers are *0-9* and *A-F*.

(*b*) **Floating Point Literals. Floating point literals** or **real literals floats** represent real numbers and are written with a decimal point dividing the integer and fractional parts are numbers having fractional parts. These can be written in fractional form *e.g.*, –13.0, .75, 7. etc. or in **Exponent form** e.g., *0.17E5, 3.E2, .6E4* etc.

(*c*) **Complex number literals** are of the form $a + bJ$, where a and b are *floats* and J (or j) represents $\sqrt{-1}$, which is an imaginary number). a is the real part of the number, and b is the imaginary part.

(iii) Boolean Literals

A Boolean literal in Python is used to represent one of the two Boolean values *i.e.*, **True** (Boolean true) or **False** (Boolean false). A Boolean literal can either have value as *True* or as *False*.

(iv) Special Literal None

Python has one special literal, which is **None**. The **None** literal is used to indicate absence of value.

Python can also store literal collections, in the form of **tuples** and **lists** etc.

1.2.4 Operators

Operators are tokens that trigger some computation / action when applied to variables and other objects in an expression.

The operators can be **arithmetic operators** (+, –, *, /, %, **, //), **bitwise operators** (&, ^, |), **shift operators** (<<, >>), **identity operators** (is, is not), **relational operators** (>, <, >=, <=, ==, !=), **logical operators** (and, or), **assignment operator** (=), **membership operators** (in, not in), and **arithmetic-assignment operators** (/=, +=, –=, */, %=, **=, //=).

1.2.5 Punctuators

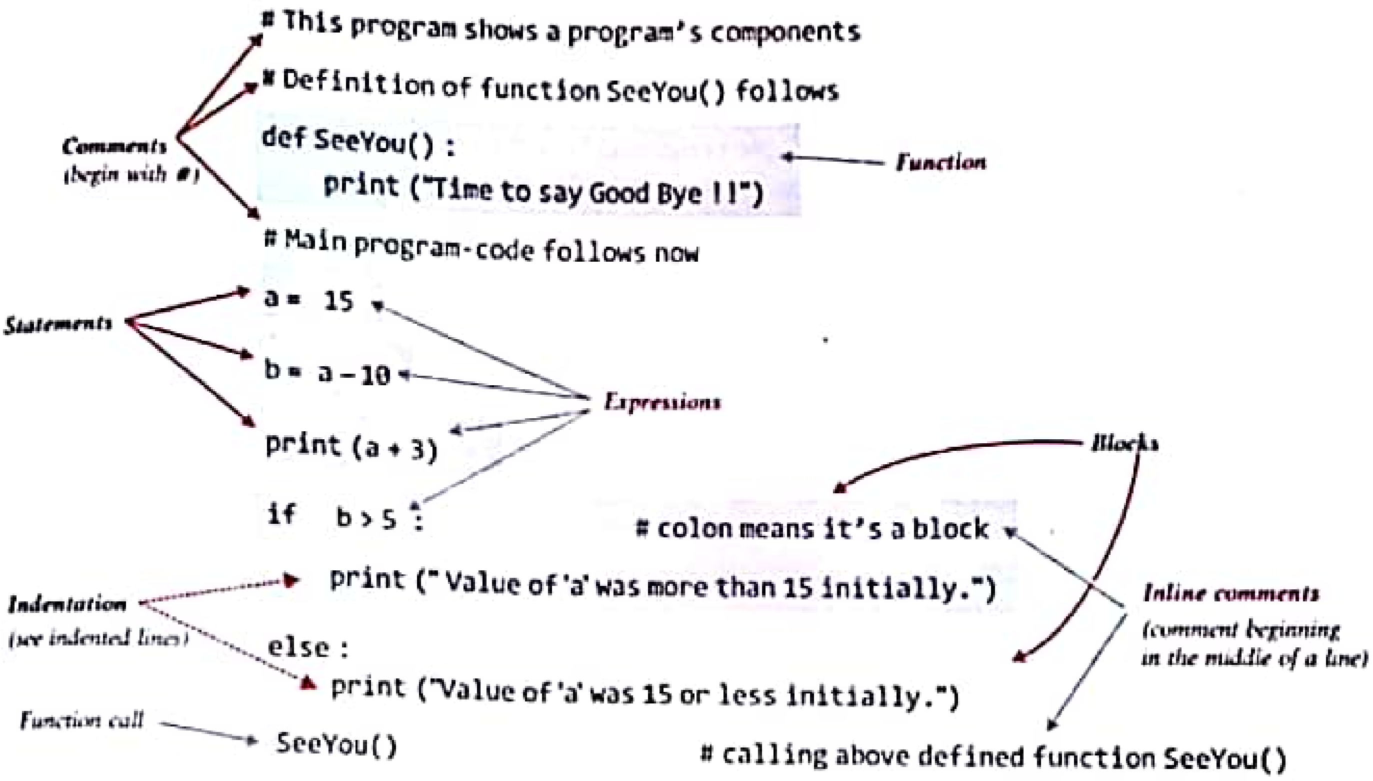
Punctuators are symbols that are used in programming languages to organize sentence structures, and indicate the rhythm and emphasis of expressions, statements, and program structure.

Most common punctuators of Python programming language are :

```
' " # \ ( ) [ ] { } @ , : . ` =
```

1.3 BAREBONES OF A PYTHON PROGRAM

A Python program may contain various elements such as *comments, statements, expressions* etc. Let us talk about the basic structure of a Python program.



```
# This program shows a program's components
# Definition of function SeeYou() follows
def SeeYou() :
    print ("Time to say Good Bye !!")
# Main program-code follows now
a= 15
b= a-10
print (a + 3)
if  b > 5 :                         # colon means it's a block
    print (" Value of 'a' was more than 15 initially.")
else :
    print ("Value of 'a' was 15 or less initially.")
SeeYou()                            # calling above defined function SeeYou()
```

As you can see that the above sample program contains various components like :

- Expressions, which are any legal combination of symbols that ***represents a value***.
- Statements, which are programming instructions.
- Comments, which are the additional readable information to clarify the source code. Comments can be **single line comments**, that start with # and **multi-line comments** that can be either triple-quoted strings or multiple # style comments.
- **Functions**, which are named code-sections and can be reused by specifying their names (*function calls*).
- **Block(s) or suite(s)**, which is a group of statements which are part of another statement or a function. All statements inside a block or suite are indented at the same level.

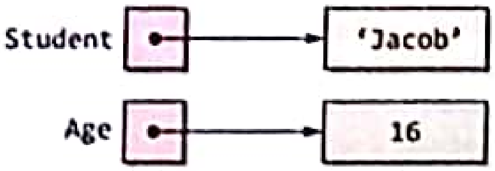
1.4 VARIABLES AND ASSIGNMENTS

Variables represent labelled storage locations, whose values can be manipulated during program run.

In Python, to create a variable, just assign to its name the value of appropriate type. For example, to create a variable namely **Student** to hold student's name and variable **age** to hold student's age, you just need to write somewhat similar to what is shown below :

```
Student = 'Jacob'
Age = 16
```

Python will internally create labels referring to these values as shown below.



1.4.1 Dynamic Typing

In Python, as you have learnt, a variable is defined by assigning to it some value (of a particular type such as numeric, string etc.). For instance, after the statement :

```
X = 10
```

We can say that variable *x* is referring to a value of integer type.

Later in your program, if you reassign a value of some other type to variable *x*, Python will not complain (no error will be raised), *e.g.*,

```
X = 10
print(X)
X = "Hello World"
print(X)
```

Above code will yield the output as :

```
10
Hello World
```

DYNAMIC TYPING

A variable pointing to a value of a certain type, can be made to point to a value/object of different type. This is called *Dynamic Typing*.

So, you can think of Python variables as labels associated with objects (literal values in our case here) ; with dynamic typing, Python makes a label refer to new value with new assignment (Fig. 1.2). Following figure illustrates it.

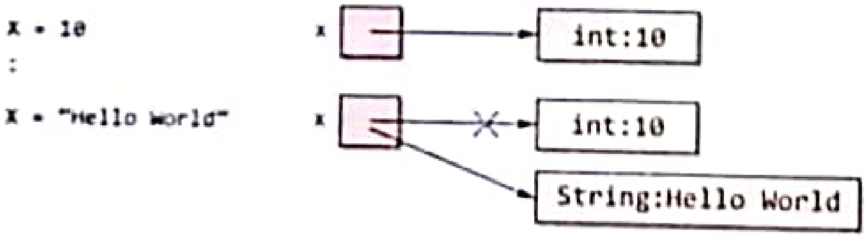


Figure 1.2 Dynamic typing in Python variables.

Dynamic Typing vs. Static Typing

Dynamic typing is different from **Static Typing**. In **Static typing**, a data type is attached with a variable when it is defined first and it is fixed. That is, data type of a variable cannot be changed in *static typing* whereas there is no such restriction in *dynamic typing*, which is supported by Python.

1.4.2 Multiple Assignments

Python is very versatile with assignments. Let's see how.

1. ***Assigning same value to multiple variables.*** You can assign same value to multiple variables in a single statement, *e.g.*,

   ```
   a = b = c = 10
   ```

   It will assign value 10 to all three variables ***a, b, c***.

2. ***Assigning multiple values to multiple variables.*** You can even assign multiple values to multiple variables in single statement, *e.g.*,

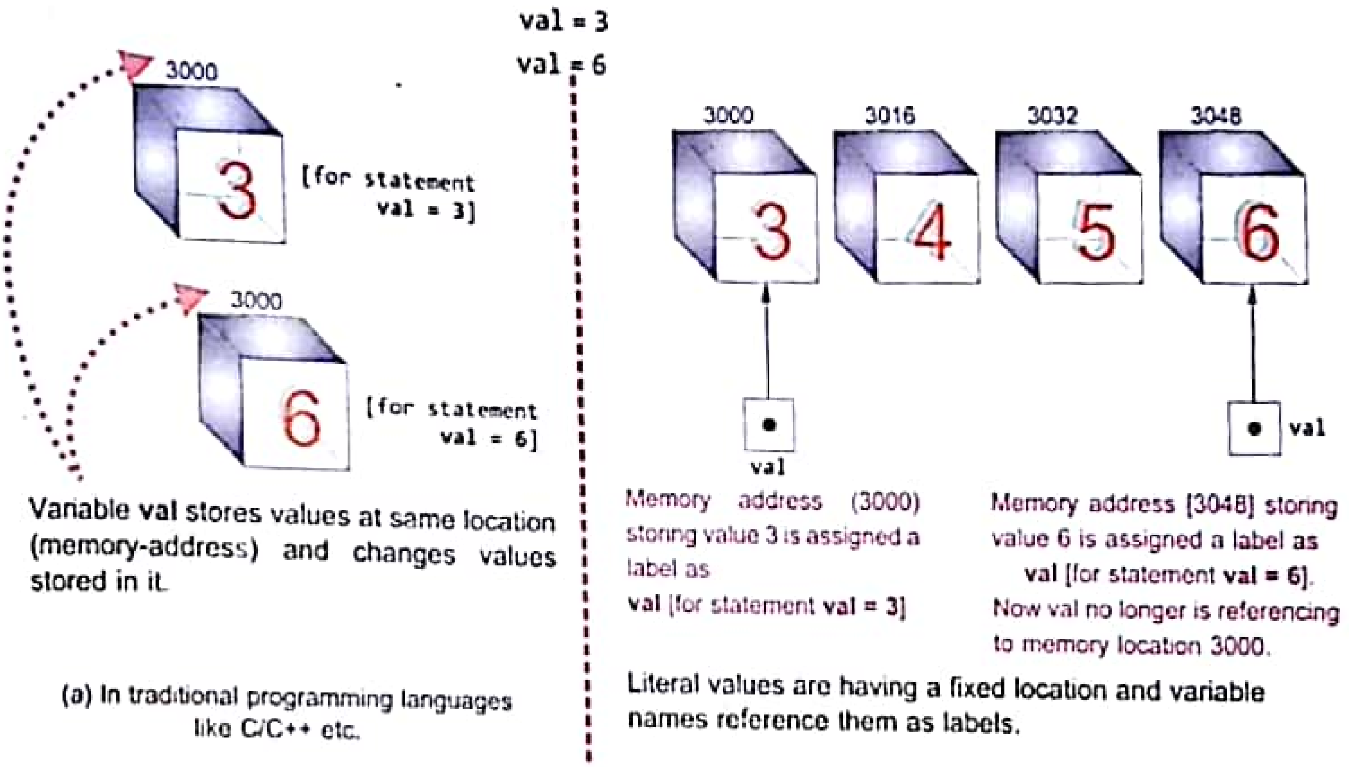
```
x, y, z = 10, 20, 30
```

It will assign the values *order wise, i.e.*, first variable is given first value, second variable the second value and so on. That means, above statement will assign value 10 to *x*, 20 to *y* and 30 to *z*.

If you want to swap values of *x* and *y*, you just need to write as follows :

```
x, y = y, x
```

In Python, assigning a value to a variable means, variable's label is referring to that value.



Variable **val** stores values at same location (memory-address) and changes values stored in it.

(*a*) In traditional programming languages like C/C++ etc.

Memory address (3000) storing value 3 is assigned a label as **val** [for statement **val = 3**]

Memory address [3048] storing value 6 is assigned a label as **val** [for statement **val = 6**]. Now val no longer is referencing to memory location 3000.

Literal values are having a fixed location and variable names reference them as labels.

(*b*) In Python programming language

Figure 1.3 How variables are stored in traditional programming languages and in Python.

1.5 SIMPLE INPUT AND OUTPUT

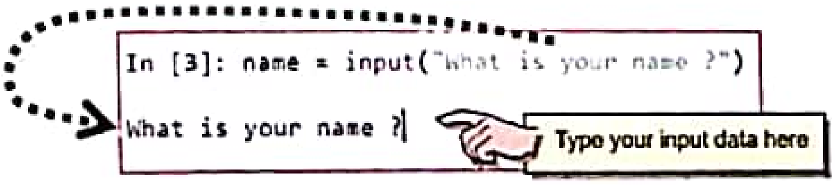
In Python 3.x, to get input from user interactively, you can use built-in function **input( )**. The function **input( )** is used in the following manner :

```
variable_to_hold_the_value = input (<prompt to be displayed>)
```

For example,

```
name = input ('What is your name ?')
```

The above statement will display the prompt as :



```
In [3]: name = input("What is your name ?")

What is your name ?
```

The **input( )** *function always returns a value of String type*. Python offers two functions **int( )** and **float( )** to be used with **input( )** to convert the values received through **input( )** into **int** and **float** types. You can :

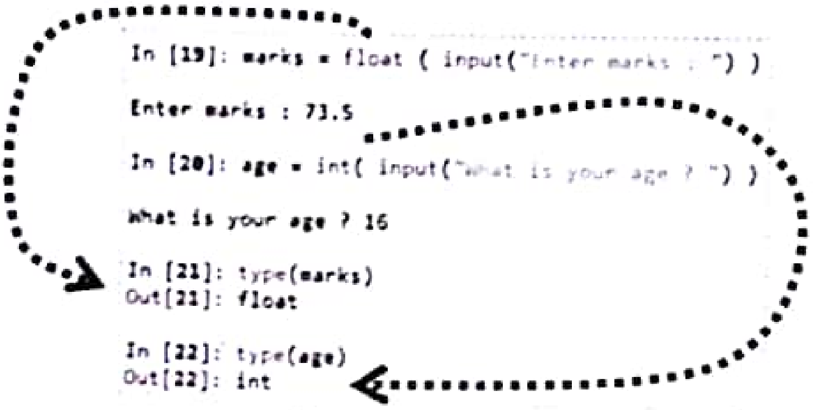
- Read in the value using **input( )** function.
- And then use **int( )** or **float( )** function with the *read value* to change the type of input value to **int** or **float** respectively.

You can also combine these two steps in a single step too, *i.e.*, as :

```
<variable_name> = int( input( <prompt string>) )
```

Or

```
<variable_name> = float( input( <prompt string>) )
```



```
In [19]: marks = float ( input("Enter marks : ") )

Enter marks : 73.5

In [20]: age = int( input("What is your age ? ") )

What is your age ? 16

In [21]: type(marks)
Out[21]: float

In [22]: type(age)
Out[22]: int
```

Function int( ) around input( ) converts the read value into int type and function float( ) around input( ) function converts the read value into float type.

While inputting integer values using **int( )** with **input( )**, make sure that the value being entered must be **int** type compatible. Similarly, while inputting floating point values using **float( )** with **input( )**, make sure that the value being entered must be **float** type compatible (*e.g.*, *'abc'* cannot be converted to *int* or *float*, hence it is not compatible).

Output Through print( ) Statement

The print( ) function of Python 3.x is a way to send output to standard output device, which is normally a monitor.

The simplified syntax to use print( ) function is as follows :

```
print(*objects, [ sep = ' ' or <separator-string> end = '\n' or <end-string> ])
```

*objects means it can be one or multiple comma separated *objects* to be printed.

Let us consider some simple examples first :

```
print ("hello")                      # a string
print (17.5)                         # a number
print (3.14159*(r*r))                # the result of a calculation, which will
                                     # be performed by Python and then printed
                                     # out (assuming that some number has been
                                     # assigned to the variable r)
print ("I\'m", 12 + 5,"years old.")  # multiple comma separated expressions
```

The print statement has a number of features :

- it auto-converts the items to strings *i.e.*, if you are printing a numeric value, it will automatically convert it into equivalent string and print it ; for numeric expressions, it first evaluates them and then converts the result to string, before printing.
- it inserts spaces between items automatically because the default value of ***sep*** argument (separator character) is space(' ').

  Consider this code :

  ```
  print ("My", "name", "is", "Amit.")
  ```

  Four different string objects with no space in them are being printed.

  will print

  ```
  My name is Amit.
  ```

  *But the output line has automatically spaces inserted in between them because default **sep** character is a space.*

  You can change the value of separator character with **sep** argument of **print( )** as per this :

  The code :

  ```
  print ("My", "name", "is", "Amit.", sep = '...' )
  ```

  will print

  ```
  My...name...is...Amit.
  ```

  *This time the **print( )** separated the items with given **sep** character, which is '...'*

- it appends a newline character at the end of the line unless you give your own **end** argument. Consider the code given below :

  ```
  print ("My name is Amit.")
  print("I am 16 years old")
  ```

  It will produce output as :

  ```
  My name is Amit.
  I am 16 years old
  ```

NOTE

A print( ) function without any value or name or expression prints a blank line.

If you explicitly give an end argument with a **print( )** function then the **print( )** will print the line and end it with the string specified with the end argument, and not the newline character, *e.g.*, the code

```
print("My name is Amit. ", end = '$')
print("I am 16 years old. ")
```

will print output as :

```
My name is Amit. $I am 16 years old.
```

*This time the **print( )** ended the line with given **end** character, which is '$' here. And because it was not **newline**, next line was printed from here itself.*

Program 1.1 *Write a program to input a number and print its cube.*

```
num = float(input('Enter a number: '))
num_cube = num * num * num
print('The cube of', num, 'is', num_cube)
```

Program 1.2 *Write a program to input a number and print its square root.*

```
num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of', num, 'is', num_sqrt)
```

PriP

PYTHON : BASIC INPUT/OUTPUT — Progress In Python 1.1

This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 1.1 under Chapter 1 after practically doing it on the computer.

>>>❖<<<

1.6 DATA TYPES

Data types are means to identify type of data and set of valid operations for it. Python offers following built-in core data types : (*i*) *Numbers* (*ii*) *String* (*iii*) *List* (*iv*) *Tuple* (*v*) *Dictionary*.

(*i*) Data types for Numbers

Python offers following data types to store and process different types of numeric data :

(*a*) Integers

- ❖ Integers (signed)
- ❖ Booleans

(*b*) Floating-Point Numbers

(*c*) Complex Numbers

(*a*) **Integers.** There are *two* types of integers in Python :

(*i*) **Integers (signed).** It is the normal integer representation of whole numbers. *Python* 3.*x* provides single data type (*int*) to store any integer, whether *big* or *small*.

It is signed representation, *i.e.*, the integers can be positive as well as negative.

(*ii*) **Booleans.** These represent the truth values *False* and *True*. The Boolean type is a subtype of plain integers, and Boolean values *False* and *True* behave like the values 0 and 1, respectively.

(*b*) **Floating Point Numbers.** In Python, floating point numbers represent machine-level **double precision floating point numbers (15 digit precision).** The range of these numbers is limited by underlying machine architecture subject to available (virtual) memory.

(*c*) **Complex Numbers.** Python represents complex numbers in the form $A + B\,j$. Complex numbers are a composite quantity made of two parts : *the real part* and the *imaginary part*, both of which are represented internally as ***float*** values (floating point numbers).

You can retrieve the two components using attribute references. For a complex number **z** :

- **z.real** gives the *real part.*
- **z.imag** gives the *imaginary part* as a float, not as a complex value.

NOTE

Python represents complex numbers as a pair of floating point numbers

Table 1.1 *The Range of Python Numbers*

Data type	Range
Integers	an unlimited range, subject to available (virtual) memory only
Booleans	*two* values **True** (1), **False** (0)
Floating point numbers	an unlimited range, subject to available (virtual) memory on underlying machine architecture.
Complex numbers	Same as floating point numbers because the real and imaginary parts are represented as floats

(*ii*) Data Type for Strings

All strings in Python 3.x are sequences of *pure Unicode characters. Unicode* is a system designed to represent every character from every language. A string can hold any type of known characters *i.e., letters, numbers,* and *special characters,* of any known scripted language.

Following are all legal strings in Python :

"abcd" , "1234", '$%^&', '????', "ŠÆËá" , "αβγ", 'दर',
"दर", "تجثشت"

NOTE

Valid string indices are *0, 1, 2*... upto *length-1* in forward direction and *−1, −2, −3*... *−length* in backward direction.

A Python string is a sequence of characters and each character can be individually accessed using its **index**.

(*iii*) Lists

A List in Python represents a group of comma-separated values of any datatype between square brackets *e.g.*, following are some lists :

```
[1, 2, 3, 4, 5]
['a', 'e', 'i', 'o', 'u']
['Neha', 102, 79.5]
```

In list too, the values internally are numbered from 0 (zero) onwards *i.e.,* first item of the list is internally numbered as 0, second item of the list as 1, 3rd item as 2 and so on.

(*iv*) Tuples

Tuples are represented as group of comma-separated values of any date type within parentheses, *e.g.*, following are some tuples :

```
p = (1, 2, 3, 4, 5)
q = (2, 4, 6, 8)
r = ('a', 'e', 'i', 'o', 'u')
h = (7, 8, 9, 'A', 'B', 'C')
```

NOTE

Values of type list are *mutable i.e.,* changeable – one can change / add / delete a list's elements. But the values of type tuple are *immutable i.e.,* non-changable ; one cannot make changes to a tuple.

(v) Dictionaries

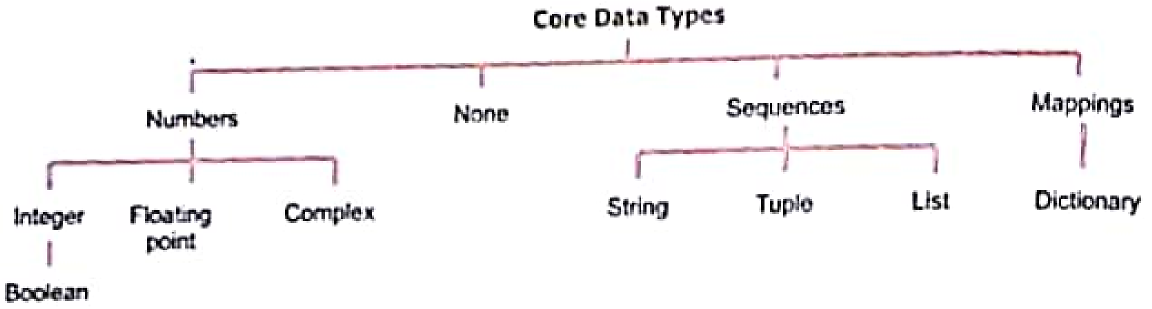
The *dictionary* is an unordered set of comma-separated **key : value pairs**, within { }, with the requirement that within a dictionary, no two keys can be the same (*i.e.*, there are unique keys within a dictionary). For instance, following are some dictionaries :

```
{'a' : 1 , 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
```

```
>>> vowels = {'a' : 1, 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
>>> vowels['a']
1
```

Here 'a', 'e', 'i', 'o' and 'u' are the keys of dictionary vowels; 1, 2, 3, 4, 5 are values for those keys respectively.

Following figure summarizes the core data types of Python.



1.7 MUTABLE AND IMMUTABLE TYPES

The Python data objects can be broadly categorized into *two* – *mutable* and *immutable* types, in simple words changeable or modifiable and non-modifiable types.

Immutable Types

The immutable types are those that can never change their value *in place*. In Python, the following types are immutable : *integers, floating point numbers, Booleans, strings, tuples.*

In immutable types, the **variable names** are stored as references to a value-object. Each time you change the value, the variable's reference memory address changes. See following explanation for sample code given below :

```
P = 5
q = P
r = 5
          ⋮
P = 10
r = 7
q = r
```

◇ Initially these three statements are executed :

```
p = 5
q = p
r = 5
```

All variables having same value reference and the same value object i.e., p, q, r will all reference same integer

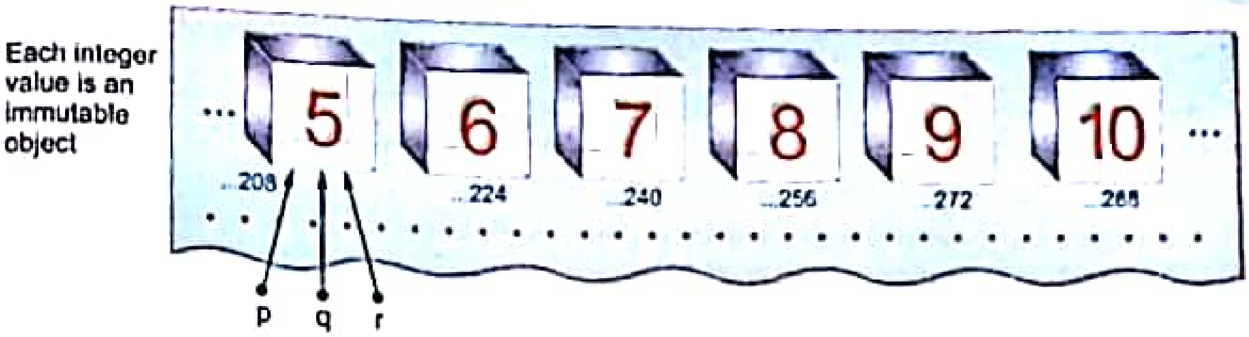


Figure 1.4

◈ When the next set of statements execute, *i.e.*,

```
p = 10
r = 7
q = r
```

then these variable names are made to point to different integer objects.

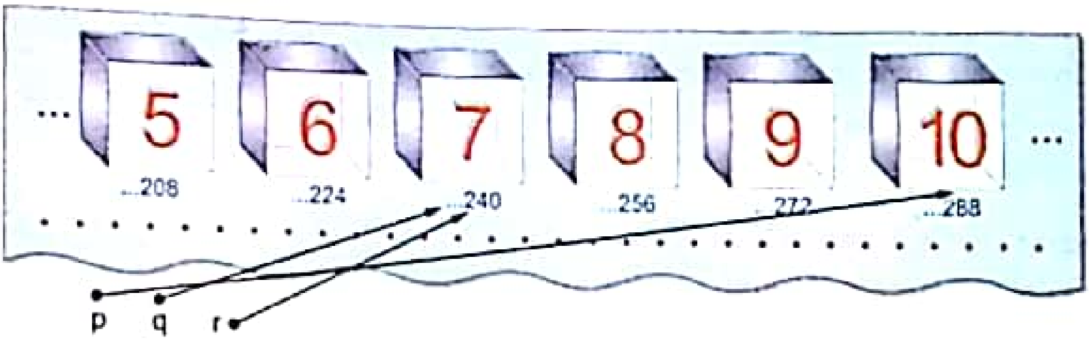


Figure 1.5

Mutable Types

Mutability means that in the same memory address, new value can be stored as and when you want. The types that do not support this property are **immutable types**.

The mutable types are those whose values can be **changed in place**. Only three types are mutable in Python.

These are : *lists, dictionaries* and *sets*.

To change a member of a list, you may write :

```
Chk = [2, 4, 6]
Chk[1] = 40
```

It will make the list namely Chk as [2, 40, 6].

NOTE

Python frontloads some commonly used values in memory. Each variable referring to that value actually stores that memory address of the value. Multiple variables/identifiers can refer to a value. Internally Python keeps count of how many identifiers/variables are referring to a value.

NOTE

Mutable objects are :
list, dictionary, set
Immutable objects:
int, float, complex, string, tuple

1.8 EXPRESSIONS

An expression in Python is any valid combination of *operators, literals* and *variables*. The expressions in Python can be of any type : *arithmetic expressions, string expressions, relational expressions, logical expressions, compound expressions* etc.

EXPRESSION

An *expression* in Python is any valid combination of operators and atoms. An expression is composed of one or more *operations*.

Arithmetic expressions involve numbers (integers, floating-point numbers, complex numbers) and arithmetic operators, *e.g.*, 2 + 5 ** 3, – 8 * 6/5

An expression having literals and/or variables of any valid type and relational operators is a **relational expression**. *For example*, these are valid relational expressions :

```
x > y,  y <= z,  z != x,  z == q,  x < y > z,  x == y != z
```

An expression having literals and/or variables of any valid type and logical operators is a **logical expression**. *For example*, these are valid logical expressions :

```
a or b ,  b and c,  a and not b ,  not c or not b
```

Python also provides two string operators + and *, when combined with string operands and integers, form **string expressions**.

Following are some legal string expressions :

```
"and" + "then"          #would result into 'andthen' - concatenation
"and" * 2               #would result into 'andand' - replication
```

1.8.1 Evaluating Arithmetic Operations

To evaluate an arithmetic expression (with operator and operands), Python follows these rules :

- Determines the order of evaluation in an expression considering the operator precedence.
- As per the evaluation order, for each of the sub-expression (generally in the form of **<value> <operator><value>** *e.g.*, 13 % 3)
  - Evaluate each of its operands or arguments.
  - Performs any implicit conversions (*e.g.*, promoting **int** to **float** or **bool** to **int** for arithmetic on mixed types). For implicit conversion rules of Python, read the text given after the rules.
  - Compute its result based on the operator.
  - Replace the subexpression with the computed result and carry on the expression evaluation.
  - Repeat till the final result is obtained.

In a mixed arithmetic expression, Python converts all operands up to the type of the largest operand (***type promotion***). In the simplest form, an expression is like *op1 operator op2* (*e.g.*, *x/y* or *p* ** *a*). Here, if both arguments are standard numeric types, the following coercions are applied :

- If either argument is a complex number, the other is converted to complex ;
- Otherwise, if either argument is a floating point number, the other is converted to floating point ;
- No conversion if both operands are integers.

NOTE

An implicit type conversion is a conversion performed by the compiler without programmer's intervention.

Table operator precedence

Operator	Description
()	Parentheses (grouping)
**	Exponentiation
~x	Bitwise nor
+x, -x	Positive, negative (unary +, -)
*, /, //, %	Multiplication, division, floor division, remainder
+, -	Addition, subtraction
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
<, <=, >, >=, <>, !=, ==, is, is not	Comparisons (Relational operators), identity operators
not x	Boolean NOT
and	Boolean AND
or	Boolean OR

Highest

Lowest

Example 1.1 *Consider below given expressions what. What will be the final result and final data type ?*

(*a*) `a, b = 3, 6`
`c = b/a`

(*b*) `a, b = 3, 6`
`c = b // a`

(*c*) `a, b = 3, 6.0`
`c = b % a`

Ans. (*a*) In expression

```
c = 6/3        b     /     a
c = 2.0        |           |
              int         int
               |_____ _____|
                     |
                floating pt
```

Here, the operator is /, which always gives floating pt result

(*b*) In expression

```
c = 6 // 3     b    //     a
c = 2          |           |
              int         int
               |_____ _____|
                     |
                    int
```

(*c*) In expression

```
c = 6.0 % 3    b     %     a
c = 0.0        |           |
             float        int
               |_____ _____|
                     |
                   float
```

1.8.2 Evaluating Relational Expressions

All comparison operations in Python have the same priority, which is lower than that of any arithmetic operations. All relational.expressions (comparisons) yield Boolean values only *i.e., True* or *False*.

Further, chained expressions like $a < b < c$ have the interpretation that is conventional in mathematics *i.e.*, comparisons in Python are chained arbitrarily, *e.g.*, $a < b < c$ is internally treated as $a < b$ and $b < c$.

Example 1.2 *How would following relational expressions be internally interpreted by Python ?*

(*i*) $p > q < y$ (*ii*) $a <= N <= b$

Solution. (*i*) $(p > q)$ and $(q < y)$ (*ii*) $(a <= N)$ and $(N <= b)$

1.8.3 Evaluating Logical Expressions

While evaluating logical expressions, Python follows these rules :

(*i*) The precedence of logical operators is lower than the arithmetic operators, so constituent arithmetic sub-expression (if any) is evaluated first and then logical operators are applied, *e.g.*,

`25/5 or 2.0 + 20/10` will be first evaluated as : `5 or 4.0`

So, the overall result will be 5.

(*ii*) The precedence of logical operators among themselves is **not, and, or.** So, the expression *a* or *b* and not *c* will be evaluated as :

`(a or (b and (not c) ))` Similarly, following expression *p* and *q* or not *r* will be evaluated as : `((p and q) or (not r))`

(*iii*) **Important.** While evaluating, Python minimizes internal work by following these rules :

(*a*) In **or** evaluation, Python only evaluates the second argument if the first one is *false*$_{tval}$

(*b*) In **and** evaluation, Python only evaluates the second argument if the first one is *true*$_{tval}$

Example 1.3 *What will be the output of following expression ?*

```
(5 < 10) and (10 < 5) or (3 < 18) and not 8 < 18
```

Solution. False

1.8.4 Type Casting (Explicit Type Conversion)

An explicit type conversion is user-defined conversion that forces an expression to be of specific type. The explicit type conversion is also known as Type Casting.

Type casting in Python is performed by <type>( ) function of appropriate data type, in the following manner :

```
<datatype> (expression)
```

where <datatype> is the data type to which you want to type-cast your expression.

For example, if we have (a = 3 and b = 5.0), then

```
int(b)
```

will cast the data-type of the expression as **int**.

TYPE CASTING

The explicit conversion of an operand to a specific type is called type casting.

1.8.5 Math Library Functions

Python's standard library provides a module namely math for math related functions that work with all number types except for complex numbers.

In order to work with functions of math module, you need to first *import* it to your program by giving statement as follows as the top line of your Python script :

```
import math
```

Then you can use math library's functions as math.<function-name> .

Table 1.2 *Some Mathematical Functions in math Module*

S. No.	Function	Prototype (General Form)	Description	Example
1.	ceil	math.ceil(num)	The ceil( ) function returns the smallest integer not less than *num*.	math.ceil(1.03) gives 2.0 math.ceil(−1.03) gives −1.0
2.	sqrt	math.sqrt (num)	The sqrt( ) function returns the square root of *num*. If num <0, domain error occurs.	math.sqrt(81.0) gives 9.0.
3.	exp	math.exp(arg)	The exp( ) function returns the natural logarithm e raised to the *arg* power.	math.exp(2.0) gives the value of e^2.
4.	fabs	math.fabs (num)	The fabs( ) function returns the absolute value of *num*.	math.fabs(1.0) gives 1.0 math.fabs(−1.0) gives 1.0.
5.	floor	math.floor (num)	The floor( ) function returns the largest integer not greater than *num*.	math.floor(1.03) gives 1.0 math.floor(−1.03) gives −2.0.
6.	log	math.log (num, [base])	The log( ) function returns the natural logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument *num* is zero.	math.log(1.0) gives the natural logarithm for 1.0. math.log(1024, 2) will give logarithm of 1024 to the base 2.
7.	log10	math.log10 (num)	The log10( ) function returns the base 10 logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument is zero.	math.log10(1.0) gives base 10 logarithm for 1.0.
8.	pow	math.pow (base, exp)	The pow( ) function returns *base* raised to *exp* power *i.e., base exp*. A domain error occurs if *base* = 0 and *exp* <= 0 ; also if *base* <0 and *exp* is not integer.	math.pow(3.0, 0) gives value of 3^0. math.pow(4.0, 2.0) gives value of 4^2.
9.	sin	math.sin(arg)	The sin( ) function returns the sine of *arg*. The value of *arg* must be in radians.	math.sin(val) (*val* is a number).
10.	cos	math.cos(arg)	The cos( ) function returns the cosine of *arg*. The value of *arg* must be in radians.	math.cos(val) (*val* is a number).
11.	tan	math.tan(arg)	The tan( ) function returns the tangent of *arg*. The value of *arg* must be in radians.	math.tan(val) (val is a number)
12.	degrees	math.degrees(x)	The degrees( ) converts angle *x* from radians to degrees.	math.degrees(3.14) would give 179.91
13.	radians	math.radians(*x*)	The radians( ) converts angle *x* from degrees to radians.	math.radians(179.91) would give 3.14

DATA HANDLING — Progress In Python 1.2

This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 1.2 under Chapter 1 after practically doing it on the computer.

>>>❖<<<

Let Us Revise

- *A Python program can contain various components like expressions, statements, comments, functions, blocks and indentation.*
- *Blocks are represented through indentation.*
- *Python supports dynamic typing i.e., a variable can hold values of different types at different times.*
- *The input( ) is used to obtain input from user ; it always returns a string type of value.*
- *Output is generated through print( ) (by calling print function) statement.*
- *Operators are the symbols (or keywords sometimes) that represent specific operations.*
- *An expression is composed of one or more operations. It is a valid combination of operators, literals and variables.*
- *Types of operators used in an expression determine its type. For instance, use of arithmetic operators makes it arithmetic expression.*
- *Expressions can be arithmetic, relational or logical, compound etc.*
- *In implicit conversion, all operands are converted up to the type of the largest operand, which is called type promotion or coercion.*
- *The explicit conversion of an operand to a specific type is called type casting and it is done using type conversion functions that is used as*

```
<type conversion function > ( <expression>)
```

1.9 STATEMENT FLOW CONTROL

In a program, statements may be executed sequentially, selectively or iteratively. Every programming language provides constructs to support *sequence, selection* or *iteration.* A **conditional** is a statement set which is executed, on the basis of result of a condition. A **loop** is a statement set which is executed repeatedly, until the end condition is satisfied.

1. Compound Statement

A compound statement represents a group of statements executed as a unit. The compound statements of Python are written in a specific pattern as shown below :

```
<compound statement header > :
        <indented body containing multiple simple and/or compound statements>
```

The conditionals and the **loops** are **compound statements**, ***i.e.,*** **they contain other statements.** For all compound statements, following points hold :

- The contained statements are not written in the same column as the control statement, rather they are indented to the right and together they are called a block.
- The first line of compound statement, *i.e.,* its header contains a colon (:) at the end of it.

For example :

```
num1 = int(input("Enter number1"))
num2 = int(input("Enter number1"))

if num2 < num1 :
    t = num2 * num2
    t = t + 10

print(num2, num1, t)
```

The colon (:) at the end of header line means it is a compound statement

The contained statements within if are indented to the right

2. Simple Statement

Compound statements are made of simple statements. Any single executable statement is a simple statement in Python.

3. Empty Statement

The simplest statement is the empty statement *i.e.,* a statement which does nothing. In Python an empty statement is the *pass* statement.

It takes the following form :

```
pass
```

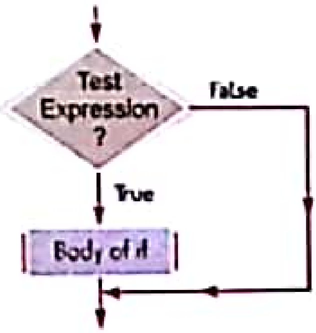
Wherever Python encounters a pass statement, Python does nothing and moves to next statement in the flow of control.

1.10 THE IF CONDITIONALS

The if conditionals of Python come in multiple forms : plain if conditional, **if-else** conditional and **if-elif** conditionals.

1.10.1 Plain if Conditional Statement

An **if** statement tests a particular condition; if the condition evaluates to *true,* a course-of-action is followed *i.e.,* a statement or set-of-statements is executed. If the condition is *false,* it does nothing :



The syntax (general form) of the if statement is as shown below :

```
if <conditional expression> :
    statement
    [statements]
```

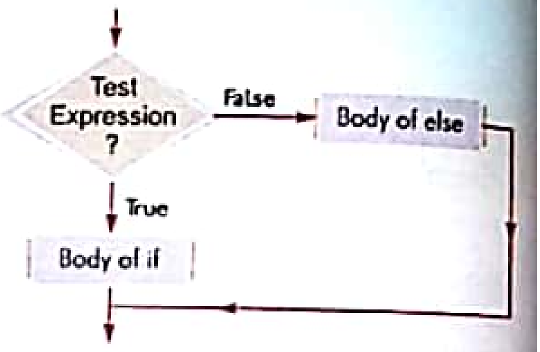
For example, consider the following code fragments using **if** conditionals :

```
if grade == 'A' :
    print("Congratulations! You did well")

:
if a > b :
    print("A has more than B has")
    print("Their difference is", (a-b))
```

1.10.2 The if-else Conditional Statement

This form of **if** statement tests a condition and if the condition evaluates to *true*, it carries out statements indented below **if** and in case condition evaluates to *false*, it carries out statements indented below **else**.

The syntax (general form) of the **if-else** statement is as shown below :



```
if <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```

For example, consider the following code fragments using **if-else** conditionals :

```
if a >= 0 :
    print(a, "is zero or a positive number")
else :
    print(a, "is a negative number")
```

The colon (:) is in both : the if header as well as else line

The statements in if-block and else are indented

1.10.3 The if-elif Conditional Statement

Sometimes, you want to check a condition when control reaches else, *i.e.*, condition test in the form of *else* if. To serve such conditions, Python provides **if-elif** and **if-elif-else** statements.

The general form of these statements is :

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
```

and

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```

Consider following two example code fragments :

```
if runs >= 100 :
    print("Batsman scored a century")
elif runs >= 50 :
    print("Batsman scored a fifty")
else :
    print("Batsman has neither scored a century nor fifty")
:
if num < 0 :
    print(num, "is a negative number.")
elif num == 0 :
    print(num, "is equal to zero.")
else :
    print(num, "is a positive number.")
```

Program 1.3

Write a program that inputs an integer in range 0 - 999 and then prints if the integer entered is a 1/2/3 digit number.

```
num = int(input("Enter a number (0..999) : "))
if num < 0:
    print("Invalid entry. Valid range is 0 to 999.")
elif num < 10:
    print("Single digit number is entered")
elif num < 100:
    print("Two digit number is entered")
elif num <= 999:
    print("Three digit number is entered")
else:
    print("Invalid entry. Valid range is 0 to 999.")
```

Sample runs of this program are given below :

```
Enter a number (0..999) : -3
Invalid entry. Valid range is 0 to 999.
=========================================
Enter a number (0..999) : 4
Single digit number is entered
=========================================
Enter a number (0..999) : 100
Three digit number is entered
=========================================
Enter a number (0..999) : 10
Two digit number is entered
=========================================
Enter a number (0..999) : 3000
Invalid entry. Valid range is 0 to 999.
```

1.10.4 Nested if Statements

Sometimes you may need to test additional conditions. For such situations, Python also supports **nested-if** form of **if**. A nested **if** is an **if** that has another **if** in its **if's** body or in **elif's** body or in its **else's** body.

Consider the following example code using **nested-if statements** :

```
x = int(input("Enter first number :"))
y = int(input("Enter second number :"))
z = int(input("Enter third number :"))

min = mid = max = None
if x < y and x < z :
    if y < z :
        min, mid, max = x, y, z
    else :
        min, mid, max = x, z, y
elif y < x and y < z :
    if x < z :
        min, mid, max = y, x, z
    else :
        min, mid, max = y, z, x
else :
    if x < y :
        min, mid, max = z, x, y
    else:
        min, mid, max = z, y, x
print("Numbers in ascending order :", min, mid, max)
```

if statements inside another if (Nested ifs)

1.10.5 Storing Conditions

Sometimes the conditions being used in code are complex and repetitive. In such cases to make your program more readable, you can use **named conditions** *i.e.*, you can store conditions in a name and then use that named conditional in the **if** statements.

Consider the following example code :

```
b, c = 2, 3

#Store condition in a name called all.
all = a == 1 and b == 2 and c == 3
#Test variable.
if all:
    print("Condition fulfilled")
#Use it again.
if all:
    print("Condition fulfilled again.")
```

this condition is given name as all.

TIP

Using Named or Stored conditionals reduces repetition. It may also improve processing speed: fewer comparisons are done

Using nested if statements, we can rewrite the previous program as follows :

Program 1.4 Write a program that inputs an integer in range 0 - 999 and then prints if the integer entered is a 1/2/3 digit number. Use Nested if statements.

```
num = int(input("Enter a number (0..999) : "))
if num < 0 or num > 999:
    print("Invalid entry. Valid range is 0 to 999.")
else:
    if num < 10:
        print("Single digit number is entered")
    else :
        if num < 100:
            print("Two digit number is entered")
        else :
            print("Three digit number is entered")
```

The sample run of this program is just the same as previous program.

1.11 LOOPING STATEMENTS

Python provides *two* kinds of loops : *for loop* and *while loop* to represent counting loop and conditional loop respectively.

1.11.1 The for Loop

The **for loop** of Python is designed to process the items of any sequence, such as a list or a string, one by one.

The General Form of for loop

The general form of **for loop** is as given below :

```
for <variable> in <sequence> :
    statements_to_repeat
```

For example, consider the following loop :

```
for element in [10, 15, 20, 25] :
    print(element + 2, end = ' ')
```

The above loop would give output as :

```
12  17  22  27
```

- The above-given **for loop** executes a sequence of statements for each of the elements of given sequence [10, 15, 20, 25].
- To keep the count, it uses a control variable (*element* in above case) in that takes a different value on each iteration. Firstly value 10, then the next value of sequence, 15, then 20 and lastly 25.

The *range( )* based for loop

For number based lists, you can specify **range( )** function to represent a list as in :

```
for val in range(3, 18) :
    print(val)
```

In the above loop, **range(3, 18)** will first generate a list [3, 4, 5, ..., 16, 17] with which for loop will work. You need not define loop variable (**val** above) beforehand in a for loop.

As mentioned, a **range( )** produces an integer number sequence. There are *three* ways to define a range :

```
range(stop)                 #elements from 0     to stop-1, incrementing by 1
range(start, stop)          #elements from start to stop-1, incrementing by 1
range(start, stop, step)    #elements from start to stop-1, incrementing by step
```

- The ***start value*** is always part of the range. The ***stop value*** is never part of the range. The ***step*** shows the difference between two consecutive values in the range.
- If ***start value*** is omitted, it supposed to be 0. If the ***step*** is omitted, it is supposed to be 1.

Consider some examples of **range( )** as given below :

`range(7)`	`0, 1, 2, 3, 4, 5, 6`
`range(5, 12)`	`5, 6, 7, 8, 9, 10, 11`
`range(5, 13, 2)`	`5, 7, 9, 11`
`range(10, 4)`	`no value`
`range(10, 4, -1)`	`10, 9, 8, 7, 6, 5`

For example, the next program shows the cube of the numbers from 15 to 20 :

Program 1.5 *Write a program to print cubes of numbers in the range 15 to 20.*

```
for i in range(15, 21):
    print("Cube of number", i, end = ' ')
    print("is", i ** 3)
```

```
Cube of number 15 is  3375
Cube of number 16 is  4096
Cube of number 17 is  4913
Cube of number 18 is  5832
Cube of number 19 is  6859
Cube of number 20 is  8000
```

Program 1.7 Write a program to print square root of every alternate number in the range 1 to 10.

```
for i in range(1, 10, 2) :
    print("square root of", i, "is", (i ** 0.5))
```

1.11.2 The while Loop

A while loop is a conditional loop that will repeat the instructions within itself as long as a conditional remains true (Boolean *True* or truth value *true*).

The general form of Python while loop is :

```
while <logicalExpression> :
      loop-body
```

where the loop-body may contain a single statement or multiple statements or an *empty statement (i.e.,* pass statement). The loop iterates while the *logical Expression* evaluates to *true*. When the expression becomes *false*, the program control passes to the line after the loop-body.

Program 1.8 Write a program that multiplies two integer numbers without using the * operator, using repeated addition.

```
n1 = int(input("Enter first number :"))
n2 = int(input("Enter second number :"))
product = 0
count = n1
while count > 0 :
      count = count - 1
      product = product + n2
print("The product of", n1, "and", n2, "is", product)
```

```
Enter first number : 4
Enter second number : 5
The product of 4 and 5 is 20
```

In general, the while loop is used when it is not possible to know in advance how many times the loop will be executed, but the termination condition is known. The while loop is an entry-controlled loop as it has a control over entry in the loop in the form of test condition.

1.12 JUMP STATEMENTS – break AND continue

Python offers two jump statements – **break** and **continue** – to be used within loops to jump out of loop-iterations.

The *break* Statement

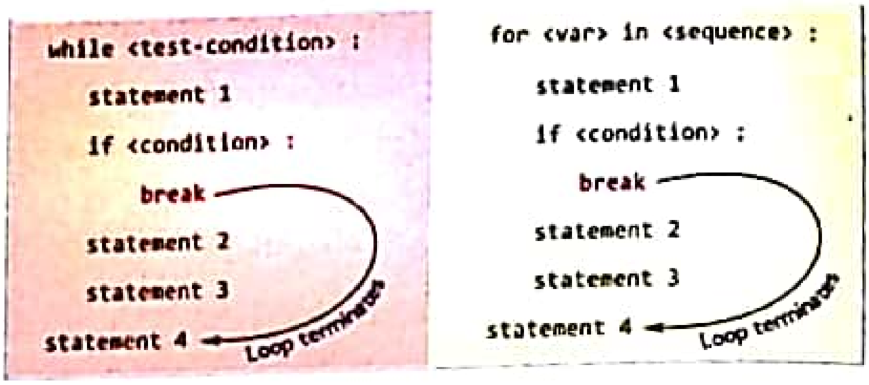
A break statement terminates the very loop it lies within. Execution resumes at the statement immediately following the body of the terminated statement.

NOTE

A *break* statement skips the rest of the loop and jumps over to the statement following the loop.

The following figure (Fig. 1.6) explains the working of a *break* statement.

The following code fragment gives you an example of a break statement :



```
while <test-condition> :
    statement 1
    if <condition> :
        break
    statement 2
    statement 3
statement 4
```

```
for <var> in <sequence> :
    statement 1
    if <condition> :
        break
    statement 2
    statement 3
statement 4
```

Figure 1.6 The working of a break statement.

```
a = b = c = 0
for i in range(1, 11) :
    a = int(input ("Enter number 1 :"))
    b = int(input ("Enter number 2 :"))
    if b == 0 :
        print("Division by zero error! Aborting!")
        break
    else :
        c = a / b
        print("Quotient = ", c)
print("Program over !")
```

range (1, 11) will generate a sequence of numbers from 1 .. 10.

The above code fragment intends to divide ten pairs of numbers by inputting two numbers *a* and *b* in each iteration. If the number *b* is zero, the loop is immediately terminated displaying message 'Division by zero error! Aborting!' otherwise the numbers are repeatedly input and their quotients are displayed.

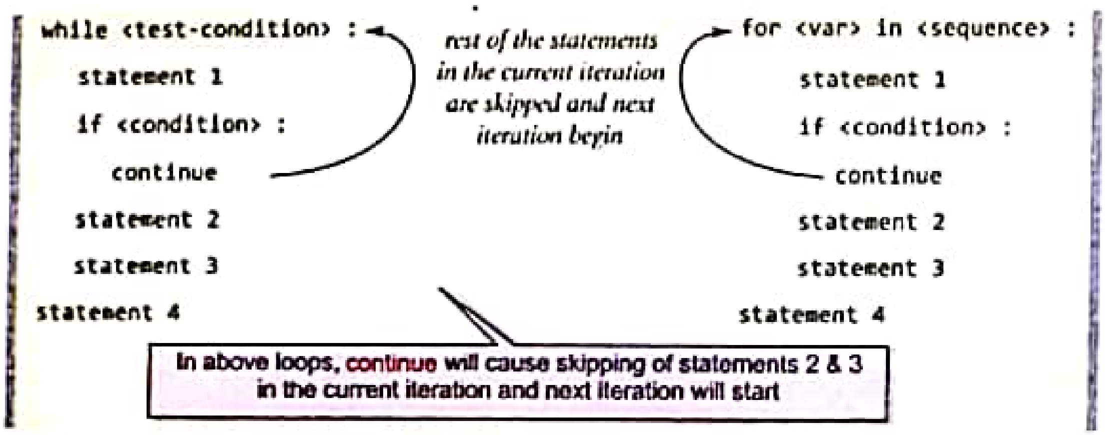
The continue Statement

Unlike **break** statement, the continue statement forces the next iteration of the loop to take place, skipping any code in between.

The following figure (Fig. 1.7) explains the working of *continue* statement :

NOTE

The continue statement skips the rest of the loop statements and causes the next iteration of the loop to take place.



```
while <test-condition> :
    statement 1
    if <condition> :
        continue
    statement 2
    statement 3
statement 4
```

```
for <var> in <sequence> :
    statement 1
    if <condition> :
        continue
    statement 2
    statement 3
statement 4
```

rest of the statements in the current iteration are skipped and next iteration begin
In above loops, continue will cause skipping of statements 2 & 3 in the current iteration and next iteration will start

Figure 1.7 The working of a continue statement.

1.13 MORE ON LOOPS

There are two more things you need to know about loops – the *loop else clause* and nested loops.

1.13.1 Loop else Statement

Python loops have an optional else clause. Complete syntax of Python loops along with else clause is as given below :

```
for <variable> in <sequence> :
    statement1
    statement2
    :
else :
    statement(s)
```

```
while <test condition> :
    statement1
    statement2
    :
else :
    statement(s)
```

The else clause of a Python loop executes when the loop terminates normally, *i.e.*, when test-condition results into *false* for a *while loop* or *for loop* has executed for the last value in the sequence; not when the break statement terminates the loop

Following figure (1.8) shows you control flow in Python loops.

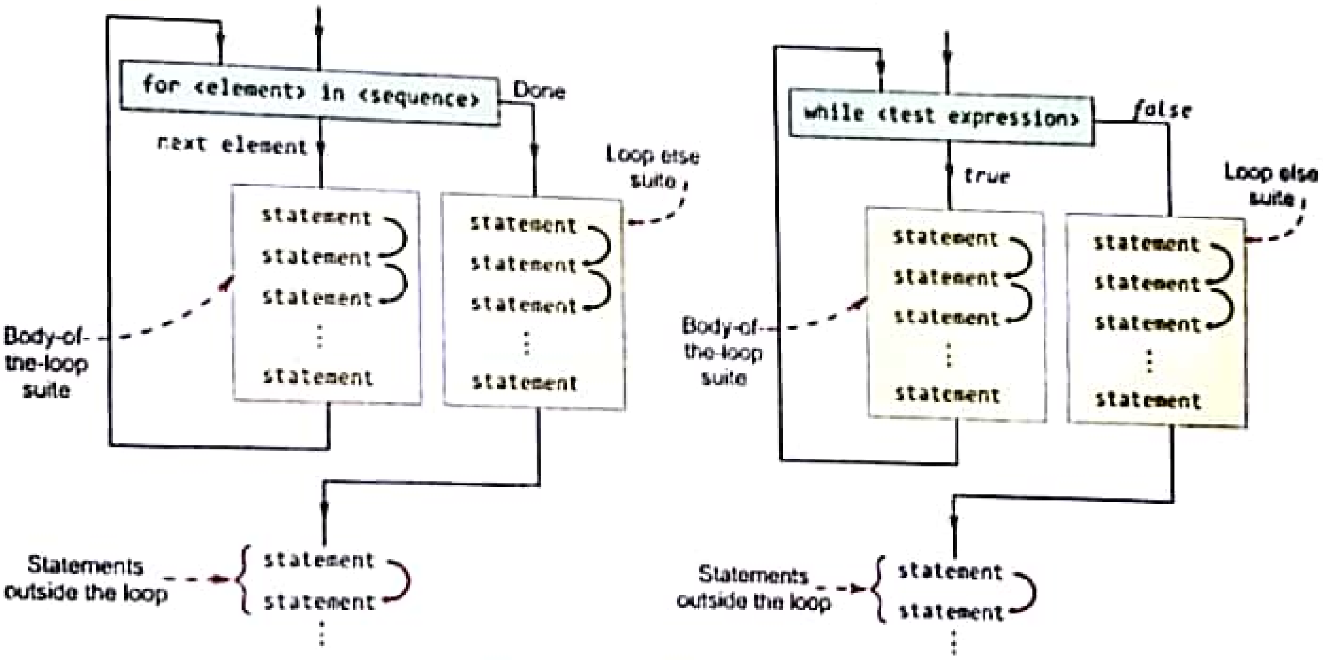


Figure 1.8 Control flow in Python loops.

Consider following example :

```
for a in range (1, 4) :
    if a % 8 == 0 :
        break
    print("Element is", end = ' ')
    print(a)
else :
    print("Ending loop after printing all elements of sequence")
```

The above will give the following output :

```
Element is 1
Element is 2
Element is 3
Ending loop after printing all elements of sequence.
```

This line is printed because the else clause of given for loop executed when the for loop was terminating, normally.

If, in above code, you change the line

```
        if a%8==0 :
```

with

```
        if a%2==0 :
```

Then the output will be :

```
Element is 1
```

Notice that for a = 2, the break got executed and loop terminated. Hence just one element got printed (for a = 1). As break terminated the loop, the else clause also did not execute, so no line after the printing of elements.

NOTE

The else clause of a Python loop executes when the loop terminates normally, not when the loop is terminating because of a break statement.

The **else clause** works identically in while loop, *i.e.*, executes if the test-condition goes *false* and in case of break statement, the **loop-else clause** is not executed.

1.13.2 Nested Loops

A loop may contain another loop in its body. This form of a loop is called nested loop. But in a nested loop, the inner loop must terminate before the outer loop.

The following is an example of a nested loop :

```
for i in range(1, 6) :
    for j in range (1, i ) :
        print("*", end = ' ')
print( )
```

In nested loops, a **break** statement will terminate the very loop it appears in. That is, if **break** statement is inside the inner loop then only the inner loop will terminate and outer loop will continue. If however, the **break** statement is in outer loop, the outer loop will terminate.

PriP CONTROL FLOW STATEMENTS

Progress In Python 1.3

This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 1.3 under Chapter 1 after practically doing it on the computer.

>>>❖<<<

Let Us Revise

- *Statements are the instructions given to the computer to perform any kind of action.*
- *Python statements can be on one of these types : empty statement, single statement and compound statement.*
- *A compound statement represents a group of statements executed as a unit.*
- *Every compound statement of Python has a header and an indented body below the header. Some examples of compound statements are : functions, if statement, while statement etc.*
- *Python provides one selection statement **if** in many forms – **if..else** and **if..elif..else**.*
- *The statements that allow a set of instructions to be performed repeatedly are iteration statements.*
- *Python provides two looping constructs – for and while. The for loop is a counting loop and while is a conditional loop.*
- *The while loop is an entry-controlled loop as it has a control over entry in the loop in the form of test condition.*
- *Loops in Python can have **else** clause too. The else clause of a loop is executed in the end of the loop only when loop terminates normally.*
- *The **break** statement can terminate a loop immediately and the control passes over to the statement following the statement containing break.*
- *In nested loops, a **break** statement terminates the very loop it appears in.*
- *The **continue** statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the beginning of the next iteration of the loop.*

Solved Problems

1. *What is the difference between a keyword and an identifier ?*

   Solution. **Keyword** is a special word that has a special meaning and purpose. Keywords are reserved and are a few. *For example*, if, elif, else etc. are keywords.

   Identifier is the user-defined name given to a part of a program *viz.* variable, object, function etc. Identifiers are not reserved. These are defined by the user but they can have letters, digits and a symbol underscore. They must begin with either a letter or underscore. *For instance, _chk, chess, trial* etc. are identifiers in Python.

2. *What are literals in Python ? How many types of literals are allowed in Python ?*

   Solution. Literals mean constants *i.e.*, the data items that never change their value during a program run. Python allows *five* types of literals :

   (*i*) String literals (*ii*) Numeric literals
   (*iii*) Boolean literals (*iv*) Special Literal *None*
   (*v*) Literal Collections like tuples, lists etc.

3. *How many ways are there in Python to represent an integer literal ?*

   Solution. Python allows *three* types of integer literals :

   (*a*) Decimal (base 10) integer literals
   (*b*) Octal (base 8) integer literals
   (*c*) Hexadecimal (base 16) integer literals

(a) **Decimal Integer Literals.** An integer literal consisting of a sequence of digits is taken to be decimal integer literal unless it begins with 0 (digit zero).

For instance, 1234, 41, +97, −17 are decimal integer literals.

(b) **Octal Integer Literals.** A sequence of digits starting with 0 (digit zero) is taken to be an octal integer. *For instance*, decimal integer 8 will be written as 010 as octal integer. $(8_{10} = 10_8)$ and decimal integer 12 will be written as 014 as octal integer $(12_{10} = 14_8)$.

(c) **Hexadecimal Integer Literals.** A sequence of digits preceded by *0x* or *0X* is taken to be an hexadecimal integer.

For instance, decimal 12 will be written as 0XC as hexadecimal integer.

Thus number 12 will be written either as 12 (as decimal), 014 (as octal) and 0XC (as hexadecimal).

4. *How many types of strings are supported in Python ?*

Solution. Python allows two string types :

(i) *Single line Strings*	Strings that are terminated in single line
(ii) *Multiline Strings*	Strings storing multiple lines of text.

5. *What is None literal in Python ?*

Solution. Python has one special literal called None.

The None literal is used to indicate something that has not yet been created. It is a legal empty value in Python.

6. *The following code is not giving desired output. We want to input value as 20 and obtain output as 40. Could you pinpoint the problem ?*

```
Number = input( "Enter Number" )
DoubleTheNumber = Number * 2
Print (DoubleTheNumber)
```

Solution. The problem is that input( ) returns value as a string, so the input value 20 is returned as string '20' and not as integer 20. So the output is 2020 in place of required output 40.

Also Print is not legal statement of Python ; it should be print.

7. *Why is following code giving errors ?*

```
name = "Rehman"
print("Greetings !!!")
     print("Hello", name)
     print("How do you do ?")
```

Solution. The problem with above code is inconsistent indentation. In Python, we cannot indent a statement unless it is inside a suite and we can indent only as much as required.

Thus, corrected code will be :

```
name = "Rehman"
print("Greetings !!!")
print("Hello", name)
print("How do you do ?")
```

8. *What are data types ? What are Python's built-in core data types ?*

Solution. The real life data is of many types. So to represent various types of real-life data, programming languages provide ways and facilities to handle these, which are known as *data types*.

Python's built-in core data types belong to :

- Numbers (integer, floating-point, complex numbers, Booleans)
- String
- List
- Tuple
- Dictionary

9. *Which data types of Python handle Numbers ?*

Solution. Python provides following data types of handle numbers (version 3.x) :

(*i*) Plain integers
(*ii*) Long integers
(*iii*) Boolean
(*iv*) Floating-point numbers
(*v*) Complex numbers

10. *Why is Boolean considered a subtype of integers ?*

Solution. Boolean values *True* and *False* internally map to integers *1* and *0*. That is, internally *True* is considered equal to 1 and *False* as equal to 0 (zero). When 1 and 0 are converted to Boolean through *bool( )* function, they return *True* and *False*. That is why Booleans are treated as a subtype of integers.

11. *What is the role of comments and indentation in a program ?*

Solution. Comments provide explanatory notes to the readers of the program. Compiler or interpreter ignores the comments but they are useful for specifying additional descriptive information regarding the code and logic of the program.

Indentation makes the program more readable and presentable. Its main role is to highlight nesting of groups of control statements.

12. *What is a statement ? What is the significance of an empty statement ?*

Solution. A statement is an instruction given to the computer to perform any kind of action.

An empty statement is useful in situations where the code requires a statement but logic does not. To fill these two requirements simultaneously, empty statement is used.

Python offers pass statement as an empty statement.

13. *If you are asked to label the Python loops as determinable or non-determinable, which label would you give to which loop ? Justify your answer.*

Solution. The 'for loop' can be labelled as ***determinable loop*** as number of its iterations can be determined beforehand as the size of the sequence, it is operating upon.

The 'while loop' can be labelled as ***non-determinable loop***, as its number of iterations cannot be determined beforehand. Its iterations depend upon the result of a test-condition, which cannot be determined beforehand.

14. *There are two types of else clauses in Python. What are these two types of else clauses ?*

Solution. The two types of Python else clauses are :

(*a*) else in an if statement
(*b*) else in a loop statement

The *else* clause of an *if* statement is executed when the condition of the if statement results into *false*.

The *else* clause of a *loop* is executed when the loop is terminating normally *i.e.*, when its test-condition has gone *false* for a *while* loop or when the *for* loop has executed for the last value in sequence.

15. *Write a program that asks the user to input number of seconds and then expresses it in terms of many minutes and seconds it contains.*

Solution.

```
#get the number of seconds from the user
numseconds = input("Enter number of seconds")
numseconds_int = int(numseconds)

# extract the number of minutes using integer division
numminutes = numseconds_int//60

# extract the number of seconds remaining since the last minute using the modulo
remainingseconds = numseconds_int % 60

print('minutes:', numminutes)
print('seconds:', remainingseconds)
```

16. *Write a program that repeatedly asks from users some numbers until string 'done' is typed. The program should print the sum of all numbers entered.*

Solution.

```
#Unknown Number of Numbers to Sum
total = 0
s = input('Enter a number or "done": ')
while s != 'done' :
    num = int(s)
    total = total + num
    s = input('Enter a number or "done": ')
print('The sum of entered numbers is', total)
```

17. *Write a program to print a square multiplication table as shown below :*

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
5    10   15   20   25   30   35   40   45
6    12   18   24   30   36   42   48   54
7    14   21   28   35   42   49   56   63
8    16   24   32   40   48   56   64   72
9    18   27   36   45   54   63   72   81
```

Solution.

```
for row in range(1, 10):
    for col in range(1, 10):
        prod = row * col
        if prod < 10:
            print(' ', prod,' ', end = ' ')    #This adds a space if the number is single digit
        else:
            print(prod, ' ', end = ' ')
print( )
```

2

Python Revision Tour-II

In This Chapter

2.1 INTRODUCTION

The purpose of *Revision Tour chapters* is to brush up all the concepts you have learnt in class XI. The previous chapter – Python Revision Tour I – covered basic concepts like : Python basics, Data handling, and control flow statements. This chapter is going to help you recall and brush up concepts like *Strings, Lists, Tuples* and *Dictionaries.*

2.2 STRINGS IN PYTHON

Strings in Python are stored as individual characters in contiguous locations, with two-way index for each location. Consider following figure (Fig. 2.1).

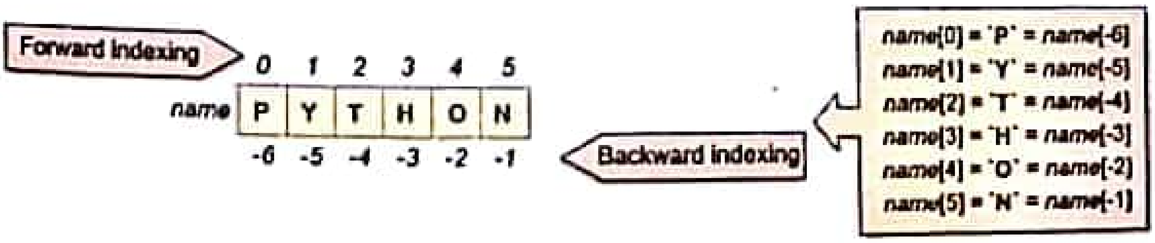


Figure 2.1 Structure of a Python String.

From Fig. 2.1 you can infer that :

- Strings in Python are stored by storing each character separately in contiguous locations.
- The characters of the strings are given two-way indices :
  - 0, 1, 2, ... size-1 in the *forward direction* and
  - -1, -2, -3, ... -size in the *backward direction*.

Thus, you can access any character as **<stringname>[<index>]** *e.g.*, to access the first character of string name shown in Fig. 2.1, you'll write **name[0]**, because the index of first character is 0. You may also write **name [-6]** for the above example *i.e.*, when string name is storing "PYTHON".

Length of string variable can be determined using function **len(<string>)**, *i.e.*, to determine length of above shown string name, you may write :

```
len(name)
```

which will give you 6, indicating that string name stores *six* characters.

2.2.1 Item Assignment not Supported

One important thing about Python strings is that you cannot change the individual letters of a string by assignment because strings are immutable and hence item assignment is not supported, *i.e.*,

```
name = 'hello'
name[0] = 'p'
```

will cause an error like :

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    name[0] = 'p'
TypeError: 'str' object does not support item assignment
```

2.2.2 Traversing a String

Traversing refers to iterating through the elements of a string, one character at a time.

To traverse through a string, you can write a loop like :

```
code = "Powerful"
for ch in code :
    print(ch, '~', end = ' ')
```

NOTE

Traversing refers to iterating through the elements of a string, one character at a time.

The above code will print :

```
P ~ o ~ w ~ e ~ r ~ f ~ u ~ l ~
```

2.2.3 String Operators

In this section, you'll be learning to work with various operators that can be used to manipulate strings in multiple ways.

1. String Concatenation Operator +

The + operator creates a new string by joining the two operand strings, *e.g.*,

```
"power" + "ful"
```

will result into

```
'powerful'
```

Caution!

The + operator has to have both operands of the same type either of number types (for addition) or of string types (for multiplication). It cannot work with one operand as string and one as a number.

2. String Replication Operator *

To use a * operator with strings, you need two types of operands – a string and a number, *i.e.*, as number * string or string * number, where ***string operand*** tells the string to be replicated and ***number operand*** tells the number of times, it is to be repeated; Python will create a new string that is a number of repetitions of the string operand.

For example,

```
3 * "Ha!"
```

will return

```
'Ha!Ha!Ha!'
```

Caution!

The * operator has to either have both operands of the *number* types (for multiplication) or one *string* type and one *number* type (for replication). It cannot work with both operands of string types.

3. Membership Operators

There are *two* membership operators for strings (in fact, for all sequence types). These are **in** and **not in**.

in	returns *True* if a character or a substring exists in the given string ; *False* otherwise
not in	returns *True* if a character or a substring does not exist in the given string ; *False* otherwise

Both membership operators (when used with strings), require that both operands used with them are of string type, *i.e.*,

```
<string> in <string>
<string> not in <string>
```

Now, let's have a look at some examples :

```
>>> "a" in "heya"
True
>>> "jap" in "heya"
False
```

```
>>> "jap" in "Japan"
False
>>> sub = "help"
False
>>> sub2 not in string
True
>>> string = 'helping hand'
>>> sub2 = 'HELP'
>>> sub in string
True
>>> sub2 in string
False
>>> sub not in string
False
```

4. Comparison Operators

Python's standard comparison operators *i.e.*, all relational operators (<, <=, >, >=, ==, !=,) apply to strings also. The comparisons using these operators are based on the standard character-by-character comparison rules for ASCII or Unicode (*i.e.*, dictionary order). Thus, you can make out that :

```
"a" == "a"          will give     True
"abc" == "abc"      will give     True
"a" != "abc"        will give     True
"A" != "a"          will give     True
"ABC" == "abc"      will give     False (letters' case is different)
"abc" != "Abc"      will give     True  (letters' case is different)
```

String comparison principles are :

- Strings are compared on the basis of lexicographical ordering (ordering in dictionary).
- Upper-case letters are considered smaller than the lower-case letters.

Determining ASCII/Unicode Value of a Single Character

Python offers a built-in function **ord( )** that takes a single character and returns the corresponding ASCII value or Unicode value :

```
ord(<single-character>)   #returns ASCII value of passed character
```

The opposite of **ord( )** function is **chr( )**, *i.e.*, while **ord( )** returns the ASCII value of a character, the **chr( )** takes the ASCII value in integer form and returns the character corresponding to that ASCII value.

The general syntax of **chr( )** function is :

```
chr(<int>)          # Gives character corresponding to passed ASCII
                    # value given as integer
```

Consider these examples :

```
>>> ord('A')
65
>>> ord(u'A')
65
>>> chr(65)
'A'
>>> chr(97)
'a'
```

2.2.4 String Slices

The term 'string slice' refers to a part of the string, where strings are sliced using a range of indices. That is, for a string say name, if we give name[n : m] where n and m are integers and legal indices, Python will return a slice of the string by returning the characters falling between indices *n* and *m* : starting at *n*, *n* + 1, *n* + 2 ...till *m* – 1.

STRING SLICE

String Slice refers to part of a string containing some contiguous characters from the string.

NOTE

For any index n, s[:n] + s[n:] will give you original string s.

Following figure (Fig. 2.2) shows some string slices using the string :

```
helloString ="Hello World" :
```

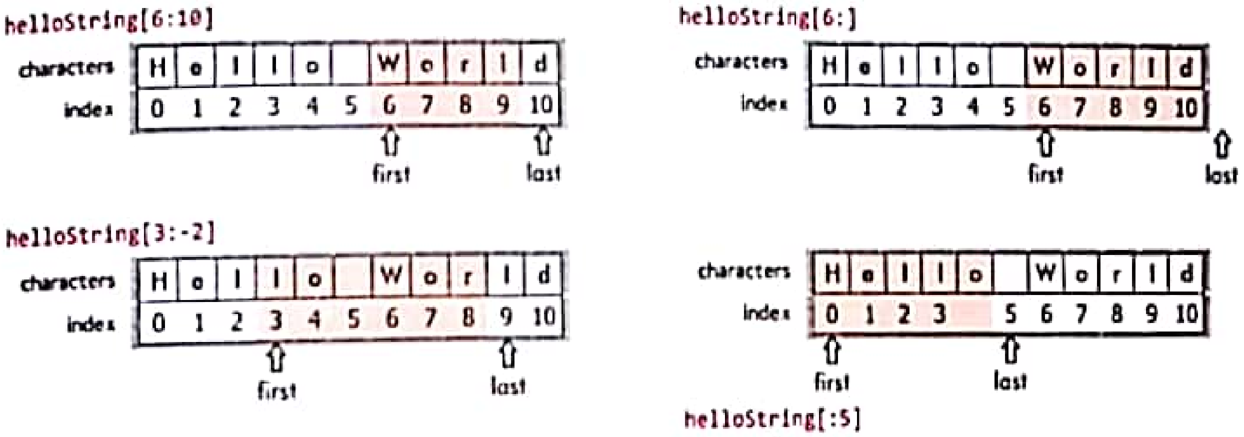


Figure 2.2 String Slicing in Python.

Interesting Inference

Using the same string slicing technique, you will find that

- for any index *n*, s[:n] + s[n:] will give you original string s.

This works even for *n* negative or out of bounds.

Consider the string namely word storing *'amazing'*.

```
>>> word[3:], word [:3]
'zing' 'ama'
```

```
>>> word[:3] + word[3:]
'amazing'
>>> word[:-7], word [-7:]
'' 'amazing'
>>> word[:-7] + word[-7:]
'amazing'
```

NOTE

String [: : –1] is an easy way to reverse a string.

◇ Index out of bounds causes error with strings but slicing a string outside the bounds does not cause error.

```
s = "Hello"
print (s[5])
```

Will cause error because 5 is invalid index-out of bounds, for string "Hello"

But if you give

```
s = "Hello"
print (s[4 : 8])
print (s[5 : 10])
```

One limit is outside the bounds (length of Hello is 5 and thus valid indexes are 0-4)

Both limits are outside the bounds

the above will not give any error and print output as :

```
o
```

empty string

i.e., letter o followed by empty string in next line.

2.2.5 String Functions

Python also offers many built-in functions and methods for string manipulation. The string manipulation methods that are being discussed below can be applied to strings as per following syntax :

```
<stringObject>.<method name> ()
```

For instance, if you have a string namely str = "Rock the World" and you want to find its length, you will write the code somewhat like shown below :

```
>>> str = "Rock the World."
>>> str.length( )
15
>>> str2 = "New World"
>>> str2.length( )
9
```

see the string object is str and method name is length( ).

Do you know that following websites and web applications have used Python extensively : ***Instagram, Dropbox, Google, Netflix, Spotify, Quora, Reddit, Facebook,*** **and many others ?**

Table 2.1 *Python's built-in string manipulation methods*

string.capitalize()	Returns a copy of the *string* with its first character capitalized.
	Example
	`>>> ' i love my India'.capitalize()`
	`I love my India`

string.find (sub[, start[, end]])	Returns the lowest index in the *string* where the substring *sub* is found within the slice range of *start* and *end*. Returns -1 if *sub* is not found.
	Example
	`>>> string = 'it goes as - ringa ringa roses'`
	`>>> sub = 'ringa'`
	`>>> string.find(sub, 15, 22)`
	`-1`
	`>>> string.find(sub, 15, 25)`
	`19`

string.isalnum()
string.isalpha()
string.isdigit()
	Returns True if the characters in the *string* are alphanumeric (alphabets or numbers) and there is at least one character, False otherwise.
	Returns True if all characters in the *string* are alphabetic and there is at least one character, False otherwise.
	Returns True if all the characters in the *string* are digits. There must be at least one digit, otherwise it returns False.
	Examples
	`>>> string = "abc123"`
	`>>> string2 = 'hello'`
	`>>> string3 = '12345'`
	`>>> string4 = ' '`
	`>>> string.isalnum()`
	`True`
	`>>> string2.isalnum()`
	`True`
	`>>> string3.isalnum()`
	`True`
	`>>> string4.isalnum()`
	`False`
	`>>> string.isalpha()`
	`False`
	`>>> string2.isalpha()`
	`True`
	`>>> string3.isalpha()`
	`False`
	`>>> string4.isalpha()`
	`False`
	`>>> string.isdigit()`
	`False`
	`>>> string2.isdigit()`
	`False`
	`>>> string3.isdigit()`
	`False`
	`>>> string4.isdigit()`
	`True`

string.isspace()	Returns True if there are only whitespace characters in the *string*. There must be at least one character. It returns False otherwise.
	Example
	`>>> string = "   "         # stores three spaces`
	`>>> string2 = ""          # an empty string`
	`>>> string.isspace()`
	`True`
	`>>> string2.isspace()`
	`False`

`string.islower()`

Returns **True** if all cased characters in the *string* are lowercase. There must be at least one cased character. It returns **False** otherwise.

`string.isupper()`

Tests whether all cased characters in the *string* are uppercase and requires that there be at least one cased character. Returns **True** if so and **False** otherwise.

Examples

```
>>> string = 'hello'
>>> string2 = 'THERE'
>>> string3 = 'Goldy'
>>> string.islower()
True
>>> string2.islower()
False
>>> string3.islower()
False
```

```
>>> string = "HELLO"
>>> string2 ="There"
>>> string3 = "goldy"
>>> string.isupper()
True
>>> string2.isupper()
False
>>> string3.isupper()
False
>>> string4.isupper()
True
>>> string5.isupper()
False
```

`string.lower()`

Returns a copy of the *string* converted to lowercase. Example

```
>>> string.lower()          #string = "HELLO"
'hello'
```

`string.upper()`

Returns a copy of the *string* converted to uppercase. Example

```
>>> string.upper()          #string = "hello"
'HELLO'
```

`string.lstrip([chars])`

Returns a copy of the *string* with leading characters removed.

If used without any argument, it removes the leading whitespaces.

One can use the optional chars argument to specify a set of characters to be removed.

The chars argument is not a prefix ; rather, all combinations of its values (all possible substrings from the given string argument chars) are stripped when they lead the *string*.

`string.rstrip([chars])`

Returns a copy of the *string* with trailing characters removed.

If used without any argument, it removes the leading whitespaces.

The chars *argument* is a *string* specifying the set of characters to be removed.

The chars argument is not a suffix; rather, all combinations of its values are stripped.

Examples

```
>>> string2 = 'There'
'There'
>>> string2.lstrip( 'The' )
're'
>>> "saregamapadhanisa".lstrip( "tears" )
'gamapadhanisa'
>>> string2.rstrip('care')
'Th'
>>> "saregamapadhanisa".rstrip( "tears" )
'saregamapadhani'
```

'The', 'Th', 'he', 'Te', 'T', 'h', 'e'' and their reversed strings are matched, if any of these found, is removed from the left of the string 'The' found , hence removed

Program 2.1 *Program that reads a line and prints its statistics like :*

Number of uppercase letters :
Number of lowercase letters:
Number of alphabets :
Number of digits:

```
line = input( "Enter a line :" )
lowercount = uppercount = 0
digitcount = alphacount = 0
for a in line :
    if a.islower() :
        lowercount += 1
    elif a.isupper() :
        uppercount += 1
    elif a.isdigit() :
        digitcount += 1
    if a.isalpha() :
        alphacount += 1
print("Number of uppercase letters :" , uppercount)
print("Number of lowercase letters :", lowercount)
print("Number of alphabets :", alphacount)
print("Number of digits :", digitcount)
```

```
Enter a line : Hello 123, ZIPPY zippy Zap
Number of uppercase letters :  7
Number of lowercase letters : 11
Number of alphabets :  18
Number of digits :  3
```

2.3 LISTS IN PYTHON

A list is a standard data type of Python that can store a sequence of values belonging to any type. The Lists are depicted through square brackets, *e.g.*, following are some lists in Python :

```
[]                          #list with no member, empty list
[1, 2, 3]                   #list of integers
[1, 2.5, 3.7, 9]            #list of numbers (integers and floating point)
['a', 'b', 'c']             # list of characters
['a', 1, 'b', 3.5, 'zero']  # list of mixed value types
['One', 'Two', 'Three']     # list of strings
```

Lists are mutable (*i.e.*, modifiable) *i.e.*, you can change elements of a list in place. List is one of the two mutable types of Python – *Lists* and *Dictionaries* are mutable types ; all other data types of Python are immutable.

2.3.1 Creating Lists

To create a list, put a number of expressions, separated by commas in square brackets. That is, to create a list you can write in the form given below :

```
L = []
L = [value, ...]
```

This construct is known as a list display construct.

Creating Empty List

The empty list is [ ]. You can also create an empty list as :

```
L = list()                    #It will generate an empty list and name that list as L.
```

Creating Lists from Existing Sequences

You can also use the built-in list type object to create lists from sequences as per the syntax given below :

```
L = list(<sequence>)
```

where *<sequence>* can be any kind of sequence object including *strings, tuples,* and *lists.*

Consider following examples :

```
>>> l1 = list('hello')          #creating list from string
>>> l1
['h', 'e', 'l', 'l', 'o']
>>> t = ('w', 'e', 'r', 't', 'y')
>>> l2 = list(t)                #creating list from tuple
>>> l2
['w', 'e', 'r', 't', 'y']
```

Creating List from Keyboard Input

You can use this method of creating lists of single characters or single digits via keyboard input. Consider the code below :

```
>>> l1 = list( input('Enter list elements:'))
Enter list elements: 234567
>>> l1
['2', '3', '4', '5', '6', '7']
```

Notice, this way the data type of all characters entered is string even though we entered digits. To enter a list of integers through keyboard, you can use the method given below.

Most commonly used method to input lists is eval(input( ))[1] as shown below :

```
list = eval(input("Enter list to be added :"))
print("List you entered :", list)
```

when you execute it, it will work somewhat like :

```
Enter list to be added : [67, 78, 46, 23]
List you entered : [67, 78, 46, 23]
```

[1] Please note, sometimes (not always) eval( ) does not work in Python shell. At that time, you can run it through a script too.

2.3.2 Lists vs. Strings

Lists and strings have lots in common yet key differences too. This section is going to summarize the similarities and differences between lists and strings.

2.3.2A Similarity between Lists and Strings

Lists are similar to strings in following ways :

- *Length* Function *len*(*L*) returns the number of items (count) in the list L.
- *Indexing and Slicing*

  L[i] returns the item at index *i* (the first item has index 0), and

  L[i:j] returns a new list, containing the objects between *i* and *j*.
- *Membership operators*

  Both '**in**' and '**not in**' operators work on Lists just like they work for other sequences such as strings. (Operator **in** tells if an element is present in the list or not, and **not in** does the opposite.)
- *Concatenation and Replication operators + and **

  The + operator adds one list to the end of another. The * operator repeats a list.
- *Accessing Individual Elements*

  Like strings, the individual elements of a list are accessed through their indexes.

  Consider following examples :

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0]
'a'
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
>>> vowels[-5]
'a'
```

2.3.2B Difference between Lists and Strings

The lists and strings are different from one another in following ways :

- *Storage* Lists are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.
- *Mutability* Strings are not mutable, while lists are. You cannot change individual elements of a string in place, but Lists allow you to do so. That is, following statement is fully valid for Lists (though not for strings) :

```
L[i] = <element>
```

For example, consider the same **vowels** list created above and have a look at following code :

```
>>> vowels[0] = 'A'
>>> vowels
['A', 'e', 'i', 'o', 'u']
>>> vowels[-4] = 'E'
>>> vowels
['A', 'E', 'i', 'o', 'u']
```

Notice, it changed the element in place. ('a' changed to 'A' and 'e' changed to 'E') no new list created – because Lists are MUTABLE.

NOTE

Lists are similar to strings in many ways like indexing, slicing and accessing individual elements but they are different in the sense that Lists are mutable while strings are not.

2.3.3 List Operations

In this section, we shall talk about most common list operations, briefly.

2.3.3A Traversing a List

Traversing a list means accessing and processing each element of it. The *for loop* makes it easy to traverse or loop over the items in a list, as per following syntax :

```
for <item> in <List>:
  process each item here
```

For example, following loop shows each item of a list L in separate lines :

```
L = ['P', 'y', 't', 'h', 'o', 'n']
for a in L :
    print(a)
```

The above loop will produce result as :

```
P
y
t
h
o
n
```

2.3.3B Joining Lists

The concatenation operator +, when used with two lists, joins two lists and returns the concatenated list. Consider the example given below :

```
>>> lst1 = [ 1, 4, 9]
>>> lst2 = [6, 12, 20 ]
>>> lst1 + lst2
[1, 4, 9, 6, 12, 20]
```

The + operator when used with lists requires that both the operands must be of list types.

2.3.3C Repeating or Replicating Lists

Like strings, you can use * operator to replicate a list specified number of times, *e.g.*, (considering the same list lst1 = [1, 3, 5])

```
>>> lst1 * 3
[1, 4, 9, 1, 4, 9, 1, 4, 9]
```

Like strings, you can only use an integer with a * operator when trying to replicate a list.

2.3.3D Slicing the Lists

List slices, like string slices are the sub part of a list extracted out. You can use indexes of list elements to create list slices as per following format :

```
seq = L[start:stop]
```

Consider the following example :

```
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> seq = lst [ 3: -3]
>>> seq
[20, 22, 24]
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> lst [3:30]
[20, 22, 24, 30, 32, 34]
>>> lst [-15 :7]
[10, 12, 14, 20, 22, 24, 30]
```

Giving upper limit way beyond the size of the list, but Python returns elements from list falling in range 3 onwards <30

Giving lower limit much lower, but Python returns elements from list falling in range -15 onwards <7

Lists also support slice steps too. That is, if you want to extract, not consecutive but every other element of the list, there is a way out – the *slice steps*. The *slice steps* are used as per following format :

```
seq = L[start:stop:step]
```

Consider some examples to understand this.

```
>>> lst
[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> lst[0 : 10 : 2]
[10, 14, 22, 30, 34]
>>> lst[2 : 10 : 3]
[14, 24, 34]
```

Include every 2nd element, i.e., skip 1 element in between. Check resulting list slice

Include every 3rd element, i.e., skip 2 elements in between

Using Slices for List Modification

You can use slices to overwrite one or more list elements with one or more other elements. Following examples will make it clear to you :

```
>>> L = ["one", "two", "THREE"]
>>> L[0:2] = [ 0, 1 ]
>>> L
[0, 1, "THREE"]

>>> L =["one", "two", "THREE"]
>>> L[0:2] = "a"
>>> L
["a", "THREE"]
```

NOTE

Like strings, in list slices, you can give start and stop beyond limits of list and it won't raise IndexError, rather it will return the elements falling between specified boundaries.

2.3.4 List Manipulation

You can perform various operations on lists like : *appending, updating, deleting* etc.

Appending Elements to a List

You can also add items to an existing sequence. The **append( )** method **adds a single item to the end of the list**. It can be done as per following format :

```
L.append(item)
```

Consider some examples :

```
>>> lst1 = [10, 12, 14]
>>> lst1.append(16)
>>> lst1
[10, 12, 14, 16]
```

The element specified as argument to append( ) is added at the end of existing list

Updating Elements to a List

To update or change an element of the list in place, you just have to assign new value to the element's index in list as per syntax :

```
L[index] = <new value>
```

Consider following example :

```
>>> lst1 = [10, 12, 14, 16]
>>> lst1[2] = 24
>>> lst1
[10, 12, 24, 16]
```

Statement updating an element (3rd element – having index 2) in the list.

Display the list to see the updated list.

Deleting Elements from a List

You can also remove items from lists. The **del** statement can be used to remove an individual item, or to remove all items identified by a slice.

It is to be used as per syntax given below :

```
del List [ <index>]                # to remove element at index
del List [<start> : <stop> ]       # to remove elements in list slice
```

e.g.,

```
>>> del lst[10:15]
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 18, 19, 20]
```

Delete all elements between indexes 10 to 15 in list namely lst. Compare the result displayed below

If you use **del <lstname>** only *e.g.,* **del lst**, it will delete all the elements and the list object too. After this, no object by the name *lst* would be existing.

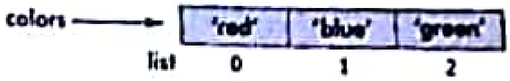
You can also use **pop( )** method to remove single element, not list slices.

The *pop( ) method* is covered in a later section, *List Functions*.

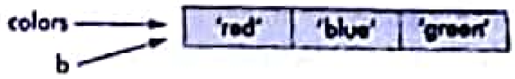
2.3.5 Making True Copy of a List

Assignment with an assignment operator (=) on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory (called *shallow copy*).

```
colors = ['red', 'blue', 'green']
```



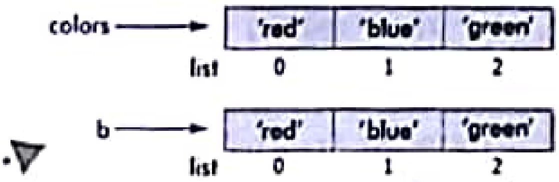
```
b = colors      ## Does not copy the list
```



So, if you make changes in one list, the other list will also report those changes because these two list-names are labels referring to same list because '=' copied the reference not the actual list.

To make **b** true copy of list **colors** *i.e.*, an independent list identical to list **colors** you should create copy of list as follows :

```
b = list(colors)
```



Now *colors* and *b* are separate lists (*deep copy*).

2.3.6 List Functions

Python also offers many built-in functions and methods for list manipulation. These can be applied to list as per following syntax :

```
<listObject>.<method name>()
```

1. The index method

This function returns the index of first matched item from the list.

```
List.index (<item>)
```

For example, for a list L1 = [13, 18, 11, 16, 18, 14],

```
>>> L1.index (18)
1
```

returns the index of first value 18, even if there is another value 18 at index 4.

However, if the given item is not in the list, it raises exception value Error

2. The append method

The *append( )* method adds an item to the end of the list. It works as per following syntax :

```
List.append(<item>)
```

– *Takes exactly one element and returns no value*

For example, to add a new item "yellow" to a list containing colours, you may write :

```
>>> colours = [ 'red', 'green', 'blue']
>>> colours.append('yellow')
>>> colours
['red', 'green', 'blue', 'yellow']
```

See the item got added at the end of the list

The *append*( ) does not return the new list, just modifies the original.

3. The extend method

The *extend( )* method is also used for adding multiple elements (given in the form of a list) to a list.

The *extend( )* function works as per following format :

```
List.extend(<list>)
```

– *Takes exactly one element (a list type) and returns no value*

That is *extend( )* takes a list as an argument and appends all of the elements of the argument list to the list object on which *extend( )* is applied. Consider the following example :

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
>>> t2
['d', 'e']
```

Extend the list t1, by adding all elements of t2

See the elements of list t2 are added at the end of list t1

But list t2 remains unchanged

Difference between append( ) and extend( ) methods

While append( ) function adds one element to a list, *extend( )* can add multiple elements from a list supplied to it as argument.

4. The insert method

The *insert( )* function inserts an item at a given position. It is used as per following syntax :

```
List.insert( <pos>, <item>)
```

– *Takes two arguments and returns no value.*

The first argument <pos> is the index of the element before which the second argument <item> is to be added. Consider the following example:

```
>>> t1 = [ 'a', 'e', 'u']
>>> t1.insert(2, 'i')              # inset element 'i' at index 2.
>>> t1
['a', 'e', 'i', 'u']
```

See element 'i' inserted at index 2

5. The pop method

The *pop( )* is used to remove the item from the list. It is used as per following syntax :

```
List.pop(<index>)                  # <index is optional argument
```

– *Takes one optional argument and returns a value – the item being deleted*

Thus, pop( ) removes an element from the given position in the list, and return it. If no index is specified, pop( ) removes and returns the last item in the list. Consider some examples :

```
>>> t1
['K', 'a', 'e', 'i', 'p', 'q', 'u']
>>> ele1 = t1.pop(0)  ←——  Remove element at index 0 i.e., first
>>> ele1                   element and store it in ele1
'K'   ☜  The removed element

>>> t1
['a', 'e', 'i', 'p', 'q', 'u']  ←——  List after removing first element
>>> ele2 = t1.pop()
>>> ele2
'u'                ←——  No index specified, it will remove
                        the last element
>>> t1
['a', 'e', 'i', 'p', 'q']
```

The pop( ) method raises an exception (runtime error) if the list is already empty.

6. The remove method

The **remove( )** method removes the first occurrence of given item from the list. It is used as per following format :

```
List.remove ( <value> )
```

– *Takes one essential argument and does not return anything*

The remove( ) will report an error if there is no such item in the list. Consider some examples :

```
>>> t1 = ['a', 'e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove('a')
>>> t1            ←——  First occurrence of 'a' is removed from the list
['e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove('p')
>>> t1            ←——  First occurrence of 'p' is removed from the list
['e', 'i', 'q', 'a', 'q', 'p']
```

7. The clear method

This method removes all the items from the list and the list becomes empty list after this function. This function returns nothing. It is used as per following format.

```
List.clear( )
```

For instance, if you have a list L1 as

```
>>> L1 = [2, 3, 4, 5]
>>> L1.clear( )  ←——  it will remove all the items from list L1.
>>> L1
[ ]  ←——  Now the L1 is an empty list.
```

Unlike **del <lstname>** statement, **clear( )** removes only the elements and not the list element. After **clear( )**, the list object still exists as an empty list.

8. The count method

This function returns the count of the item that you passed as argument. If the given item is not in the list, it returns *zero*.

It is used as per following format :

```
List.count(<item> )
```

For instance, for a list L1 = [13, 18, 20, 10, 18, 23]

```
>>> L1.count(18)
2 ←———————— returns 2 as there are two items with value 18 in the list.
>>> L1.count(28)
0 ←———————— No item with value 28 in the list, hence it returned 0 (zero)
```

9. The reverse method

The *reverse( )* reverses the items of the list. This is done **"in place"**, *i.e.*, it does not create a new list.

The syntax to use reverse method is :

```
List.reverse()
```

– *Takes no argument, returns no list ; reverses the list 'in place' and does not return anything.*

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.reverse()
>>> t1      ←———————— The reversed list
['p', 'q', 'a', 'q', 'i', 'e']
```

10. The sort method

The *sort( )* function sorts the items of the list, by default in increasing order. This is done **"in place"**, *i.e.*, it does not create a new list.

It is used as per following syntax :

```
List.sort()
```

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.sort()
>>> t1 ←———————— Sorted list in default ascending order
['a', 'e', 'i', 'p', 'q', 'q']
```

Like reverse( ), sort( ) also performs its function and does not return anything.

To sort a list in decreasing order using sort( ), you can write :

```
>>> List.sort(reverse = True)
```

2.4 TUPLES IN PYTHON

The Tuples are depicted through parentheses *i.e.*, round brackets, *e.g.*, following are some tuples in Python :

```
()                              # tuple with no member, empty tuple
(7,)                            # tuple with one member
(1, 2, 3)                       # tuple of integers
(1, 2.5, 3.7, 9)                # tuple of numbers (integers and floating point)
('a', 'b', 'c')                 # tuple of characters
('a', 1, 'b', 3.5, 'zero')      # tuple of mixed value types
('One', 'Two', 'Three')         # tuple of strings
```

Tuples are **immutable sequences** *i.e.*, you cannot change elements of a tuple in place.

2.4.1 Creating Tuples

To create a tuple, put a number of expressions, separated by commas in parentheses. That is, to create a tuple you can write in the form given below :

```
T = ()
T = (value, ...)
```

This construct is known as a **tuple display construct**.

Creating *Empty* Tuple

The empty tuple is ( ). You can also create an empty tuple as :

```
T = tuple()
```

Creating *Single Element* Tuple*

Making a tuple with a single element is tricky because if you just give a single element in round brackets, Python considers it a value only, *e.g.*,

```
>>> t = (1)
>>> t
1
```

(1) was treated as an integer expression, hence t stores an integer 1, not a tuple

To construct a tuple with one element just add a comma after the single element as shown below :

```
>>> t = 3,
>>> t
(3,)
```

To create a one-element tuple, make sure to add comma at the end

Now t stores a tuple, not integer.

Creating Tuples from Existing Sequences

You can also use the built-in tuple type object (tuple( )) to create tuples from sequences as per the syntax given below :

```
T = tuple(<sequence>)
```

where *<sequence>* can be any kind of sequence object including *strings, lists* and *tuples*.

Consider following examples :

```
>>> t1 = tuple('hello')                    #creating tuple from a string
>>> t1
('h', 'e', 'l', 'l', 'o')
>>> L = ['w', 'e', 'r', 't', 'y']
>>> t2 = tuple(L)                          # creating tuple from a list
>>> t2
('w', 'e', 'r', 't', 'y')
```

Creating Tuple from Keyboard Input

You can use this method of creating tuples of single characters or single digits via keyboard input.

Consider the code below :

```
t1 = tuple(input('Enter tuple elements:'))
Enter tuple elements : 234567
>>> t1
('2', '3', '4', '5', '6', '7')
```

But most commonly used method to input tuples is eval(input( )) as shown below :

```
tuple = eval(input("Enter tuple to be added:"))
print("Tuple you entered :", tuple)
```

when you execute it, it will work somewhat like :

```
Enter tuple to be added: (2, 4, "a", "hjkjl, "[3, 4])
Tuple you entered : (2, 4, "a", "hjkjl", [3, 4])
```

2.4.2 Tuples vs. Lists

Tuples and lists are very similar yet different. This section is going to talk about the same.

2.4.2A Similarity between Tuples and Lists

Tuples are similar to lists in following ways :

- *Length* Function len(T) returns the number of items (count) in the tuple T.
- *Indexing and Slicing*

  T[*i*] returns the item at index *i* (the first item has index 0), and *T*[*i*:*j*] returns a new tuple, containing the objects between *i* and *j*.
- *Membership operators*

  Both **'in'** and **'not in'** operators work on Tuples also. That is, **in** tells if an element is present in the tuple or not and **not in** does the opposite.
- *Concatenation and Replication operators + and **

  The + operator adds one tuple to the end of another. The * operator repeats a tuple.

◆ *Accessing Individual Elements*

The individual elements of a tuple are accessed through their indexes given in square brackets. Consider the following examples :

```
>>> vowels = ('a', 'e', 'i', 'o', 'u')
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
```

2.4.2B Difference between Tuples and Lists

Tuples are not mutable, while *lists* are. You cannot change individual elements of a *tuple* in place, but *lists* allow you to do so. That is, following statement is fully valid for lists (BUT not for tuples).

If we have a list L and a tuple T, then

```
L[i] = element                    # is valid
```

is VALID for Lists. BUT

```
T[i] = element                    # is invalid
```

is INVALID for tuples as you cannot perform item-assignment in immutable types.

2.4.3 Tuple Operations

In this section, we shall talk about most common tuple operations, briefly.

2.4.3A Traversing a Tuple

Traversing a tuple means accessing and processing each element of it. The *for loop* makes it easy to traverse or loop over the items in a tuple, as per following syntax :

```
for <item> in <Tuple>:
    process each item here
```

For example, following loop shows each item of a tuple *T* in separate lines :

```
T = ('P', 'u', 'r', 'e')
for a in T :
    print(T[a])
```

The above loop will produce result as :

```
P
u
r
e
```

2.4.3B Joining Tuples

The + operator, the concatenation operator, when used with two tuples, joins two tuples. Consider the example given below :

```
>>> tpl1 = (1, 3, 5)
>>> tpl2 = (6, 7, 8)
>>> tpl1 + tpl2
(1, 3, 5, 6, 7, 8)
```

IMPORTANT

Sometimes you need to concatenate a tuple (say *tpl*) with another tuple containing only one element. In that case, if you write statement like :

```
>>> tpl + (3)
```

Python will return an error like :

```
TypeError : can only concatenate tuple (not "int") to tuple
```

The reason for above error is : a single value in ( ) is treated as single value not as tuple. That is, expressions (3) and ('a') are integer and string respectively but (3,) and ('a',) are single element tuples. Thus, following expression won't give any error :

```
>>> tpl + (3, )
```

2.4.3C Repeating or Replicating Tuples

Like strings and lists, you can use * operator to replicate a tuple specified number of times, *e.g.*,

```
>>> tpl1 * 3
(1, 3, 5, 1, 3, 5, 1, 3, 5)
```

Like strings and lists, you can only use an integer with a * operator when trying to replicate a tuple.

2.4.3D Slicing the Tuples

Tuple slices, like list-slices or string slices are the sub part of the tuple extracted out. You can use indexes of tuple elements to create tuple slices as per following format :

```
seq = T[start:stop]
```

Recall that index on last limit is not included in the tuple slice. Consider the following example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> seq = tpl [ 3:-3]
>>> seq
(20, 22, 24)
```

If you want to extract, not consecutive but every other element of the tuple, there is a way out – **the slice steps**. The slice steps are used as per following format :

```
seq = T[start:stop:step]
```

Consider some examples to understand this.

```
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl[0 : 10 : 2]
(10, 14, 22, 30, 34)
```

Include every 2nd element, i.e., skip 1 element in between. Check resulting tuple slice

```
>>> tpl[2 : 10 : 3]
(14, 24, 34)
```

Include every 3rd element, i.e., skip 2 element in between

```
>>> tpl[:: 3]
(10, 20, 30)
```

No start and stop given. Only step is given as 3. That is, from the entire tuple, pick every 3rd element for the tuple

2.4.3E Unpacking Tuples

Creating a tuple from a set of values is called *packing* and its reverse, *i.e.*, creating individual values from a tuple's elements is called *unpacking*.

Unpacking is done as per syntax :

```
<variable1>, <variable2>, <variable3>, … = t
```

where the number of variables in the left side of assignment must match the number of elements in the tuple.

For example, if we have a tuple as :

```
t = (1, 2, 'A', 'B')
```

The length of above tuple *t* is 4 as there are four elements in it. Now to unpack it, we can write :

```
w, x, y, z = t
print(w, "-", x, "-", y, "-", z)
```

The output will be :

```
1-2-A-B
```

2.4.4 Tuple Functions and Methods

1. The len( ) method

This method returns length of the tuple, *i.e.*, the count of elements in the *tuple*.

Syntax : `len(<tuple>)`

```
>>> employee = ('John', 10000, 24, 'Sales')
>>> len(employee)
4 ←———————— The len( ) returns the count of elements in the tuple
```

2. The max( ) method

This method returns the element from the tuple having **maximum value.**

Syntax : `max(<tuple>)`

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> max(tpl)
34 ←———————— Maximum value from tuple tpl is returned
```

3. The min( ) method

This method returns the element from the tuple having **minimum value.**

Syntax : `min(<tuple>)`

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> min(tpl)
10 ←———————— Maximum value from tuple tpl is returned
```

NOTE

Like max( ), for min( ) to work, the elements of tuple should be of same type.

4.The index( ) method

It returns the index of an existing element of a tuple.

Syntax : `<tuplename> . index (<item>)`

```
>>> t1 = [3, 4, 5, 6.0]
>>> t1.index(5)
2
```

But if the given item does not exist in tuple, it raises **ValueError** exception.

5. The count( ) function

The count( ) method returns the count of a member element/object in a given sequenc (list/tuple).

Syntax : `<sequence name>.count(<object>)`

```
>>> t1 = (2, 4, 2, 5, 7, 4, 8, 9, 9, 11, 7, 2)
>>> t1.count(2)
3
```

There are 3 occurrences of element 2 in given tuple, hence count() return 3 here

NOTE

With tuple( ), the argument must be a sequence type *i.e.,* a string or a list or a dictionary.

6. The tuple( ) method

This method is actually constructor method that can be used to create tuples from different types of values.

Syntax : `tuple(<sequence>)`

- *Creating empty tuple*

```
>>> tuple()
()
```

- *Creating tuple from a string*

```
>>> t = tuple("abc")
>>> t
('a', 'b', 'c')
```

- *Creating a tuple from a list*

```
>>> t = tuple([1,2,3])
>>> t
(1, 2, 3)
```

- *Creating a tuple from keys of a dictionary*

```
>>> t1 = tuple ( {1:"A", 2:"B"})
>>> t1
(1, 2)
```

PYTHON SEQUENCES : STRINGS, LIST & TUPLES — Progress In Python 2.1

This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.1 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

2.5 DICTIONARIES IN PYTHON

Python dictionaries are a collection of some *key-value* pairs. Dictionaries are mutable, unordered collections with elements in the form of a **key : value** pairs that associate keys to values.

2.5.1 Creating a Dictionary

To create a dictionary, you need to include the **key : value** pairs in curly braces as per following syntax :

```
<dictionary-name> = {<key>:<value>, <key>:<value>...}
```

Following is an example dictionary by the name teachers that stores the names of teachers as keys and the subjects being taught by them as values of respective keys.

```
teachers = { "Dimple" : "Computer Science", "Karen" : "Sociology",
             "Harpreet" : "Mathematics", "Sabah" : "Legal Studies" }
```

Notice that

- the curly brackets mark the beginning and end of the dictionary,
- each entry (*Key : Value*) consists of a pair separated by a colon – the key and corresponding value is given by writing colon (:) between them,
- the key-value pairs are separated by commas (,).

Internally, dictionaries are indexed (*i.e.*, arranged) on the basis of keys.

2.5.2 Accessing Elements of a Dictionary

In dictionaries, the elements are accessed through the *keys* defined in the **key:value** pairs, as per the syntax shown below :

```
<dictionary-name> [ <key>]
```

Thus to access the value for key defined as "Karen" in above declared teachers dictionary, you will write :

```
>>> teachers["Karen"]
```

and Python will return

```
Sociology
```

Attempting to access a key that doesn't exist causes an error. Consider the following statement that is trying to access a non-existent key (13) from dictionary teachers.

```
>>> teachers["Kushal"]
teachers KeyError : 13
```

In Python dictionaries, the elements (key : value pairs) are unordered ; one cannot access elements as per specific order.

Accessing Keys or Values Simultaneously

To see all the keys in a dictionary in one go, you may write **<dictionary>.keys( )** and to see all values in one go, you may write **<dictionary>.values( )**, as shown below :

```
>>> d = {"Vowel1" : "a", "Vowel2" : "e", "Vowel3" : "i", "Vowel4" : "o", "Vowel5" : "u"}
>>> d.keys()
['Vowel5', 'Vowel4', 'Vowel3', 'Vowel2', 'Vowel1']
>>> d.values()
['u', 'o', 'i', 'e', 'a']
```

Python lists keys in an arbitrary order.

2.5.3 Characteristics of a Dictionary

Dictionaries like lists are mutable and that is the only similarity they have with lists. Otherwise, dictionaries are different type of data structures with following characteristics :

(*a*) A dictionary is a unordered set of key : value pairs.

(*b*) Unlike the string, list and tuple, a dictionary is not a sequence because it is unordered set of elements.

(*c*) Dictionaries are indexed by keys and its keys must be of any non-mutable type.

(*d*) Each of the keys within a dictionary must be unique.

(*e*) Like lists, dictionaries are also mutable. We can change the value of a certain key "in place" using the assignment statement as per syntax :

```
<dictionary>[<key>] = <value>
```

2.5.4 Dictionary Operations

In this section, we shall briefly talk about various operations possible on Python dictionaries.

2.5.4A Traversing a Dictionary

Traversal of a collection means accessing and processing each element of it. The *for loop* makes it easy to traverse or loop over the items in a dictionary, as per following syntax :

```
for <item> in <Dictionary> :
    process each item here
```

Consider following example that will illustrate this process. A dictionary namely *d1* is defined with three keys – a number, a string, a tuple of integers.

```
d1 = { 5 : "number", \
    "a" : "string", \
    (1,2) : "tuple" }
```

To traverse the above dictionary, you can write for loop as :

```
for key in d1 :
    print(key, ":", d1[key])
```

The above loop will produce the output as shown below :

```
a : string
(1, 2) : tuple
5 : number
```

2.5.4B Adding Elements to Dictionary

You can add new elements (**key : value pair**) to a dictionary using assignment as per the following syntax. BUT the *key* being added must not exist in dictionary and must be unique. If the *key* already exists, then this statement will change the value of existing *key* and no new entry will be added to dictionary.

```
<dictionary>[<key>] = <value>
```

Consider the following example :

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee['dept'] = 'Sales'
>>> Employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
```

Using this method, you can create new dictionary, if you are adding elements to an empty dictionary.

2.5.4C Updating Existing Elements in a Dictionary

Updating an element is similar to what we did just now. That is, you can change value of an existing key using assignment as per following syntax :

```
dictionary>[<key>] = <value>
```

Consider the following example :

```
>>> Employee = {'name' : 'John' , 'salary' : 10000, 'age' : 24}
>>> Employee['salary'] = 20000
>>> Employee
{'salary' : 20000, 'age' : 24, 'name' : 'John'}
```

But make sure that the *key* must exist in the dictionary, otherwise new entry will be added to the dictionary.

Using this technique of adding key : value pairs, you can create dictionaries interactively at runtime by accepting input from user.

Program 2.2 *Write a program to create a dictionary containing names of competition winner students as keys and number of their wins as values.*

```
n = int(input("How many students ?"))
CompWinners = { }
for a in range(n) :
    key = input("Name of the student :")
    value = int(input("Number of competitions won :"))
    CompWinners[key] = value
print("The dictionary now is :")
print(CompWinners)
```

```
How many students ? 5
Name of the student : Naval
Number of competitions won : 5
Name of the student : Zainab
Number of competitions won : 3
Name of the student : Nita
Number of competitions won : 3
Name of the student : Rosy
Number of competitions won : 1
Name of the student : Jamshed
Number of competitions won : 5
The dictionary now is :
{'Nita' : 3, 'Naval' : 5, 'Zainab' : 3, 'Rosy' : 1, 'Jamshed' : 5}
```

2.5.4D Deleting Elements from a Dictionary

There are two methods for deleting elements from a dictionary.

(*i*) To delete a dictionary element or a dictionary entry, *i.e.*, a **key:value pair**, you can use **del** command. The syntax for doing so is as given below :

```
del <dictionary>[ <key>]
```

Consider the following example :

```
>>> empl3
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> del empl3['age']
>>> empl3
{'salary' : 10000, 'name' : 'John'}
```

But with **del** statement, the key that you are giving to delete must exist in the dictionary, otherwise Python will return an error. See below :

```
>>> del empl3['new']
del empl3['new']
KeyError : 'new'
```

(*ii*) Another method to delete elements from a dictionary is by using pop( ) method as per following syntax :

```
<dictionary>.pop(<key>)
```

The pop( ) method will not only delete the **key:value pair** for mentioned *key* but also return the corresponding value.

Consider the following code example

```
>>> employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> employee.pop('age')
24
>>> employee
{'salary' : 10000, 'name' : 'John'}
```

If you try to delete a *key* which does not exist, the Python returns error. See below :

```
>>> employee.pop('new')
employee.pop('new')
KeyError : 'new'
```

However, pop( ) method allows you to specify what to display when the given *key* does not exist, as per following syntax :

```
<dictionary>.pop(<key>, <in-case-of-error-show-me>)
```

For example :

```
>>> employee.pop('new', "Not Found")
'Not Found'
```

2.5.4E Checking for Existence of a Key

Usual membership operators in and not in work with dictionaries as well. But they can check for the existence of keys only. You may use them as per syntax given below :

```
<key> in <dictionary>
<key> not in <dictionary>
```

- The in operator will return *True* if the given key is present in the dictionary, otherwise *False*.
- The not in operator will return *True* if the given key is not present in the dictionary, otherwise *False*.

Consider the following examples :

```
>>> empl = {'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> 'age' in empl
True
>>> 'John' in empl
False
>>> 'John' not in empl
True
>>> 'age' not in empl
False
```

2.5.5 Dictionary Functions and Methods

1. The len( ) method

This method returns length of the *dictionary* , *i.e.,* the count of elements (*key:value* pairs) in the *dictionary*. The syntax to use this method is given below :

```
len(<dictionary>)
```

e.g.,

```
>>> employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> len(employee)
3
```

2. The clear( ) method

This method removes all items from the *dictionary* and the dictionary becomes empty dictionary post this method.

```
<dictionary>.clear()
```

e.g.,

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee.clear()
>>> Employee
{ }
```

See, now the dictionary is empty

NOTE

The clear( ) removes all the elements of a dictionary and makes it empty dictionary while del statement removes the complete dictionary as an object. After del statement with a dictionary name, that dictionary object no more exists, not even empty dictionary.

3. The get( ) method

With this method, you can get the item with the given key, similar to **dictionary [key]**. If the key is not present, Python will give error.

```
<dictionary>.get( key , [ default ])
```

e.g.,

```
>>> empl1
{'salary' : 10000, 'dept' : 'Sales', 'age': 24, 'name' : 'John'}
>>> empl1.get('dept')
'Sales'
```

4. The items( ) method

This method returns all of the items in the *dictionary* as a sequence of (key, value) tuples. Note that these are returned in no particular order.

```
<dictionary>.items()
```

e.g.,

```
employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
myList = employee.items()
for x in myList :
    print(x)
```

The adjacent code gives output as :

```
( 'salary', 10000)
('age', 24)
( 'name', 'John' )
```

5. The keys( ) method

This method returns all of the *keys* in the *dictionary* as a sequence of *keys* (in form of a list). Note that these are returned in no particular order.

```
<dictionary>.keys()
```

e.g.,

```
>>> employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
>>> employee.keys()
['salary', 'dept', 'age', 'name']
```

6. The values( ) method

This method returns all the values from the *dictionary* as a sequence (a list). Note that these are returned in no particular order.

```
<dictionary>.values()
```

e.g.,

```
>>> employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
>>> employee.values()
[10000, 'Sales', 24, 'John']
```

7. The update( ) method

This method merges *key : value* pairs from the new *dictionary* into the original *dictionary*, adding or replacing as needed. The items in the new dictionary are added to the old one and override any items already there with the same keys.

The syntax to use this method is given below :

Dictionary to be updated

This dictionary's items will be taken for updating other dictionary.

```
<dictionary>.update ( <other-dictionary>)
```

e.g.,

See, the elements of dictionary ***employee2*** *have overridden the elements of dictionary* ***employee1*** *having the same keys i.e., of keys* ***'name'*** *and "salary"*

```
>>> employee1 = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> employee2 = {'name' : 'Diya' , 'salary' : 54000, 'dept' : 'Sales'}
>>> employee1.update(employee2)
>>> employee1
{'salary' : 54000, 'dept' : 'Sales', 'name' : 'Diya', 'age' : 24}
>>> employee2
{'salary' : 54000, 'dept' : 'Sales', 'name' : 'Diya'}
```

Program 2.3 *Given three lists as list1 = ['a','b','c'] , list2 = ['h','i','t'] and list3 = ['0','1','2']. Write a program that adds lists 2 and 3 to list1 as single element each. The resultant list should be in the order of* ***list3, elements of list1, list2.***

```
list1 =['a','b','c']
list2 = ['h','i','t']
list3 = ['0','1', '2']
print("Originally :")
print("List1 = ", list1)
print("List2 = ", list2)
print("List3 = ", list3)

# adding list2 as single element at the end of list1
list1.append(list2)        #list2 gets added as one element at the end of list1
```

```
# adding list3 as single element at the start of list1
list1.insert(0,list3) #list3 gets added as one element at the beginning of list1
print("After adding two lists as individual elements, list now is :")
print(list1)
```

The output produced by above program is as follows :

```
Originally :
List1 = ['a', 'b', 'c']
List2 = ['h', 'i', 't']
List3 = ['0', '1', '2']
After adding two lists as individual elements, list now is :
[['0', '1', '2'], 'a', 'b', 'c', ['h', 'i', 't']]
```

Program 2.4 *Given three lists as list1 = ['a','b','c'], list2 = ['h','i','t'] and list3 = ['0','1','2']. Write a program that adds individual elements of lists 2 and 3 to list1. The resultant list should be in the order of elements of list3, elements of list1, elements of list2.*

```
list1 =['a','b','c']
list2 = ['h','i','t']
list3 = ['0','1', '2']
print("Originally :")
print("List1 = ", list1)
print("List2 = ", list2)
print("List3 = ", list3)

# adding elements of list1 at the end of list3
list3.extend(list1)

# adding elements of list2 at the end of list3
list3.extend(list2)
print("After adding elements of two lists individually, list now is :")
print(list3)
```

The output produced by above program is as follows :

```
Originally :
List1 = ['a', 'b', 'c']
List2 = ['h', 'i', 'y']
List3 = ['0', '1', '2']
After adding elements of two lists individually, list now is :
['0', '1', '2', 'a', 'b', 'c', 'h', 'i', 't']
```

Program 2.5 Write a program that finds an element's index/position in a tuple WITHOUT using index().

```
tuple1 = ('a','p','p','l','e',)
char = input("Enter a single letter without quotes : ")
```

```
        if char in tuple1:
            count = 0
            for a in tuple1:
                if a != char:
                    count += 1
                else:
                    break
            print(char, "is at index", count, "in", tuple1)
        else:
            print(char, "is NOT in", tuple1)
```

```
Enter a single letter without quotes : l
l is at position 3 in ('a', 'p', 'p', 'l', 'e')
==============================================
Enter a single letter : p
p is at position 1 in ('a', 'p', 'p', 'l', 'e')
```

Program 2.6 *Write a program that checks for presence of a value inside a dictionary and prints its key.*

```
info = {'Riya':'CSc.', 'Mark':'Eco', 'Ishpreet':'Eng', 'Kamaal':'Env.Sc'}
inp = input("Enter value to be searched for :")
if inp in info.values():
    for a in info:
        if info[a] == inp :
            print("The key of given value is", a)
            break
    else:
        print("Given value does not exist in dictionary")
```

```
Enter value to be searched for : Env.Sc
The key of given value is Kamaal
======================================
Enter value to be searched for : eng
Given value does not exist in dictionary
```

Program 2.7 *The code of previous will not work if the cases of the given value and value inside dictionary are different. That is, the result (of previous program) will be like :*

```
Enter value to be searched for : eng
Given value does not exist in dictionary
```

Make changes in above program so that the program returns the key, even if the cases differ, i.e., match the two values ignoring their cases

```
info = {'Riya':'CSc.', 'Mark':'Eco', 'Ishpreet':'Eng', 'Kamaal':'Env.Sc'}
inp = input("Enter value to be searched for :")
```

```
for a in info:
    if info[a].upper() == inp.upper() :
        print("The key of given value is", a)
        break
else:
    print("Given value does not exist in dictionary")
```

The sample run of above program is as given below :

```
Enter value to be searched for : eng
The key of given value is Ishpreet
```

2.6 SORTING TECHNIQUES

Sorting in computer terms means arranging elements in a specific order — ascending or increasing order or descending or decreasing order.

There are multiple ways or techniques or algorithms that you can apply to sort a group of elements such as *selection sort, insertion sort, bubble sort, heap sort, quick sort* etc.

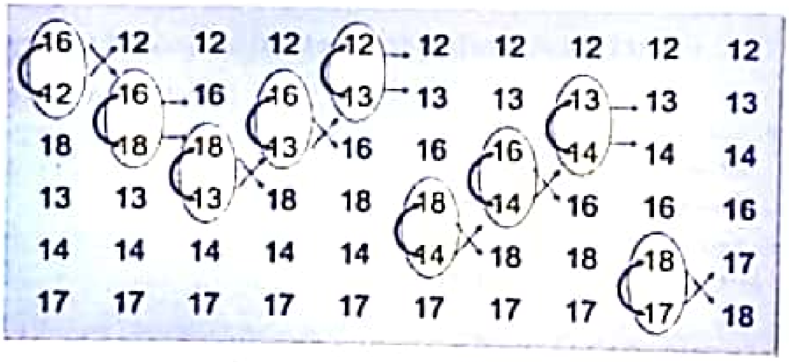
SORTING

Sorting, in computer terms, refers to arranging elements in a specific order — ascending or descending.

We shall cover *Bubble sort* and *insertion sort*, as recommended by syllabus.

2.6.1 Bubble Sort

The basic idea of bubble sort is to compare two adjoining values and exchange them if they are not in proper order. To understand this, have a look at figure 2.3 that visually explains the process of *bubble sort*.



16	12	12	12	12	12	12	12	12	12
12	16	16	16	13	13	13	13	13	13
18	18	18	13	16	16	16	14	14	14
13	13	13	18	18	18	14	16	16	16
14	14	14	14	14	14	18	18	18	17
17	17	17	17	17	17	17	17	17	18

Figure 2.3

Program 2.8 *Program to sort a list using Bubble sort.*

```
aList = [ 15, 6, 13, 22, 3, 52, 2] 
print ("Original list is :", aList)
n = len(aList)
# Traverse through all list elements
for i in range(n):
    # Last i elements are already in place
```

We have taken a pre-initialized list. You can even input a list prior to sorting it.

```
    for j in range(0, n-i-1):
        # traverse the list from 0 to n-i-1
        # Swap if the element found is greater
        # than the next element
        if aList[j] > aList[j + 1] :
            aList[j], aList[j + 1] = aList[j + 1], aList[j]
print ("List after sorting :", aList)
```

This expression will ensure that we do not compare the heavier elements that have already settled at correct position.

The output produced by above code is like :

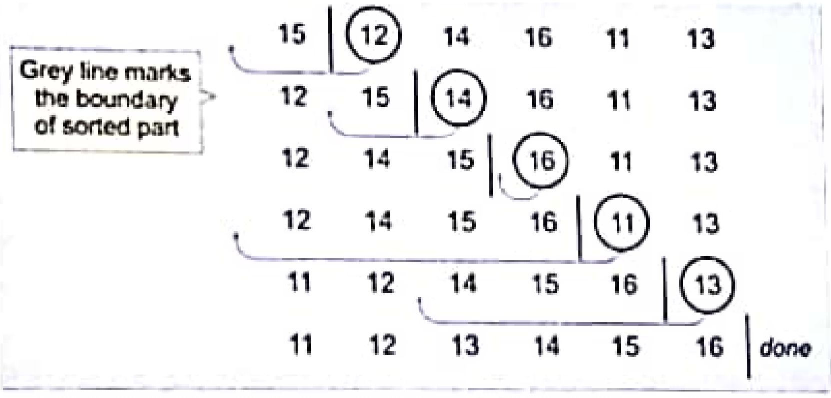
```
Original list is : [15, 6, 13, 22, 3, 52, 2]
List after sorting : [2, 3, 6, 13, 15, 22, 52]
```

INSERTION SORT

Insertion sort is a sorting algorithm that builds a sorted list one element at a time from the unsorted list by inserting the element at its correct position in sorted list.

2.6.2 Insertion Sort

Insertion sort is a sorting algorithm that builds a sorted list one element at a time from the unsorted list by inserting the element at its correct position in sorted list.



15 | 12 14 16 11 13
12 15 | 14 16 11 13
12 14 15 | 16 11 13
12 14 15 16 | 11 13
11 12 14 15 16 | 13
11 12 13 14 15 16 |

Figure 2.4

Program 2.9 Program to sort a sequence using insertion sort.

```
aList = [15, 6, 13, 22, 3, 52, 2]
print ("Original list is :", aList)
for i in range(1, len(aList)) :
    key = aList[i]
    j = i - 1
    while j >= 0 and key < aList[j] :
        aList[j + 1] = aList[j]     #shift elements to right to make room for key
        j = j - 1
    else:
        aList[j + 1] = key
print("List after sorting :", aList)
```

```
Original list is : [15, 6, 13, 22, 3, 52, 2]
List after sorting : [2, 3, 6, 13, 15, 22, 52]
```

PriP PYTHON SEQUENCES : DICTIONARIES, SORTING Progress In Python 2.2

This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.2 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Python strings are stored in memory by storing individual characters in contiguous memory locations.*
- *The index (also called subscript sometimes) is the numbered position of a letter in the string.*
- *In Python, indices begin **0** onwards in the forward direction up to **length-1** and **-1, -2, ...** up to **– length** in the backward direction. This is called **two-way indexing**.*
- *The string slice refers to a part of the string **s[start:end]** is the element beginning at start and extending up to but not including end.*
- *Lists are mutable sequences of Python i.e., you can change elements of a list in place.*
- *Lists index their elements just like strings, i.e., two way indexing.*
- *Lists are similar to strings in many ways like **indexing, slicing and accessing individual elements** but they are different in the sense that Lists are mutable while strings are not.*
- *Membership operator **in** tells if an element is present in the sequence or not and **not in** does the opposite.*
- *List slice is an extracted part of a list; list slice is a list in itself.*
- ***L[start:stop]** creates a list slice out of list L with elements falling between indexes start and stop , not including stop.*
- *Tuples are immutable sequences of Python i.e., you cannot change elements of a tuple in place.*
- *To create a tuple, put a number of comma-separated expressions in round brackets. The empty round brackets i.e., **( )** indicate an empty tuple.*
- *Tuples index their elements just like strings or lists, i.e., **two way indexing**.*
- *Tuples are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.*
- *Tuple slice is an extracted part of tuple; tuple slice is a tuple in itself.*
- ***T[start:stop]** creates a tuple slice out of tuple T with elements falling between indexes start and stop, not including stop.*
- *Dictionaries are mutable with elements in the form of a **key:value** pair that associate keys to values.*
- *The keys of a dictionary must be of immutable types.*
- *In Python dictionaries, the elements (key:value pairs) are unordered ; one cannot access element as per specific order.*

- *Keys of a dictionary must be unique.*
- *In Dictionaries, the updation and addition of elements are similar in syntax. But for addition, the key must not exist in the dictionary and for updation, the key must exist in the dictionary.*
- *Sorting of an array means arranging the array elements in a specified order.*
- *In bubble sort, the adjoining values are compared and exchanged if they are not in proper order. This process is repeated until the entire array is sorted.*
- *In insertion sort, each successive element is picked & inserted at an appropriate position in the previously sorted array.*

Solved Problems

1. *What is indexing in context to Python strings ? Why is it also called two-way indexing ?*

   Solution. In Python strings, each individual character is given a location number, called index and this process is called indexing.

   Python allocates indices in two directions :

   - in forward direction, the indexes are numbered as 0, 1, 2,.... length-1.
   - in backward direction, the indexes are numbered as −1, −2, −3.... length.

   This is known as two-way indexing.

2. *What is a string slice ? How is it useful ?*

   Solution. A sub-part or a slice of a string, say s, can be obtained using s [n : m] where *n* and *m* are integers. Python returns all the characters at indices *n*, *n*+1, *n*+2...*m*−1 *e.g.*,

   ```
   'Well done'[1 : 4]    will give    'ell'
   ```

3. *Figure out the problem with following code fragment. Correct the code and then print the output.*

   ```
   1.  s1 = 'must'
   2.  s2 = 'try'
   3.  n1 = 10
   4.  n2 = 3
   5.  print(s1 + s2)
   6.  print(s2 * n2)
   7.  print(s1 + n1)
   8.  print(s2 * s1)
   ```

   Solution. The problem is with lines 7 and 8.

   - Line 7 – **print(s1 + n1)** will cause error because **s1** being a string cannot be concatenated with a number **n1**.

     This problem can be solved either by changing the operator or operand *e.g.*, all the following statements will work :

     (*a*) `print (s1 * n1)`

     (*b*) `print (s1 + str(n1))`

     (*c*) `print (s1 + s2)`

- **Line 8 – print(s2 * s1)** will cause error because two strings cannot be used for replication. The corrected statement will be :

```
print(s2 + s1)
```

If we replace the Line 7 with its suggested solution (*b*), the output will be :

```
must try
try try try
must 10
try must
```

4. *Consider the following code :*

```
string = input( "Enter a string :" )
count = 3
while True :
    if string[0] == 'a' :
        string = string[2 :]
    elif string[-1] == 'b' :
        string = string [: 2]
    else :
        count += 1
        break
print(string)
print(count)
```

What will be the output produced, if the input is : (i) aabbcc (ii) aaccbb (iii) abcc

Solution.

(*a*) *bbcc* (*b*) *cc* (*c*) *cc*
4 4 4

5. *Consider the following code :*

```
Inp = input("Please enter a string :" )
while len(Inp) <= 4 :
    if Inp[-1] == 'z' :                  #condition 1
        Inp = Inp [0 : 3] + 'c'
    elif 'a' in Inp :                    #condition 2
        Inp = Inp[0] + 'bb'
    elif not int(Inp[0]) :               #condition 3
        Inp = '1' + Inp[1 :] + 'z'
    else :
        Inp = Inp + '*'
print(Inp)
```

What will be the output produced if the input is (i) 1bzz, (ii) ' 1a ' (iii) ' abc ' (iv) ' 0xy ', (v) ' xyz '.

Solution.

(*i*) 1*bzc** (*ii*) 1*bb***

(*iii*) endless loop because '*a*' will always remain at index 0 and condition 3 will be repeated endlessly.

(*iv*) 1×*yc** (*v*) Raises an error as *Inp[0]* cannot be converted to int.

6. *Write a program that takes a string with multiple words and then capitalizes the first letter of each word and forms a new string out of it.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
a = 0
end = length
string2 = ''          #empty string
while a < length :
    if a == 0 :
        string2 += string[0].upper()
        a += 1
    elif (string[a] == '' and string[a+1] != '') :
        string2 += string[a]
        string2 += string[a+1].upper()
        a +=2
    else :
        string2 += string[a]
        a += 1
print("Original String :", string)
print("Captilized words String", string2)
```

7. *Write a program that reads a string and checks whether it is a palindrome string or not.*

Solution.

```
string = input("Enter a string :")
length = len(string)
mid = length/2
rev = -1
for a in range(mid) :
    if string[a] == string[rev] :
        a += 1
        rev -= 1
    else :
        print(string, "is not a palindrome")
        break
else :        #loop else
    print(string, "is a palindrome")
```

8. *How are lists different from strings when both are sequences ?*

Solution. The lists and strings are different in following ways :

(*i*) The lists are mutable sequences while strings are immutable.

(*ii*) In consecutive locations, a string stores the individual characters while a list stores the references of its elements.

(*iii*) Strings store single type of elements – all characters while lists can store elements belonging to different types.

9. *What are nested lists ?*

Solution. When a list is contained in another list as a member-element, it is called nested list, *e.g.*,

```
a = [2, 3, [4, 5]]
```

The above list a has three elements – an integer 2, an integer 3 and a list [4, 5], hence it is nested list.

10. *What does each of the following expressions evaluate to?*

Solution. Suppose that *L* is the list

["These", ["are", "a"], ["few", "words"], "that", "we", "will", "use"].

(*a*) `L[3:4] + L[1:2]`

(*b*) `"few" in L[2:3]`

(*c*) `"few" in L[2]`

(*d*) `L[2][1:]`

(*e*) `L[1] + L[2]`

Solution.

(*a*)
```
L[3:4] = ['that']
L[1:2] = [['are', 'a']]
L[3:4] + L[1:2] = ['that', ['are', 'a']]
```

(*b*) **False.** The string "few" is not an element of this range. *L*[2 : 3] returns a list of elements from *L*–> [['few', 'words']], this is a list with one element, a list.

(*c*) **True.** *L*[2] returns the list ['few', 'words'], "few" is an element of this list.

(*d*)
```
L[2] = ['few', 'words']
L[2][1:] = ['words']
```

(*e*)
```
L[1] = ['are', 'a']
L[2] = ['few', 'words']
L[1] + L[2] = ['are', 'a', 'few', 'words']
```

11. *What is the output produced by the following code snippet ?*

```
aLst =[1,2,3,4,5,6,7,8,9]
print(aLst[::3])
```

Solution.

```
[1,4,7]
```

12. *What will be the output of the following code snippet ?*

```
Lst =[1,2,3,4,5,6,7,8,9]
Lst[::2]=10,20,30,40,50,60
print(Lst)
```

(*a*) *ValueError* : attempt to assign sequence of size 6 to extended slice of size 5

(*b*) [10, 2, 20, 4, 30, 6, 40, 8, 50, 60]

(*c*) [1, 2, 10, 20, 30, 40, 50, 60]

(*d*) [1, 10, 3, 20, 5, 30, 7, 40, 9, 50, 60]

Solution. (*a*)

13. *What will be the output of the following code snippet ?*

```
values = []
for i in range (1,4):
    values.append(i)
    print (values)
```

Solution.

```
[1]
[1, 2]
[1, 2, 3]
```

14. *What will be the output of the following code ?*

```
rec = {"Name" : "Python", "Age":"20"}
r = rec.copy()
print(id(r) == id(rec))
```

(*a*) True (*b*) False (*c*) 0 (*d*) 1

Solution. (*b*)

15. *What will be the output of the following code snippet?*

```
dc1 = {}
dc1[1] = 1
dc1['1'] = 2
dc1[1.0] = 4

sum = 0
for k in dc1:
    sum += dc1[k]
print (sum)
```

Solution. 6

16. *Predict the output of following code fragment :*

```
fruit = {}
f1 = ['Apple','Banana', 'apple', 'Banana']
for index in f1 :
    if index in fruit:
        fruit[index] += 1
    else:
        fruit[index] = 1
    print(fruit)
print (len(fruit))
```

Solution.

```
{'Apple': 1}
{'Apple': 1, 'Banana': 1}
{'Apple': 1, 'Banana': 1, 'apple': 1}
{'Apple': 1, 'Banana': 2, 'apple': 1}
3
```

17. *Find the error in following code. State the reason of the error.*

```
aLst = {'a':1,'b':2,'c':3}
print (aLst['a','b'])
```

Solution.

The above code will produce **KeyError**, the reason being that there is no key same as the list ['a', 'b'] in dictionary **aLst**.

It seems that the above code intends to print the values of two keys 'a' and 'b', thus we can modify the above code to perform this as :

```
aLst = {'a':1,'b':2,'c':3}
print (aLst['a'], aLst['b'])
```

Now it will give the result as :

```
1 2
```

18. *Find the error in the following code fragment. State the reason behind the error.*

```
box = {}
jars = {}
crates = {}
box['biscuit'] = 1
box['cake'] = 3
jars['jam'] = 4
crates['box'] = box
crates['jars'] = jars
print (crates[box])
```

Solution. The above code will produce error with **print( )** because it is trying to print the value from dictionary **crates** by specify a key which is of a mutable type dictionary(**box** is a dictionary). There can never be a key of mutable type in a dictionary, hence the error.

The above code can be corrected by changing the **print( )** as :

```
print (crates['box'])
```

19. *Write the most appropriate list method to perform the following tasks.*

(*a*) *Delete a given element from the list.* (*b*) *Delete 3rd element from the list.*

(*c*) *Add an element in the end of the list.*

(*d*) *Add an element in the beginning of the list.*

(*e*) *Add elements of a list in the end of a list.*

Solution. (*a*) remove( ) (*b*) pop( ) (*c*) append( ) (*d*) insert( ) (*e*) extend( )

20. *How are tuples different from lists when both are sequences ?*

Solution. The tuples and lists are different in following ways :

- The tuples are immutable sequences while lists are mutable.
- Lists can grow or shrink while tuples cannot.

21. *How can you say that a tuple is an ordered list of objects ?*

Solution. A tuple is an ordered list of objects. This is evidenced by the fact that the objects can be accessed through the use of an ordinal index and for a given index, same element is returned everytime.

22. *Following code is trying to create a tuple with a single item. But when we try to obtain the length of the tuple is, Python gives error. Why? What is the solution ?*

```
>>> t = (6)
>>> len(t)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    len(t)
TypeError: object of type 'int' has no len()
```

Solution. The syntax for a tuple with a single item requires the item to be followed by a comma as shown below :

```
t = ("a",)
```

Thus, above code is not creating a tuple in t but an integer, on which *len( )* cannot be applied. To create a tuple in t with single element, the code should be modified as :

```
>>> t = (6,)
>>> len(t)
```

23. *What is the length of the tuple shown below ?*

```
t = (((('a', 1), 'b', 'c'), 'd', 2), 'e', 3)
```

Solution. The length of this tuple is 3 because there are just three elements in the given tuple. Because a careful look at the given tuple yields that tuple *t* is made up of :

```
t1 = "a", 1
t2 = t1, "b", "c"
t3 = t2, "d", 2
t = ( t3,"e", 3)
```

24. *Can tuples be nested ?*

Solution. Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

25. *How are dictionaries different from lists ?*

Solution. The dictionary is similar to lists in the sense that it is also a collection of data-items just like lists BUT it is different from lists in the sense that lists are *sequential collections* (ordered) and dictionaries are *non-sequential collections* (unordered).

In lists, where there is an order associated with the data-items because they act as storage units for other objects or variables you've created. Dictionaries are different from lists and tuples because the group of objects they hold aren't in any particular order, but rather each object has its own unique name, commonly known as a key.

26. *How are objects stored in lists and dictionaries different ?*

Solution. The objects or values stored in a dictionary can basically be anything (even the nothing type defined as None), but keys can only be immutable type-objects. *e.g.*, strings, tuples, integers, etc.

27. *When are dictionaries more useful than lists ?*

Solution. Dictionaries can be much more useful than lists. *For example*, suppose we wanted to store all our friends' cell-phone numbers. We could create a list of pairs (name of friend, phone number), but once this list becomes long enough searching this list for a specific phone number will get time-consuming. Better would be if we could index the list by our friend's name. This is precisely what a dictionary does.

28. *Can sequence operations such as slicing and concatenation be applied to dictionaries ? Why ?*

Solution. No, sequence operations like slicing and concatenation cannot be applied on dictionaries. The reason being, a dictionary is not a sequence. Because it is not maintained in any specific order, operations that depend on a specific order cannot be used.

28. *Why can't Lists be used as keys ?*

Solution. Lists cannot be used as keys in a dictionary because they are mutable. And a Python dictionary can have only keys of immutable types.

30. *If the addition of a new key:value pair causes the size of the dictionary to grow beyond its original size, an error occurs. True or false ?*

Solution. **False.** There cannot occur an error because Dictionaries being the mutable types, they can grow or shrink on an as-needed basis.

31. *Consider a dictionary* ***my_points*** *with single-letter keys, each followed by a 2-element tuple representing the coordinates of a point in an x-y coordinate plane.*

```
my_points = { 'a' : (4, 3), 'b' : (1, 2), 'c' : (5, 1) }
```

Write a program to calculate the maximum value from within all of the values tuples at same index.

For example, maximum for 0th index will be computed from values 4, 1 and 5 – all the entries at 0th index in the value-tuple.

Print the result in following format :

```
Maximum Value at index(my_points, 0) = 5
Maximum Value at index(my_points, 1) = 3
```

Solution.

```
my_points = { 'a' : (4, 3), 'b' : (1, 2), 'c' : (5, 1) }
highest = [0, 0]
init = 0
for a in range(2) :
    init = 0
    for b in my_points.keys():
        val = my_points[b][a]
        if init == 0 :
            highest[a] = val
        init += 1
        if val > highest[a] :
            highest[a] = val
    print("Maximum Value at index(my_points, ", a, ") = ", highest[a])
```

GLOSSARY

Dictionary	A mutable, unordered collection with elements in the form of a key:value pairs that associate keys to value.
Index	An integer variable that is used to identify the position of an element and access the element.
List	A mutable sequence of Python that can store objects of any type.
Lookup	A dictionary operation that takes a key and finds the corresponding value.

3

Working with Functions

In This Chapter

3.1 INTRODUCTION

Large programs are generally avoided because it is difficult to manage a single list of instructions. Thus, a large program is broken down into smaller units known as functions. A function is a named unit of a group of program statements. This unit can be invoked from other parts of the program.

The most important reason to use functions is to make program handling easier as only a small part of the program is dealt with at a time, thereby avoiding ambiguity. Another reason to use functions is to reduce program size. Functions make a program more readable and understandable to a programmer thereby making program management much easier.

In this chapter, we shall talk about functions, especially, how a function works ; how you can create your own functions in Python ; and how you can use the functions created by you.

FUNCTION

A *Function* is a subprogram that acts on data and often returns a value.

3.2 UNDERSTANDING FUNCTIONS

In order to understand what a function is, in terms of a programming language, read the following lines carefully.

You have worked with polynomials in Mathematics. Say we have following polynomial :

$$2x^2$$

For $x = 1$, it will give result as $2 \times 1^2 = 2$
For $x = 2$, it will give result as $2 \times 2^2 = 8$
For $x = 3$, it will give result as $2 \times 3^2 = 18$

and so on.

Now, if we represent above polynomial as somewhat like

$$f(x) = 2x^2$$

Then we can say (from above calculations) that

$$f(1) = 2 \quad \text{...(1)}$$

$$f(2) = 8 \quad \text{...(2)}$$

$$f(3) = 18 \quad \text{...(3)}$$

The notation $f(x) = 2x^2$ can be termed as a **function**, where for function namely ***f***, ***x*** is its argument *i.e.*, value given to it, and **$2x^2$** is its functionality, *i.e.*, the functioning it performs. For different values of argument *x* function $f(x)$ will return different results (refer to equations **(1)**, **(2)** and **(3)** given above).

On the similar lines, programming languages also support functions. You can create functions in a program, that :

- can have arguments (*values given to it*), if needed
- can perform certain functionality (*some set of statements*)
- can return a result

For instance, above mentioned mathematical function $f(x)$ can be written in Python like this :

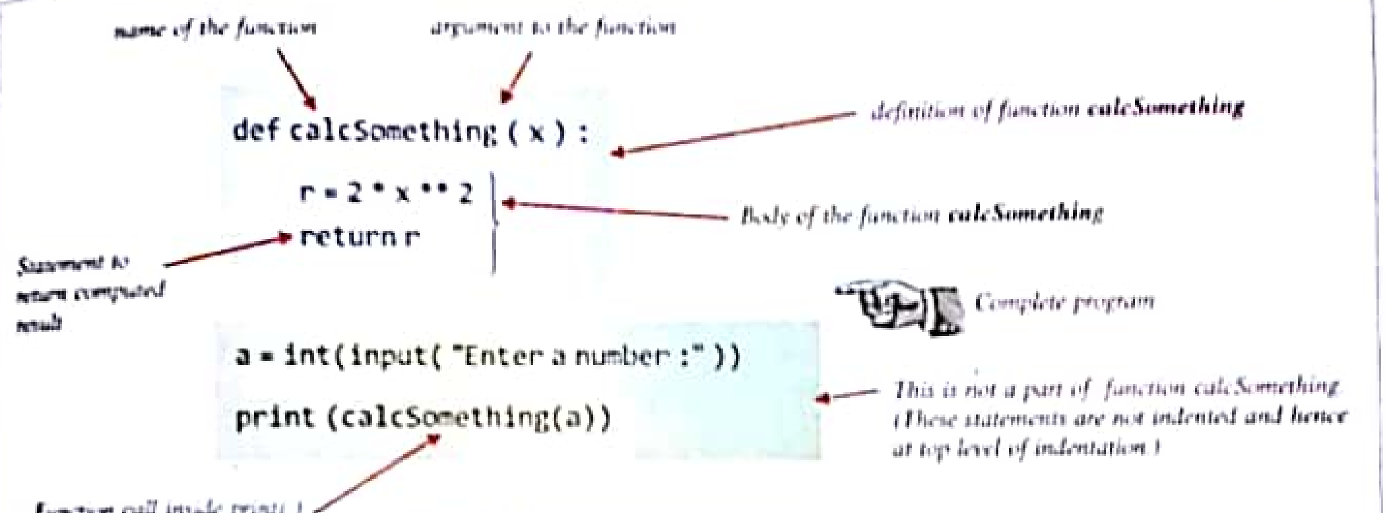
```
def calcSomething ( x ) :
    r = 2 * x ** 2
    return r
```

where

- **def** means a function definition is starting
- identifier following 'def' is the name of the function, *i.e.*, here the function name is ***calcSomething***
- the variables/identifiers inside the parentheses are the *arguments* or *parameters* (values given to function), *i.e.*, here ***x*** is the argument to function ***calcSomething***.
- there is a colon at the end of *def* line, meaning it requires a block

- the statements indented below the function, (*i.e.*, block below *def* line) define the functionality (working) of the function. This block is also called ***body-of-the-function***. Here, there are ***two statements in the body of function calcSomething***.
- The **return** statement returns the computed result.

The non-indented statements that are below the function definition are not part of the function **calcSomething's** definition. For instance, consider the example-function given in Fig. 3.1 below :



```
def calcSomething ( x ) :
    r = 2 * x ** 2
    return r

a = int(input( "Enter a number :" ))
print (calcSomething(a))
```

name of the function
argument to the function
definition of function calcSomething
Body of the function calcSomething
Statement to return computed result
Complete program
This is not a part of function calcSomething (These statements are not indented and hence at top level of indentation)
Function call inside print( )

Figure 3.1 Python Function Anatomy

3.2.1 Calling/Inworking/Using a Function

To use a function that has been defined earlier, you need to write a *function call* statement in Python. A *function call* statement takes the following form :

```
<function-name>(<value-to-be-passed-to-argument>)
```

For example, if we want to call the function **calcSomething( )** defined above, our function call statement will be like :

```
calcSomething(5)          # value 5 is being sent as argument
```

Another function call for the same function, could be like :

```
a = 7
calcSomething(a)          # this time variable a is being sent as argument
```

Carefully notice that number of values being passed is same as number of parameters.

Also notice, in Fig. 3.1, the last line of the program uses a function call statement. (*print* ( ) is using the function call statement.)

Consider one more function definition given below :

```
def cube(x) :
    res = x ** 3          # cube of value in x
    return res            # return the computed value
```

As you can make out that the above function's name is **cube( )** and it takes one argument. Now its function call statement(s) would be similar to the ones shown below :

(*i*) Passing literal as argument in function call

```
cube(4)              # it would pass value as 4 to argument x
```

(*ii*) Passing variable as argument in function call

```
num = 10
cube(num)            # it would pass value as variable num to argument x
```

(*iii*) taking input and passing the input as argument in function call

```
mynum = int (input ("Enter a number :"))
cube(mynum)          # it would pass value as variable mynum to argument x
```

(*iv*) using function call inside another statement

```
print(cube(3))    # cube(3) will first get the computed result
                  # which will be then printed
```

(*v*) using function call inside expression

```
doubleOfCube = 2 * cube(6)
                  # function call's result will be multiplied with 2
```

NOTE

The syntax of the function call is very similar to that of the declaration, except that the key word ***def*** and colon (:) are missing.

3.2.2 Python Function Types

Python comes preloaded with many *function-definitions* that you can use as per your needs. You can even create new functions. Broadly, Python functions can belong to one of the following *three* categories :

1. Built-in functions — These are pre-defined functions and are always available for use. You have used some of them – *len( ), type( ), int( ), input( )* etc.

2. Functions defined in modules — These functions are pre-defined in particular modules and can only be used when the corresponding module is *imported. For example*, if you want to use pre-defined functions inside a module, say **sin( )**, you need to first *import* the module *math* (that contains definition of **sin( )**) in your program.

3. User defined functions — These are defined by the programmer. As programmers you can create your own functions.

In this chapter, you will learn to write your own Python functions and use them in your programs.

STRUCTURE OF FUNCTIONS

Progress In Python 3.1

This PriP session is aimed at making anatomy of Python functions clear to you.

You'll be required to practice about structure of Functions.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.1 under Chapter 3 after practically doing it on the computer.

3.3 DEFINING FUNCTIONS IN PYTHON

As you know that we write programs to do certain things. Functions can be thought of as key-doers within a program. A function once defined can be invoked as many times as needed by using its name, without having to rewrite its code.

In the following lines, we are about to give the general form *i.e.*, syntax of writing function code in Python. Before we do that, just remember these things.

In a syntax language :

- item(s) inside angle brackets < > has to be provided by the programmer.
- item(s) inside square brackets [ ] is optional, *i.e.*, can be omitted.
- items/words/punctuators outside < > and [ ] have to be written as specified.

A function in Python is defined as per following general format :

```
def <function name> ( [parameters] ) :
      [ " " "<function's docstring>" " " ]
      <statement>
      [<statement>]
      :
```

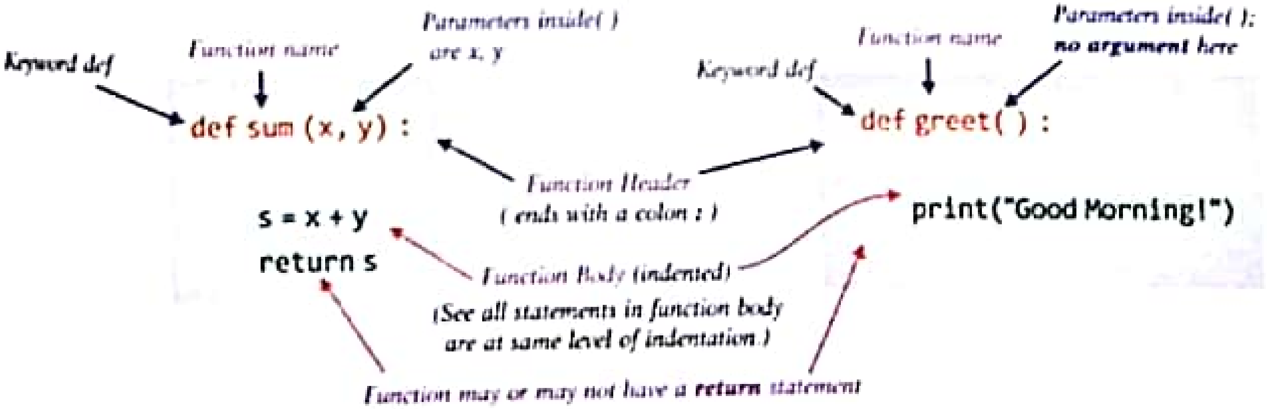
For example, consider some function definitions given below:

```
def sum (x, y) :
      s = x + y
      return s
```

Or

```
def greet( ):
      print("Good Morning!")
```

Though you know about various elements in a function-definition, still let us talk about it again. Let us dissect these functions' definitions to know about various components.



```
def sum (x, y) :

      s = x + y
      return s
```

```
def greet( ) :
      print("Good Morning!")
```

Keyword def
Function name
Parameters inside( ) are x, y
Keyword def
Function name
Parameters inside( ); **no argument here**
Function Header (ends with a colon :)
Function Body (indented)
(See all statements in function body are at same level of indentation.)
Function may or may not have a **return** statement

Let us define these terms formally :

Function Header The first line of function definition that begins with keyword *def* and ends with a colon (:), specifies the ***name of the function*** and its ***parameters***

Parameters Variables that are listed within the parentheses of a function header

Function Body The block of statements / indented-statements beneath function header that defines the action performed by the function.

The function body may or may not return any value. A function returns a value through a *return* statement, *e.g.*, above given ***sum( )*** is returning a value stored in variables, but function ***greet( )*** is not returning a value.

A function not returning any value can still have a ***return*** statement without any expression or value. Examples below will make it clearer.

Indentation The blank space in the beginning of a statement (convention is four spaces) within a block. All statements within same block have same indentation.

Let us now have a look at some more function definitions.

```
# Sample Code 1

def sumOf3Multiples1( n ) :
    s = n * 1 + n * 2 + n * 3
    return s


# Sample Code 2
def sumOf3Multiples2( n ) :
    s = n * 1 + n * 2 + n * 3
    print(s)
```

Both these functions are doing the same thing BUT first one is ***returning*** *the computed value using return statement and second function is* ***printing*** *the computed value using print( ) statement*

Consider some more function definitions:

```
# Sample Code 3
def areaOfSquare ( a ) :
    return a * a
```

```
# Sample Code 4
def areaOfRectangle ( a, b) :
    return a * b
```

```
# Sample Code 5
def perimeterCircle( r ) :
    return (2 * 3.1459 * r)
```

```
# Sample Code 6
def perimeterRectangle( l, b) :
    return 2 * ( l + b )
```

```
# Sample Code 7
def Quote( ) :
    print("\t Quote of the Day")
    print("Act Without Expectation!!")
    print("\t -Lao Tzu")
```

For all these function definitions, try identifying their parts. (Not as an exercise, just do it casually, while reading them.)

A function definition defines a user-defined object function. The function definition does not execute the function body; this gets executed only when the function is called or invoked. In the following lines, we are discussing how to invoke functions, but before that it would be useful to know the basic structure of a Python program.

3.3.1 Structure of a Python Program

In a Python program, generally all function definitions are given at the top followed by statements which are not part of any functions. These statements are not indented at all. These are often called from the top-level statements (the ones with no indentation). The Python interpreter starts the execution of a program/script from the top-level statements. The top level statements are part of the main program. Internally Python gives a special name to top-level statements as *__main__* .

> **NOTE**
>
> By default, Python names the segment with top-level statements (main program) as **__main__** .

The structure of a Python program is generally like the one shown below :

```
def function1( ) :
      :
def function2( ) :
      :
def function3( ) :
      :
      :
# top-level statements here
statement1
statement2
      :
```

Python names the segment with top-level statements (no identation) as __main__. Python begins execution of a program from the top-level statements i.e., from __main__.

Python stores this name in a built-in variable called __name__ (*i.e.*, you need not declare this variable ; you can directly use it). You can see it yourself. In the *__main__* segment of your program if you give a statement like :

```
print(__name__)
```

Python will show you this name. For example, run the following code and see it yourself.

```
def greet( ) :
    print("Hi there!")

print("At the top-most level right now")
print("Inside" , __name__)
```

The top-level statements, i.e., the __main__ segment of this Python program. Python will start execution of this program from the segment.

Upon executing above program, Python will display :

```
At the top-most level right now
Inside __main__
```

Notice word ' __main__ ' in the output by Python interpreter. This is the result of statement :

print(... __name__)

3.4 FLOW OF EXECUTION IN A FUNCTION CALL

Let us now talk about how the control flows (*i.e.*, the flow of execution of statements) in case of a function call. You already know that a function is called (or invoked, or executed) by providing the function name, followed by the values being sent enclosed in parentheses. For instance, to invoke a function whose header looks like :

```
def sum (x, y) :
```

the *function call statement* may look like as shown below :

```
sum (a, b)
```

where *a*, *b* are the values being passed to the function *sum*( ).

Let us now see what happens when Python interpreter encounters a function call statement.

The *Flow of Execution* refers to the order in which statements are executed during a program run.

Recall that a block is a piece of Python program text that is executed as a unit (denoted by line indentation). A **function body** is also a block. In Python, a block is executed in an **execution frame**.

An execution frame contains :

- some internal information (used for debugging)
- name of the function
- values passed to function
- variables created within function
- information about the next instruction to be executed.

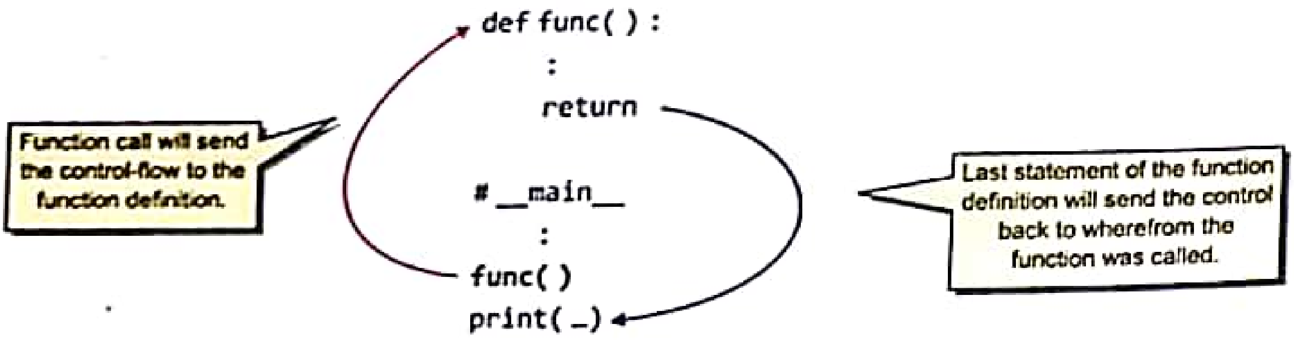
FLOW OF EXECUTION

The ***Flow of Execution*** refers to the order in which statements are executed during a program run.

NOTE

The Flow of Execution refers to the order in which statements are executed during a program run.

Whenever a function call statement is encountered, an *execution frame* for the called function is created and the control (program control) is transferred to it. Within the function's execution frame, the statements in the function-body are executed, and with the *return statement* or the last statement of function body, the control returns to the statement wherefrom the function was called, *i.e.*, as :



```
def func( ) :
    :
    return

#__main__
    :
func( )
print(_)
```

Function call will send the control-flow to the function definition.

Last statement of the function definition will send the control back to wherefrom the function was called.

Let us now see how all this is done with the help of an example. Consider the following program 3.1 code.

Program 3.1 *Program to add two numbers through a function*

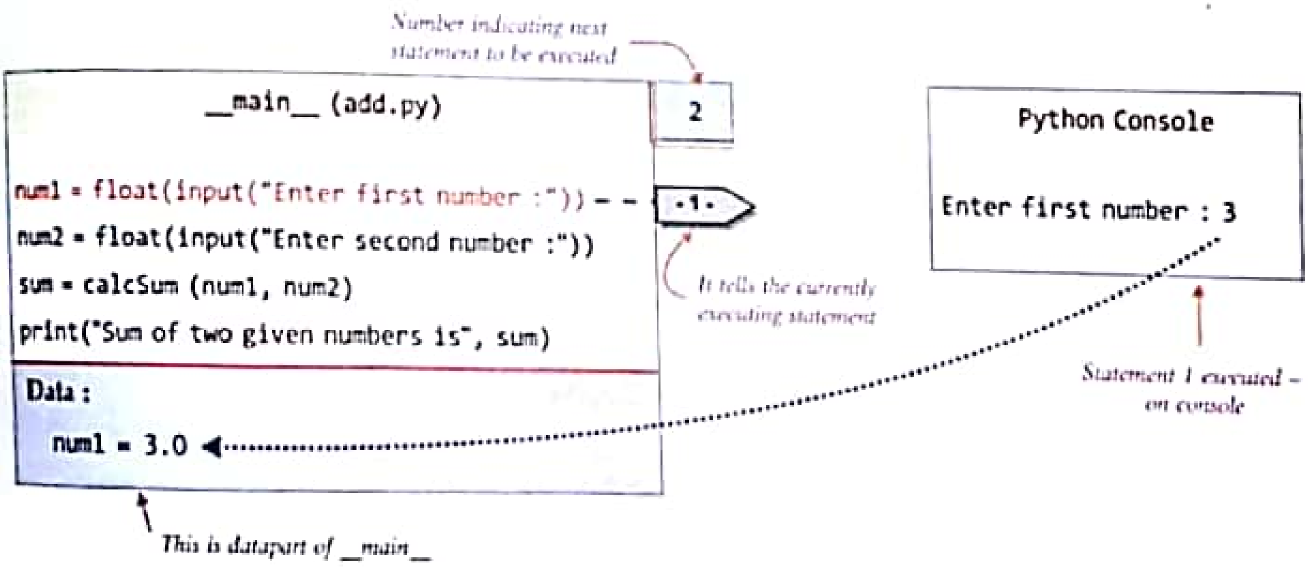
```
# program add.py to add two numbers through a function
def calcSum (x, y) :
    s = x + y                    # statement 1
    return s                     # statement 2

num1 = float(input( "Enter first number :" ) )      # 1 (statement 1)
num2 = float(input( "Enter second number :" ) )     # 2 (statement 2)
sum = calcSum(num1, num2)                           # 3 (statement 3)
print("Sum of two given numbers is", sum)           # 4 (statement 4)
```

Program execution begins with first statement of *__main__* segment. (def statements are also read but ignored until called. It will become clear to you in a few moments. Just read on.)

(Please note that in the following lines, we have put up some execution frames for understanding purposes only; these are not based on any standard diagram.)

NOTE

Program execution begins with first statement of *__main__* segment.



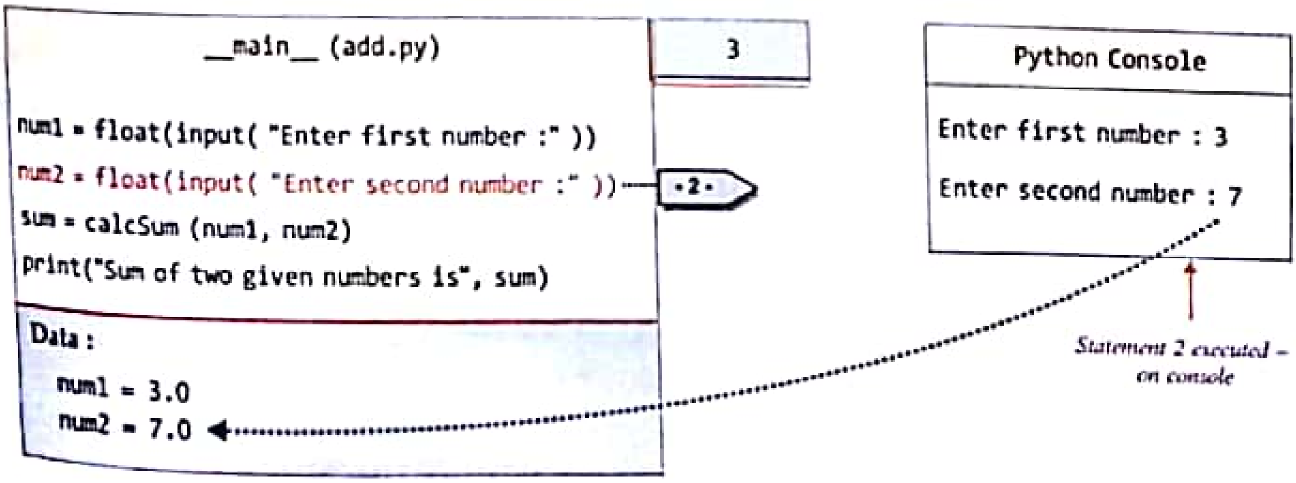
```
__main__ (add.py)

num1 = float(input("Enter first number :"))
num2 = float(input("Enter second number :"))
sum = calcSum (num1, num2)
print("Sum of two given numbers is", sum)
```

Data :
num1 = 3.0

Python Console

```
Enter first number : 3
```

Number indicating next statement to be executed
2
•1•
It tells the currently executing statement
Statement 1 executed – on console
This is datapart of __main__

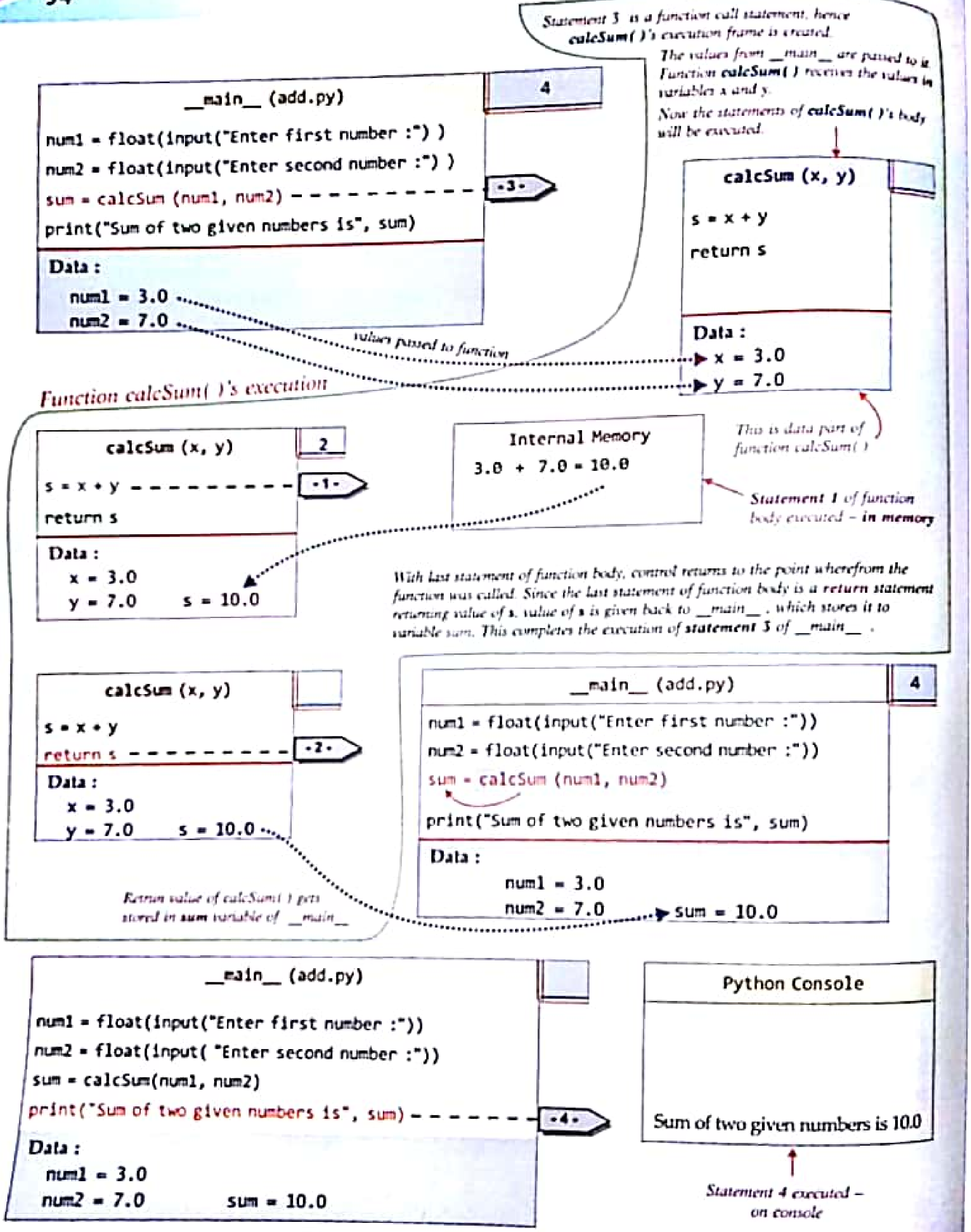


```
__main__ (add.py)

num1 = float(input( "Enter first number :" ))
num2 = float(input( "Enter second number :" ))
sum = calcSum (num1, num2)
print("Sum of two given numbers is", sum)
```

Data :
num1 = 3.0
num2 = 7.0

Python Console

```
Enter first number : 3
Enter second number : 7
```



Statement 3 is a function call statement, hence ***calcSum( )****'s execution frame is created.*

The values from __main__ are passed to it. Function ***calcSum( )*** *receives the values in variables x and y.*

Now the statements of ***calcSum( )****'s body will be executed.*

```
__main__ (add.py)
num1 = float(input("Enter first number :") )
num2 = float(input("Enter second number :") )
sum = calcSum (num1, num2)
print("Sum of two given numbers is", sum)
Data :
   num1 = 3.0
   num2 = 7.0
```

```
calcSum (x, y)
s = x + y
return s
Data :
   x = 3.0
   y = 7.0
```

This is data part of function calcSum( )

Function calcSum( )'s execution

```
calcSum (x, y)
s = x + y
return s
Data :
   x = 3.0
   y = 7.0       s = 10.0
```

```
Internal Memory
3.0 + 7.0 = 10.0
```

Statement 1 *of function body executed –* ***in memory***

With last statement of function body, control returns to the point wherefrom the function was called. Since the last statement of function body is a ***return statement*** *returning value of* ***s****, value of* ***s*** *is given back to __main__ , which stores it to variable sum. This completes the execution of* ***statement 3*** *of __main__ .*

```
calcSum (x, y)
s = x + y
return s
Data :
   x = 3.0
   y = 7.0       s = 10.0
```

```
__main__ (add.py)
num1 = float(input("Enter first number :"))
num2 = float(input("Enter second number :"))
sum = calcSum (num1, num2)
print("Sum of two given numbers is", sum)
Data :
        num1 = 3.0
        num2 = 7.0          sum = 10.0
```

Return value of calcSum( ) gets stored in ***sum*** *variable of __main__*

```
__main__ (add.py)
num1 = float(input("Enter first number :"))
num2 = float(input( "Enter second number :"))
sum = calcSum(num1, num2)
print("Sum of two given numbers is", sum)
Data :
  num1 = 3.0
  num2 = 7.0          sum = 10.0
```

```
Python Console

Sum of two given numbers is 10.0
```

Statement 4 executed – on console

So we can say that for above program the statements were executed as :

main.1 → main.2 → main.3 → calcSum.1 → calcSum.2 → main.3 → main.4

(As you can see that we have shown a statement as its ***<segment-name>.<statement-number>***)

Now that you know how functions are executed internally, let us discuss about *actual flow of execution*.

In a program, Python starts reading from line 1 downwards. Statements are executed one at a time, in order from top to bottom. While executing a program, Python follows these guidelines :

- Execution always begins at the first statement of the program.
- Comment lines (lines beginning with a #) are ignored, *i.e.*, not executed. All other non-blank lines are executed.
- If Python notices that it is a function definition, (**def statements**) then Python just executes the function header line to determine that it is proper function header and skips/ignores all lines in the function body.
- The statements inside a function-body are not executed until the function is called.
- In Python, a function can define another function inside it. But since the inner function definition is inside a function-body, the inner definition isn't executed until the outer function is called.
- When a code-line contains a *function-call*, Python first jumps to the function header line and then to the first line of the function body and starts executing it.
- A function ends with a **return** statement or the last statement of function body, whichever occurs earlier.
- If the called function returns a value *i.e.*, has a statement like return <variable/value/expression> (*e.g.*, **return a** or **return 22/7** or **return a + b** etc.) then the control will jump back to *the function call statement* and completes it (*e.g.*, if the returned value is to be assigned to variable or to be printed or to be compared or used in any type of expression etc. ; whole function call is replaced with the return value to complete the statement).
- If the called function does not return any value *i.e.*, the **return** statement has no variable or value or expression, then the control jumps back to the line following the function call statement.

If we give line number to each line in the program then flow of execution can be represented just through the line numbers, *e.g.*,

```
1.  # program add.py to add two numbers through a function
2.  def calcSum (x, y) :
3.          s = x + y                  # statement 1
4.          return s                   # statement 2
5.
6.  num1 = float(input("Enter first number :" ))       # 1 (statement 1)
7.  num2 = float(input("Enter second number :" ))      # 2 (statement 2)
8.  sum = calcSum (num1, num2)                         # 3 (statement 3)
9.  print("Sum of two given numbers is", sum)          # 4 (statement 4)
```

Determining flow of execution on paper is also sometimes known as *tracing the program*. As per above discussion the flow of execution for above program can also be represented as follows :

$$2 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow \boxed{2 \rightarrow 3 \rightarrow 4} \rightarrow 8 \rightarrow 9$$

function called and executed

Line 1 is ignored because it is a comment ; *line 2* is executed and determined that it is a function header, so entire function-body (*i.e., lines* 3 and 4) is ignored; *lines 6, 7* and *8* executed; *line 8* has a function call, so control jumps to the function header (*line 2*) and then to first line of function-body, *i.e., line 3*, function returns after *line 4* to line containing function call statement *i.e., lines* and then to *line 9*.

Please note that the function calling another function is called the **caller** and the function being called is the **called function** (or **callee**). In above code, the *__main__* is the caller of *calcSum( )* function.

3.4.1 Arguments and Parameters

As you know that you can pass values to functions. For this, you define variables to receive values in *function definition* and you send values via a *function call statement*. For example, consider the following program :

```
def multiply ( a, b ) :
    print (a * b)

y = 3
multiply ( 12, y )           # function call -1-
multiply ( y, y )            # function call -2-
x = 5
multiply (y, x )             # function call -3-
```

You can see that above program has a function namely **multiply( )** that receives two values. This function is being called thrice by passing different values. The *three* function calls for multiply( ) are :

```
multiply ( 12, y )           # function call -1-
multiply ( y, y )            # function call -2-
multiply (y, x )             # function call -3-
```

With *function-call 1,* the variables *a* and *b* in function header will receive values 12 and *y* respectively.

With *function-call 2,* the variables *a* and *b* in function header will receive values *y* and *y* respectively.

With *function-call 3,* the variables *a* and *b* in function header will receive values *y* and *x* respectively.

As you can see that there are **values being passed** (through function call) and **values being received** (in function definition). Let us define these *two* types of values more formally.

- *arguments* : Python refers to the values being passed as **arguments** and
- *parameters* : values being received as **parameters**.

So you can say that *arguments* appear in *function call statement* and *parameters* appear in *function header*.

Arguments in Python can be one of these value types :

- literals
- variables
- expressions

But the *parameters* in Python have to be some names *i.e.*, variables to hold incoming values.

The alternative names for **argument** are ***actual parameter*** and ***actual argument***. Alternative names for **parameter** are ***formal parameter*** and ***formal argument***. So either you use the word combination of *argument and parameter* or you can use the combination *actual parameter and formal parameter* ; or the combination *actual arguments* and *formal arguments*.

Thus for a function as defined below :

```
def multiply ( a, b ) :
    print (a * b)
```

The following are some valid function call statements :

```
multiply( 3, 4 )        # both literal arguments
p = 9
multiply( p, 5)         # one literal and
                        # one variable argument
multiply( p, p + 1)     # one variable and
                        # one expression argument
```

NOTE

The values being passed through a function-call statement are called *arguments* (or *actual parameters* or *actual arguments*). The values received in the function definition/ header are called *parameters* (or *formal parameters* or *formal arguments*).

But a function header like the one shown below is invalid :

```
def multiply ( a + 1, b ) :
    :
```

Error!! A function header cannot have expressions. It can have just names or identifiers to hold the incoming values.

Please remember one thing – if you are passing values of *immutable types* (*e.g.*, *numbers, strings* etc.) to the called function then the called function cannot alter the values of passed arguments but if you are passing the values of *mutable types* (*e.g., list* or *dictionaries*) then called function would be able to make changes in them[1].

FUNCTIONS' BASICS

Progress In Python 3.2

This '*Progress in Python*' session is aimed at strengthening the functions' basics like : *functions' terminology, function anatomy, argument vs parameter* and *flow of execution during function calls.*

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.2 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

[1] This is somewhat similar to function call mechanisms, which are of *two* types : *Call by Value* and *Call by Reference*. In *Call by Value* mechanism, the called function makes a separate copy of passed values and then works with them, so original values remain unchanged. But with *Call by Reference* mechanism, the called function works with original values passed to it, thus any changes made, take place in original values only.
In Python, immutable types implement *Call By Value* mechanism and mutable types implement *Call By Reference* mechanism.

3.5 PASSING PARAMETERS

Uptill now you learnt that a function call must provide all the values as required by function definition. For instance, if a function header has *three* parameters named in its header then the function call should also pass *three* values. Other than this, Python also provides some other ways of sending and matching arguments and parameters.

Python supports *three* types of formal arguments/parameters :

1. Positional arguments (*Required arguments*)
2. Default arguments
3. Keyword (or *named*) arguments

Let us talk about these, one by one.

3.5.1 Positional/Required Arguments

Till now you have seen that when you create a function call statement for a given function definition, you need to match the number of arguments with number of parameters required. *For example*, if a function definition header is like :

```
def check (a, b, c) :
      :
```

then possible function calls for this can be :

```
check ( x, y, z)              # 3 values (all variables) passed
check (2, x, y)               # 3 values (literal + variables) passed
check (2, 5, 7)               # 3 values (all literals) passed
      :
```

See, in all the above function calls, the number of passed values (arguments) has matched with the number of received values (parameters). Also, the values are given (or matched) position-wise or order-wise, *i.e.*, the first parameter receives the value of first argument, second parameter, the value of second argument and so on *e.g.*,

In function call 1 above :

- *a* will get value of *x*
- *b* will get value of *y*
- *c* will get value of *z*

In function call 2 above :

- *a* gets value of 2 ;
- *b* gets value of *x* ;
- *c* gets value of *y*

In function call 3 above :

- *a* gets value 2 ;
- *b* gets value 5 ;
- *c* gets value 7

Thus, through such function calls,

- the arguments must be provided for all parameters (*Required*)
- the values of arguments are matched with parameters, position (order) wise (*Positional*)

This way of parameter and argument specification is called ***Positional arguments*** or ***Required arguments*** or ***Mandatory arguments*** as no value can be skipped from the function call or you cannot change the order *e.g.*, you cannot assign value of first argument to third parameter.

NOTE

When the function call statement must match the number and order of arguments as defined in the function definition, this is called the positional argument matching.

3.5.2 Default Arguments

What if we already know the value for a certain parameter, *e.g.*, in an interest calculating function, we know that mostly the rate of interest is 10%, then there should be a provision to define this value as the default value.

Python allows us to assign default value(s) to a function's parameter(s) which is useful in case a matching argument is not passed in the function call statement. **The default values are specified in the function header of function definition.** Following is an example of function header with default values :

```
def interest (principal, time, rate = 0.10) :
```

*This is default value for parameter **rate**. If in a function call, the value for **rate** is not provided, Python will fill the missing value (for **rate only**) with this value.*

The ***default value*** is specified in a manner syntactically similar to a variable initialization. The above function declaration provides a default value of 0.10 to the parameter ***rate***.

Now, if any function call appears as follows :

NOTE

Non-default arguments cannot follow default argument.

```
si_int = interest (5400, 2)          # third argument missing
```

then the value **5400** is passed to the parameter ***principal***, the value **2** is passed to the second parameter ***time*** and since the third argument ***rate*** is missing, its default value **0.10** is used for ***rate***. But if a function call provides all *three* arguments as shown below :

```
si_int = interest (6100, 3, 0.15)          # no argument missing
```

then the parameter ***principal*** gets value **6100**, ***time*** gets **3** and the parameter ***rate*** gets value ***0.15***.

That means the default values (values assigned in function header) are considered only if no value is provided for that parameter in the function call statement.

DEFAULT PARAMETER

A parameter having default value in the function header is known as a default parameter.

One very important thing you must know about default parameters is :

In a function header, any parameter cannot have a default value unless all parameters appearing on its right have their default values.

For instance, in the above mentioned declaration of function ***interest( )***, the parameter ***principal*** cannot have its default value unless the parameters on its right, ***time*** and ***rate*** also have their default values. Similarly, the parameter ***time*** cannot have its default value unless the parameter on its right, *i.e.*, ***rate*** has its default value. There is no such condition for ***rate*** as no parameter appears on its right.

Thus, *required parameters* should be before *default parameters*.

NOTE

A parameter having a default value in function header becomes optional in function call. Function call may or may not have value for it.

Following are examples of function headers with default values :

```
def interest (prin, time, rate = 0.10) :            # legal
def interest (prin, time = 2, rate) :               # illegal (default parameter
                                                    # before required parameter)
def interest (prin = 2000, time = 2, rate) :        # illegal
                                                    # (same reason as above)
def interest (prin, time = 2, rate = 0.10) :        # legal
def interest (prin = 200, time = 2, rate = 0.10) :  # legal
```

Default arguments are useful in situations where some *parameters* always have the same value. Also they provide greater flexibility to the programmers.

Some advantages of *default parameters* are listed below :

- They can be used to add new parameters to the existing functions.
- They can be used to combine similar functions into one.

NOTE

The default values for parameters are considered only if no value is provided for that parameter in the function call statement.

3.5.3 Keyword (Named) Arguments

The default arguments give you flexibility to specify the default value for a parameter so that it can be skipped in the function call, if needed. However, still you cannot change the order of the arguments in the function call ; you have to remember the correct order of the arguments.

To have complete control and flexibility over the values sent as arguments for the corresponding parameters, Python offers another type of arguments : *keyword arguments*.

Python offers a way of writing function calls where you can write any argument in any order **provided you name the arguments** when calling the function, as shown below :

```
interest (prin = 2000, time = 2, rate = 0.10)
interest (time = 4, prin = 2600, rate = 0.09)
interest (time = 2, rate = 0.12, prin = 2000)
```

All the above function calls are valid now, even if the order of arguments does not match the order of parameters as defined in the function header.

In the 1st function call above,

prin gets value 2000, *time* gets value as 2 and *rate* as 0.10.

In the 2nd function call above,

prin gets value 2600, *time* gets value as 4 and *rate* as 0.09.

In the 3rd function call above,

prin gets value 2000, *time* gets value as 2 and *rate* as 0.12.

This way of specifying names for the values being passed, in the function call is known as **keyword arguments.**

KEYWORD ARGUMENTS

Keyword arguments are the named arguments with assigned values being passed in the function call statement.

3.5.4 Using Multiple Argument Types Together

Python allows you to combine multiple argument types in a function call. Consider the following function call statement that is using both *positional (required)* and *keyword arguments* :

```
interest (5000, time = 5)
```

The first argument value (5000) in above statement is representing a positional argument as it will be assigned to first parameter on the basis of its position. The second argument (time = 5) is representing *keyword argument* or *named argument*. The above function call also skips an argument (rate) for which a default value is defined in the function header.

Rules for combining all three types of arguments

Python states that in a function call statement :

- an argument list must first contain *positional (required) arguments* followed by any *keyword argument*.
- *Keyword arguments* should be taken from the *required arguments* preferably.
- You cannot specify a value for an argument more than once.

NOTE

Having a positional arguments after keyword arguments will result into error.

For instance, consider the following function header :

```
def interest( prin, cc, time = 2, rate = 0.09) :
    return prin * time * rate
```

It is clear from above function definition that values for parameters *prin* and *cc* can be provided either as positional arguments or as keyword arguments but these values cannot be skipped from the function call.

Now for above function, consider following call statements :

Function call statement	Legal / illegal	Reason
`interest(prin = 3000, cc = 5)`	legal	non-default values provided as named arguments
`interest(rate = 0.12, prin = 5000, cc = 4)`	legal	keyword arguments can be used in any order and for the argument skipped, there is a default value
`interest(cc = 4, rate = 0.12, prin = 5000)`	legal	with keyword arguments, we can give values in any order
`interest(5000, 3, rate = 0.05)`	legal	positional arguments before keyword argument; for skipped argument there is a default value
`interest(rate = 0.05, 5000, 3 )`	illegal	keyword argument before positional arguments
`interest(5000, prin = 300, cc =2)`	illegal	Multiple values provided for *prin* ; once as positional argument and again as keyword argument
`interest(5000, principal = 300, cc = 2 )`	illegal	undefined name used (*principal* is not a parameter)
`interest(500, time = 2, rate = 0.05 )`	illegal	A required argument (*cc*) is missing.

Now consider the following program that creates and uses the function interest(), we have been discussing so far.

Program 3.2 Program to calculate simple interest using a function interest( ) that can receive principal amount, time and rate and returns calculated simple interest. Do specify default values for rate and time as 10% and 2 years respectively.

```
def interest(principal, time = 2, rate = 0.10) :
    return principal * rate * time

# __main__
prin = float(input("Enter principal amount :"))
print("Simple interest with default ROI and time values is :")
si1 = interest(prin)
print("Rs.", si1)
roi = float(input("Enter rate of interest (ROI) :"))
time = int(input("Enter time in years :"))
print("Simple interest with your provided ROI and time values is :")
si2 = interest(prin, time, roi/100)
print("Rs.", si2)
```

Sample run of above program is as shown below :

```
Enter principal amount : 6700
Simple interest with default ROI and time values is :
Rs. 1340.0
Enter rate of interest (ROI) : 8
Enter time in years : 3
Simple interest with your provided ROI and time values is :
Rs. 1608.0
```

3.6 RETURNING VALUES FROM FUNCTIONS

Functions in Python may or may not return a value. You already know about it. There can be broadly *two* types of functions in Python :

- Functions returning some value (*non-void functions*)
- Functions not returning any value (*void functions*)

1. Functions returning some value (Non-void functions)

The functions that return some computed result in terms of a value, fall in this category. The computed value is returned using **return** statement as per syntax :

```
return <value >
```

The value being returned can be one of the following :

- ◆ a literal
- ◆ a variable
- ◆ an expression

For example, following are some legal return statements :

```
return 5                # literal being returned
return 6+4              # expression involving literals being returned
return a                # variable being returned
return a**3             # expression involving a variable and literal, being returned
return (a + 8**2) / b   # expression involving variables and literals, being returned
return a + b /c         # expression involving variables being returned
```

When you call a function that is returning a value, the returned value is made available to the caller function/program by internally substituting the function call statement. Confused ? Well, don't be. Just read on, please ☺ .

Suppose if we have a function :

```
def sum (x, y) :
    s = x + y
    return s
```

And we are invoking this function as :

```
result = sum(5, 3)
```

The returned value from sum( ) will replace this function call.

After the function call to **sum( )** function is successfully completed, (*i.e.*, the return statement of function has given the computed sum of 5 and 3) **the returned value** (8 in our case) **will internally substitute the function call** statement. That is, now the above statement will become (internally) :

```
result = 8
```

This is the returned value after successful completion of sum(5, 3). Thus result will now store value 8

IMPORTANT

◆ **The returned value of a function should be used in the caller function/program inside an expression or a statement** *e.g.*, for the above mentioned *sum( )* function, following statements are using the returned value in right manner :

```
add_result = sum (a, b)
```

The returned valued being used in assignment statement

```
print(sum(3, 4))
```

The returned valued being used in print statement

```
sum (4, 5) > 6
```

The returned valued being used in a relational expression

If you do not use their value in any of these ways and just give a stand-alone function call, **Python will not report an error but their return value is completely wasted.**

NOTE

Functions returning a value are also known as fruitful functions.

- **The return statement ends a function execution even if it is in the middle of the function.** A function ends the moment it reaches a *return* statement or all statements in function-body have been executed, whichever occurs earlier, *e.g.*, following function will never reach print( ) statement as *return* is reached before that.

```
def check (a) :
    a = math.fabs(a)
    return a
    print(a)

check(-15)
```

This statement is unreachable because check() function will end with return and control will never reach this statement

Caution! If you do not use a function call of a function-returning some value inside any other expression or statement, function will be executed but its return value will be wasted ; Python will not report any error for it.

2. Functions not returning any value (Void functions)

The functions that perform some action or do some work but do not return any computed value or final value to the caller are called **void functions**. A void function may or may not have a return statement. If a *void function* has a return statement, then it takes the following form :

```
return
```

For a void function, return statement does not have any value/expression being returned.

that is, keyword **return** without any value or expression. Following are some examples of void functions :

void function but no return statement

```
def greet( ):
    print("helloz")
```

Another void function with no return statement

```
def greet1(name) :
    print("hello", name)
```

void function with a return statement

```
def quote( ) :
    print("Goodness counts!!")
    return
```

```
def prinSum (a, b, c) :
    print("Sum is", a + b + c)
    return
```

Another void function with a return statement

The void functions are generally not used inside a statement or expression in the caller ; their function call statement is standalone complete statement in itself, *e.g.*, for the all four above defined *void functions*, the function-call statements can take the form :

```
greet( )
greet1( )
quote( )
prinSum(4, 6)
```

As you can see that all these function call statements are standalone, i.e., these are not part of any other expression or statement

The void functions do not return a value but they return a legal empty value of Python *i.e.*, **None**. Every void function returns value **None** to its caller. So if need arises you can assign this return value somewhere as per your needs, *e.g.*, consider following program code :

```
def greet( ):
    print("helloz")

a = greet( )
print(a)
```

The above program will give output as :

```
helloz
None
```

NOTE

A void function (sometimes called non-fruitful functions) returns legal empty value of Python *i.e.*, None to its caller.

Yes, you guessed it right – **helloz** is printed because of *greet( )*'s execution and **None** is printed as value stored in **a** because *greet( )* returned value **None**, which is assigned to variable **a**.

Consider the following example :

```
# Code 1
def replicate() :
    print("$$$$$")

print(replicate())
```

```
# Code 2
def replicate1() :
    return "$$$$$"

print(replicate1())
```

Here the outputs produced by above two codes will be :

Outputs : *Code 1*

```
$$$$$
None
```

Code 2

```
$$$$$
```

I know that you know the reason, why ?

So, now you know that in Python you can have following *four* possible combinations of functions :

(*i*) non-void functions without any arguments

(*ii*) non-void functions with some arguments

(*iii*) void functions without any arguments

(*iv*) void functions with some arguments

Please note that a function in a program can call any other function in the same program.

3.6.1 Returning Multiple Values

Unlike other programming languages, Python lets you return more than one value from a function. Isn't that useful ? You must be wondering, how ? Let's find out.

To return multiple values from a function, you have to ensure following things :

(*i*) The return statement inside a *function body* should be of the form given below :

```
return <value1/variable1/expression1>, <value2/variable2/expression2>, ...
```

(*ii*) The *function call statement* should receive or use the returned values in one of the following ways :

(*a*) Either receive the returned values in form a tuple variable, *i.e.*, as shown below :

```
def squared(x, y, z):
    return x * x, y * y, z * z

t = squared(2,3,4)
print(t)
```

The return statement returning comma separated multiple values (expressions)

Variable t that receives the returned values is a tuple (recall that comma separated values are taken as a tuple)

Now you can use the tuple t with usual operations

Tuple t will be printed as:
(4, 9, 16)

(*b*) Or you can directly unpack the received values of tuple by specifying the same number of variables on the left-hand side of the assignment in function call, *e.g.*,

```
def squared(x, y, z):
    return x * x, y * y, z * z

v1, v2, v3 = squared(2,3,4)
print("The returned values are as under:")
print(v1, v2, v3)
```

Now the received values are in the form of three different variables, not as a tuple

Output produced as::

```
The returned values are as under:
4 9 16
```

Now consider the following example program.

Program 3.3 Program that receives two numbers in a function and returns the results of all arithmetic operations (+, -, *, /, %) on these numbers.

```
def arCalc(x, y) :
    return x+y, x-y, x*y, x/y, x%y

#__main__
num1 = int(input("Enter number 1 : "))
num2 = int(input("Enter number 2 : "))
add, sub, mult, div, mod = arCalc(num1, num2)
print("Sum of given numbers :", add)
print("Subtraction of given numbers :", sub)
print("Product of given numbers :", mult)
print("Division of given numbers :", div)
print("Modulo of given numbers :", mod)
```

Sample run of above program is as shown below :

```
Enter number 1 : 13
Enter number 2 : 7
Sum of given numbers : 20
Subtraction of given numbers : 6
Product of given numbers : 91
Division of given numbers : 1.8571428571428572
Modulo of given numbers : 6
```

3.7 COMPOSITION

Composition in general refers to using an expression as part of a larger expression; or a statement as a part of larger statement. In functions' context, we can understand composition as follows :

The arguments of a function call can be any kind of expression :

- an arithmetic expression *e.g.*,

```
greater( (4 + 5), (3 + 4) )
```

- a logical expression *e.g.*,

```
test( a or b)
```

- a function call (function composition) *e.g.*,

```
int(str(52))
int(float("52.5") * 2)
int(str(52) + str(10))
```

Function call as part of larger function call i.e., composition

NOTE

Composition in general refers to using an expression as part of a larger expression; or a statement as a part of larger statement.

The above examples show you composition of function calls – a function call as part of larger function call.

3.8 SCOPE OF VARIABLES

The scope rules of a language are the rules that decide, in which part(s) of the program, a particular piece of code or data item would be known and can be accessed therein. To understand Scope, let us consider a real-life situation.

Suppose you are touring a historical place with many monuments. To visit a monument, you have to buy a ticket. Say, you buy a ticket (let us call it *ticket1*) to go see a *monumentA*. As long as, you are inside *monumentA*, your ticket1 is valid. But the moment you come out of *monumentA*, the validity of *ticket1* is over. You cannot use *ticket1* to visit any other monument. To visit *monumentB*, you have to buy another ticket, say *ticket2*. So, we can say that scope of *ticket1* is *monumentA* and scope of *ticket2* is *monumentB*.

Say, to promote tourism, the government has also launched a city-based ticket (say *ticket3*). A person having city-based ticket can visit all the monuments in that city. So we can say that the scope of *ticket3* is the whole city and all the monuments within city including *monumentA* and *monumentB*.

Now let us understand scope in terms of Python. In programming terms, we can say that, scope refers to part(s) of program within which a name is legal and accessible. If it seems confusing, I suggest you read on the following lines and examples and then re-read this section.

SCOPE

Part(s) of program within which a name is legal and accessible, is called scope of the name.

There are broadly *two* kinds of scopes in Python, as being discussed below.

1. Global Scope

A name declared in top level segment (__main__) of a program is said to have a **global scope** and is *usable inside the whole program* and all blocks (functions, other blocks) contained within the program.

(*Compare with real-life example given above, we can say that ticket3 has global scope within a city as it is usable in all blocks within the city.*)

2. Local Scope

A name declared in a function-body is said to have **local scope** *i.e.*, it can be *used only within this function and the other blocks contained under it.* The *names of formal arguments* also have local scope.

(*Compare with real-life example given above, we can say that ticket1 and ticket2 have local scopes within monumentA and monumentB respectively.*)

A local scope can be multi-level; there can be an enclosing local scope having a nested local scope of an inside block. All this would become clear to you in coming lines.

NOTE

A *global variable* is a variable defined in the 'main' program (__main__ section). Such variables are said to have *global scope*.

A *local variable* is a variable defined within a function. Such variables are said to have *local scope*.

Scope Example I

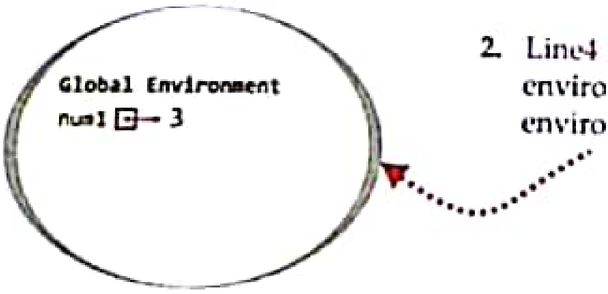
Consider the following Python program (program 3.1 of section 3.4) :

```
1.  def calcSum (x, y) :
2.      z = x + y           # statement -1-
3.      return z            # statement -2-

4.  num1 = int( input( "Enter first number :" ) )       # statement -1-
5.  num2 = int( input( "Enter second number :" ) )      # statement -2-
6.  sum = calcSum ( num1, num2 )                         # statement -3-
7.  print('Sum of given numbers is', sum)               # statement -4-
```

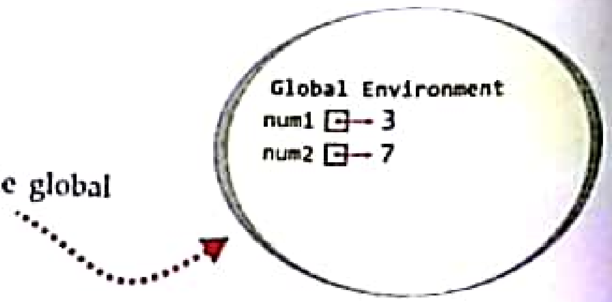
A careful look on the program tells that there are *three* variables *num1, num2* and *sum* defined in the *main program* and three variables *x, y* and *z* defined in the function *calcSum( )*. So, as per definition given above, *num1, num2* and *sum* are global variables here and *x, y* and *z* are local variables (local to function *calcSum( )*).

Let us now see how there would be different scopes for variables in this program by checking the status after every statement executed. We'll check the status as per the flow of execution of above program (refer to section 3.4)

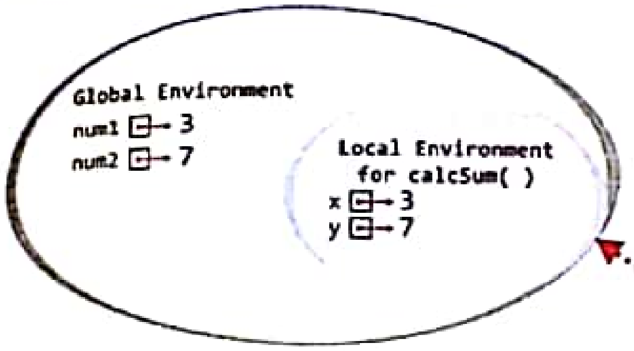
1. Line1 : *def* encountered and lines 2-3 are ignored.



2. Line4 (Main.1) : Execution begins and global environment is created. *num1* is added to this environment.

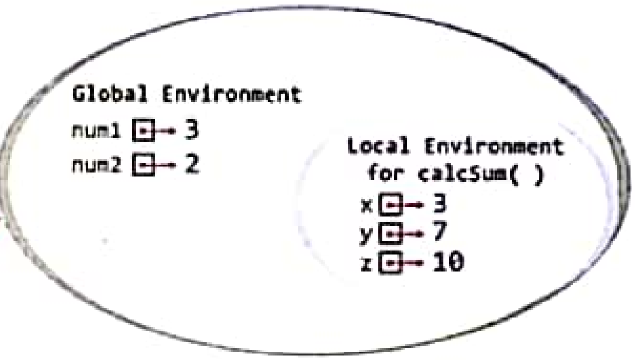


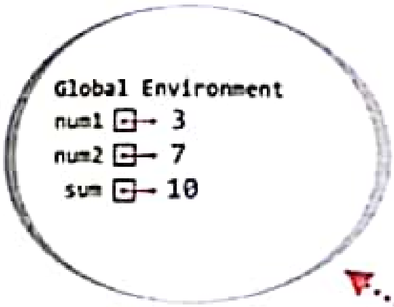
3. Line5 (Main.2) : *num2* is also added to the global environment.



4. **Line6 (Main.3) : calcSum( ) is invoked, so a local environment for calcSum( ) is created; *formal arguments* x and y are created in local environment.**

5. Line2 (calcSum.1) : variable *z* is created in the local environment.





6. Line3 (calcSum.2) : value of *z* is returned to caller (**return** ends the function, hence after sending value of *s* to caller in variable *sum* (when control is back to *Main.3*), the local environment is removed and so are all its constituents).

7. Line7 (Main.4) : the print statement picks value of *sum* from its own environment.

8. Program over. Global environment is also removed with the end of the program.

As you can see from above that scope of names *num1*, *num2* and *sum* is global and scope of names *x*, *y* and *z* is local.

Variables defined outside all functions are global variables

These variables can be defined even before all the function definitions.

Consider the following example :

```
x = 5
def func(a):
    b = a + 1
    return b

y = input( "Enter number" )
z = y + func(x)
print(z)
```

Variable x defined above all functions. It is also a global variable along with y and z

Scope Example 2

Let us take one more example. Consider the following code :

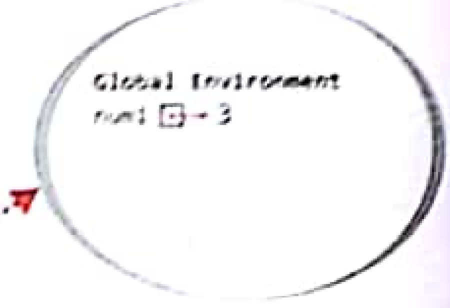
```
1.  def calcSum(a, b, c) :
2.          s = a + b + c                          # statement -1-
3.          return s                               # statement -2-

4.  def average ( x, y, z ) :
5.          sm = calcSum (x, y, z)                 # statement -1-
6.          return sm / 3                          # statement -2-

7.  num1 = int (input ( "Number 1 :" ) )          # statement -1-
8.  num2 = int (input ( "Number 2 :" ) )          # statement -2-
9.  num3 = int (input ( "Number 3 :" ) )          # statement -3-
10. print("Average of these numbers is", average( num1, num2, num3))
                                                               # statement -4-
```

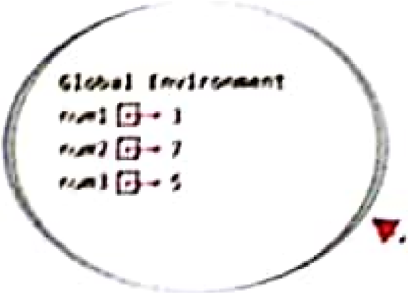
Internally the global and local environments would be created as per flow of execution :

1. Line1 : *def* encountered ; lines 2, 3 ignored.

2. Line4 : *def* encountered ; lines 5, 6 ignored.

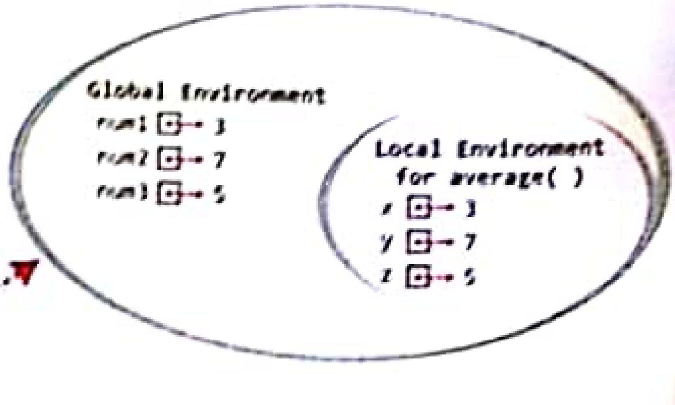
3. Line7 (**Main.1**) : execution of main program begins ; global environment created ; ***num1*** added to it.



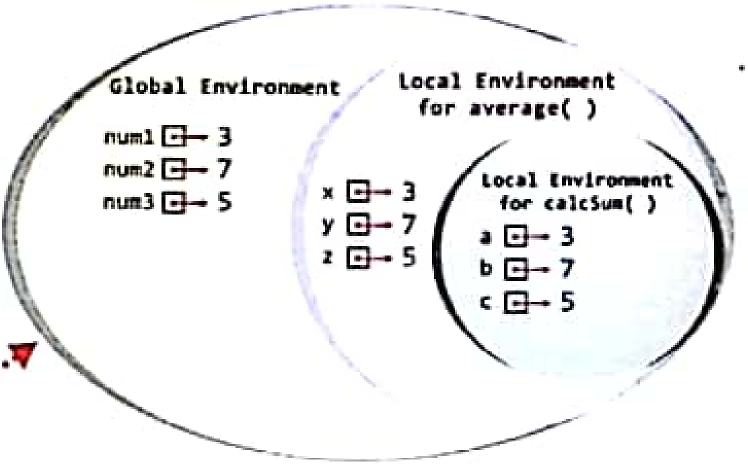


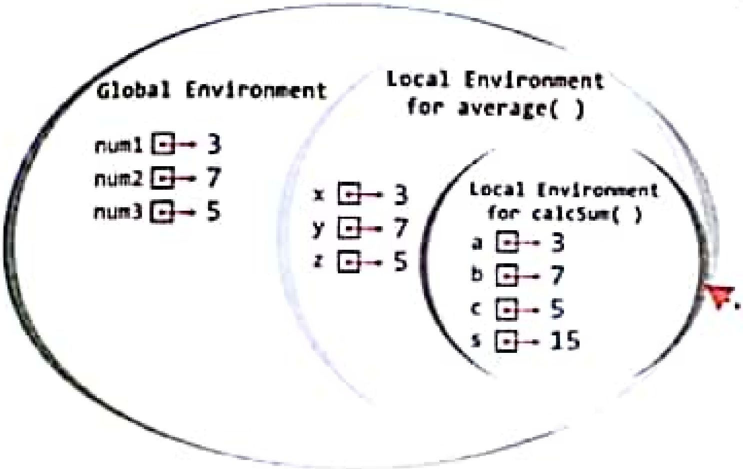
4. Lines 8, 9 (**Main.2** and **Main.3**) : add ***num2*** and ***num3*** to global environment.

5. Line10 (**Main.4**) : Function **average( )** is invoked, so a local environment for **average( )** is created ; formal arguments ***x***, ***y*** and ***z*** are created in local environment.



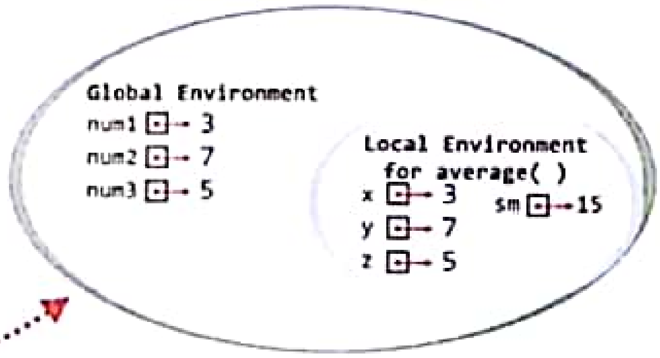
6. Line5 (average.1) : Function *calcSum( )* is invoked, so a local environment for *calcSum( )* is created, nested within local environment of *average( )* ; its formal arguments (*a, b, c*) are created in it.



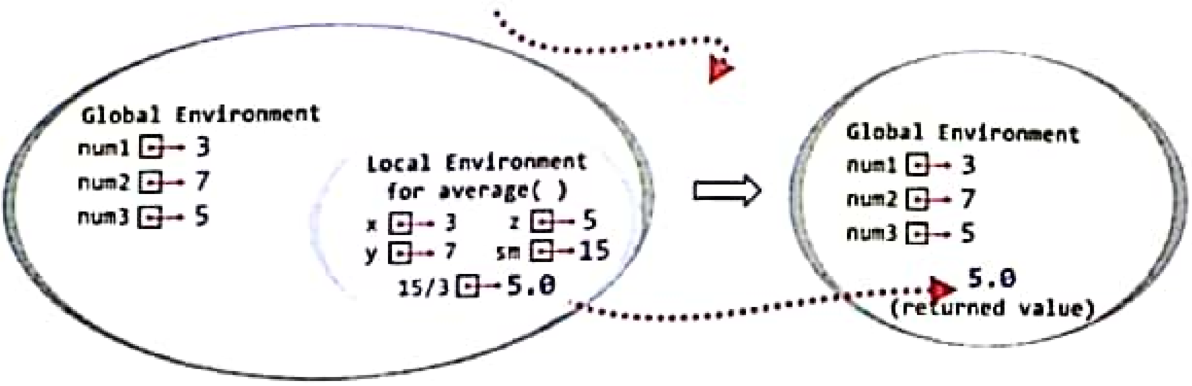


7. Line1 (calcSum.1) : Variable *s* is created within local environment of *calcSum( )*.

8. Line2 (calcSum.2) : Value of *s* is returned to *sm* of *average( )* and *calcSum( )* is over, hence the local environment of *calcSum( )* is removed.



9. Line6 (average.2) : Return value is calculated as *sm / 3* (*i.e.,* 15/3 = 5.0) and returned to caller (*main.4*) statement ; *average( )* is over so its local environment is removed.



10. Line10 (Main.4) : The *print statement* receives computed value 5.0, prints it and program is over. (with this global environment of the program will also be removed.)

Here we want to introduce another term – **lifetime** of a variable. The lifetime of variable is the time for which a variable lives in memory. For global variables, **lifetime is entire program run** (*i.e.*, they live in memory as long as the program is running) and for local variables, **lifetime is their function's run** (*i.e.*, as long as their function is being executed.)

LIFETIME

The time for which a variable or name remains in memory is called Lifetime of variable.

3.8.1 Name Resolution (Resolving Scope of a Name)

For every name reference within a program, *i.e.*, when you access a variable from within a program or function, Python follows name resolution rule, also known as **LEGB rule.** That is, for every name reference, Python does the following to resolve it :

(*i*) It checks within its **Local environment** (LEGB) (or *local namespace*) if it has a variable with the same name ; if yes, Python uses its value.

If not, then it moves to step (*ii*).

(*ii*) Python now checks the **Enclosing environment** (LEGB) (*e.g.*, if whether there is a variable with the same name) ; if yes, Python uses its value.

If the variable is not found in the current environment, Python repeats this step to higher level enclosing environments, if any.

If not, then it moves to step (*iii*).

(*iii*) Python now checks the **Global environment** (LEGB) whether there is a variable with the same name ; if yes, Python uses its value.

If not, then it moves to step (*iv*).

(*iv*) Python checks its **Built-in environment** (LEGB) that contains all built-in variables and functions of Python, if there is a variable with the same name ; if yes, Python uses its value.

Otherwise Python would report the error :

```
name < variable > not defined.
```

As you can make out that LEGB rule means checking in the order of Local, Enclosing, Global, Built-in environments (namespaces) to resolve a name reference. (Fig 3.2)

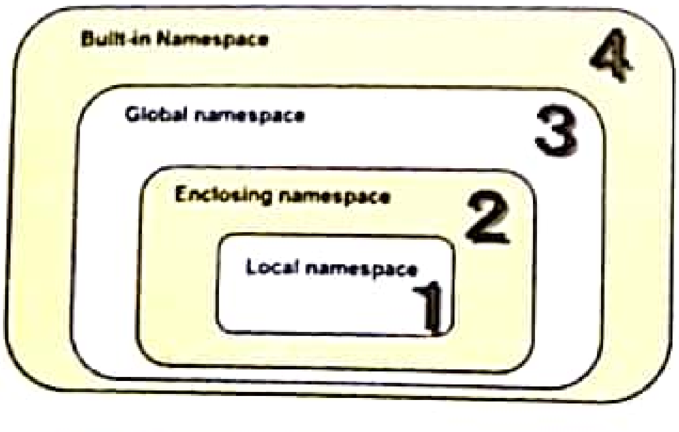


Figure 3.2 LEGB rule for name resolution (scope)

Let us consider some examples to understand this.

Case 1 : Variable in global scope but not in local scope

Let us understand this with the help of following code :

```
def calcSum (x, y) :
    s = x + y                    # statement -1-
    print(num1)                  # statement -2-
    return s                     # statement -3-

num1 = int(input( "Enter first number :" ) )
num2 = int(input( "Enter second number :" ) )
print("Sum is", calcSum (num1, num2))
```

*variable **num1** is a global variable, not a local variable*

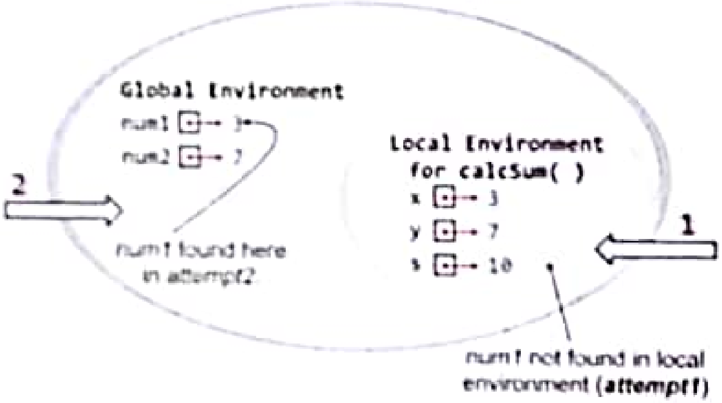
Consider statement 2 of function *calcSum( )*. Carefully notice that *num1* has not been created in calcSum( ) and still statement2 is trying to print its value. The internal memory status at time of execution of statement 2 of *calcSum( )* would be somewhat like :

1. Python will first check the *Local environment* of *calcSum( )* for ***num1*** ;

   num1 is not found there.

2. Python now checks for ***num1***, the parent environment of ***calcSum( )***, which is *Global environment* (there is not any intermediate enclosing environment).

   Python finds ***num1*** here ; so it picks its value and prints it.


Global Environment
num1 → 3
num2 → 7
Local Environment for calcSum( )
x → 3
y → 7
s → 10
2
1
num1 found here in attempt2
num1 not found in local environment (attempt1)

Case 2 : Variable neither in local scope nor in global scope

What if the function is using a variable which is neither in its local environment nor in its parent environment ? Simple! Python will return an error, *e.g.*, Python will report error for variable name in the following code as it not defined anywhere :

```
def greet( ):
    print("hello", name)

greet( )
```

This would return error as name is neither in local environment nor in global environment

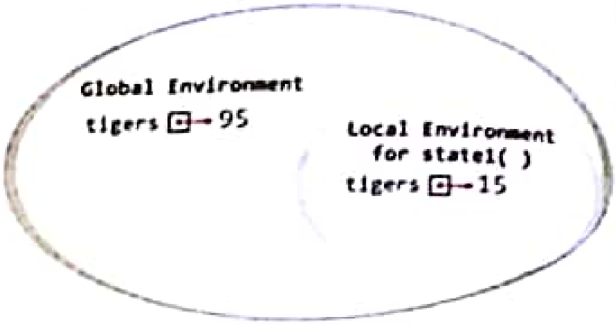
Case 3 : Same variable name in local scope as well as in global scope

If inside a function, you assign a value to a name which is already there in a higher level scope, Python won't use the higher scope variable because it is an assignment statement and assignment statement creates a variable by default in current environment.

For instance, consider the following code ; read it carefully :

```
def state1( ):
    tigers = 15
    print(tigers)

tigers = 95
print tigers
state1()
print(tigers)
```

This statement will create a local variable with name tigers as it is assignment statement. It won't refer to tigers of main program.



The above program will give output as :

```
95
15
95
```

*Result of print statement inside state1( ) function, thus, value of local **tigers** is printed.*

*Result of print statement inside main program, thus, value of global **tigers** is printed.*

That means a local variable created with same name as that of global variable, it hides the global variable. As in above code, local variable **tigers** hides the global variable **tigers** in function ***state1( )***.

What if you want to use the global variable inside local scope?

If you want to use the value of already created global variable inside a local function without modifying it, then simply use it. Python will use LEGB rule and reach to this variable.

But if you want to assign some value to the global variable without creating any local variable, then what to do ? This is because, if you assign any value to a name, Python will create a local variable by the same name. For this kind of problem, Python makes available **global** statement.

To tell a function that for a particular name, do not create a local variable but use global variable instead, you need to write :

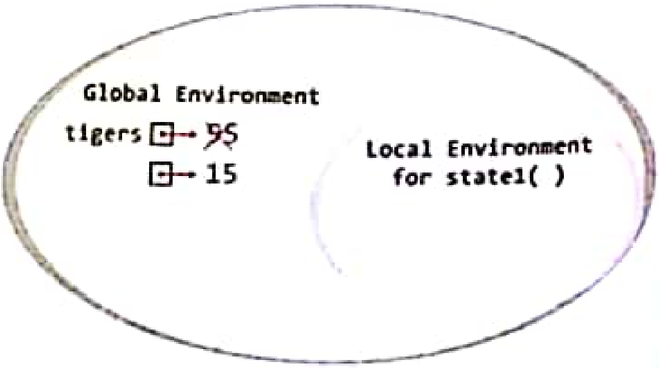
```
global <variable name>
```

The **global** statement is a declaration which holds for the entire current code block. It means that the listed identifiers are part of the *global namespace or global environment.*

For example, in above code, if you want function state1( ) to work with global variable **tigers**, you need to add global statement for **tigers** variable to it as shown below :

```
def state1( ):
    global tigers
    tigers = 15
    print(tigers)

tigers = 95
print(tigers)
state1()
print(tigers)
```

This is an indication not to create local variable with the name tigers, rather use global variable tigers.



The above program will give output as :

```
95
15
15
```

Result of print statement inside state1( ) function, value of global ***tigers*** *is printed (which was modified to* ***15*** *in previous line).*

Result of print statement inside main program, thus, value of global ***tigers*** *(which is 15 now) is printed.*

Once a variable is declared *global in a function, you cannot undo the statement*. That is, after a global statement, the function will always refer to the global variable and local variable cannot be created of the same name.

But for good programming practice, the use of global statement is always discouraged as with this programmers tend to lose the control over variables and their scopes.

NOTE

The global statement cannot be reverted in a program run. One should avoid using global statement in Python program.

TIP

Although global variables can be accessed through local scope, but it is not a good programming practice. So, keep global variables global, and local variables local.

CALLING FUNCTIONS, ARGUMENT TYPES, SCOPE OF VARIABLES

PriP Progress In Python 3.3

This '*Progress in Python*' session is aimed at these concepts : *calling or invoking functions, different types of arguments, scope of variables* and *Name resolution by Python*.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.3 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

3.9 MUTABLE/IMMUTABLE PROPERTIES OF PASSED DATA OBJECTS

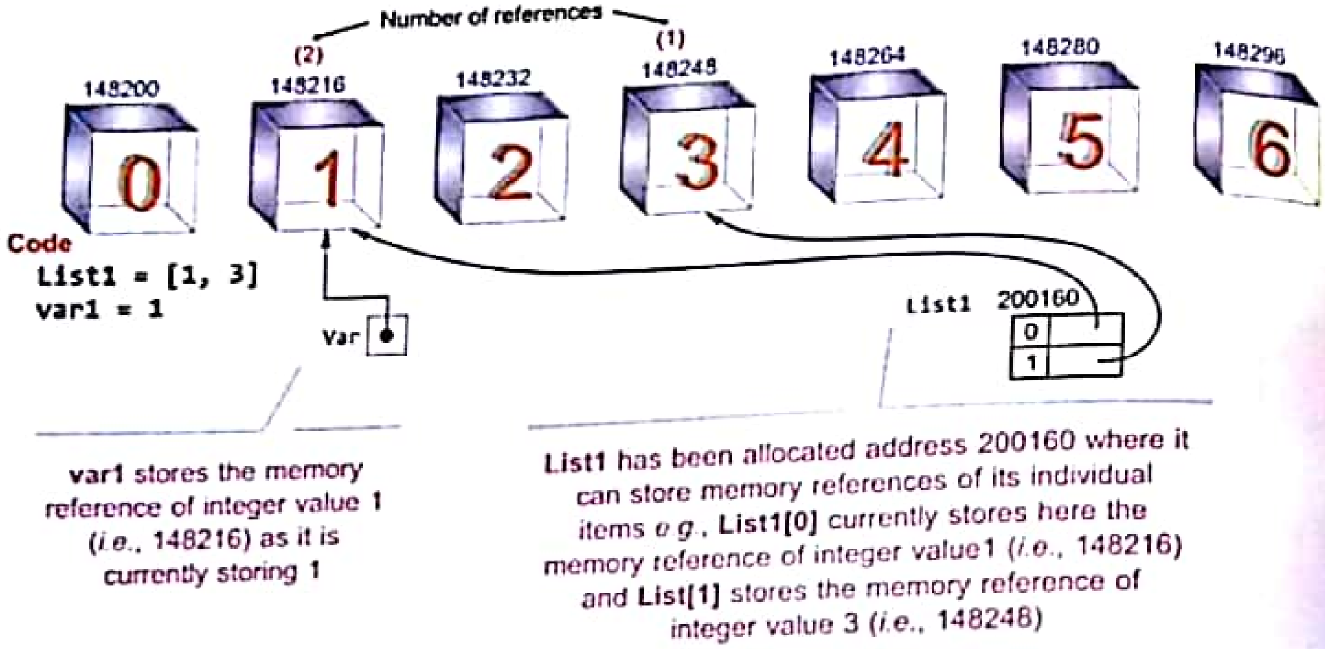
So now you know how you can pass values to a function and how a function returns value(s) to its caller. But here it is important to recall following things about variables :

- Python's variables are not *storage containers*, rather Python variables are like memory references; they refer to the memory address where the value is stored.
- Depending upon the mutability/immutability of its data type, a variable behaves differently. That is, if a variable is referring to an immutable type then any change in its value will also change the memory address it is referring to, but if a variable is referring to mutable type then any change in the value of mutable type will not change the memory address of the variable (recall section 1.7). Following figure also summarizes the same.

IMPORTANT

Another important thing to know here is that the *scope rules* apply to the *identifiers* or the *name labels* and not on the values. For example, if you pass an argument *a* with value 5 to a function that receives it in parameter *b*, then both *a* and *b* will refer to same value (*i.e.*, 5) in data space (even with different scopes, they may refer to same data in data space depending upon mutability), but, you can use name *b* only inside the called function and not in the calling function. This will become clearer to you when we discuss mutability with respect to arguments and parameters in the section 3.8.

(Integer literals are stored at some predefined locations)



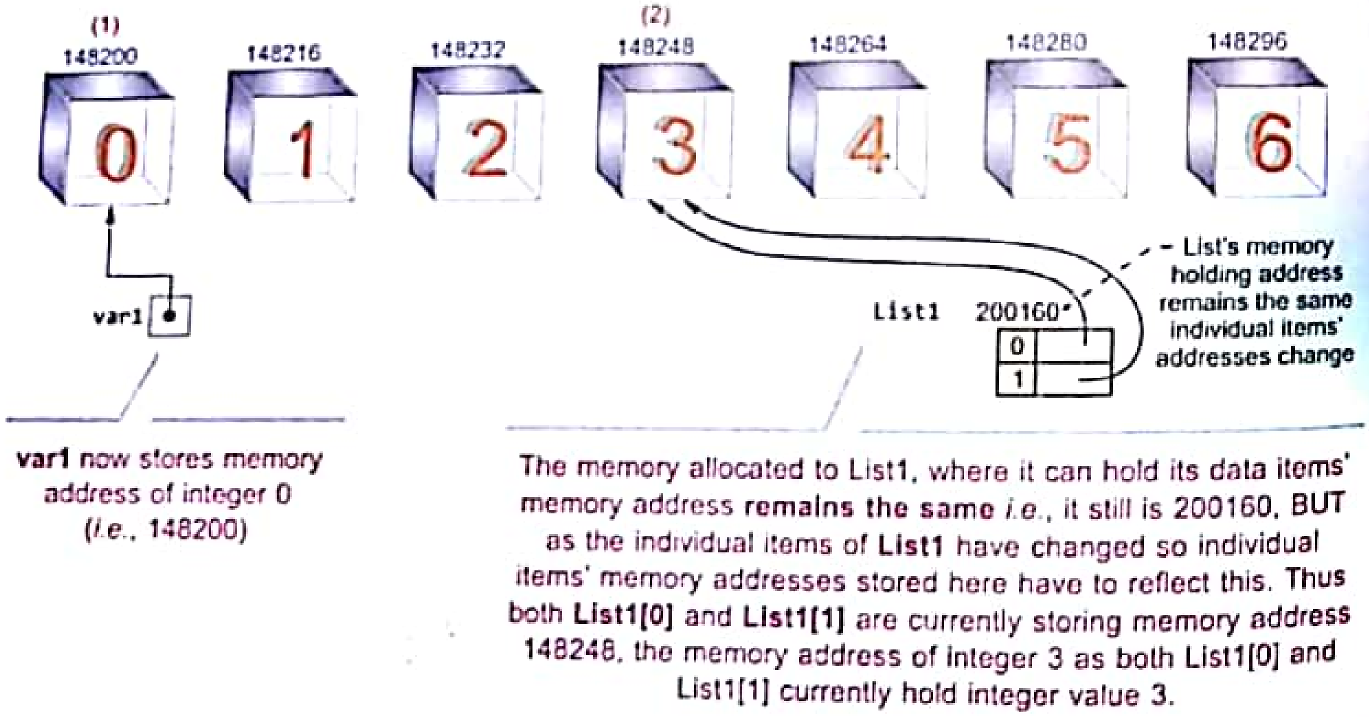
```
List1 = [1, 3]
var1 = 1
```

var1 stores the memory reference of integer value 1 (*i.e.*, 148216) as it is currently storing 1

List1 has been allocated address 200160 where it can store memory references of its individual items *e.g.*, **List1[0]** currently stores here the memory reference of integer value1 (*i.e.*, 148216) and **List[1]** stores the memory reference of integer value 3 (*i.e.*, 148248)

Now if you make following changes to **List1** and **Var1** :

```
var1 = var1 - 1        (Now var1 holds value 0)
List1[0] = 3           (Now first item of List1 is also 3)
```



List's memory holding address remains the same individual items' addresses change

var1 now stores memory address of integer 0 (*i.e.*, 148200)

The memory allocated to List1, where it can hold its data items' memory address **remains the same** *i.e.*, it still is 200160, BUT as the individual items of **List1** have changed so individual items' memory addresses stored here have to reflect this. Thus both **List1[0]** and **List1[1]** are currently storing memory address 148248, the memory address of integer 3 as both List1[0] and List1[1] currently hold integer value 3.

Figure 3.4 **Impact of Mutability and Immutability**

3.9.1 Mutability/Immutability of Arguments/Parameters and Function Calls

When you pass values through arguments and parameters to a function, mutability/ immutability also plays an important role there.

Let us understand this with the help of some sample codes.

Sample Code 1.1

Passing an Immutable Type Value to a function.

```
def myFunc1(a):
    print("\t Inside myFunc1()")
    print("\t Value received in 'a' as", a)
    a = a + 2
    print("\t Value of 'a' now changes to", a)
    print("\t returning from myFunc1()")
#__main__
num = 3
print("Calling myFunc1() by passing 'num' with value", num)
myFunc1(num)
print("Back from myFunc1(). Value of 'num' is", num)
```

Now have a look at the output produced by above code as shown below :

```
Calling myFunc1( ) by passing 'num' with value 3
        Inside myFunc1( )
        Value received in 'a' as 3
        Value of 'a' now changes to 8
        returning from myFunc1( )
Back from myFunc1( ). Value of 'num' is 3
```

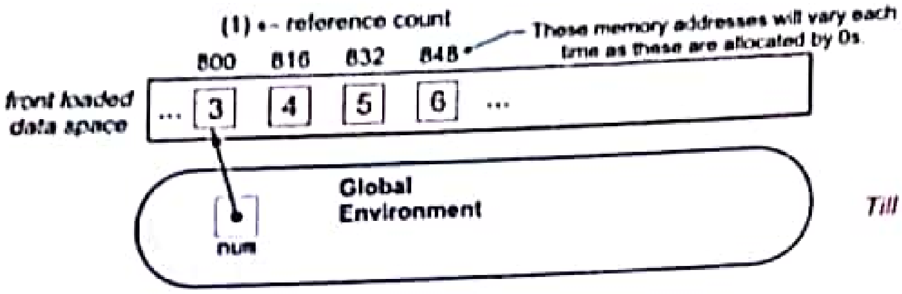
The value got changed from 3 to 8 inside function BUT NOT got reflected to __main__

As you can see that the function *myFunc1()* received the passed value in parameter *a* and then changed the value of *a* by performing some operation on it. Inside **myFunc1( )**, the value (of *a*) got changed but after returning from **myFunc1( )**, the originally passed variable *num* remains unchanged.

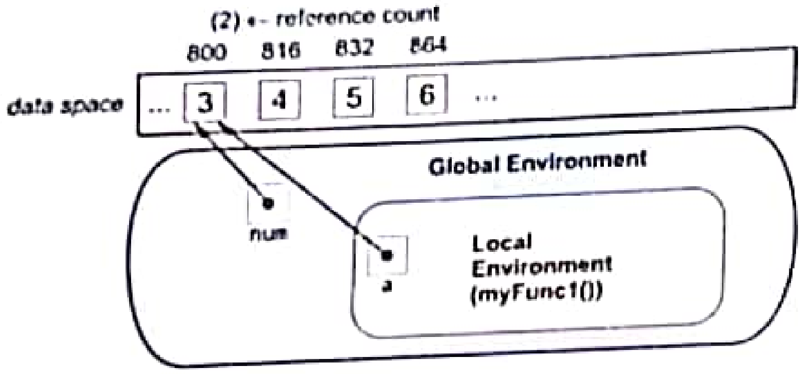
Let us see how the memory environments are created for above code (*i.e., **sample code1***).

Memory Environment For Sample Code 1.1

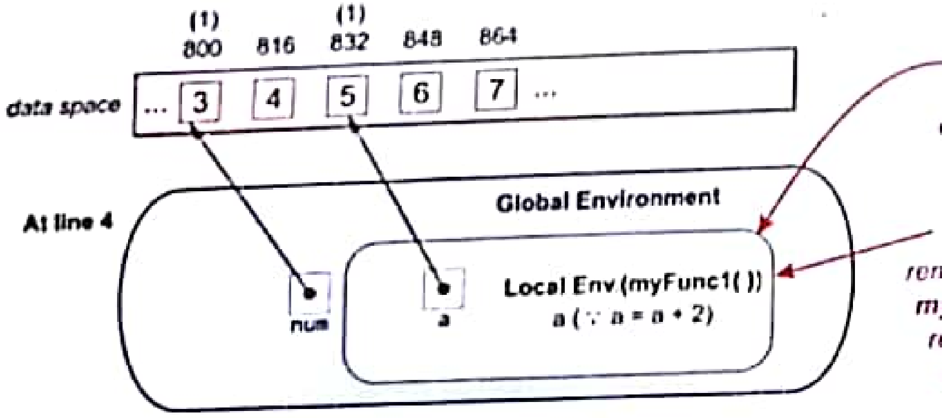
(Note : Passed value is an integer, an immutable type)



Till Lines 7-9 of code of --main--

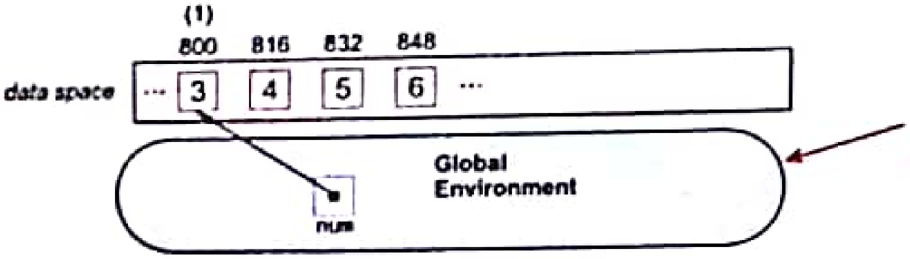


*At **line10** (function is called) argument **num** is received in parameter **a** and for **lines 1, 2, 3**, environment remains the same*



*See, the global environment's **num** remains unaffected from changes to variable **a** of **myfunc1***

*Local memory environment remains the same till **line6** and the **myfunc1( )** gets over and control returns to --main-- part's **line11** and the local environment of **myfunc1( )** is removed.*



*At **line11**, when **num's** value is printed Python prints 3 as the **num** remained unchanged in its scope and thus always remained 3.*

So you just saw how Python processed an immutable data type when it is passed as argument. Let us see what happens inside memory if you pass a mutable type such as a *list*. (Recall that a sequence/collection such as a list internally is stored as a container that holds the references of individual items.)

Sample Code 2.1

Passing a Mutable Type Value to a function–Making changes in place)

```
def myFunc2(myList):
    print("\n\t Inside CALLED Function now")
    print("\t List received:", myList)
    myList[0] += 2
    print("\t List within called function, after changes:", myList)
    return

List1 = [1]
print("List before function call : ", List1)
myFunc2(List1)
print("\nList after function call : ", List1)
```

Now have a look at the output produced by above code as shown below :

```
List before function call : [1]

  Inside CALLED Function now
  List received:  [1]
  List within called function, after changes : [3]

List after function call : [3]
```

The value got changed from [1] to [3] inside function and change GOT REFLECTED to __main__

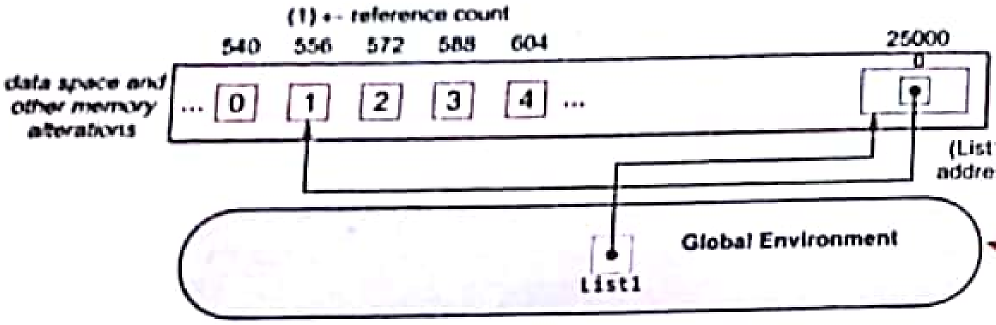
As you can see that the function myFunc2( ) receives a mutable type, *a list*, this time. The passed list (List1) contains value as [1] and is received by the function in parameter mylist. The changes made inside the function in the list *mylist* get reelected in the original list passed, *i.e.*, in list1 of __main__ .

So when you print its value after returning from function, it shows the changed value. The reason is clear – list is a mutable type and thus changes made to it are refelected back in the caller function.

Let us see how the memory environments are created for above code (*i.e., sample code2.1*).

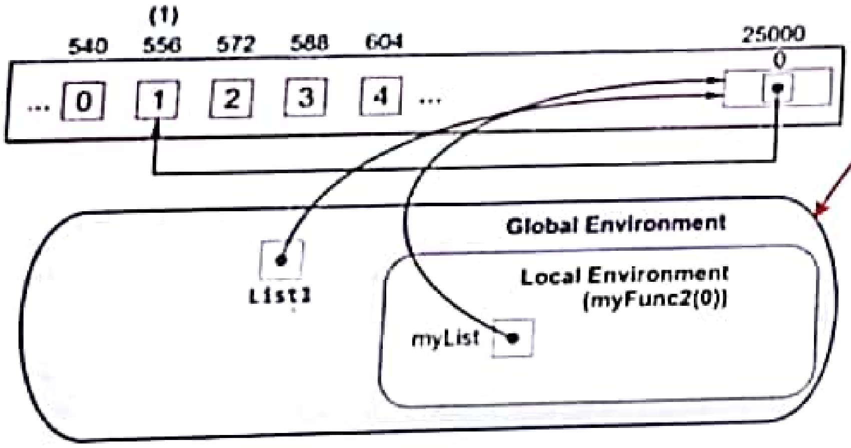
Memory Environment For Sample Code 2.1

(Note : Passed value is a *list*, a mutable type)

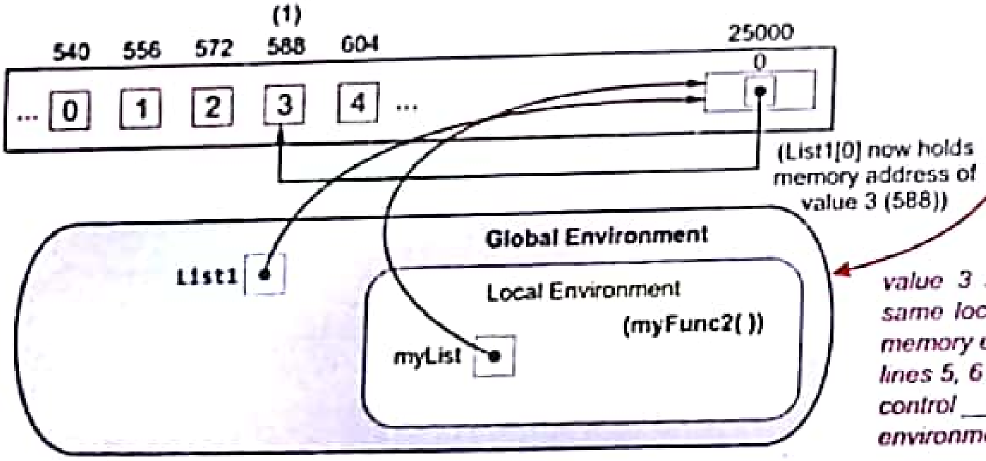

(1) ←- reference count
540
556
572
588
604
25000
0
data space and other memory alterations
... 0 1 2 3 4 ...
(List1[0] holds memory address of value 1 (556))
Global Environment
List1

Memory addresses vary on each computer and even for each session.

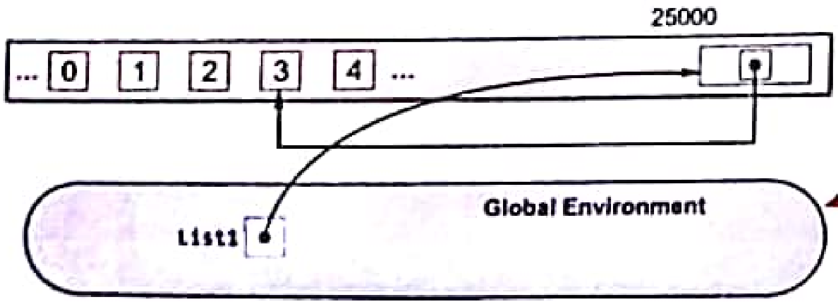
Till Lines 7-9 of code of __main__



*At line9, function myFunc2( ) gets called ; argument **List1** is received in parameter **myList**. Both now point to address 25000. For lines 1, 2, 3, the environment remains the same.*



*At line4, myList[0]'s value changes to 3 (myList[0] += 2) and thus the 0th item of **myList** now holds reference of value 3. BUT the memory location of list **myList** remains the same (25000). The change of memory address from value 1 to value 3 has been done in place i.e., at same location of mylist (i.e., 25000). The memory environment remains the same for lines 5, 6 of code and then function returns control __main__'s line 10, remaining local environment of function myFunc2( ).*



*At line 10, when **List1** is printed, it is referring to address 25000 that currently holds reference of value 3. Hence changed list is printed (as [3])*

So changes in memory references occurred at the same address **25000** for both lists (argument and parameter) and hence got reflected in __main__, the very essence of mutable types.

Similarly, if you make other changes such as adding or deleting items from a passed list, these will also get reflected back as both the passed list argument and received parameter list work with the same memory address where changes are done in place. Check following code and its output – changes are reflected back in __main__

Sample Code 2.2

Passing a Mutable Type Value to a function – Adding/Deleting items to it)

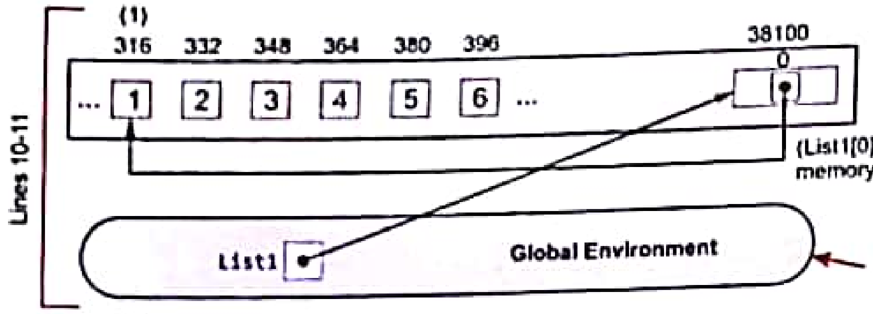
```
def myFunc3(myList):
    print("\t Inside CALLED Function now")
    print("\t List received:", myList)
    myList.append(2)
    myList.extend([5,1])
    print("\t List after adding some elements:", myList)
    myList.remove(5)
    print("\t List within called function, after all changes:", myList)
    return
List1 = [1]
print("List before function call :", List1)
myFunc3(List1)
print("\List after function call :", List1)
```

Now have a look at the output produced by above code as shown below :

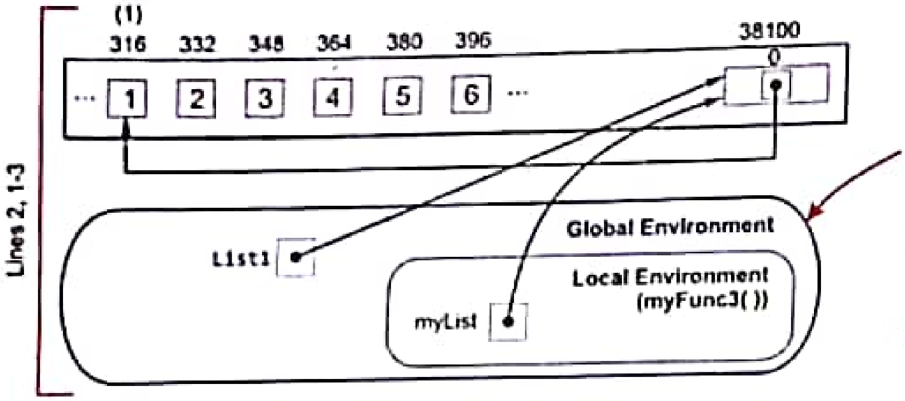
```
List before function call :  [1]
  Inside CALLED Function now
  List received:  [1]
  List after adding some elements: [1, 2, 5, 1]
  List within called function, after all changes: [1, 2, 1]
List after function call : [1, 2, 1]
```

The value got changed from [1] to [1, 2,1] inside function and change GOT REFLECTED to __main__

Memory Environment For Sample Code 2.2

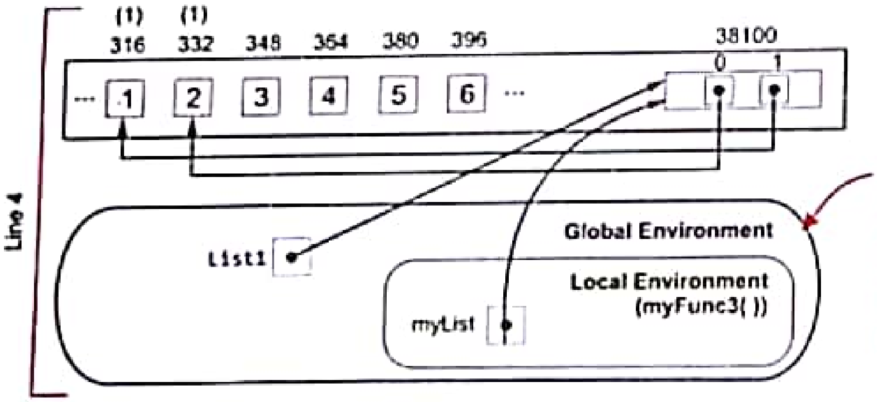


Till Lines 10-11 of code of __main__



*At line 12, function myFunc3( ) gets called ; argument **List1** is received in parameter **myList**. Both **List1** and **myList** point to same memory address i.e., 38100.*

Environment remains the same for lines 1, 2, 3.

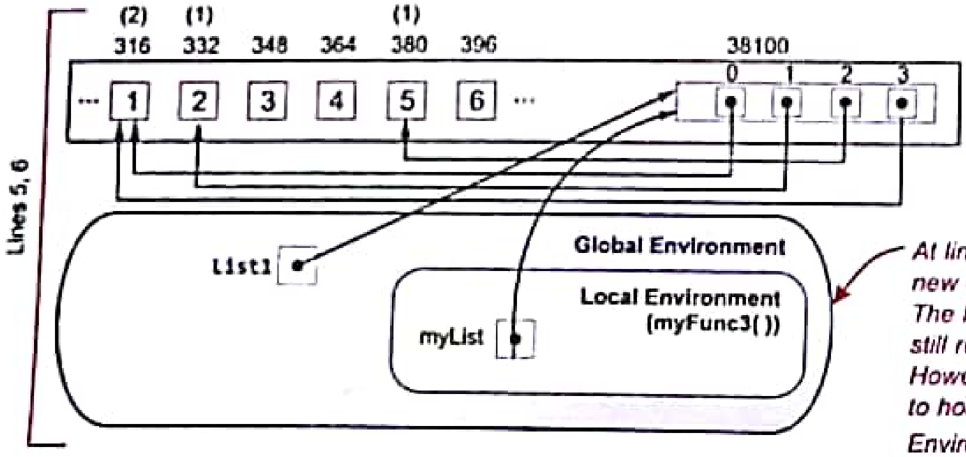


*At line 4, statement **myList.append(2)** gets executed, so a new element (index 1) is appended to mylist's same memory address 38100.*

This new element now references to value2.

Notice even after appending a new element the list's address is the same *i.e.*, 38100. It, however, now accommodates place to hold new value.

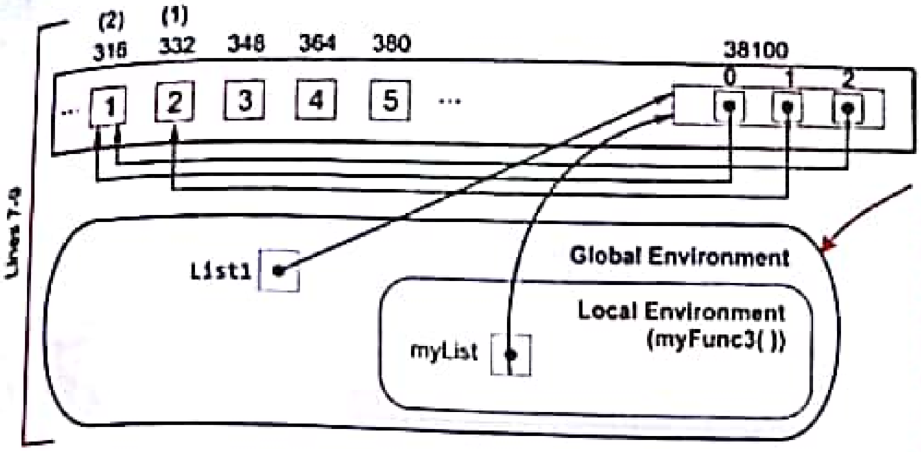
Mutable types act like containers. Container remains unchanged, however, its contents may change. Hence their overall address remains the same as it is the container's address.



At line5, the list is extended with two new elements (myList.extend(5, 1)). The list's(myList) memory address still remains the same i.e., 38100. However, it now accomodates place to hold two new elements.

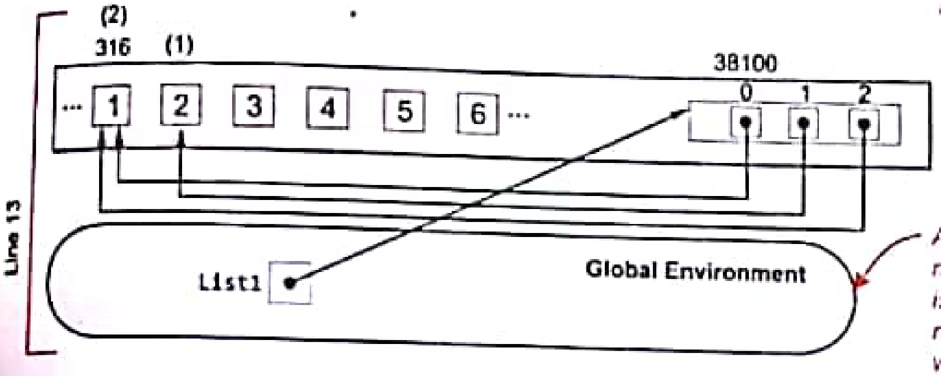
Environment remains same for line 6 too.

Memory Environment For Sample Code 2.2 (Contd....)



At line 7, statement **myList.remove(s)** removes element 5 from list and thus only three elements are left in List(**myList**). But the memory address of the list (myList) remains the same even after changes in its contents.

The environment remains the same till line 9 and then control returns to line13.



At line 13, after returning from myFunc3( ) the local environment is removed. The list. List1 is still referencing the address 38100 with all the changes intact.

Sample Code 2.3

Passing a Mutable Type Value to a function – Assigning parameter to a new value/variable.

```
1.  def myFunc4(myList):
2.      print("\n\t Inside CALLED Function now")
3.      print("\t List received:", myList)
4.      new = [3,5]
5.      myList = new        ←  Parameter list get assigned a new value (a list here)
6.      myList.append(6)
7.      print("\t List within called function, after changes:", myList)
8.      return

9.  List1 = [1, 4]
10. print("List before function call :", List1)
11. myFunc4(List1)
12. print("\nList after function call :", List1)
```

Now carefully look at the output produced by above code as shown below :

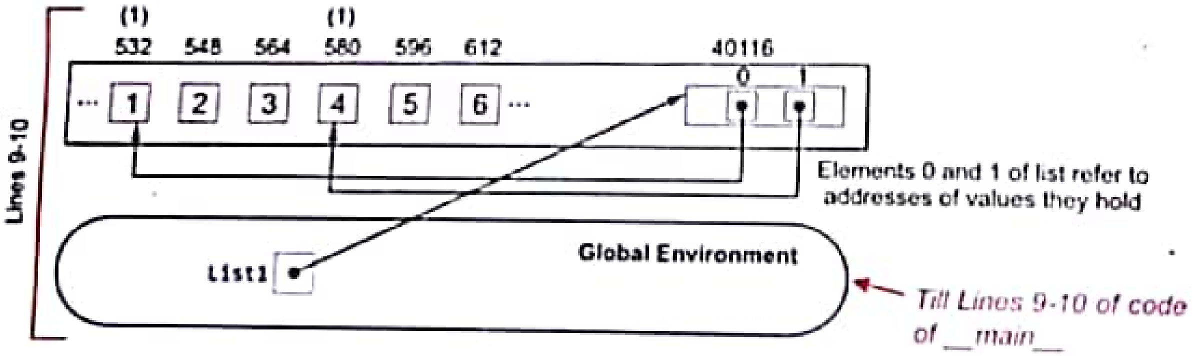
```
List before function call :  [1, 4]

  Inside CALLED Function now
  List received:  [1, 4]
  List within called function, after changes: [3, 5, 6]

List after function call : [1, 4]
```

The value got changed from [1, 4] to [3,5, 6] inside function and change DID NOT GET REFLECTED to __main__

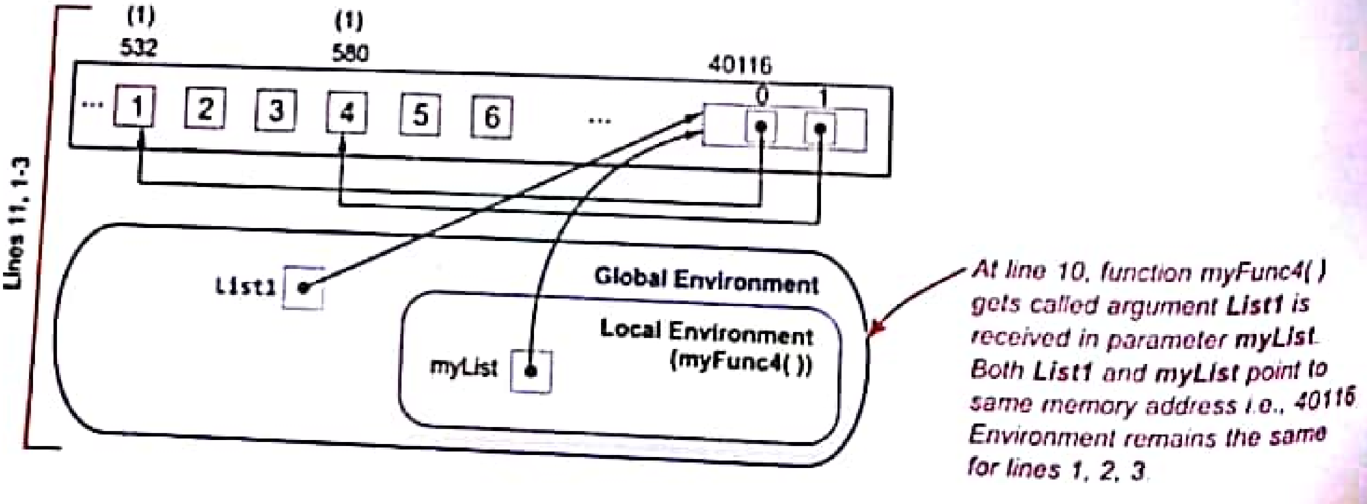
Isn't this unexpected, the passed value is of a mutable type, a list. Right ? But still the changes made in the function do not get reflected back to __main__. Why ? No worries. Let's find out. But first have a look at the memory environments created for above code.

Memory Environment For Sample Code 2.3

(Note : Passed value is a list, a mutable type)

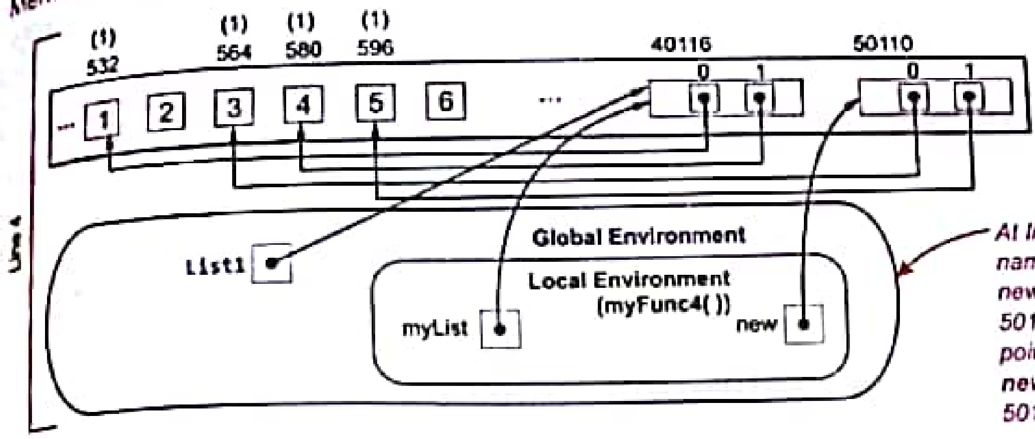


Till Lines 9-10 of code of __main__

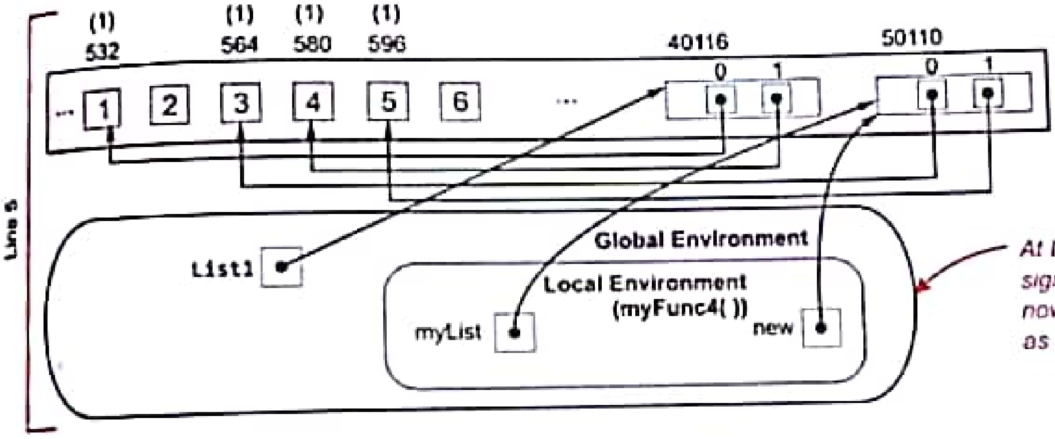


At line 10, function myFunc4( ) gets called argument **List1** is received in parameter **myList**. Both **List1** and **myList** point to same memory address i.e., 40116 Environment remains the same for lines 1, 2, 3.

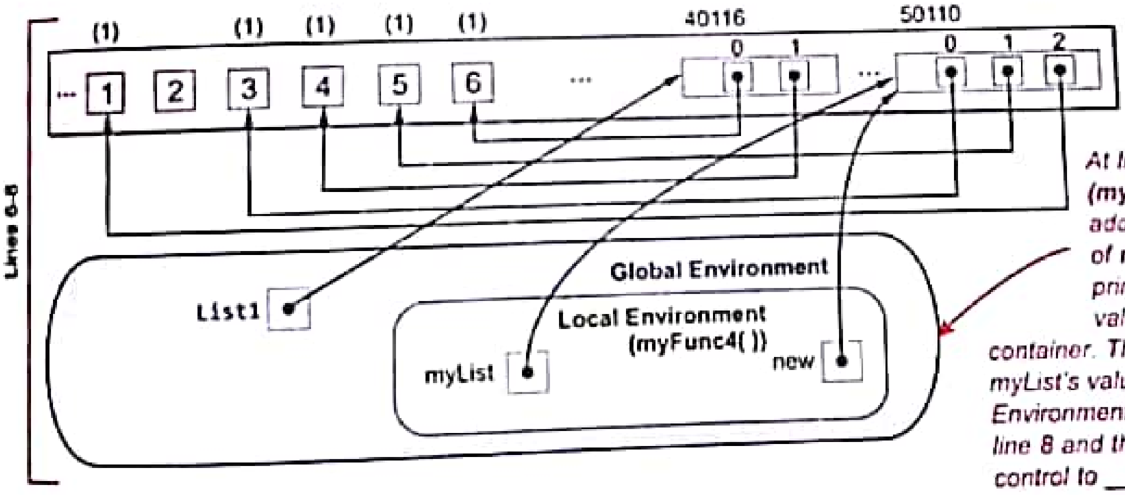
Memory Environment For Sample Code 2.3 (Contd....)



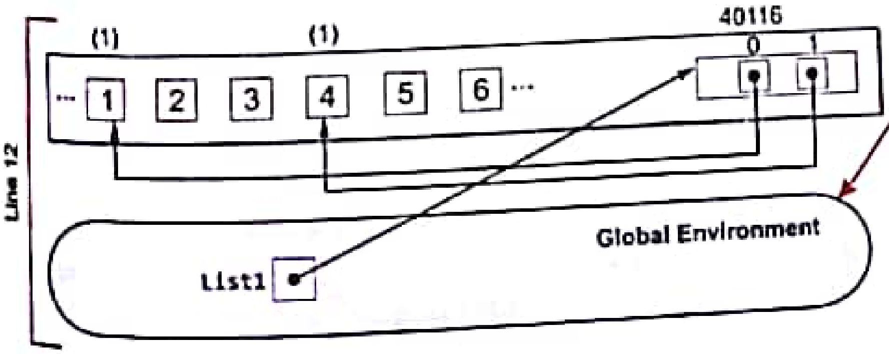
*At line 4, a new list by the name new is created for which new memory is allocated at 50110. While **List1** and **myList** point to memory address 40116, **new** has memory address as 50110.*



*At line 5, **myList** is assigned **new**, which means it now points to same address as **new** i.e., 50110.*

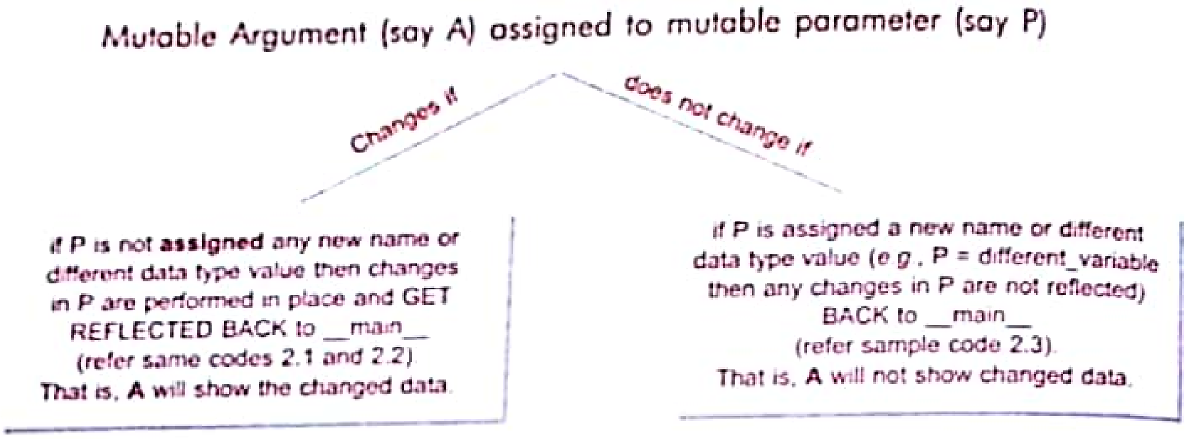


*At line 6, statement **(myList.append(6))** adds value 6 at the end of **myList**. Since **myList** is printing to address 50110, value 6 gets added to this container. Thus when you print myList's value, it shows [3, 5, 6]. Environment remains the same till line 8 and then function returns control to __main__ at line 12.*



*After myFunc4( ) gets over, the local environment is removed. The list, **List1** is still referring to address 40116 and prints whatever is contained at this very address i.e., [1, 4]. Changes that were made to address 50110 are not reflected.*

As it is clear from the memory environs of above code that the moment you make the parameter refer to another address by assigning a different variable name, it loses the address of the actual argument passed to it. So, whatever changes are made to it post this will not get reflected to original argument even though it was mutable because the addresses became different. So we can summarize the mutable behaviour as summarized below :


Mutable Argument (say A) assigned to mutable parameter (say P)

if P is not **assigned** any new name or different data type value then changes in P are performed in place and GET REFLECTED BACK to __main__ (refer same codes 2.1 and 2.2). That is, A will show the changed data.

if P is assigned a new name or different data type value (e.g., P = different_variable then any changes in P are not reflected) BACK to __main__ (refer sample code 2.3). That is, A will not show changed data.

Figure 3.5 Mutable types' behaviour with functions.

Check Point

3.1

1. If return statement is not used inside the function, the function will return:

   (a) 0 (b) None object

   (c) an arbitrary integer

   (d) Error! Functions in Python must have a return statement.

2. Which of the following keywords marks the beginning of the function block?

   (a) func (b) define (c) def (d) function

3. What is the area of memory called, which stores the parameters and local variables of a function call ?

   (a) a heap (b) storage area

   (c) a stack (d) an array

4. Find the errors in following function definitions :

   (a)
```
def main( )
    print ("hello")
```
   (b)
```
def func2() :
    print (2 + 3)
```
   (c)
```
def compute( ) :
    print (x * x)
```
   (d)
```
square (a)
    return a * a
```

That means, we can say that :

- Changes in immutable types are not reflected in the caller function at all.
- Changes, if any, in mutable types
  - are reflected in caller function if its name is not assigned a different variable or datatype.
  - are not reflected in the caller function if it is assigned a different variable or datatype.

MUTABILITY/IMMUTABILITY OF ARGUMENTS

Progress In Python 3.4

This 'Progress in Python' session is aimed at these concepts : *Mutability/Immutability of the arguments, parameters and their behaviour in various situations.*

⋮

>>>❖<<<

Solved Problem

1. *What is the significance of having functions in a program ?*

   Solution. Creating functions in programs is very useful. It offers following advantages :

   (i) *The program is easier to understand.*

   Main block of program becomes compact as the code of functions is not part of it, thus is easier to read and understand.

   (ii) *Redundant code is at one place, so making changes is easier.*

   Instead of writing code again when we need to use it more than once, we can write the code in the form of a function and call it more than once. If we later need to change the code, we change it in one place only. Thus it saves our time also.

   (iii) *Reusable functions can be put in a library in modules.*

   We can store the reusable functions in the form of modules. These modules can be imported and used when needed in other programs.

2. *From the program code given below, identify the parts mentioned below :*

```
def processNumber(x):
    x = 72
    return x + 3

y = 54
res = processNumber(y)
```

Identify these parts : *function header, function call, arguments, parameters, function body, main program*

Solution.

Function header	:	`def processNumber(x) :`
Function call	:	`processNumber(y)`
Arguments	:	`y`
Parameters	:	`x`

Function body	:	`x = 72`
		`return x + 3`

Main program	:	`y = 54`
		`res = processNumber(y)`

3. *Trace the following code and predict output produced by it.*

```
def power(b, p) :
    y = b ** p
    return y

def calcSquare(x) :
    a = power(x, 2)
    return a

```

```
9.  n = 5
10. result = calcSquare(n) + power(3, 3)
11. print(result)
```

Solution. Flow of execution for above code will be :

1 → 5 → 9 → 10 → 5 → 6 → 1 → 2 → 3 → 6 → 7 → 10 → 1 → 2 → 3 → 10 → 11

The output produced by above code will be :

```
52
```

4. *Trace the flow of execution for following programs :*

(a)

```
1   def power(b, p):
2       r = b ** p
3       return r
4
5   def calcSquare(a):
6       a = power(a, 2)
7       return a
8
9   n = 5
10  result = calcSquare(n)
11  print (result)
```

(b)

```
1. def increment(x) :
2.     x = x + 1
3.
4. # main program
5. x = 3
6. print(x)
7. increment(x)
8. print(x)
```

(c)

```
1.  def increment(x):
2.      z = 45
3.      x = x + 1
4.      return x
5.
6.  # main
7.  y = 3
8.  print(y)
9.  y = increment(y)
10. print(y)
11. q = 77
12. print(q)
13. increment(q)
14. print(q)
15. print(x)
16. print(z)
```

Solution.

(a) 1 → 5 → 9 → 10 → 5 → 6 → 1 → 2 → 3 → 6 → 7 → 10 → 11

(b) 1 → 5 → 6 → 7 → 1 → 2 → 8

Control didnot return to function call statement (7) as nothing is being returned by increment( )

(c) 1 → 7 → 8 → 9 → 1 → 2 → 3 → 4 → 9 → 10 → 11 → 12 → 13 → 1 → 2 → 3 → 4 → 14 → 15 → 16

5. *What is the difference between the formal parameters and actual parameters? What are their alternative names? Also, give a suitable Python code to illustrate both.*

Solution. ***Actual Parameter*** is a parameter, which is used in *function call* statement to send the value from *calling function* to the *called function*. It is also known as ***Argument***.

Formal Parameter is a parameter, which is used in *function header* of the *called function* to receive the value from *actual parameter*. It is also known as ***Parameter***.

For example,

```
def addEm(x, y, z):
    print(x + y + z)

addEm(6, 16, 26)
```

In the above code, *actual parameters* are *6, 16* and *26* ; and *formal parameters* are *x, y* and *z*.

6. *Consider a function with following header :*

```
def info(object, spacing = 10, collapse = 1):
```

Here are some function calls given below. Find out which of these are correct and which of these are incorrect stating reasons :

```
a.  info( obj1 )
b.  info(spacing = 20)
c.  info( obj2, 12 )
d.  info( obj11, object = obj12)
e.  info( obj3, collapse = 0)
f.  info( )
g.  info(collapse = 0, obj3 )
e.  info( spacing = 15, object = obj4 )
```

Solution.

(*a*)	Correct	*obj1* is for positional parameter **object** ; **spacing** gets its default value of *10* and **collapse** gets its default value of *1*.
(*b*)	Incorrect	Required positional argument (object) missing ; required arguments cannot be missed.
(*c*)	Correct	Required parameter **object** gets its value as *obj2* ; **spacing** gets value *12* and for skipped argument **collapse**, default value *1* is taken.
(*d*)	Incorrect	Same parameter object is given multiple values – one through positional argument and one through keyword(named) argument.
(*e*)	Correct	Required parameter **object** gets its value as *obj3* ; **collapse** gets value *0* and for skipped argument **spacing**, default value *10* is taken.
(*f*)	Incorrect	Required parameter **object's** value cannot be skipped.
(*g*)	Incorrect	Positional arguments should be before keyword arguments.
(*h*)	Correct	Required argument **object** gets its value through a keyword argument.

7. *What is the default return value for a function that does not return any value explicitly ?*

   (*a*) None (*b*) int (*c*) double (*d*) null

Solution. (*a*)

8. *What will following code print ?*

```
def addEm(x, y, z):
    print(x + y + z)

def prod (x, y, z) :
    return x * y * z

a = addEm(6, 16, 26)
b = prod ( 2, 3, 6)
print(a, b)
```

Solution.

```
None 36
```

9. *In the previous question's code, identify the void and non-void functions. The previous code stores the return values of both void and non-void functions in variables. Why did Python not report an error when void functions do not return a value?*

Solution.

Void function	:	addEm( )
Non-void function	:	prod( )

In Python, void functions do not return a value; rather they report the absence of returning value by returning None, which is legal empty value in Python. Thus, variable *a* stores None and it is not any error.

10. *Consider below given function headers. Identify which of these will cause error and why ?*

    (*i*) `def func(a = 1, b):`

    (*ii*) `def func(a = 1, b, c = 2):`

    (*iii*) `def func(a = 1, b = 1, c = 2):`

    (*iv*) `def func(a = 1, b = 1, c = 2, d):`

Solution. Function headers (*i*), (*ii*) and (*iv*) will cause error because **non-default arguments cannot follow default arguments.**

Only function header (*ii*) will not cause any error.

11. *What is the difference between a local variable and a global variable ? Also, give a suitable Python code to illustrate both.*

Solution. The differences between a local variable and global variable are as given below :

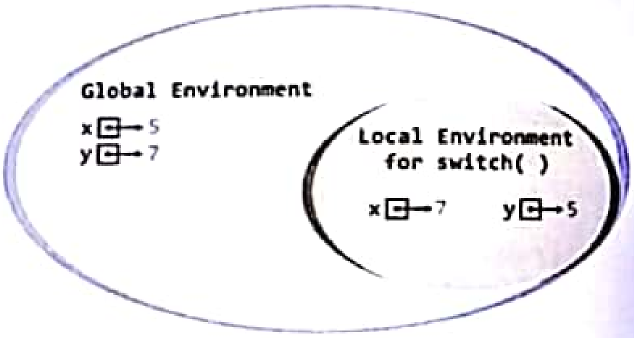
	Local Variable	Global Variable
1.	It is a variable which is declared within a function or within a block	It is variable which is declared outside all the functions
2.	It is accessible only within a function/block in which it is declared	It is accessible throughout the program

For example, in the following code, *x, xCubed* are global variables and *n* and *cn* are local variables.

```
def cube(n):
    cn = n * n * n
    return cn

x = 10
xCubed = cube(x)
print(x, "cubed is", xCubed)
```

12. *Following code intends to swap the values of two variables through a function, but upon running the code, the output shows that the values are swapped inside the switch() function but back again in main program, the variables remain un-swapped. What could be the reason? Suggest a remedy.*

```
def switch(x, y):
    x, y = y, x
    print("inside switch :", end = ' ')
    print("x =", x, "y =", y)

x = 5
y = 7
print("x =", x, "y =", y)
switch(x, y)
print("x =", x, "y =", y)
```



Solution. The reason for un-reflected changes in the main program is that although both main program and *switch*( ) have variables with same names *i.e., x* and *y*, but their scopes are different as shown in the adjacent figure.

The scope of *x* and *y* of *switch( )* is local. Though they are swapped in the namespace of *switch( )* but their namespace is removed as soon as control returns to main program. The global variables *x* and *y* remain unchanged as *switch( )* worked with a different copy of values not with original values.

The remedy of above problem is that the *switch( )* gets to work with global variables so that changes are made in the global variables. This can be done with the help of global statement as shown below:

```
def switch(x, y):
    global x, y
    x, y = y, x
    print("inside switch :", end = ' ')
    print("x =", x, "y =", y)
x = 5
y = 7
print("x =", x, "y =", y)
switch (x, y)
print("x =", x, "y=", y)
```

Now the above program will be able to swap the values of variables through switch( ).
(Though, now passing parameter is redundant.)

13. *Following code intends to add a given value to global variable a. What will the following code produce?*

```
1.  def increase(x) :
2.        a = a + x
3.        return
4.
5.  a = 20
6.  b = 5
7.  increase(b)
8.  print(a)
```

Solution. The above code will produce an error.

The reason being whenever we assign something to a variable inside a function, the variable is created as a local variable. Assignment creates a variable in Python.

Thus at line 2, when variable *a* is incremented with the passed value *x*, Python tries to create a local variable *a* by assigning to it the value of expression on the right hand side of assignment. But variable *a* also appears in the right hand side of assignment, which results in error because a is undeclared so far in function.

To assign some value to a global variable from within a function, without creating a local variable with the same name, global statement can be used. So, if we add

```
global a
```

in the first line of function body, the above error will be corrected. Python won't create a local variable *a*, rather will work with global variable *a*.

14. *Which names are local, which are global and which are built-in in the following code fragment?*

```
invaders = 'Big names'
pos = 200
level = 1

def play( ) :
    max_level = level + 10
    print(len (invaders ) == 0)
    return max_level


res = play()
print(res)
```

Solution.

Global names	:	invaders, pos, level, res
Local names	:	max_level
Built in	:	len

15. *Predict the output of the following code fragment ?*

```
def func(message, num = 1):
    print(message * num)

func('Python')
func('Easy', 3)
```

Solution.

```
Python
EasyEasyEasy
```

16. *Predict the output of the following code fragment ?*

```
def check(n1 = 1, n2 = 2):
    n1 = n1 + n2
    n2 += 1
    print(n1, n2)

check()
check(2, 1)
check(3)
```

Solution.

```
3 3
3 2
5 3
```

17. *What is the output of the following code?*

```
a = 1
def f ( ) :
    a = 10
print(a)
```

Solution. The code will print 1 to the console.

18. *What will be the output of following code?*

```
def interest (prnc, time =2 , rate = 0.10) :
    return (prnc * time * rate)

print(interest (6100, 1))
print(interest (5000, rate = 0.05))
print(interest (5000, 3, 0.12 ))
print(interest (time = 4, prnc = 5000))
```

Solution.

```
610.0
500.0
1800.0
2000.0
```

19. *Is return statement optional ? Compare and comment on the following two return statements :*

```
return
return val
```

Solution. The return statement is optional ONLY WHEN the function is *void* or we can say that when the function does not return a value. A function that returns a value, must have at least one *return* statement.

From given two return statements, statement

```
return
```

is not returning any value, rather it returns the control to caller along with empty value *None*. And the statement

```
return val
```

is returning the control to caller along with the value contained in variable *val*.

20. *Write a function that takes a positive integer and returns the one's position digit of the integer.*

Solution.

```
def getOnes(num):
    # return the ones digit of the integer num
    onesDigit = num % 10
    return onesDigit
```

21. *Write a function that receives an octal number and prints the equivalent number in other number bases i.e., in decimal, binary and hexadecimal equivalents.*

Solution.

```
def oct2others( n) :
    print("Passed octal number :", n)
    numString = str(n)
    decNum = int( numString, 8)
    print("Number in Decimal :", decNum)
    print("Number in Binary :", bin(decNum))
    print("Number in Hexadecimal :", hex(decNum))

num = int(input("Enter an octal number :"))
oct2others(num)
```

Please recall that bin( ) and hex( ) do not return numbers but return the string-representations of equivalent numbers in binary and hexadecimal number systems respectively.

22. *Write a program that generates 4 terms of an AP by providing initial and step values to a function that returns first four terms of the series.*

Solution.

```
def retSeries(init, step ):
    return init, init+step, init+2*step, init+3*step

ini = int(input("Enter initial value of the AP series :"))
st = int(input("Enter step value of the AP series :"))
print("Series with initial value", ini, "& step value", st, "goes as:")
t1, t2, t3, t4 = retSeries(ini, st)
print(t1, t2, t3, t4)
```

4

Using Python Libraries

In This Chapter

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.

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.

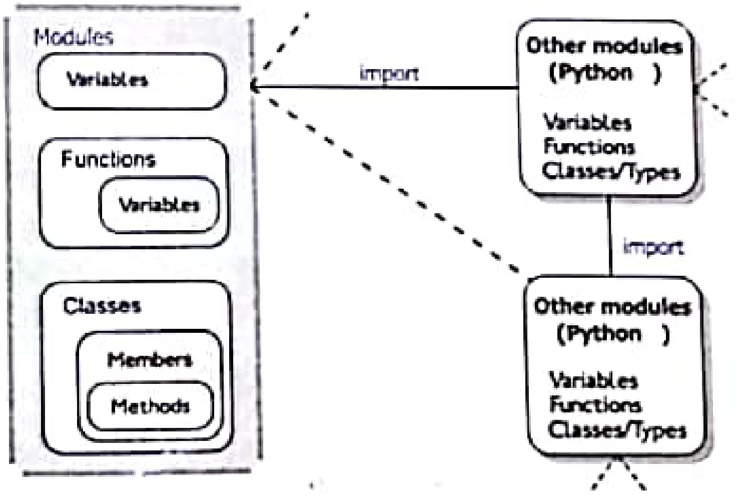


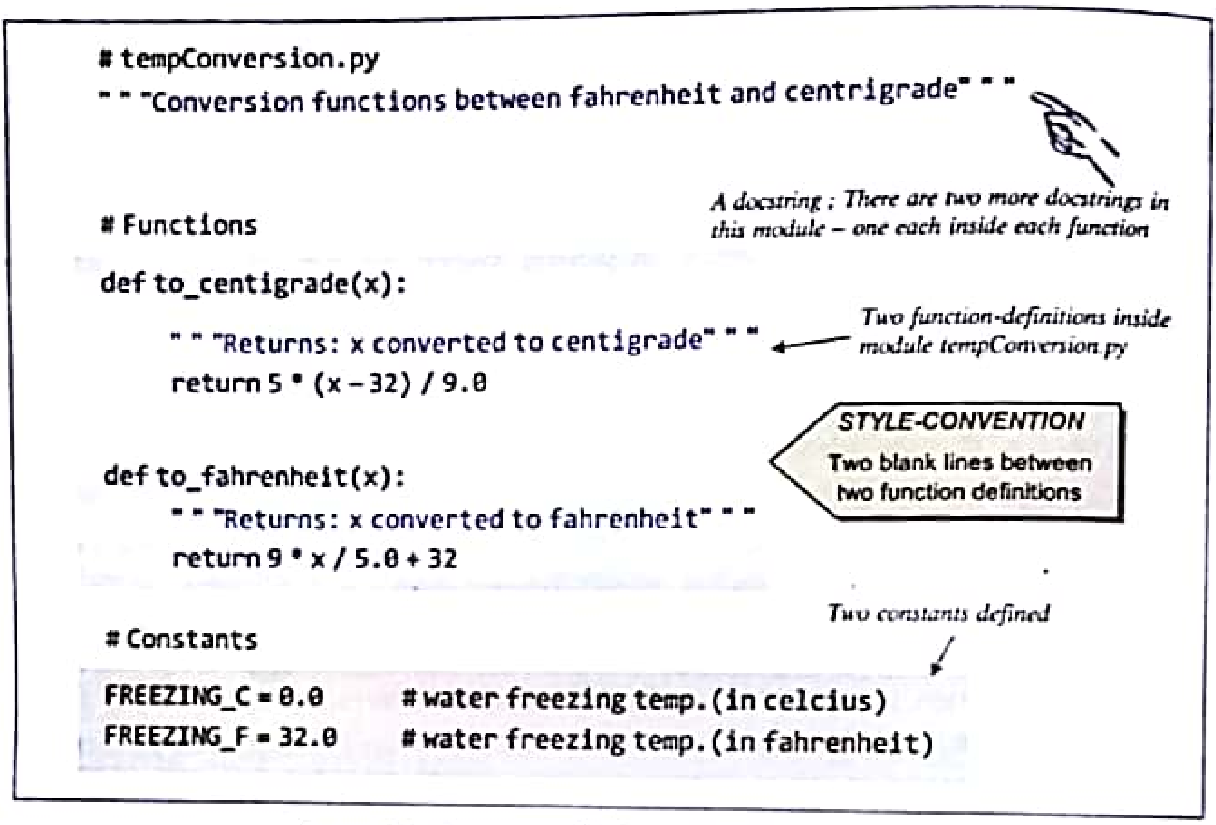
Figure 4.1 Composition/Structure of a Python Module

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"""


# Functions
def to_centigrade(x):
    """Returns: x converted to centigrade"""
    return 5 * (x - 32) / 9.0


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)
```

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

Two function-definitions inside module tempConversion.py

STYLE-CONVENTION

Two blank lines between two function definitions

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

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

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.

```
>>> 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.

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

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

> **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.

```
>>> 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 :

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.

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

calling function to_centrigrade( ) of imported module tempConversion

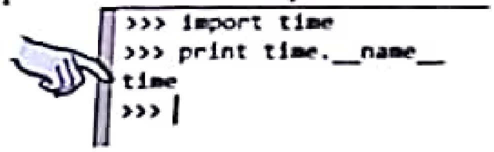
The name of a module is stored inside a constant **__name__** (prefix & suffix are having two underscores). You can use it like :

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

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

NOTE

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



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

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].

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

- 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*. Figure 4.3 illustrates this difference.

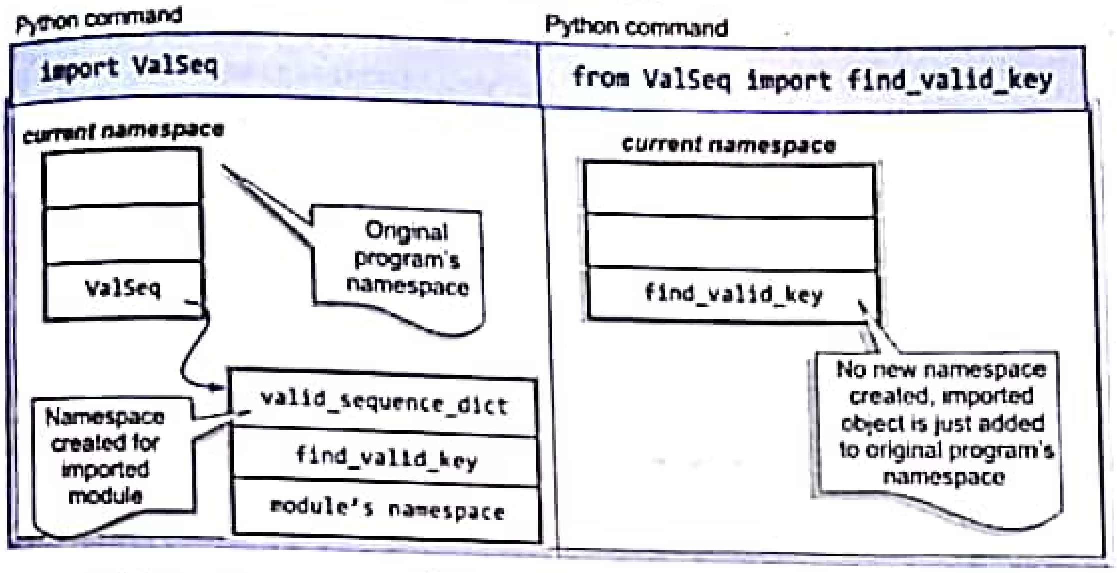

Python command
import ValSeq
current namespace
ValSeq
Original program's namespace
valid_sequence_dict
find_valid_key
module's namespace
Namespace created for imported module
Python command
from ValSeq import find_valid_key
current namespace
find_valid_key
No new namespace created, imported object is just added to original program's namespace

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

PriP 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.

Mathematical and String Functions

Consider the following example program that uses some built-in functions of Python :

◇ **oct(<integer>)** returns octal string for given numbers *i.e.*, 00 + octal equivalent of number.

◇ **hex(<integer>)** returns hex string for given numbers *i.e.*, 0x + hexadecimal equivalent of number.

Program 4.1 *Write a program that reads a number, then converts it into octal and hexadecimal equivalent numbers using built-in functions of Python.*

```
# program using built in functions
num = int(input( "Enter a number :" ))
print("Number entered =", num)
onum = oct(num)                 # oct( ) converts to octal number-string
hnum = hex(num)                 # hex( ) converts to hexadecimal number-string
print("Octal conversion yields", onum)
print("Hexadecimal conversion yields", hnum)
```

The output produced by above program would be :

```
Enter a number : 17
Number entered = 17
Octal conversion yields  0o21
Hexadecimal conversion yields  0x11
```

In the above code, we have used functions *hex( )* and *oct( )* for hexadecimal and octal conversions of a number respectively. These are Python's built-in functions that take a number and return string-representations of hexadecimal and octal equivalents respectively of the given number. There is one more function *bin( )* that gives string representation of binary equivalent of given number.

Please note that *oct( ), hex( )* and *bin( )* do not return a number ; they return a string representation of converted number.

Similarly, 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 4.2 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.*

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.*,

```
In[ ]:"*".join("Hello")
Out[ ]: 'H*e*l*l*o'
```

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

```
In[ ]:"***".join("TRIAL")
Out[ ]: '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.

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

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

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

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

```
In[ ]:"###".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.*,

```
In[ ] :"I Love Python".split()
Out[ ]: ['I', 'Love', 'Python']

In[ ] :"I Love Python".split(" ")
Out[ ]: ['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.*,

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

The given string is divided from positions containing "o"

<str>.replace( )

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

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

4.3 Program

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.

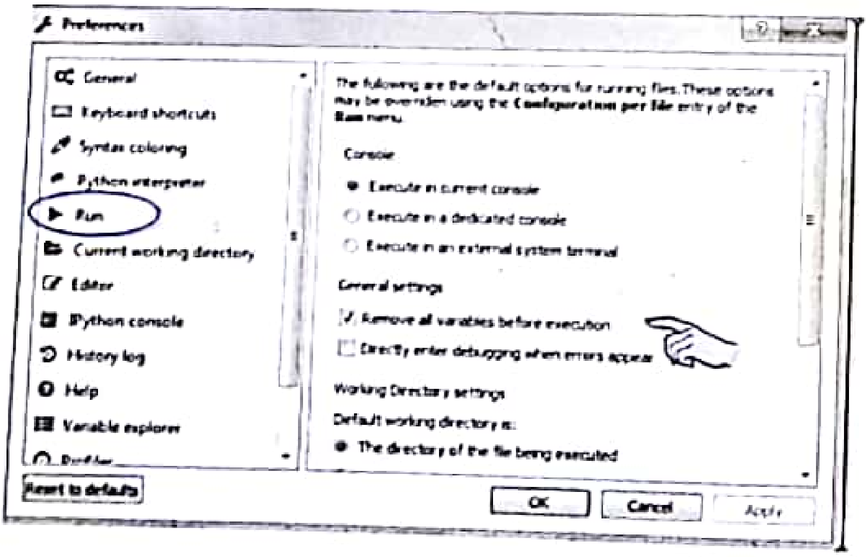
The best way to avoid this is that you clear all the variables in memory before running your program. For this you can do one of the following *two* things :

(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)



The following are the default options for running files. These options may be overriden using the **Configuration per file** entry of the **Run** menu.

Console

● Execute in current console

○ Execute in a dedicated console

○ Execute in an external system terminal

General settings

☑ Remove all variables before execution

□ Directly enter debugging when errors appear

Working Directory settings

Default working directory is:

● The directory of the file being executed

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 *random* and *urllib* 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
```

Two 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.,* $0.0 \leq N < 1.0$. 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.,* $a \leq N \leq b$ (both range-limits are inclusive). Remember, it generates an integer.

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())  ———  The output generated is between range [0.0, 1.0)
0.022353193431 ←
```

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

   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
```

[6] Please note that learning how to use modules is important for learning how to use libraries. Also, the modules being covered in this section are required as per practical suggestions given in the syllabus.

[7] In fact, pseudo-random numbers because it is generated via some algorithm or procedure and hence deterministic somewhere.

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 urllib and Webbrowser Modules

Python offers you module **urllib** using which you can send http requests and receive the result from within your Python program. To use **urllib** in your program to get details about a website, you need to first import it using command :

```
import urllib
```

The urllib module is a collection of sub-modules like *request*, *error*, *parse* etc. While we shall not go in detailed discussion of urllib modules, you shall learn to use *urllib.request* that is primarily used for opening and fetching URLs. You can use following functions of urllib for the purpose mentioned below :

(*i*)	`urllib.request.urlopen(<URL>)`	opens a website or network object denoted by URL for reading and returns a file-like object (say **wurL**), using which other functions are often used
(*ii*)	`<urlopen's returnval>.read( )`	returns the html or the source code of given url opened via urlopen( )
(*iii*)	`<urlopen' return val>.getcode( )`	returns HTTP status code where 200 means 'all okay'. 301, 302 mean some kind of redirection happened.
(*iv*)	`<urlopen's return val>.headers`	Stores metadata about the opened URL
(*v*)	`<urlopen's return val>.info( )`	returns same information as stored by headers
(*vi*)	`<copy>.geturl( )`	returns the url string

Following program will use all above functions and *urllib*, but before that let us talk about webbrowser module, using which you can open a URL in a window/tab.

The webbrowser module provides functionality to open a website in a window or tab of webbrowser on your computer, from within your program. In order to use webbrowser module, you need to import it in your program by giving following command.

```
import webbrowser
```

Once imported you can use any of the following functions :

```
webbrowser.open(<URL in quotes>)
webbrowser.open_new(<URL in quotes>)
webbrowser.open_new_tab(<URL in quotes>)
```

All these open functions will open the given URL in the same window or a new window or a new tab respectively.

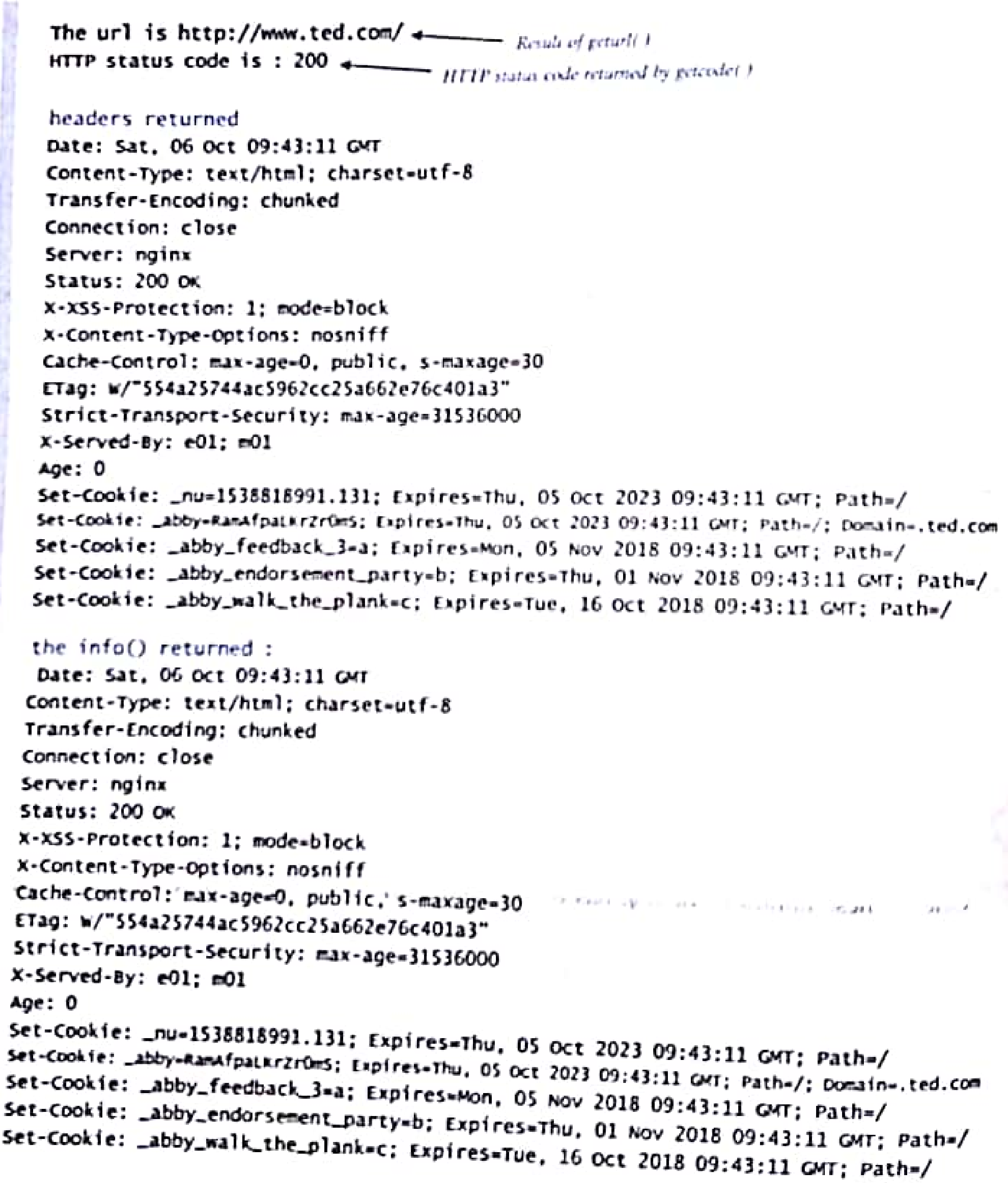
Following program uses the above mentioned functions of **urllib** and **webbrowser** modules to open a website.

Program **4.4** Write a program to get http request information from url www.ted.com and open it from within your program.

```
import urllib
import webbrowser
weburl = urllib.request.urlopen('http://www.ted.com/')
html = weburl.read()
data = weburl.getcode()
url = weburl.geturl()
hd = weburl.headers
inf = weburl.info()
print("The url is", url)
print("HTTP status code is :", data)
print("headers returned \n", hd)
print("the info() returned :\n",inf)
print("Now opening the url", url)
webbrowser.open_new(url)
```

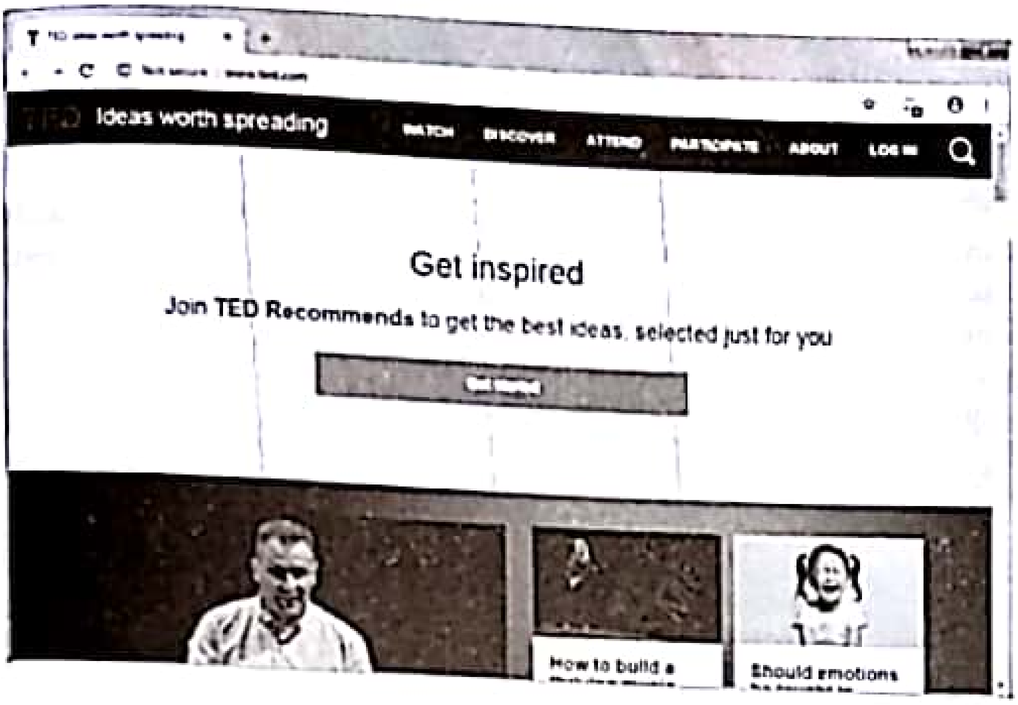
We have not printed the html code obtained via read( ) as it is too large to print here in the book. You , however, can print it in your Python code to see it.

The output produced by above code is :



```
The url is http://www.ted.com/ ←——— Result of geturl( )
HTTP status code is : 200 ←——— HTTP status code returned by getcode( )

headers returned
Date: Sat, 06 Oct 09:43:11 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: close
Server: nginx
Status: 200 OK
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Cache-Control: max-age=0, public, s-maxage=30
ETag: W/"554a25744ac5962cc25a662e76c401a3"
Strict-Transport-Security: max-age=31536000
X-Served-By: e01; m01
Age: 0
Set-Cookie: _nu=1538818991.131; Expires=Thu, 05 Oct 2023 09:43:11 GMT; Path=/
Set-Cookie: _abby=RamAfpaLkrZrOmS; Expires=Thu, 05 Oct 2023 09:43:11 GMT; Path=/; Domain=.ted.com
Set-Cookie: _abby_feedback_3=a; Expires=Mon, 05 Nov 2018 09:43:11 GMT; Path=/
Set-Cookie: _abby_endorsement_party=b; Expires=Thu, 01 Nov 2018 09:43:11 GMT; Path=/
Set-Cookie: _abby_walk_the_plank=c; Expires=Tue, 16 Oct 2018 09:43:11 GMT; Path=/

the info() returned :
Date: Sat, 06 Oct 09:43:11 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: close
Server: nginx
Status: 200 OK
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Cache-Control: max-age=0, public, s-maxage=30
ETag: W/"554a25744ac5962cc25a662e76c401a3"
Strict-Transport-Security: max-age=31536000
X-Served-By: e01; m01
Age: 0
Set-Cookie: _nu=1538818991.131; Expires=Thu, 05 Oct 2023 09:43:11 GMT; Path=/
Set-Cookie: _abby=RamAfpaLkrZrOmS; Expires=Thu, 05 Oct 2023 09:43:11 GMT; Path=/; Domain=.ted.com
Set-Cookie: _abby_feedback_3=a; Expires=Mon, 05 Nov 2018 09:43:11 GMT; Path=/
Set-Cookie: _abby_endorsement_party=b; Expires=Thu, 01 Nov 2018 09:43:11 GMT; Path=/
Set-Cookie: _abby_walk_the_plank=c; Expires=Tue, 16 Oct 2018 09:43:11 GMT; Path=/
```

```
Now opening the url http://www.ted.com/
```



Get inspired

Join TED Recommends to get the best ideas, selected just for you

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.

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.

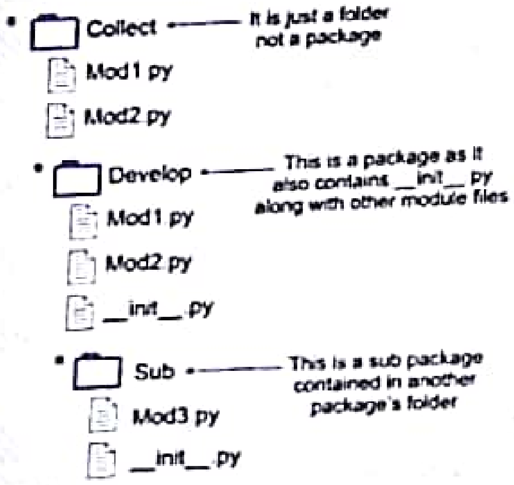

Collect
It is just a folder not a package
Mod1.py
Mod2.py
Develop
This is a package as it also contains __init__.py along with other module files
Mod1.py
Mod2.py
__init__.py
Sub
This is a sub package contained in another package's folder
Mod3.py
__init__.py

Figure 4.4 A package vs. folder.

4.5.2 Procedure For Creating Packages

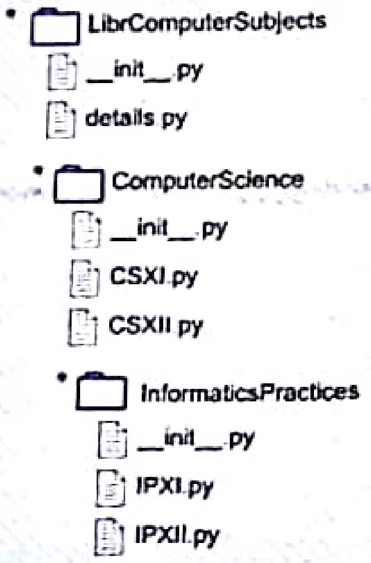
In order to create a package, you need to follow a certain procedure as discussed below.

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.

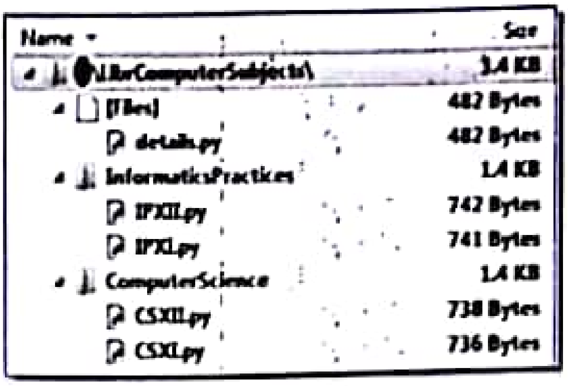
(Just keep in mind that while naming the folders/subfolders, keep the words in the name underscore-separated; don't use any other word separators, at all (not even hyphens))

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).



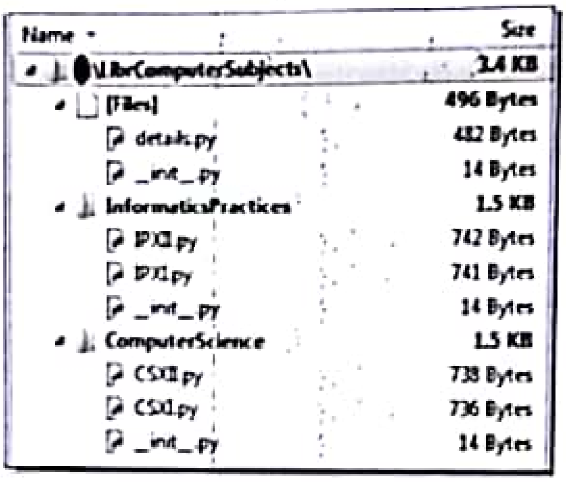
Name	Size
\LibrComputerSubjects\	3.4 KB
[Files]	482 Bytes
details.py	482 Bytes
InformaticsPractices	1.4 KB
IPXII.py	742 Bytes
IPXI.py	741 Bytes
ComputerScience	1.4 KB
CSXII.py	738 Bytes
CSXI.py	736 Bytes

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.

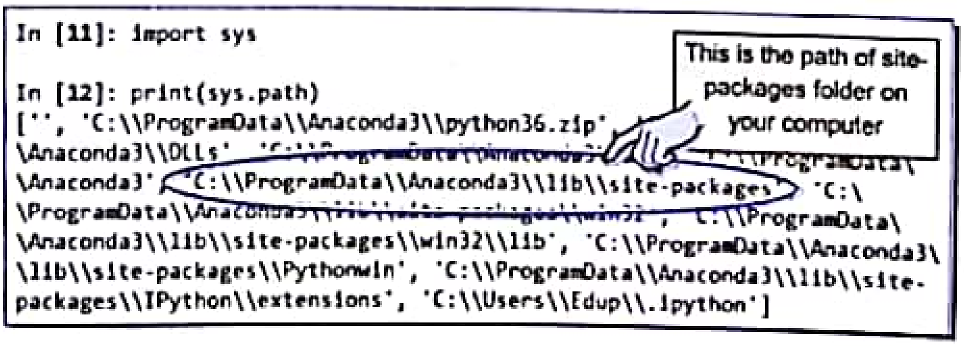


Name	Size
\LibrComputerSubjects\	3.4 KB
[Files]	496 Bytes
details.py	482 Bytes
init.py	14 Bytes
InformaticsPractices	1.5 KB
IPXII.py	742 Bytes
IPXI.py	741 Bytes
init.py	14 Bytes
ComputerScience	1.5 KB
CSXII.py	738 Bytes
CSXI.py	736 Bytes
init.py	14 Bytes

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' [illegible]
\Anaconda3\\DLLs', 'C:\\ProgramData\\Anaconda3 [illegible] 'C:\\ProgramData\
\Anaconda3', 'C:\\ProgramData\\Anaconda3\\lib\\site-packages', 'C:\
\ProgramData\\Anaconda3\\lib\\site-packages\\win32', '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 library/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( )*.

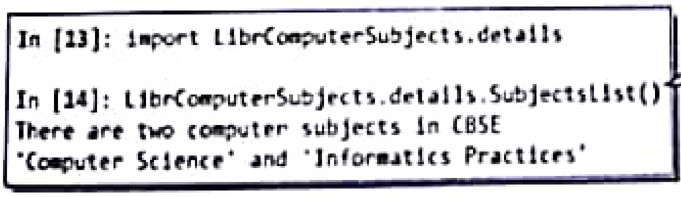
(*i*) To import module *details*, you can write import command as :

```
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

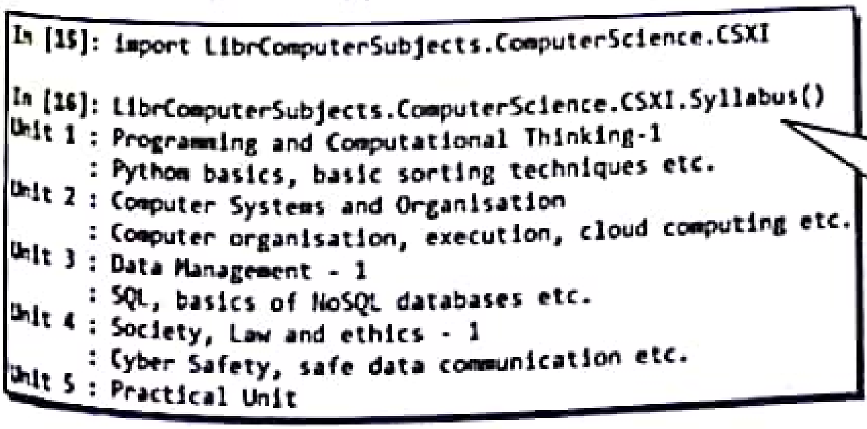
(*ii*) To import module *CSXI*, you can write import command as :

```
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
       : Python 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

PriP

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 urllib module lets you send and receive http requests and their results and **webbrowser** lets you open a webpage from within a Python program.*
- *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.*

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. *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[1] 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
```

8. *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

9. *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))))
```

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.

10. *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

[1] Refer to Appendix A for more details.

(*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.

11. *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.

12. *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.

13. *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.

5

File Handling

In This Chapter

5.1 INTRODUCTION

Most computer programs work with files. This is because files help in storing information permanently. Word processors create document files ; Database programs create files of information ; Compilers read source files and generate executable files. So, we see, it is the files that are mostly worked with, inside programs. *A file in itself is a bunch of bytes stored on some storage device like hard-disk, thumb-drive etc.* Every programming language offers some provision to use and create files through programs. Python is no exception and in this chapter, you shall be learning to work with data files through Python programs.

5.2 DATA FILES

The data files are the files that store data pertaining to a specific application, for later use. The data files can be stored in *two* ways :

- Text files
- Binary files

1. Text Files

A text file stores information in ASCII or Unicode characters (the one which is default for your programming platform). In text files, each line of text is terminated, (delimited) with a special character known as EOL (End of Line) character. In text files, some internal translations take place when this EOL character is read or written. In Python, by default, this EOL character is the newline character ('\n') or carriage-return, newline combination ('\r\n').

2. Binary Files

A binary file is just a file that contains information in the same format in which the information is held in memory, *i.e.*, the file content that is returned to you is raw (with no translation or no specific encoding). In binary file, there is no delimiter for a line. Also no translations occur in binary files. As a result, binary files are faster and easier for a program to read and write than are text files. As long as the file doesn't need to be read by people or need to be ported to a different type of system, binary files are the best way to store program information.

5.3 OPENING AND CLOSING FILES

In order to work with a file from within a Python program, you need to open it in a specific mode as per the file manipulation task you want to perform. The most basic file manipulation tasks include *adding, modifying or deleting data* in a file, which in turn include any one or combination of the following operations :

- reading data from files
- writing data to files
- appending data to files

Python provides built-in functions to perform each of these tasks. But before you can perform these functions on a file, you need to first open the file.

5.3.1 Opening Files

In data file handling through Python, the first thing that you do is open the file. It is done using *open*( ) function as per one of the following syntaxes :

```
<file_objectname> = open(<filename>)
<file_objectname> = open(<filename>, <mode>)
```

For example,

```
myfile = open("taxes.txt")
```

Python will look for this file in current working directory

The above statement opens file "**taxes.txt**" in file mode as ***read mode*** (default mode) and attaches it to ***file object namely myfile.***

Consider another statement :

```
file2 = open("data.txt", "r")
```

The above statement opens the file "**data.txt**" in ***read mode*** (because of "r" given as mode) and attaches it to file object namely ***file2.***

Consider one more file-open statement :

```
file3 = open("e:\\main\\result.txt", "w")
```

Python will look for this file in E:\main folder

The above statement opens file **"result.txt"** (stored in folder E:\ main) in write mode (because of "w" given as mode) and attaches it to file object namely *file3*.

(Notice that in above file open statement, the file path contains double slashes in place of single slashes.)

NOTE

A ***file-object*** is also known as ***file-handle.***

The above given *three* file-open statements must have raised these questions in your mind :

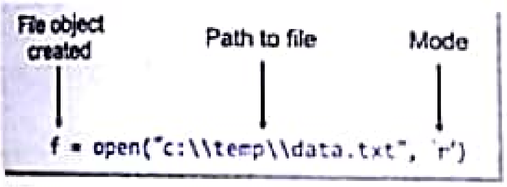
(i) *What is file-object ?* (ii) *What is mode or file-mode ?*

The coming lines will have answers to all your questions, but for now, let us summarize the file *open( )* function.

- Python's *open( )* function creates a *file object* which serves as a link to a file residing on your computer.
- The first parameter for the *open( )* function is a path to the file you'd like to open. If just the file name is given, then Python searches for the file in the current folder.
- The second parameter of the open function corresponds to a mode which is typically read ('r'), write ('w'), or append ('a'). If no second parameter is given, then by default it opens it in read ('r') *mode.*

NOTE

Please note that when you open a file in *readmode*, the given file must exist in the folder, otherwise Python will raise FileNotFoundError.



```
f = open("c:\\temp\\data.txt", 'r')
```

Figure 5.1 Working of File open( ) function.

Figure 5.1 summarizes the *open( )* function of Python for you.

As you can see in Fig. 5.1, the slashes in the path are doubled. This is because the slashes have special meaning and to suppress that special meaning escape sequence for slash *i.e.,* \\ is given.

However, if you want to write with single slash, you may write in raw string as :

with r here, you can give single slashes in pathnames

```
f = open(r "c :\temp\data.txt", "r")
```

The prefix *r* in front of a string makes it *raw string* that means there is no special meaning attached to any character. Remember, following statement might give you incorrect result :

NOTE

The prefix *r* in front of a string makes it ***raw string*** that means there is no special meaning attached to any character.

```
f = open("c :\temp\data.txt", "r")
```

This might give incorrect result as \t is tab character

Reason being that '\t' will be treated as tab character and '\d' on some platforms as numeric-digit or some error.

Thus the *two* ways to give paths in filenames correctly are :

(i) Double the slashes *e.g.,*

```
f = open("c:\\temp\\data.txt", "r")
```

NOTE

The default file-open mode is read mode, *i.e.,* If you do not provide any file open mode, Python will open it in read mode ("r").

(*ii*) Give raw string by prefixing the file-path string with an *r* *e.g.*,

```
f = open(r"c:\temp\data.txt", "r")
```

NOTE

If you just give the file name without its path (e.g., just "data.txt" rather than "c :\\temp\\data.txt"), Python will open/create the file in the same directory in which the module file is stored.

5.3.1A File Object/File Handle

File objects are used to read and write data to a file on disk. The file object is used to obtain a reference to the file on disk and open it for a number of different tasks.

File object (also called ***file-handle***) is very important and useful tool as through a *file-object* only, a Python program can work with files stored on hardware. All the functions that you perform on a data file are performed through file-objects.

FILE OBJECT

A file object is a reference to a file on disk. It opens and makes it available for a number of different tasks.

When you use file open( ), Python stores the reference of mentioned file in the file-object. A file-object of Python is a stream of bytes where the data can be read either byte by byte or line by line or collectively. All this will be clear to you in the coming lines (under topic – *File access modes*).

5.3.1B File Access Modes

When Python opens a file, it needs to know the file-mode in which the file is being opened. A file-mode governs the type of operations (such as *read* or *write* or *append*) possible in the opened file *i.e.*, it refers to how the file will be used once it's opened. File modes supported by Python are being given in Table 5.1.

Table 5.1 *File-modes*

Text File Mode	Binary File Mode	Description	Notes
'r'	'rb'	read only	❖ File must exist already, otherwise Python raises I/O error.
'w'	'wb'	write only	❖ If the file does not exist, file is created. ❖ If the file exists, Python will truncate existing data and over-write in the file. So this mode must be used with caution.
'a'	'ab'	append	❖ File is in write only mode. ❖ If the file exists, the data in the file is retained and new data being written will be appended to the end. ❖ If the file does not exist, Python will create a new file.
'r+'	'r+b' or 'rb+'	read and write	❖ File must exist otherwise error is raised. ❖ Both reading and writing operations can take place.
'w+'	'w+b' or 'wb+'	write and read	❖ File is created if does not exist. ❖ If file exists, file is truncated (past data is lost). ❖ Both reading and writing operations can take place.
'a+'	'a+b' or 'ab+'	write and read	❖ File is created if does not exist. ❖ If file exists, file's existing data is retained ; new data is appended. ❖ Both reading and writing operations can take place.

Adding a 'b' to text-file mode makes it binary-file mode.

To create a file, you need to open a file in a mode that supports *write* mode (*i.e.*, 'w', or 'a' or 'w+' or 'a+' modes).

NOTE

A file-mode governs the type of operations (e.g., read/write/append) possible in the opened file *i.e.*, it refers to how the file will be used once it's opened.

5.3.2 Closing Files

An open file is closed by calling the *close( )* method of its file-object. Closing of file is important. In Python, files are automatically closed at the end of the program but it is good practice to get into the habit of closing your files explicitly. Why ? Well, the operating system may not write the data out to the file until it is closed (this can boost performance). What this means is that if the program exits unexpectedly there is a danger that your precious data may not have been written to the file! So the moral is : once you finish writing to a file, close it.

The *close( )* function accomplishes this task and it takes the following general form :

```
<fileHandle>.close( )
```

For instance, if a file **Master.txt** is opened *via* file-handle ***outfile***, it may be closed by the following statement :

```
outfile.close( )
```

The close( ) must be used with filehandle

NOTE

A close( ) function breaks the link of file-object and the file on the disk. After close( ), no tasks can be performed on that file through the file-object (or file-handle).

Please remember, **open( )** is a built-in function (used standalone) while **close( )** is a method used with file-handle object.

5.4 READING AND WRITING FILES

Python provides many functions for reading and writing the open files. In this section, we are going to explore these functions. Most common file reading and writing functions are being discussed in coming lines.

5.4.1 Reading from Files

Python provides mainly *three* types of *read functions* to read from a data file. But before you can read from a file, the file must be opened and linked via a ***file-object*** or ***file handle***. Most common file reading functions of Python are listed below in Table 5.2.

Table 5.2 *Python data files — reading writing functions*

S.No.	Method	Syntax	Description

1.	read( )	`<filehandle>.read([n])`	reads at most *n* bytes ; if no *n* is specified, reads the entire file.
			Returns the read bytes in the form of a *string*.
			`In [11]: file1 = open("E:\\mydata\\info.txt")`
			`In [12]: readInfo = file1.read(15)`
			`In [13]: print(readInfo)`
			`It's time to wo` 15 bytes read
			`In [14]: type(readInfo)`
			`Out[14]: str` Bytes read into string type

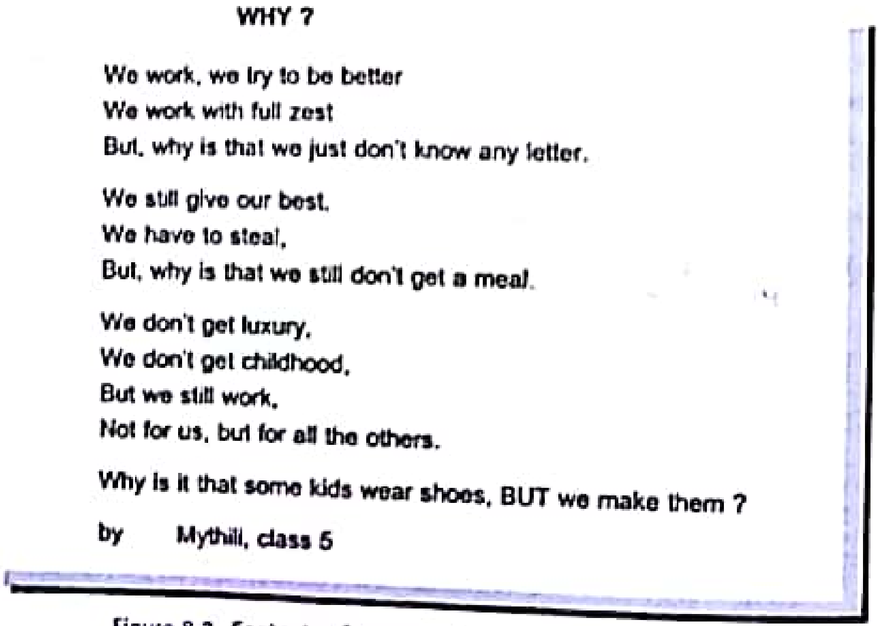
S.No.	Method	Syntax	Description

2.	readline( )	`<filehandle>.readline([n])`	reads a line of input ; if *n* is specified reads at most *n* bytes.
			Returns the read bytes in the form of a *string* ending with In(line) character or returns a blank string if no more bytes are left for reading in the file.
			`In [20]: file1 = open("E:\\mydata\\info.txt")`
			`In [21]: readInfo = file1.readline()`
			`In [22]: print(readInfo)`
			`It's time to work with files.`
			1 line read

3.	readlines( )	`<filehandle>.readlines()`	reads all lines and returns them in a *list*
			`In [23]: file1 = open("E:\\mydata\\info.txt")`
			`In [24]: readInfo = file1.readlines()`
			`In [25]: print(readInfo)`
			`["It's time to work with files.\n", 'Files offer and ease and power to store your work/data/information for later use.\n', 'simply create a file and store(write) in it .\n', 'Or open an existing file and read from it.\n']`
			All lines read
			`In [26]: type(readInfo)`
			`Out[26]: list`
			Read into list type

The `<filehandle>` in above syntaxes is the file-object holding open file's reference.

Let us consider some examples now. For the examples and explanations below, we are using a file namely *poem.txt* storing the content shown in Fig. 5.2.



WHY ?

We work, we try to be better
We work with full zest
But, why is that we just don't know any letter.

We still give our best.
We have to steal,
But, why is that we still don't get a meal.

We don't get luxury,
We don't get childhood,
But we still work,
Not for us, but for all the others.

Why is it that some kids wear shoes, BUT we make them ?

by Mythili, class 5

Figure 8.2 Contents of a sample file *poem.txt*

Code Snippet 1 *Reading a file's first 30 bytes and printing it*

```
myfile = open(r'E:\poem.txt', "r")
str = myfile.read( 30 )
print(str)
```

File-object created

File-object being used

See the number of bytes to be read specified as argument of read( )

When we specify number of bytes with read( ), Python will read only the specified number of bytes from the file. The next read( ) will start reading onwards from the last position read. *Code snippet* 2 illustrates this fact.

The output produced by above code is :

```
>>>
    WHY?
We work, we try
```

TIP

You may combine the file( ) or open( ) with the file-object's function, if you need to perform just a single task on the open file.

If need to perform just a single task with the open file, then you may combine the two steps of opening the file and performing the task, *e.g.*, following statement :

```
>>> file(r"E :\poem.txt', "r").read(30)
```

First function will open the file and the second function will perform with the result of first function, i.e., the reference of open file

would give the same result as that of above code.

For example, the following code will return the first line of file *poem.txt* :

```
open("poem.txt", "r").readline()
```

Code Snippet 2 *Reading n bytes and then reading some more bytes from the last position read*

```
myfile = open(r'E:\poem.txt', 'r')
str = myfile.read( 30 )
print(str)
str2 = myfile.read( 50 )
print(str2)
myfile.close()
```

reading 30 bytes

reading next 50 bytes

The output produced by above code is :

```
>>>
    WHY?
We work, we try

 to be better
We work with full zest
But, why is t
```

Output by first print statement, i.e., print(str)

print has entered a new line after its output i.e., here

Output by second print statement, i.e., print(str2)

If you do not want to have the newline characters inserted by print statement in the output then use end = ' ' argument in the end of *print*( ) statement so that print does not put any additional newline characters in the output. Refer to ***code snippets 3 and 4*** below.

Code Snippet 3 *Reading a file's entire content*

```
myfile = open(r'E :\poem.txt', "r")
str = myfile.read() ←
print(str)
myfile.close()
```

See when no value is specified as read()'s argument, entire file is read

The output produced by above code is :

```
>>>
     WHY?
We work, we try to be better
We work with full zest
But, why is that we just don't know any letter.
We still give our best.
We have to steal,
But, why is that we still don't get a meal.
We don't get luxury,
We don't get childhood,
But we still work,
Not for us, but for all the others.
Why is it that some kids wear shoes, BUT we make them ?
by      Mythili, class 5
```

Code Snippet 4 *Reading a file's first three lines - line by line*

```
myfile = open(r'E:\poem.txt', "r")
str = myfile.readline()
print(str, end = ' ')
str = myfile.readline()
print(str, end = ' ') ←
str = myfile.readline()
print(str, end = ' ')
myfile.close()
```

Argument end = ' ' at the end of print( ) statement will ensure that output shows exact content of the data file and no print-inserted newline characters are

The *readline( )* function will read a full line. A line is considered till a newline character (EOL) is encountered in the data file.

The output produced by above code is :

```
>>>
     WHY? ←———— Line 1
     ←———————— Line 2
We work, we try to be better ←—— Line 3
```

Code Snippet 5 *Reading a complete file - line by line*

```
myfile = open(r'E:\poem.txt', "r")
str = " "            #initially storing a space (any non-None value)
while str :
    str = myfile.readline( )
    print(str, end = ' ')
myfile.close( )
```

The output produced by the above code will print the entire content of file **poem.txt.**

```
>>>
   WHY?

We work, we try to be better
We work with full zest
But, why is that we just don't know any letter.
We still give our best.
We have to steal,
But, why is that we still don't get a meal.
We don't get luxury,
We don't get childhood,
But we still work,
Not for us, but for all the others.
Why is it that some kids wear shoes, BUT we make them ?
by      Mythili, class 5
```

The ***readline()*** function reads the leading and trailing spaces (if any) along with trailing newline character('\n') also while reading the line. You can remove these leading and trailing white spaces (spaces or tabs or newlines) using *strip( )* (without any argument) as explained below. Recall that *strip( )* without any argument removes leading and trailing whitespaces.

There is another way of printing a file line by line. This is a simpler way where after opening a file you can browse through the file using its file handle line by line by writing following code :

```
<filehandle> = open(<filename>, [<any read mode>])
for <var> in <filehandle> :
    print(<var>)
```

For instance, for the above given file *poem.txt*, if you write following code, it will print the entire file line by line :

```
myfile = open(r'E:\poem.txt', "r")
for line in myfile :
    print(line)
```

The output produced by above code is just the same as the output produced by above program. The reason behind this output is that when you iterate over a file-handle using a for loop, then the for loop's variable moves through the file line by line where a line of a file is considered as a sequence of characters up to and including a special character called the newline character (\n). So the *for loop's* variable starts with first line and with each iteration, it moves to the next line. As the *for loop* iterates through each line of the file the loop variable will contain the current line of the file as a string of characters.

Code Snippet 6 *Displaying the size of a file after removing EOL characters, leading and trailing white spaces and blank lines*

```
myfile = open(r'E:\poem.txt', "r")
str1 = " "              #initially storing a space (any non-None value)
size = 0
tsize = 0
while str1 :
    str1 = myfile.readline()
    tsize = tsize + len(str1)
    size = size + len(str1.strip())
print("Size of file after removing all EOL characters & blank lines:", size)
print("The TOTAL size of the file:", tsize)
myfile.close()
```

The output produced by the above code fragment is :

```
Size of file after removing all EOL characters & blank lines : 360
The TOTAL size of the file : 387
```

All the above code fragments read the contents from file in a string. However, if you use *readlines( )* (notice 's' in the end of the function), then the contents are read in a List. Go through next code fragment.

Code Snippet 7 *Reading the complete file-in a list*

```
myfile = open(r'E:\poem.txt', "r")
s = myfile.readlines()  ←———— notice it is readlines( ) not readline( )
print(s)
myfile.close()
```

Now carefully look at the output. The *readlines( )* has read the entire file in a list of strings where each line is stored as one string :

```
['  WHY?\n', '\n', 'We work, we try to be better\n', 'We work with
full zest\n', "But, why is that we just don't know any letter.\n",
'\n', 'We still give our best.\n', 'We have to steal,\n', "But, why
is that we still don't get a meal.\n", '\n', "We don't get luxury,
\n", "We don't get childhood,\n", 'But we still work,\n', 'Not for
us, but for all the others.\n', '\n', 'Why is it that some kids wear
shoes, BUT we make them ?\n', '\n', 'by      Mythili, class 5\n']
```

Now that you are familiar with these reading functions' working, let us write some programs.

P 5.1 Write a program to display the size of a file in bytes.

Program

```
myfile = open(r'E:\poem.txt', "r")
str = myfile.read()
size = len(str)
print("Size of the given file poem.txt is")
print(size, "bytes")
```

```
>>>
Size of the given file poem.txt is
387 bytes
```

P 5.2 Write a program to display the number of lines in the file.

Program

```
myfile = open(r'E:\poem.txt', "r")
s = myfile.readlines()
linecount = len(s)
print("Number of lines in poem.txt is", linecount)
myfile.close()
```

```
>>>
Number of lines in poem.txt is 18
```

5.4.2 Writing onto Files

After working with file-reading functions, let us talk about the writing functions for data files available in Python. (See Table 5.3). Like reading functions, the writing functions also work on open files, *i.e.*, the files that are opened and linked via a *file-object* or *file-handle*.

Table 5.3 *Python Data Files – Writing Functions*

S.No.	Name	Syntax	Description
1.	write( )	`<filehandle>.write(str1)`	writes string *str1* to file referenced by *<filehandle>*
2.	writelines( )	`<filehandle>.writelines(L)`	writes all strings in list L as lines to file referenced by *<filehandle>*

The `<filehandle>` in above syntaxes is the *file-object* holding open file's reference

Appending a File

When you open a file in "w" or *write mode*, Python overwrites an existing file or creates a non-existing file. That means, for an existing file with the same name, the earlier data gets lost. If, however, you want to write into the file while retaining the old data, then you should open the file in "a" or *append mode*. A file opened in *append mode* retains its previous data while allowing you to add newer data into. You can also add a plus symbol (+) with file read mode to facilitate reading as well as writing.

That means, in Python, writing in files can take place in following forms :

(*i*) In an existing file, while retaining its content

(*a*) if the file has been opened in append mode ("a") to retain the old content.

(*b*) if the file has been open in 'r+' or 'a+' modes to facilitate reading as well as writing.

(*ii*) to create a new file or to write on an existing file after truncating/ overwriting its old content

(*a*) if the file has been opened in write-only mode ("w")

(*b*) if the file has been open in 'w+' mode to facilitate writing as well as reading

(*iii*) Make sure to use close( ) function on *file-object* after you have finished writing as sometimes, the content remains in memory buffer and to force-write the content on file and closing the link of *file-handle* from file, *close( )* is used.

Let us consider some examples now.

Code Snippet 8 *Create a file to hold some data*

```
fileout = open("Student.dat", "w")
for i in range(5) :
    name = input("Enter name of student :")
    fileout.write(name)
fileout.close()
```

The write() will simply write the content in file without adding any extra character.

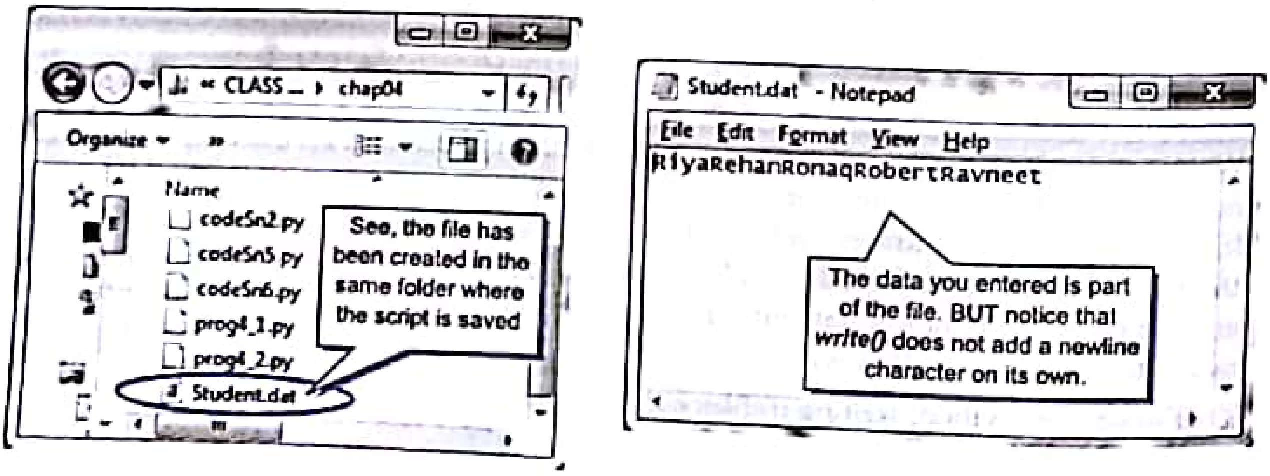
It's important to use close()

The sample run of above code is as shown below :

```
>>>
Enter name of student : Riya
Enter name of student : Rehan
Enter name of student : Ronaq
Enter name of student : Robert
Enter name of student : Ravneet
```

Now you can see the file created in the same folder where this Python script/program is saved (see Fig. 5.3(*a*) below). However, if you want to create the file in specific folder then you must specify the file-path as per the guidelines discussed earlier.

Also, you can open the created file ("student.dat" in above case) in *Notepad* to see its contents. Fig. 5.3(*b*) shows the contents of file created through *code snippet 5*.



See, the file has been created in the same folder where the script is saved

RiyaRehanRonaqRobertRavneet

The data you entered is part of the file. BUT notice that write() does not add a newline character on its own.

Figure 5.3 (*a*) File created through open( ) with "w" mode is stored in the same folder as that of script file (*b*) The write( ) function does not add any extra character in the file.

IMPORTANT

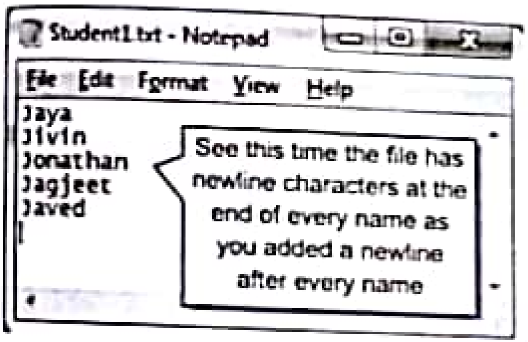
Carefully look at Fig. 5.3(*b*) that is showing contents of a file created through write( ). It is clear from the file-contents that *write( )* does not add any extra character like newline character('\n') after every write operation. Thus to group the contents you are writing in file, line wise, make sure to write newline characters on your own as depicted in ***code snippet 9.***

Code Snippet 9 *Create a file to hold some data, separated as lines*

(This code is creating a different file than created with code snippet 8)

```
fileout = open("Student1.txt", "w")
for i in range(5) :
    name = input("Enter name of student :")
    fileout.write(name)
    fileout.write('\n')
fileout.close()
```

The newline character '\n' written after every name



```
Jaya
Jivin
Jonathan
Jagjeet
Javed
```

See this time the file has newline characters at the end of every name as you added a newline after every name

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

```
>>>
Enter name of student : Jaya
Enter name of student : Jivin
Enter name of student : Jonathan
Enter name of student : Jagjeet
Enter name of student : Javed
```

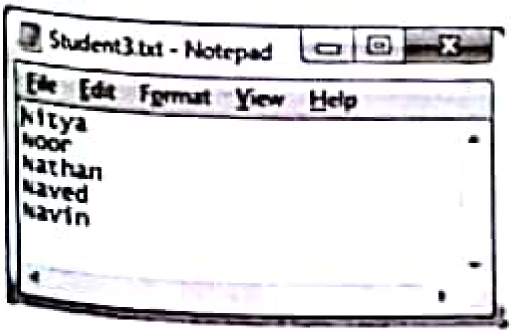
The file created by ***code snippet 9*** is shown above.

Code snippet 10 *Creating a file with some names separated by newline characters without using write( ) function.*

(For this, we shall use ***writelines( )*** *inplace of* ***write( )*** *function which writes the content of a list to a file. Function writelines() also, does not add newline character, so you have to take care of adding newlines to your file.)*

```
fileout = open("Student3.txt", "w")
List1 = []
for i in range(5) :
    name = input("Enter name of student :")
    List1.append(name + '\n')
fileout.writelines(List1)
fileout.close()
```

Responsibility to add newline character is of programmer's



```
Nitya
Noor
Nathan
Naved
Navin
```

Sample run of the above code is as shown below :

```
>>>
Enter name of student : Nitya
Enter name of student : Noor
Enter name of student : Nathan
Enter name of student : Naved
Enter name of student : Navin
```

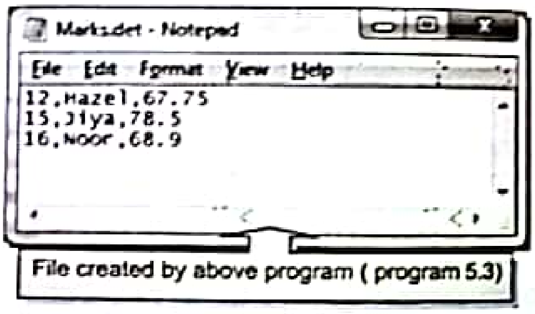
Now that you are familiar with these writing functions' working, let us write some programs based on the same.

Program 5.3 Write a program to get roll numbers, names and marks of the students of a class (get from user) and store these details in a file called "Marks.det".

```
count = int(input("How many students are there in the class?"))
fileout = open ("Marks.det", "w")

for i in range(count) :
    print("Enter details for student", (i+1), "below:")
    rollno = int(input("Rollno:))"
    name = input("Name :")
    marks = float(input("Marks:"))
    rec = str(rollno) + "," + name + "," + str(marks) +'\n'
    fileout.write(rec)
fileout.close()
```

Joining individual information by adding commas in between

```
>>>
How many students are there in the class? 3
Enter details for student 1  below :
Rollno : 12
Name : Hazel
Marks : 67.75
Enter details for student 2  below :
Rollno : 15
Name : Jiya
Marks : 78.5
Enter details for student 3  below :
Rollno : 16
Name : Noor
Marks : 68.9
```



```
12,Hazel,67.75
15,Jiya,78.5
16,Noor,68.9
```

File created by above program (program 5.3)

Figure 5.4 File created through program.

If you carefully notice, we have created comma separated values in one student record while writing in file. So we can say that the file created by program 5.4 is in CSV (comma separated values) format or it is a delimited file where comma is the delimiter.

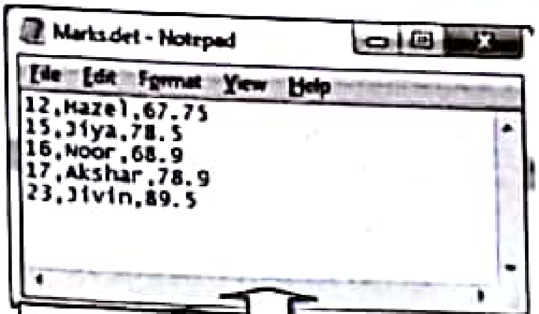
Program 5.4 Write a program to add two more students' details to the file created in program 5.3.

```
fileout = open ("Marks.det", "a")
for i in range(2) :
    print("Enter details for student", (i+1), "below :")
    rollno = int(input("Rollno:"))
    name = input("Name :")
    marks = float(input("Marks:"))
    rec = str(rollno) + "," + name + "," + str(marks) +'\n'
    fileout.write(rec)
fileout.close()
```

Notice the file is opened in append mode ("a") this time so as to retain old content

We want to add two records this time

```
>>>
Enter details for student 1 below :
Rollno : 17
Name : Akshar
Marks : 78.9
Enter details for student 2 below :
Rollno : 23
Name : Jivin
Marks : 89.5
```



```
12,Hazel,67.75
15,Jiya,78.5
16,Noor,68.9
17,Akshar,78.9
23,Jivin,89.5
```

Same file after adding two more records

Program 5.5 Write a program to display the contents of file "Marks .det" created through programs 5.3 and 5.4.

```
fileinp = open("Marks.det", "r")
while str :
    str = fileinp.readline()
    print(str)
fileinp.close()
```

```
>>>
12,Hazel,67.75
15,Jiya,78.5
16,Noor,68.9
17,Akshar,78.9
23,Jivin,89.5
```

5.4.3 The flush( ) Function

When you write onto a file using any of the write functions, Python holds everything to write in the file in buffer and pushes it onto actual file on storage device a later time. If however, you want to force Python to write the contents of buffer onto storage, you can use **flush( )** function.

Python automatically flushes the files when closing them *i.e.*, this function is implicitly called by the **close( )** function. But you may want to flush the data before closing any file. The syntax to use flush( ) function is :

```
<fileObject>.flush()
```

FLUSH( )

The flush( ) function forces the writing of data on disc still pending in *output buffer*.

Consider the following example code :

```
f = open('out.log', 'w+')
f.write('The output is \n')
f.write("My" + "work-status "+" is ")
f.flush()
s = 'OK.'
f.write(s)
f.write('\n')
# some other work
f.write('Finally Over\n')
f.flush()
f.close()
```

With this statement, the strings written so far, i.e., 'The output is' and 'My work-status is' have been pushed on to actual file on disk.

These write statements' strings may still be pending to be written on to disk

The flush( ) function ensures that whatever is held in Output buffer, is written on to the actual file on disk

5.4.4 Removing Whitespaces after Reading from File

The ***read( )*** and ***readline( )*** functions discussed above, read data from file and return it in string form and the ***readlines( )*** function returns the entire file content in a list where each line is one item of the list.

All these read functions also read the leading and trailing whitespaces *i.e.*, spaces or tabs or newline characters. If you want to remove any of these trailing and leading whitespaces, you can use strip( ) functions [rstrip( ), lstrip( ) and strip( )] as shown below.

Recall that :

- the *strip*( ) removes the given character from both ends.
- the *rstrip*( ) removes the given character from trailing end *i.e., right end.*
- the *lstrip*( ) removes the given character from leading end *i.e., left end.*

To understand this, consider the following examples :

1. **Removing EOL '\n' character from the line read from the file.**

```
fh = file("poem.txt, "r")
line = fh.readline()
line = line.rstrip('\n')
```

Will remove the end of line newline character '\n'

2. **Removing the leading whilespaces from the line read from the file**

```
line = file("poem.txt, "r").readline()
line = line.lstrip()
```

Now can you justify the output of following code that works with first line of file *poem.text* shown above where first line containing leading 8 spaces followed by word 'WHY?' and a '\n' in the end of line.

```
>>> fh = file("e :\\poem.txt", "r")
>>> line = fh.readline()
>>> len(line)
14

>>> line2 = line.rstrip('\n')
>>> len(line2)
13

>>> line3 = line.strip()
>>> len(line3)
4
```

Steps to Process a File

Following lines list the steps that need to be followed in the order as mentioned below. The five steps to use files in your Python program are :

1. Determine the type of file usage

Under this step, you need to determine whether you need to open the file for reading purpose (input type of usage) or writing purpose (output type of usage). If the data is to be brought in from a file to memory, then the file must be opened in a mode that supports reading such as "r" or "r+" etc.

Similarly, if the data (after some processing) is to be sent from memory to file, the file must be opened in a mode that supports writing such as "w" or "w+" or "a" etc.

2. Open the file and assign its reference to a *file-object* or file-handle

Next, you need to open the file using **open( )** and assign it to a file-handle on which all the file-operations will be performed. Just remember to open the file in the file-mode that you decided in step 1.

3. Now process as required

As per the situation, you need to write instructions to process the file as desired. For example, you might need to open the file and then read it one line at a time while making some computation, and so on.

4. Close the file

This is very important step especially if you have opened the file in write mode. This is because, sometimes the last lap of data remains in buffer and is not pushed on to disk until a close( ) operation is performed.

5.4.5 Significance of File Pointer in File Handling

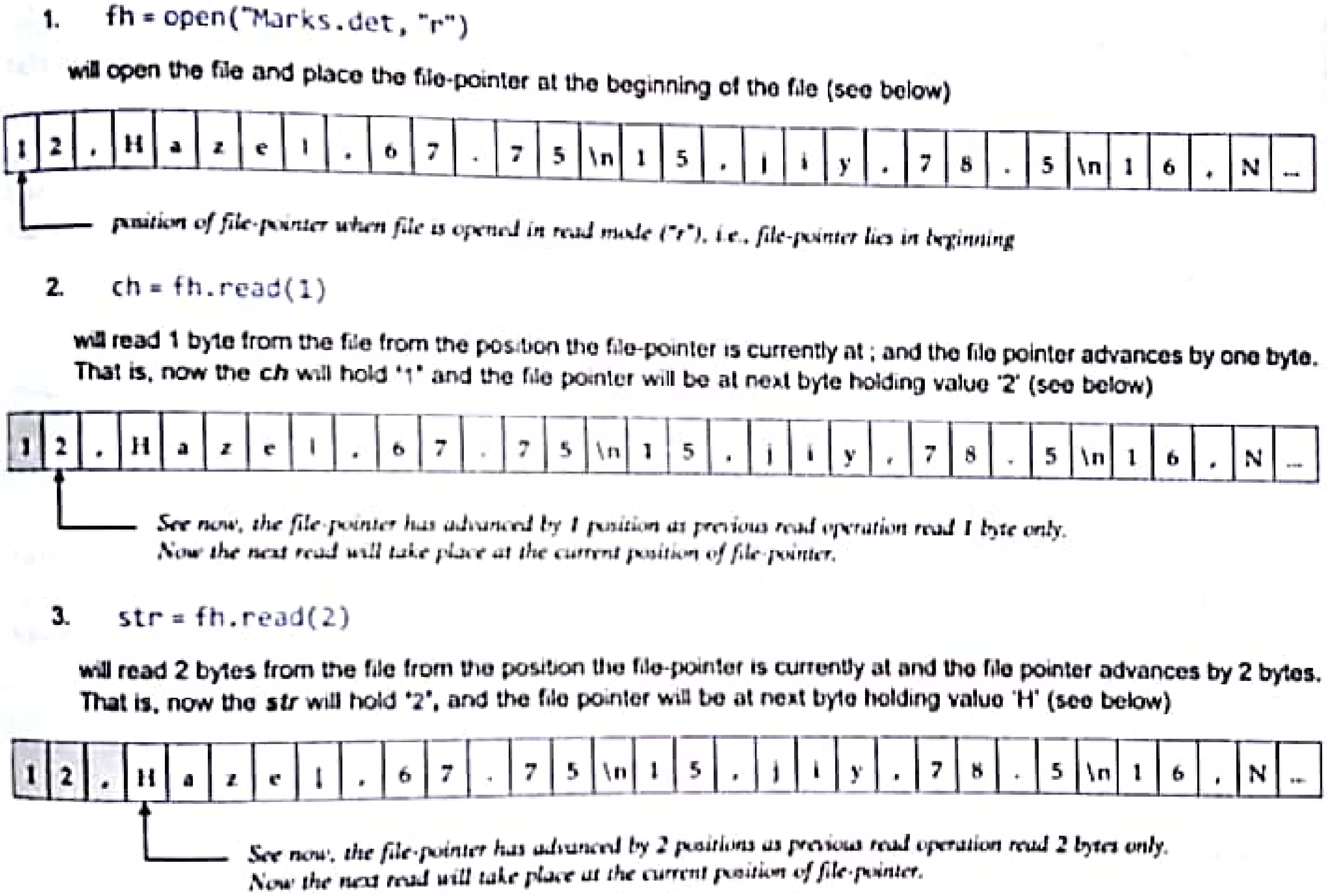
Every file maintains a *file pointer* which tells the current position in the file where writing or reading will take place. (A file pointer in this context works like a book-mark in a book).

Whenever you read something from the file or write onto a file, then these *two* things happen involving file-pointer :

(*i*) this operation takes place at the position of *file-pointer* and

(*ii*) *file-pointer* advances by the specified number of bytes

Figure 5.5 illustrates this process. It is important to understand how a *file pointer* works in Python files. The working of file-pointer has been described in Fig. 5.5.



1.

```
fh = open("Marks.det, "r")
```

will open the file and place the file-pointer at the beginning of the file (see below)

1 2 . H a z e l . 6 7 . 7 5 \n 1 5 . J i y . 7 8 . 5 \n 1 6 . N ...
position of file-pointer when file is opened in read mode ("r"), i.e., file-pointer lies in beginning

2.

```
ch = fh.read(1)
```

will read 1 byte from the file from the position the file-pointer is currently at ; and the file pointer advances by one byte. That is, now the *ch* will hold '1' and the file pointer will be at next byte holding value '2' (see below)

1 2 . H a z e l . 6 7 . 7 5 \n 1 5 . J i y . 7 8 . 5 \n 1 6 . N ...
See now, the file-pointer has advanced by 1 position as previous read operation read 1 byte only.
Now the next read will take place at the current position of file-pointer.

3.

```
str = fh.read(2)
```

will read 2 bytes from the file from the position the file-pointer is currently at and the file pointer advances by 2 bytes. That is, now the *str* will hold '2', and the file pointer will be at next byte holding value 'H' (see below)

1 2 . H a z e l . 6 7 . 7 5 \n 1 5 . J i y . 7 8 . 5 \n 1 6 . N ...
See now, the file-pointer has advanced by 2 positions as previous read operation read 2 bytes only.
Now the next read will take place at the current position of file-pointer.

Figure 5.5 Working of a file-pointer.

5.4.5A File Modes and Ope.ing Position of File-Pointer

The position of a file-pointer is governed by the *filemode* it is opened in. Following table lists the opening position of a file-pointer as per *filemode*.

Table 5.4 *File modes and opening position of file-pointer*

File Modes	Opening position of file - pointer
r, rb, r+, rb+, r+b	*beginning of the file*
w, wb, w+, wb+, w+b	*beginning of the file* (Overwrites the file if the file exists).
a, ab, a+, ab+, a+b	*at the end of the file* if the file exists otherwise creates a new file.

5.5 STANDARD INPUT, OUTPUT AND ERROR STREAMS

If someone asks you to give input to a program interactively or by typing, you know what device you need for it – *the keyboard*. Similarly, if someone says that the output is to be displayed, you know which device it will be displayed on – *the monitor*. So, we can safely say that the *Keyboard* is the **standard input device** and the *monitor* is **standard output device**. Similarly, any error if occurs is also displayed on the monitor. So, *the monitor* is also **standard error device**. That is,

- standard input device **(stdin)** – reads from the keyboard
- standard output device **(stdout)** – prints to the display and can be redirected as standard input.
- standard error device **(stderr)** – Same as *stdout* but normally only for errors. Having error output separately allows the user to divert regular output to a file and still be able to read error messages.

Do you know internally how these devices are implemented in Python? These standard devices are implemented as files called *standard streams*. In Python, you can use these standard stream files by using **sys** module. After importing, you can use these standard streams **(stdin, stdout and stderr)** in the same way as you use other files.

Interesting : Standard Input, Output Devices as Files

If you import sys module in your program then, **sys.stdin.read( )** would let you read from keyboard. This is because the keyboard is the *standard input device* linked to **sys.stdin**. Similarly, **sys.stdout.write( )** would let you write on the standard output device, *the monitor*. **sys.stdin** and **sys.stdout** are standard input and standard output devices respectively, treated as files.

Thus **sys.stdin** and **sys.stdout** are like files which are opened by the Python when you start Python. The **sys.stdin** is always opened in read mode and **sys.stdout** is always opened in write mode. Following code fragment shows you interesting use of these. It prints the contents of a file on monitor without using **print** statement :

```
import sys
fh = open(r"E:\poem.txt")
line1 = fh.readline()
line2 = fh.readline()
sys.stdout.write(line1)
sys.stdout.write(line2)
sys.stderr.write("No errors occurred\n")
```

These statements would write on file/device associated with sys.stdout, which is the monitor

Output produced is :

```
>>> ========================
>>>
         WHY ?
We work, we try to be better
No errors occurred
>>>
```

See, stderr also displayed its text on monitor.

Python Data Files : with Statement

Python's with statement for files is very handy when you have two related operations which you'd like to execute as a pair, with a block of code in between. The syntax for using with statement is :

```
with open(<filename>, <filemode>) as <filehandle> :
    <file manipulation statements>
```

The classic example is opening a file, manipulating the file, then closing it :

```
with open('output.txt', 'w') as f:
    f.write('Hi there!')
```

The above with statement will automatically close the file after the nested block of code. The advantage of using a ***with statement*** is that it is guaranteed to close the file no matter how the nested block exits. Even if an exception (a runtime error) occurs before the end of the block, it will close the file.

Absolute and Relative Paths

Full name of a file or directory or folder consists of path\primaryname.extension.

Path is a sequence of directory names which give you the hierarchy to access a particular directory or file name. Let us consider the following directory structure :

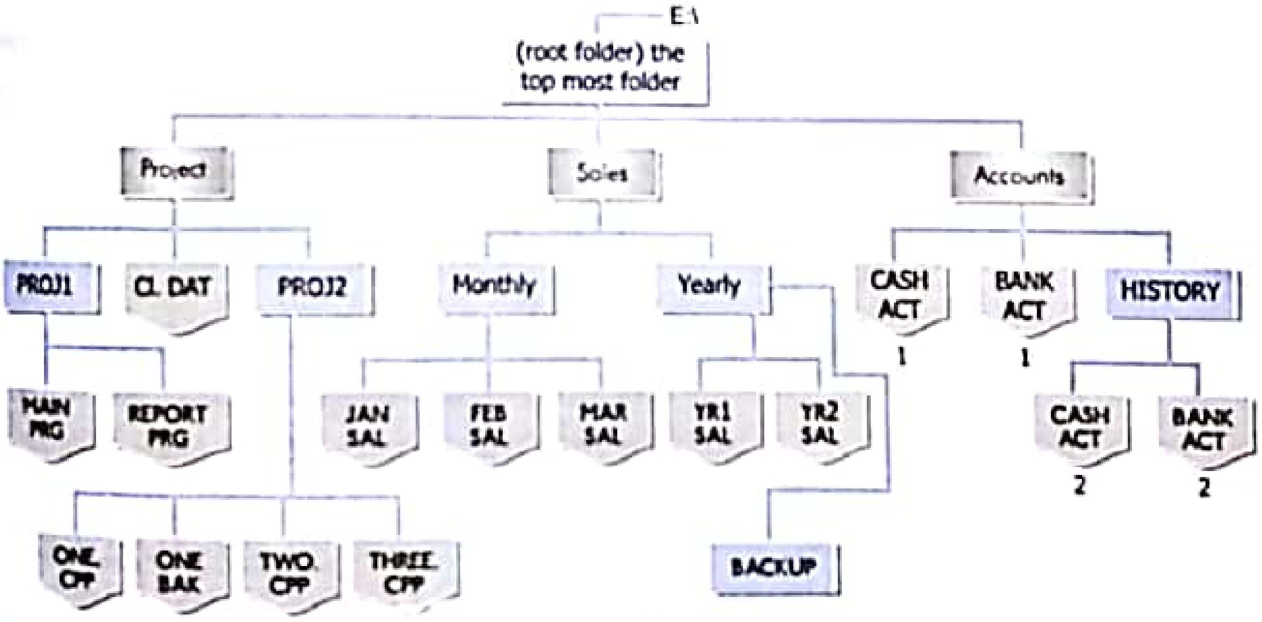


Figure 5.7 A Sample Directory Structure.

Now the *full name* of directories PROJECT, SALES and ACCOUNTS will be

```
E:\PROJECT, E:\SALES and E:\ACCOUNTS respectively.
```

So format of path can be given as

```
Drive-letter:\directory [\directory...]
```

where first \ (backslash) refers to root directory and other ('\'s) separate a directory name from the previous one.

Now the directory BACKUP'S path will be :

```
E:\SALES\YEARLY\BACKUP
```

As to reach BACKUP the sequence one has to follow is : under drive E, under root directory (first \), under SALES subdirectory of root, under YEARLY subdirectory of SALES, there lies BACKUP directory.

Similarly full name of ONE.VBP file under PROJ2 subdirectory will be :

```
E:\ PROJECT\PROJ2\ONE.VBP
  ↓
refers to
root directory
```

PATHNAME

The full name of a file or a directory is also called *pathname*.

Now see there are *two* files with the same name CASH.ACT, one under ACCOUNTS directory and another under HISTORY directory but according to Windows rule no two files can have same names (*path names*). Both these files have same names but their path names differ, therefore, these can exist on system. Therefore, CASH.ACT$_1$'s path will be

```
E:\ACCOUNTS\CASH.ACT
```

and CASH.ACT$_2$'s path will be

```
E:\ACCOUNTS\HISTORY\CASH.ACT
```

Above mentioned *path names* are *Absolute Pathnames* as they mention the *paths* from the top most level of the directory structure.

NOTE

The absolute paths are from the topmost level of the directory structure. The relative paths are relative to current working directory denoted as a dot(.) while its parent directory is denoted with two dots(..).

Relative pathnames mention the paths relative to current working directory. Let us assume that current working directory now is, say, PROJ2. The symbols . (*one dot*) and .. (*two dots*) can be used now in relative paths and pathnames. The symbol . can be used in place of current directory and .. denotes the parent directory.

So, with PROJ2 as current working folder, pathname of TWO.DOC will be :

```
.\TWO.DOC  ← TWO.DOC in current folder        (PROJ2 being working Folder)
```

which means under current working folder, there is file TWO.PAS. Similarly, path name for file CL.DAT will be

```
..\CL.DAT  ← CL.DAT in parent folder          (PROJ2 being working Folder)
```

which means under parent folder of current folder, there lies a file CL.DAT. Similarly, path name for REPORT.PRG will be

```
..\PROJ1\REPORT.PRG                           (PROJ2 being working Folder)
```

that is under parent folder's subfolder PROJ1, there lies a file REPORT.PRG.

Check Point 5.1

1. In which of the following file modes, the existing data of file will not be lost ?
   (*a*) 'rb' (*b*) ab (*c*) w
   (*d*) w + b (*e*) 'a + b' (*f*) wb
   (*g*) wb+ (*h*) w+ (*i*) r+
2. What would be the data type of variable data in following statements ?
   (*a*) data = f.read( )
   (*b*) data = f.read(10)
   (*c*) data = f.readline( )
   (*d*) data = f.readlines( )
3. How are following statements different ?
   (*a*) f.readline( )
   (*b*) f.readline( ).rstrip( )
   (*c*) f.readline( ).strip( )
   (*d*) f.readline.rstrip('\n')

PriP DATA FILES IN PYTHON

Progress In Python 5.1

This PriP session aims at giving you practical exposure to file handling in Python.

⋮

Fill it in PriP 5.1 under Chapter 5 of practical component-book – Progress in Computer Science with Pythonafter practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *A file in itself is a bunch of bytes stored on some storage devices like hard-disk, thumb-drive etc.*
- *The data files can be stored in two ways : (i) Text files (ii) Binary files.*
- *A text file stores information in ASCII or Unicode characters, where each line of text is terminated, (delimited) with a special character known as EOL (End of Line) character. In text files some internal translations take place when this EOL character is read or written.*
- *A binary file is just a file that contains information in the same format in which the information is held in memory, i.e., the file content that is returned to you is raw (with no translation or no specific encoding).*
- *The open( ) function is used to open a data file in a program through a file-object (or a file-handle).*
- *A file-mode governs the type of operations (e.g., read / write / append) possible in the opened file i.e., it refers to how the file will be used once it's opened.*
- *A text file can be opened in these file modes : 'r' , 'w', 'a', 'r+', 'w+', 'a+'*
- *A binary file can be opened in these file modes : 'rb' , 'wb', 'ab', 'r+b' ('rb+'), 'w+b'('wb+'), 'a+b'('ab+').*
- *The three file reading functions of Python are : read( ), readline( ), readlines( )*
- *While read( ) reads some bytes from the file and returns it as a string, readline( ) reads a line at a time and readlines( ) reads all the lines from the file and returns it in the form of a list.*
- *The two writing functions for Python data files are write( ) and writelines( ).*
- *While write( ) writes a string in file, writelines( ) writes a list in a file.*
- *The input and output devices are implemented as files, also called standard streams.*
- *There are three standard streams : stdin (standard input), stdout (standard output) and stderr (standard error)*
- *The absolute paths are from the topmost level of the directory structure. The relative paths are relative to current working directory denoted as a dot(.) while its parent directory is denoted with two dots(..).*

Solved Problems

1. *What is the difference between read( ) and readlines( ) function?*

Solution. The read( ) reads from a file in read mode, and stores its contents in a string type variable. The readlines( ) function, reads from a file in read mode and returns a list of all lines in the file.

2. *Differentiate between the following :* [CBSE D 2015]

(i) `f = open('diary.txt', 'r')` (ii) `f = open('diary.txt', 'w')`

Solution. (i) File has been opened in *read mode* with file handle *f*.

(ii) File has been opened in *write mode* with file handle *f*.

3. *What is the difference between readline( ) and readlines( ) function?*

   Solution. The ***readline( )*** function reads from a file in read mode and returns the next line in the file or a blank string if there are no more lines. (The returned data is of string type)

   The ***readlines( )*** function, also reads from a file in read mode and returns a list of all lines in the file. (The returned data is of list type)

4. *Write a single loop to display all the contents of a text file c :\poem.txt after removing leading and trailing whitespaces.*

   Solution.
```
for line in file("poem.txt") :
    print(line.strip())
```

5. *Write a function stats( ) that accepts a filename and reports the file's longest line.*

   Solution.
```
def stats(filename) :
    longest = ""
    for line in file(filename) :
        if len(line) > len(longest) :
            longest = line
    print("Longest line's length =", len(longest))
    print(longest)
```

6. *What is the output of following code fragment ? Explain.*
```
out = file("output.txt", "w")
out.write("Hello, world!\n")
out.write("How are you?")
out.close()
file("output.txt").read()
```

   Solution. The output will be :
```
'Hello, world!\nHow are you?'
```

   The first line of the code is opening the file in write mode ; the next two lines write text to the file. The last line opens the file and from that reference reads the file-content. Function file( ) does the same as that of open( ). Thus file("output.txt") will give the reference to open file, on which read( ) is applied.

7. *Write a function remove_lowercase( ) that accepts two filenames, and copies all lines that do not start with a lowercase letter from the first file into the second.*

   Solution.
```
def remove_lowercase(infile, outfile) :
    output = file(outfile, "w")
    for line in file(infile) :
        if not line[0] in "abcdefghijklmnopqrstuvwxyz" :
            output.write(line)
    output.close()
```

8. *What is the output of following code ?*
```
file("e:\\poem.txt", "r").readline().split()
```

   Recall that poem.txt has some leading and trailing whitespaces.

   Solution. [WHY?]

9. *What is the output of following code ?*

```
file("e:\\poem.txt", "r").readline()
```

Solution. '   WHY?\n'

10. *What is the output of following code ?*

```
fh = file("poem.txt", "r")
size = len(fh.read())
print(fh.read(5))
```

Solution. No output

Explanation. The **fh.read( )** of line 2 will read the entire file content and place the file pointer at end of file. For the **fh.read(5)**, it will return nothing as there are no bytes to be read from EOF thus **print( )** statement prints nothing.

11. *Write a program to display all the records in a file along with line/record number.*

Solution.

```
fh = open("Result.det", "r")
count = 0
rec = " "
while True :
    rec = fh.readline()
    if rec == " " :
        break
    count = count + 1
print(count, rec, end = '')          # to suppress extra newline by print
fh.close()
```

12. *What is the output produced by following code ?*

```
obj = open("New.txt","w")
obj.write("A poem by Paramhansa Yogananda) "
obj.write("Better than Heaven or Arcadia")
obj.write("I love thee, O my India!")
obj.write("And thy love I shall give")
obj.write("To every brother nation that lives.")
obj.close()
obj1=open("New.txt","r")
s1 = obj1.read(48)
print(s1)
obj1.close()
```

Solution. The output produced by above code will be :

```
A poem by Paramhansa Yogananda Better than Heaven
```

13. *The file "New.txt" contains the following :*

```
Better than Heaven or Arcadia
I love thee, O my India!
And thy love I shall give
To every brother nation that lives.
```

Considering the given file, what output will be produced by the following code ?

```
obj1=open("New.txt","r")
s1 = obj1.readline()
s2 = obj1.readline()
s3 = obj1.readline()
s4 = obj1.read(15)
print(s4)
obj1.close()
```

Solution. The output produced by above code is : `And thy love I`

14. *Two identical files (p1.txt and p2.txt) were created by following two codes (carefully go through the two codes given below)*

(*a*)
```
obj = open("p1.txt","w")
obj.write("Better than Heaven or Arcadia")
obj.write("I love thee, O my India!")
obj.write("And thy love I shall give")
obj.write("To every brother nation that lives.")
obj.close( )
```

(*b*)
```
obj = open("p2.txt","w")
obj.write("Better than Heaven or Arcadia\n")
obj.write("I love thee, O my India!\n")
obj.write("And thy love I shall give\n")
obj.write("To every brother nation that lives.\n")
obj.close( )
```

What would be the output produced if the files are read and printed with following code.

Solution. The output produced by code (a) will be :

```
Better than Heaven or ArcadiaI love thee, O my India!And thy love
I shall giveTo every brother nation that lives.
```

The output produced by code (b) will be :

```
A poem by Paramhansa Yogananda
Better than Heaven or Arcadia
I love thee, O my India!
And thy love I shall give
To every brother nation that lives.
```

15. *Considering the two files p1.txt and p2.txt created in previous question, what output will be produced by following code fragments ?*

(*a*)
```
obj1 = open("p1.txt","r")
s1 = obj1.readline()
s2 = obj1.read(15)
print(s2)
obj1.close()
```

(*b*)
```
obj1 = open("p2.txt","r")
s1 = obj1.readline()
s2 = obj1.read(15)
print(s2)
obj1.close()
```

Solution. No output or blank output will be produced by code (*a*).

For code(*b*), the output produced will be : `Better than Hea`

16. *Consider the file p2.txt created above. Now predict the output of following code that works with p2.txt. Explain the reason behind this output.*

```
fp1 = open("p2.txt", "r")
print(fp1.readline(20))
s1 = fp1.readline(30)
print(s1)
print(fp1.readline(25))
```

Solution. The output produced by above code will be :

```
A poem by Paramhansa
    Yogananda
Better than Heaven or Arc
```

The reason behind this output is that the first file-read line (*i.e.*, **fp1.readline(20)**) read 20 bytes from the file pointer. As just after opening the file, the file-pointer is at the beginning of the file, the 20 bytes are read from the beginning of the file which returned string as "A poem by Paramhansa\n" – this is because readline( ) returns the read string by adding an end-line character to it (\n).

So the first line of output was printed as : `A poem by Paramhansa`

After the first readline( ), the file pointer was at the space following word 'Paramhansa', so the next readline( ) started reading from there and read 15 character or end-of the-line, whichever is earlier. So the read string was "Yogananda\n" – notice the space before word Yogananda. Hence came the second line of the output.

Now the file-pointer was at the beginning of the third line and the next readline (*i.e.*, **fp1.readline(25)**) read 25 characters from this line and gave the last line of output.

17. *A text file contains alphanumeric text (say an.txt). Write a program that reads this text file and prints only the numbers or digits from the file.*

Solution.

```
F = open("an.txt", "r")
for line in F:
    words = line.split()
    for i in words:
        for letter in i:
            if(letter.isdigit()):
                print(letter)
```

18. *Read the code given below and answer the question :*

```
fh = open("main.txt", "w")
fh.write("Bye")
fh.close()
```

If the file contains "GOOD" before execution, what will be the contents of the file after execution of this code ?

Solution. The file would now contain "Bye" only, because when an existing file is opened in write mode ("w"), it truncates the existing data the file.

19. *A given text file "data.txt" contains :*

```
Line 1\n
\n
Line 3
Line 4
\n
Line 6
```

What would be the output of following code?

```
fh = open ("data.txt", "r")
lst = fh.readlines()
print(lst[0], end = '')
print(lst[2], end = '')
print(lst[5], end = '')
print(lst[1], end = '')
print(lst[4], end = '')
print(lst[3])
```

Solution.

```
Line 1
  Line 3
  Line 6  Line 3

  Line 4
```

20. *Write code to print just the last line of a text file "data.txt".*

Solution.

```
fin = open("data.txt", "r")
lineList = fin.readlines()
print("Last line =", lineList[-1])
```

21. *Write a program that copies a text file "source.txt" onto "target.txt" barring the lines starting with a "@" sign.*

Solution.

```
def filter(oldfile, newfile) :
    fin = open(oldfile, "r")
    fout = open(newfile, "w")
    while True :
        text = fin.readline()
        if len(text) == 0:
            break
        if text[0] == "@":
            continue
        fout.write(text)
    fin.close()
    fout.close()
filter("source.txt", "target.txt")
```

GLOSSARY

File	A bunch of bytes stored on some storage device.
File mode	A constant describing how a file is to be used.
Stream	A sequence of bytes.

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is the difference between "w" and "a" modes ?
2. What is the significance of file-object ?
3. How is file open( ) function different from close( ) function ?
4. Write statements to open a binary file C:\Myfiles\Text1.txt in read and write mode by specifying the file path in two different formats.
5. When a file is opened for output, what happens when

   (*i*) the mentioned file does not exist (*ii*) the mentioned file does exist ?
6. What role is played by file modes in file operations ? Describe the various file mode constants and their meanings.
7. What are the advantages of saving data in : (*i*) binary form (*ii*) text form ?
8. When do you think text files should be preferred over binary files ?
9. Write a statement in Python to perform the following operations : [CBSE D 16]

   (*a*) To open a text file "BOOK.TXT" in read mode (*b*) To open a text file "BOOK.TXT" in write mode

Type B : Application Based Questions

1. How are following codes different from one another ?

(a)
```
my_file = open('poem.txt','r')
my_file.read()
```

(b)
```
my_file = open('poem.txt','r')
my_file.read(100)
```

2. If the file 'poemBTH.txt' contains the following poem (by Paramhans Yoganand) :

God made the Earth;
Man made confining countries
And their fancy-frozen boundaries.
But with unfound boundless Love
I behold the borderland of my India
Expanding into the World.
Hail, mother of religions, Lotus, scenic beauty,and sages!

Then what outputs will be produced by both the code fragments given in question1.

3. Consider the file poemBTH.txt given above (in previous question). What output will be produced by following code fragment ?

```
obj1 = open("poemBTH.txt","r")
s1 = obj1.readline()
s2.readline(10)
s3 = obj1.read(15)
print(s3 )
print(obj1.readline())
obj1.close()
```

4. Consider the file "poemBTH.txt" and predict the outputs of following code fragments if the file has been opened in filepointer file1 with code:

```
file1 = open("E:\\mydata\\poemBTH.txt","r+")
```

(a)
```
print("A. Output 1")
print(file1.read())
print()
```

(b)
```
print("B. Output 2")
print(file1.readline() )
print()
```

(c)
```
print("C. Output 3")
print(file1.read(9))
print()
```

(d)
```
print("D. Output 4")
print(file1.readline(9) )
```

(e)
```
print("E. Output of Readlines function is")
print(file1.readlines() )
print()
```

NOTE. Consider the code fragments in succession, *i.e.*, code (*b*) follows code (*a*), which means changes by code (*a*) remain intact. Similarly, code (*c*) follows (*a*) and (*b*), and so on.

5. What is following code doing ?

```
file = open("contacts.csv", "a")
name = input("Please enter name.")
phno = input("Please enter phone number.")
file.write(name + "," + phno + "\n")
```

6. Write code to open file created in previous question and print it in following form :

```
Name : <name>        Phone :<phone number>
```

7. Consider the file "contacts.csv" created in above question and figure out what the following code is trying to do?

```
name = input("Enter name :")
file = open("contacts.csv", "r")
for line in file:
  if name in line:
      print(line)
```

8. Consider the file poemBTH.txt and predict the output of following code fragment. What exactly is following code fragment doing ?

```
f  = open("poemBTH.txt", "r")
nl = 0
for line in f:
  nl += 1
print(nl)
```

9. If you use the code of Q.8 with p1.txt created in solved problem 14, what would be its output ?

10. Write a method in python to read the content from a text file diary.txt line by line and display the same on screen. [CBSE D 2019]

11. Write a method in python to write multiple line of text contents into a text file mylife.txtline. [CBSE D 2019]

Type C : Programming Practice/Knowledge based Questions

1. Write a program that reads a text file and creates another file that is identical except that every sequence of consecutive blank spaces is replaced by a single space.
2. A file sports.dat contains information in following format : Event ~ Participant

   Write a function that would read contents from file *sports.dat* and creates a file named *Atheletic.dat* copying only those records from *sports.dat* where the event name is "*Atheletics*".
3. A file contains a list of telephone numbers in the following form :

   ```
   Arvind 7258031
   Sachin 7259197
   ```

   The names contain only one word the names and telephone numbers are separated by white spaces Write program to read a file and display its contents in two columns.
4. Write a program to count the words "to" and "the" present in a text file "Poem.txt".
5. Write a program to count the number of upper- case alphabets present in a text file "Article.txt".
6. Write a program that copies one file to another. Have the program read the file names from user ?
7. Write a program that appends the contents of one file to another. Have the program take the filenames from the user.
8. Write a program that reads characters from the keyboard one by one. All lower case characters get stored inside the file LOWER, all upper case characters get stored inside the file UPPER and all other characters get stored inside file OTHERS.
9. Write a function in Python to count and display the number of lines starting with alphabet 'A' present in a text file " LINES.TXT". *e.g.*, the file "LINES.TXT" contains the following lines :

   ```
   A boy is playing there.
   There is a playground.
   An aeroplane is in the sky.
   Alphabets & numbers are allowed in password.
   ```

   the function should display the output as 3.

6

Recursion

In This Chapter

6.1 INTRODUCTION

You have learnt how to create and invoke methods/functions in Python. Inside the function bodies, we can include function-calls of other functions. But have you ever thought : Could a method invoke or call itself ?

Self-invocation may at first sound useless or illegal : Isn't this defining something in terms of itself — what is called a circular definition? But self-invocation is legal, and it's actually quite useful. In fact, it's so useful that it gets its own special name : **recursion.**

This chapter explores recursion and recursive functions along with some classic recursive examples.

RECURSION

Recursion refers to a programming technique in which a function calls itself either directly or indirectly.

6.2 RECURSIVE FUNCTION

In many of your programs, you must have written some functions that **call or invoke** other functions, *e.g.*,

```
def A( ) {
    B ( )
```

Function A( ) is calling another function, B( )

Now, if you write a function definition that **calls itself**, then this function would be known as **recursive function**.

That is, a function is said to be recursive if the function definition includes a call to itself *e.g.*,

```
def A( ) :
    A( )
```

*Now function A( ) is called itself from its own function-body, so it is a **recursive function** now.*

See, another recursive function :

```
def recur( ) :
    recur( )           # calling itself
```

Recursion can be indirect also. The above example, where a function calls itself directly from within its body is an example of *direct recursion*. However, if a function calls another function which calls its caller function from within its body, it is known as *indirect recursion e.g.*,

```
def A( ) :
    B( )
def B( ) :
    A( )
```

Function A( ) calling B( ), which calls A( )

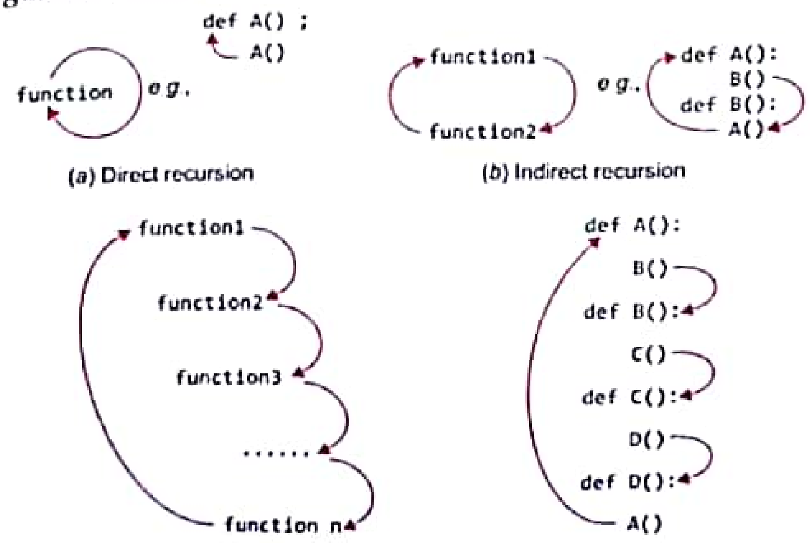
RECURSIVE FUNCTION

A function is said to be **Recursive** Function if it calls itself.

So, you can say that recursion can be :

- **direct recursion**, if a function calls itself directly from its function body.
- **indirect recursion**, if a function calls another function, which calls its caller function.

Following figure illustrates it.



(*a*) Direct recursion

(*b*) Indirect recursion

(c) Another example of indirect recursion

Figure 6.1 Types of Recursion.

The above shown sample code is an example of *indirect recursion.*

In mathematics also, you have seen recursive problems *e.g.*, sum of *n* natural numbers can be written as

sum of *n* natural numbers = *n* + sum of *n* – 1 natural numbers

Similarly, sum of *n* – 1 natural numbers = *n* – 1 + sum of *n* – 2 natural numbers

Similarly, sum of *n* – 2 natural numbers = *n* – 2 + sum of *n* – 3 natural numbers,

and so on.

This section is going to discuss recursion and recursive functions. We shall be learning how to write recursive functions and implement recursion. But before we proceed, carefully go through the following code and figure out its output.

```
def func1( )
     print("Hello func2")
     func2( )

def func2( )
     print("Yes func1")
     func1( )
```

Yes, you guessed it right. This code will print the following ENDLESSLY :

```
Hello func2
Yes func1
Hello func2
Yes func1
```

The above functions (*func1( )* and *func2( )*) will keep on calling one another infinitely and the entire memory will be used up – No stop!

Python will report following runtime error for it :

```
RuntimeError: maximum recursion depth[1] exceeded . . .
```

The problem with above code is that it is a Recursive program but recursion code is NOT SENSIBLE. You must write a sensible recursive code that has clearly defined 'where to stop'. Let us see how.

RECURSION

Recursion is a technique for solving a large computational problem by repeatedly applying the same procedure(s) to reduce it to successively smaller problems. A recursive procedure has two parts: one or more base cases and a recursive step.

Right or Sensible Recursive code is the one that fulfills following requirements :

- it must have a case, whose result is known or computed without any recursive calling – the **BASE CASE. The BASE CASE must be reachable for some argument/parameter.**
- it also have **Recursive Case**, where by function calls itself later section, (Section 6.3, explains this clearly).

[1] Recursion limit is platform dependent and can be set via *sys.setrecursionlimit(limit)*.

To understand the above points, let us consider an example program first and see how it works.

Program 6.1 Write a recursive function that computes the sum of numbers 1..n ; get the value of last number *n* from user.

```
1.  # program to illustrate recursion
2.
3.  def compute (num):
4.      if (num == 1) :
5.          return 1
6.      else:
7.          return (num + compute (num - 1) )
8.
9.  # __main__
10. last = 4
11. ssum = compute (last)
12. print ("The sum of the series from 1 ..", last, "is", ssum)
```

For **num's** *value 1, the result is pre-known and is available (or calculated) without any recursive call*

compute( ) *calling itself – recursive code*

Initial call to function ***compute( )***

The output produced by above code is :

```
The sum of the series from 1 .. 4 is 10
```

Let us now understand how it (program 6.1) works. Lines 3-7 of program 6.1 define a function **compute( )**, which is a recursive function as it calls itself on *line 7*.

1. Program starts __main__ part at *line 9* and then calls **compute( )** with variable **last** that has value 4, *i.e.*, **compute(4)** is invoked at line 11 initially.

2. With function call compute(4) at *line 11*, control shifts to *line 3*, where the code of **compute( )** begins ; parameter **num** takes value 4.

   2.1 At *line 4*, condition **num == 1** is *false*, so control shifts to else part (*line 6-7*)

   2.2 *Line 7* calculates *return value* as **num + compute(num-1)**, *i.e.*, as 4 = computer (4-1), *i.e.*, **4+compute(3)**.

   Since the value of **compute(3)** is not known, function compute( ) is to be called with value 3.

      2.2.1 So compute( ) is again called with value 3 so **num** takes value **3**. In line 4, *num* (which is 3) is still not 1 (condition 3 == 1 is *false*,) so line 7 (its return value) gets computed as **3 + compute (3-1)** *i.e.*, **3 + compute(2)**.

         2.2.1.1A For **compute(2)**, line 4 condition **num == 1** is false (as num is 2) so line 7 (its return value) is executed as **2 + compute(1)**

         2.2.1.1B For **compute(1)** condition of line 4 (num == 1) is true as **num is 1** now. So **compute (1)** returns value 1 as line 5 gets executed and control returns to **2+compute(1)** (2.2.1.1A) with value 1 for **compute (1)**.

2.2.1.1-R So after returning to 2.2.1.1.A, the compute (2)'s return value is calculated as 2 + **compute(1)** = 2 + 1 = 3 (value of **compute(1)** is 1). It calculates the **compute(2)'s** return value as 3. So the control returns to 2.2.1.

2.2.1.R **compute(2)** returns value 3. So return value of 3 gets computed as **3+compute(2) = 3 + 3 = 6**. Now the control returns to 2.2.

2.2R **compute(3)** returns 6 and thus the return value of **compute (4)** gets computed as 4 + 6 = **10** and control returns to line **11** that called **compute(4)** as **ssum = compute(4).**

3. Now variable **ssum** gets the return value of **compute(4)** as 10 and then at line 12, it prints the *sum*.

What this program manages to do is to display the value of 4 + (3 + (2 + 1)). That is, it displays the sum of the numbers from 4 down to 1. More generally, what this *compute* method does is to return the sum of all the integers between 1 and its parameter *n*.

6.3 HOW RECURSION WORKS

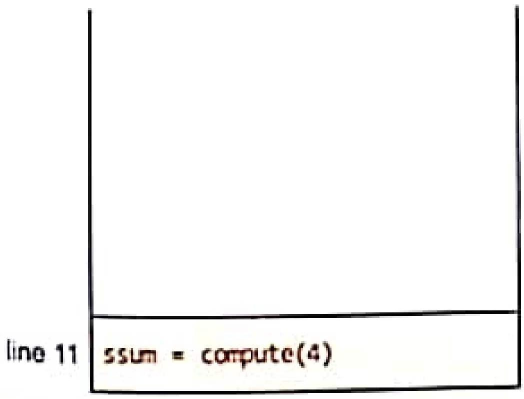
You'll notice that the *compute* function doesn't *always* recur: when its parameter *n* is 1, the function simply returns immediately without any recursive invocations. (*line* 5)

With a bit of thought, you'll realize that any functional recursive function must have such a situation, since otherwise, the recursive function will never finish. In fact, these situations are important enough to designate special case(s) where result is pre-known without invoking the function again : Any condition where a recursive function does not invoke itself is called a **base case.**

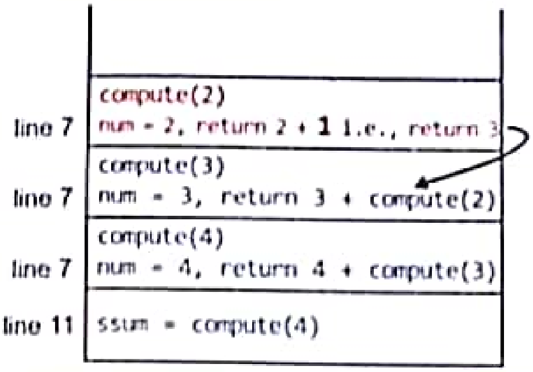
But what exactly happens when a recursive function lacks a base case ? To understand this, we need to get some idea about how a computer handles function invocations.

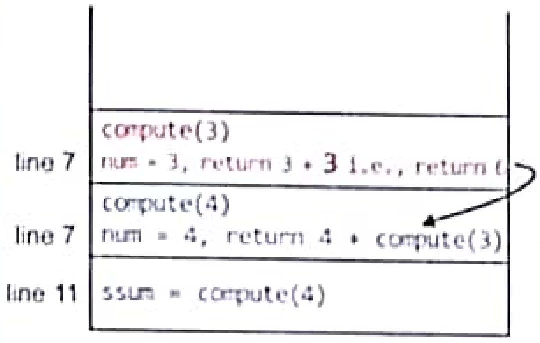
In executing a program, the computer creates what is called the **function stack.** The function stack is a stack of **frames**, each frame corresponding to a function invocation. At all times, the computer works on executing whichever function is at the stack's top; but when there is a function invocation, the computer creates a new frame and places it atop the stack. When the function at the stack's top returns, the computer removes that function's frame from the stack's top, and resumes its work on the function now on the frame's top.

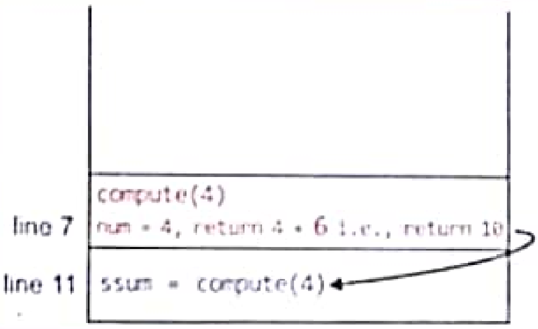
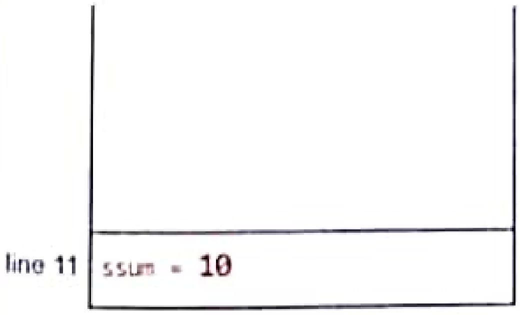
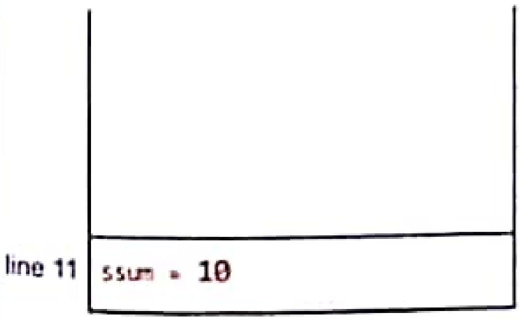
To see how this works, let's diagram how the program 6.1 operated.

1.		To start off the program, the program pushes a frame corresponding to an invocation to. Notice how this frame includes room for return value of compute(4) in variable ssum.

2.	compute(4) line 7 num = 4, return 4 + compute(3) line 11 ssum = compute(4)	When the computer invokes *compute*(4), the computer places a new frame atop the stack corresponding to *compute* ; this frame will include the variable *num*, whose value is initially 4.
3.	compute(3) line 7 num = 3, return 3 + compute(2) compute(4) line 7 num = 4, return 4 + compute(3) line 11 ssum = compute(4)	When the computer sees that *compute*(4) invokes *compute*(3), the computer places a new frame atop the stack, containing the variable *num*, whose value is initially 3.
4.	compute(2) line 7 num = 2, return 2 + compute(1) compute(3) line 7 num = 3, return 3 + compute(2) compute(4) line 7 num = 4, return 4 + compute(3) line 11 ssum = compute(4)	When the computer sees that *compute*(3) invokes *compute*(2), the computer places a new frame atop the stack, containing the variable *num*, whose value is initially 2.
5.	compute(1) line 7 num = 1, return 1 line 5 compute(2) line 7 num = 2, return 2 + compute(1) compute(3) line 7 num = 3, return 3 + compute(2) compute(4) line 7 num = 4, return 4 + compute(3) line 11 ssum = compute(4)	When the computer sees that *compute*(2) invokes *compute*(1), the computer places a new frame atop the stack, containing the variable *n*, whose value is initially 1.

6.	 compute(2)
	line 7
	num = 2, return 2 + 1 i.e., return 3
	compute(3)
	line 7
	num = 3, return 3 + compute(2)
	compute(4)
	line 7
	num = 4, return 4 + compute(3)
	line 11
	ssum = compute(4)
		When the computer sees that *compute*(1) returns, it pops the top frame off the stack and resumes with whatever frame is now at the top — which happens to be the frame for *compute*(2). Value for compute (1) is replaced with its return value which is 1

7.	 compute(3)
	line 7
	num = 3, return 3 + 3 i.e., return 6
	compute(4)
	line 7
	num = 4, return 4 + compute(3)
	line 11
	ssum = compute(4)
		When the computer sees that *compute*(2) returns with value3, it pops the top frame off the stack and resumes with *compute*(3).

8.	 compute(4) line 7 num = 4, return 4 + 6 i.e., return 10 line 11 ssum = compute(4)	When the computer sees that *compute*(3) returns with value 6, it pops the top frame off the stack and resumes with *compute*(4).
9.		When the computer sees that *compute*(4) returns with value 10, it pops the top frame off the stack and resumes with statement. `ssum = compute(4) i.e., ssum = 10`
10.		Now the last line of program is executed and result is printed as The sum of series from 1..4 is 10

And after this, the program ends and its frame is also removed from stack and stack becomes empty.

So we can say that above given compute(4) was executed as follows :

compute (4)

= 4 + compute (3)

= 4 + (3 + compute (2))

= 4 + (3 + (2 + compute(1)))

= 4 + (3 + (2 + 1))

= 4 + (3 + 3)

= 4 + 6

= 10

NOTE

It is mandatory to have a base case in order to write a sensible recursion code.

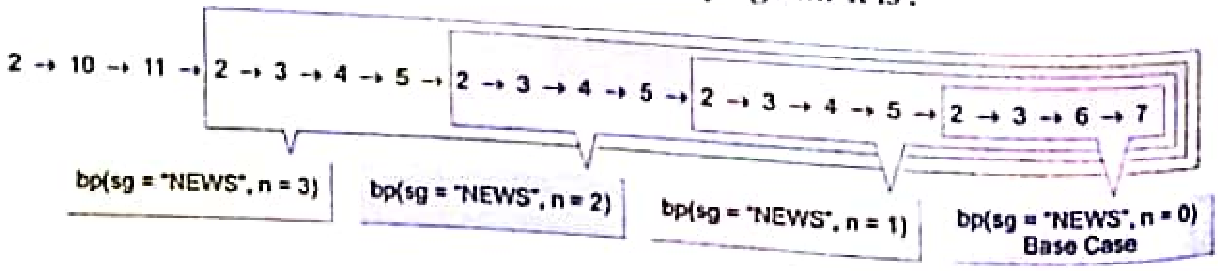
Program 6.2 Write a recursive function to print a string backwards.

```
1.  # simple recursion
2.  def bp(sg, n):
3.      if n > 0:
4.          print(sg[n], end = '')
5.          bp(sg, n-1)          ← Recursive call to function bp( )
6.      elif n == 0 :
7.          print(sg[0])
8.
9.  # __main__
10. s = input("Enter a string : ")
11. bp(s, len(s)-1)
```

The output produced by above program is as shown below :

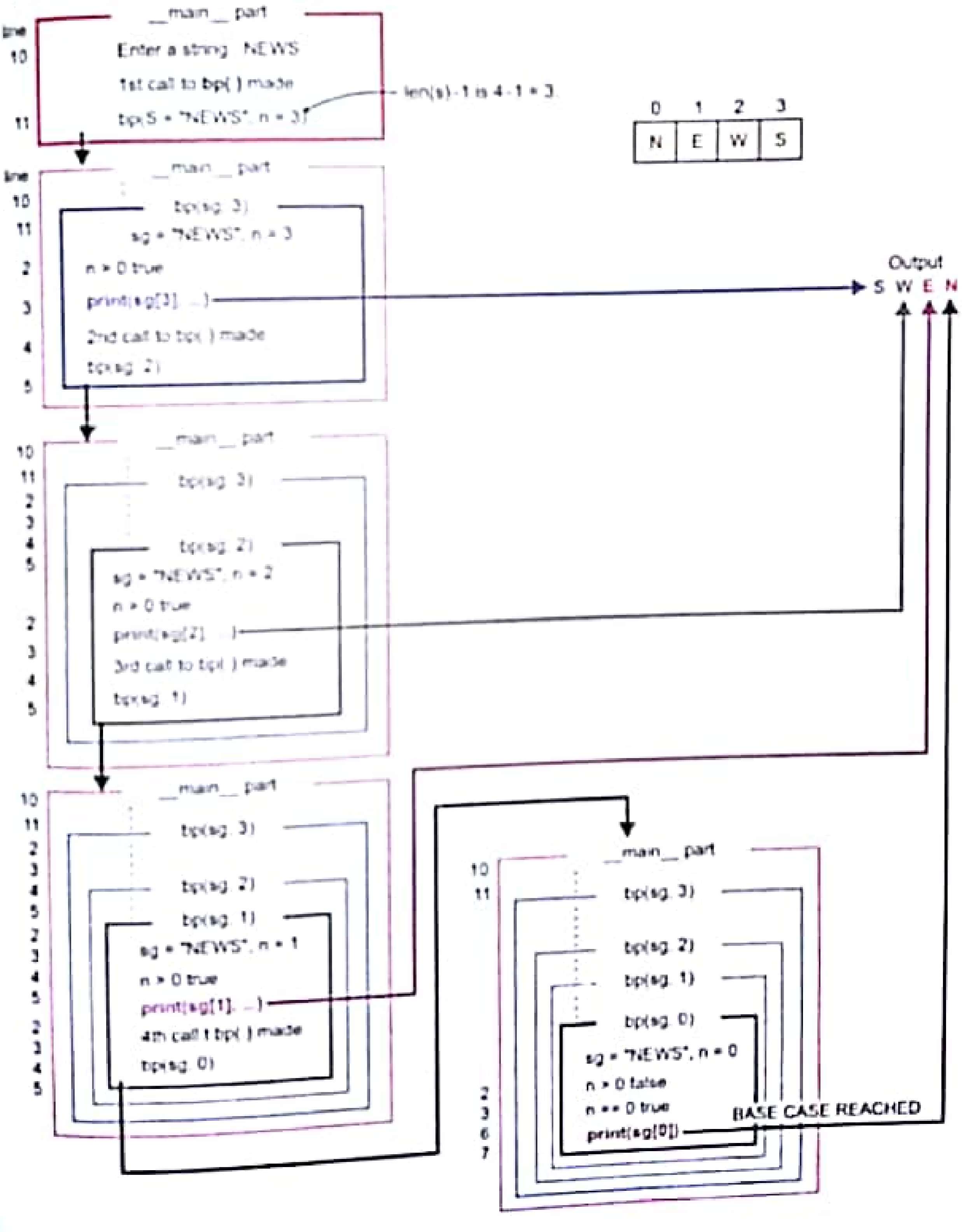
```
Enter a string : NEWS
SWEN
```

Can you figure out the flow of execution of above program? It is :



Let us see how this recursive program executed itself.

So what happens if a recursive method never reaches a base case? Hmm, you guessed it right – the stack will never stop growing. The computer, however, limits the stack to a particular height, so that no program eats up too much memory. If a program's stack exceeds this size, the computer initiates an exception, which typically would crash the program. (From the operating system's point of view, crashing the program is preferable to allowing a program to eat up too much memory and interfere with other better-behaved programs that may be running.) The exception is labelled as **maximum recursion limit exceeded**.



Consider another recursive function that also does the same thing as program 6.2 *i.e.*, backward printing of a string.

Program 6.3 Write a recursive function to print a string backwards.

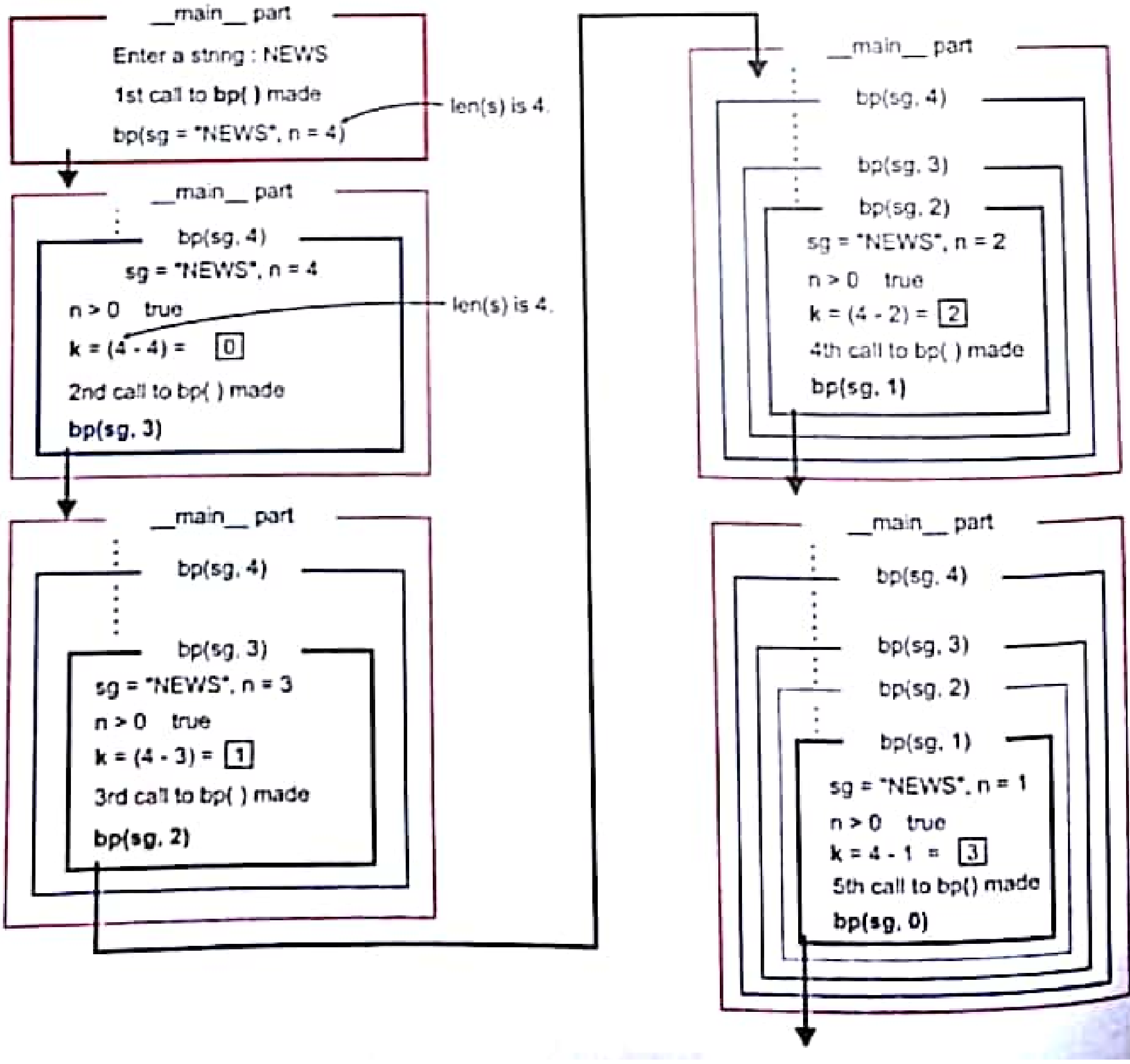
```
# simple recursion
def bp (sg, n):
    if n > 0:
        k = len(sg) - n
        bp (sg, n-1)
        print(sg[k], end = ' ')
    elif n == 0 :
        return

# __main__
s = input("Enter a string : ")
bp(s, len(s))
```

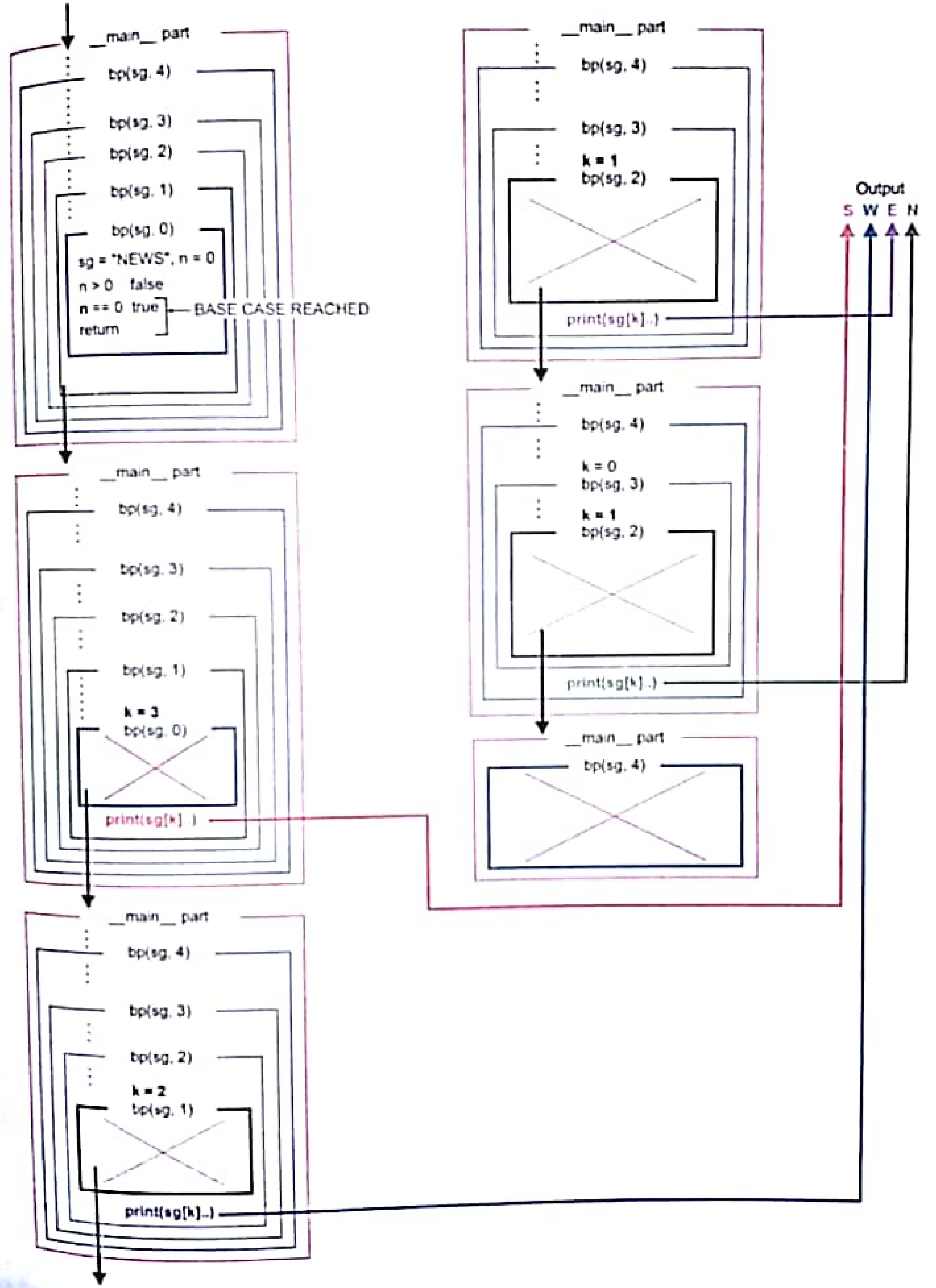
Recursive call to function **bp( )**

The output produced by above program is same as previous program, i.e. :

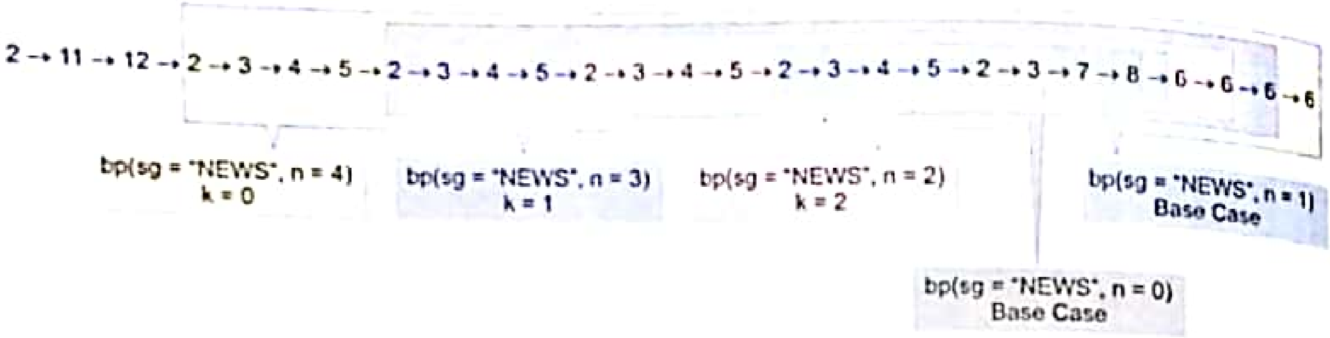
```
Enter a string : NEWS
SWEN
```

Let us see how above recursive program executed itself. Although it does the same thing as previous program 6.2, yet it is different from program 6.2.




__main__ part
bp(sg, 4)
bp(sg, 3)
bp(sg, 2)
bp(sg, 1)
bp(sg, 0)
sg = "NEWS", n = 0
n > 0 false
n == 0 true
return
BASE CASE REACHED
__main__ part
bp(sg, 4)
bp(sg, 3)
bp(sg, 2)
bp(sg, 1)
k = 3
bp(sg, 0)
print(sg[k]..)
__main__ part
bp(sg, 4)
bp(sg, 3)
bp(sg, 2)
k = 2
bp(sg, 1)
print(sg[k]..)
__main__ part
bp(sg, 4)
bp(sg, 3)
k = 1
bp(sg, 2)
print(sg[k]..)
__main__ part
bp(sg, 4)
k = 0
bp(sg, 3)
k = 1
bp(sg, 2)
print(sg[k]..)
__main__ part
bp(sg, 4)
Output
S W E N

Thus, as you can see that flow of execution of above program is as :


2 → 11 → 12 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 5 → 2 → 3 → 7 → 8 → 6 → 6 → 6 → 6
bp(sg = "NEWS", n = 4) k = 0
bp(sg = "NEWS", n = 3) k = 1
bp(sg = "NEWS", n = 2) k = 2
bp(sg = "NEWS", n = 1) Base Case
bp(sg = "NEWS", n = 0) Base Case

Technically, Python arranges the memory spaces needed for each function call in a *stack*. The memory area for each new call is placed on the top of the stack, and then taken off again when the execution of the call is completed. So, you can say that, the stack goes through the following sequence in recursive calls :

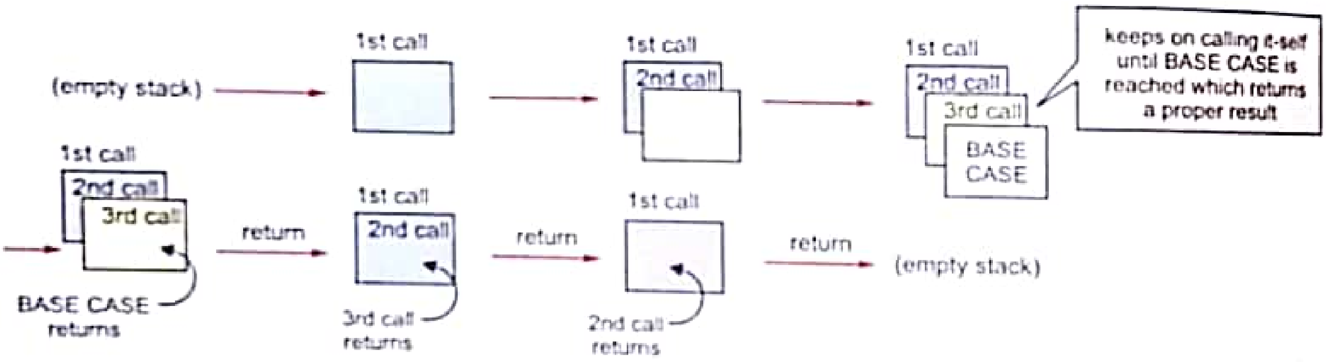


Figure 6.2 Memory manipulation in recursion.

Python uses this stacking principle for all nested function calls — not just for recursively-defined functions. A stack is an example of a "last in/first out" structure.

6.4 RECURSION IN PYTHON

As you have seen, recursion occurs when a function calls itself. In Python, as in other programming languages that support it, recursion is used as a *form of repetition that does not involve iteration*.

A **Recursive Definition** is a definition that is made in terms of a smaller version of itself. Have a look at following definition of x^n, which is non-recursive :

$$x^n = x * x * \ldots * x$$ (Iterative definition – **nonrecursive**)

Now, it can be represented in terms of recursive definition as follows :

$$\begin{aligned} x^n &= x * (x^{n-1}) && \text{for } n > 1 \\ &= x && \text{for } n = 1 \\ &= 1 && \text{for } n = 0 \end{aligned}$$ (**Recursive definition**)

Writing a Recursive Function. Before you start working recursive functions, you must know that every recursive function must have at least *two* cases :

(*i*) the **Recursive Case** (or the **inductive case**)

(*ii*) the **Base Case** (or the **stopping case**) – ALWAYS REQUIRED

- The **Base Case** is a small problem that we know how to solve and is the case that causes the recursion to end. In other words, it is the case whose solution is pre-known (either as a value or formula) and used directly.
- The **Recursive Case** is the more general case of the problem we're trying to solve using recursive call to same function.

NOTE

The Base Case in a recursive program MUST BE REACHABLE.

As an example, with the power function x^n, the **recursive case** would be :

```
Power (x, n) = x * Power (x, n - 1)
```

and the **base cases** would be

```
     Power (x, n) = x    when n = 1
and  Power (x, n) = 1    when n = 0
```

NOTE

Every recursive function consists of one or more base cases and a general, recursive case.

Other cases (when $n < 0$) we are ignoring for now for simplicity sake.

Consider the following example (program 6.4) of Power Function :

Program 6.4 *Program to show the use of recursion in calculation of power e.g., a^b*

```
# power a to b using recursion 6_4
def power(a, b) :
    if b == 0 :          # BASE CASE
        return 1
    else :
        return a * power(a, b-1)

# __main__
print("Enter only the positive numbers below")
num = int(input("Enter base number :"))
p = int(input("raised to the power of :"))
result = power(num , p)
print(num, "raised to the power of", p, "is", result)
```

The output produced by above program is as :

```
Enter only the positive numbers below
Enter base number : 7
raised to the power of : 3
7 raised to the power of 3 is 343
```

If *there is no base case, or if the base case is never executed,* ***an infinite recursion occurs.*** An Infinite Recursion is when a recursive function calls itself endlessly. Infinite recursion can happen in one of the following cases.

(i) **Base Case Not Defined.** When the recursive function has no BASE CASE defined, *infinite recursion* will occur.

(ii) **Base Case Not reached.** Sometimes you have written a base case but the condition to execute it, is never satisfied, hence the **infinite recursion** occur.

For example, the code of above program 6.4 will face infinite recursion if you enter a negative value for power. In that case, the condition for base case, *i.e.,*

```
if b == 0 :   ←—— This condition will never be true for a
                   negative value of b
    return 1
```

will never be satisfied and hence infinite recursion will occur.

To avoid infinite recursion, your code should take care of possible values that may cause infinite recursion, *e.g.,* above program (6.4)'s code can be modified as follows to avoid infinite recursion :

NOTE

There can be one or more base cases in a recursive code.

NOTE

An Infinite Recursion is when a recursive function calls itself endlessly.

```
def power(a, b) :
    if b <= 0 :   ←—— Now it takes care of negative
                       values of b
        return 1
    else :
        return a * power(a, b-1)
# __main__
:
```

Or

You may check for negative value of power in *__main__ part* as well, *i.e.,* as :

```
def power(a, b) :
    :
# __main__
num = int(input("Enter base number :"))
p = int(input("raised to the power of :"))
if p < 0 :                                    ←—— Now it takes care of negative values
    print ("Sorry, negative power not allowed")    of power
else :
    result = power(num , p)
    print(num, "raised to the power of", p, "is", result)
```

You can easily determine the flow of execution in above program.

Iterative Version

A recursive code may also be written in non-recursive way, which is the iterative version of the same. For instance, the iterative version of above program(6.4)'s code is given in program 6.5. below :

Program **6.5** Program to calculate a^b using iterative code.

```
# power a to b using iteration

def power(a, b) :
    res = 1
    if b == 0 :
        return 1
    else :
        for i in range(b):
            res = res * a
    return res

#__main__
print("Enter only the positive numbers below")
num = int(input("Enter base number :"))
p = int(input("raised to power of :"))
result = power(num , p)
print(num, "raised to power of", p, "is", result)
```

Iterative version to calculate a^b .

The output produced by above program is as :

```
Enter base number : 3
raised to power of : 4
3 raised to power of 4 is 81
```

6.4.1 Some Recursive Codes

Let us write some Python codes that use recursion carrying out its task.

A. Computing Factorial Recursively

Now, consider another example. The factorial of a non-negative number is defined as the product of all the values from 1 to the number :

$$n! = 1 * 2 * ... * n$$

It can also be defined recursively as

$$\begin{aligned} n! &= 1 && \text{if } n < 2 \\ &= n * (n-1)! && \text{if } n >= 2 \end{aligned}$$

Recursively factorial function would be written as follows :

```
# recursive factorial function 6.6
def factorial (n) :
    if n < 2:
        return 1
    return n * factorial(n-1)
```

```
#__main__
n = int(input("Enter a number (> 0) : "))
print("Factorial of", n, "is", factorial(n))
```

Sample runs of above program code are :

```
Enter a number (> 0) : 4
Factorial of 4 is 24
==========================
Enter a number (> 0) : 5
Factorial of 5 is 120
```

The iterative version of above given factorial function is (you already have written this) :

```
def factorial (n) :
    fact = n
    for I in range(1 , n):
        fact = fact * I
    return fact
```

B. Computing GCD recursively

We can efficiently compute the gcd using the following property, which holds for positive integers *p* and *q* :

```
If p > q
```

the gcd of p and q is the same as the gcd of p and p % q. That is, our Python code to compute GCD recursively would be :

```
def gcd (p, q) :
    if q == 0 :
        return p
    return gcd(q, p % q)
```

The base case is *when q is 0, with gcd(p, 0) = p*. To see that the reduction step converges to the base case, observe that the second input strictly decreases in each recursive call since p % q < q. If p < q the first recursive call switches the arguments.

```
gcd(1440, 408)
    gcd(408, 216)
        gcd(216, 24)
            gcd(192, 24)
                gcd(24, 0)
                return 24
            return 24
        return 24
    return 24
return 24
```

This recursive solution to the problem of computing the greatest common divisor is known as *Euclid's algorithm* and is one of the oldest known algorithms – it is over 2000 years old.

You can write recursive code for GCD computation yourself. (also refer to solved problem 15)

C. Fibonacci Numbers

You have written program to compute and display Fibonacci series using a loop. Here we are going to develop a recursive method to compute numbers in the Fibonacci sequence. This infinite sequence starts with 0 and 1, which we'll think of as the zeroth and first Fibonacci numbers, and each succeeding number is the sum of the two preceding *Fibonacci numbers*. Thus, the ***third term*** is 0 + 1 = 1. And to get the ***fourth term*** Fibonacci number, we'd sum the *2nd term* (1) and the *3rd term* (1) to get 2. And the ***fifth term*** is the sum of the *3rd term* (1) and the *4th term* (2), which is 3. And so on.

n	:	0	1	2	3	4	5	6	7	8	9	10	11	...
nth Fibonacci	:	0	1	1	2	3	5	8	13	21	34	55	89	...

We want to write a method *fib* that takes some integer *n* as a parameter and returns the *n*th Fibonacci number, where we think of the first 1 as the first Fibonacci number. Thus, an invocation of *fib*(6) should return 8, and the invocation of *fib*(7) should return 13.

```
def fib(n) :
    if n == 1 :            # 1st term is 0
        return 0
    elif n == 2 :          # 2nd term is 1
        return 1
    else :
        return fib(n -1) + fib(n -2)
```

These are the BASE CASES – value is returned without invoking the function

RECURSIVE CASE – the fib( ) function is invoking itself from its own body

In talking about recursive procedures such as this, it's useful to be able to diagram the various method calls performed. We'll do this using a recursion tree. The recursion tree for computing *fib*(5) is in Fig. 6.3.

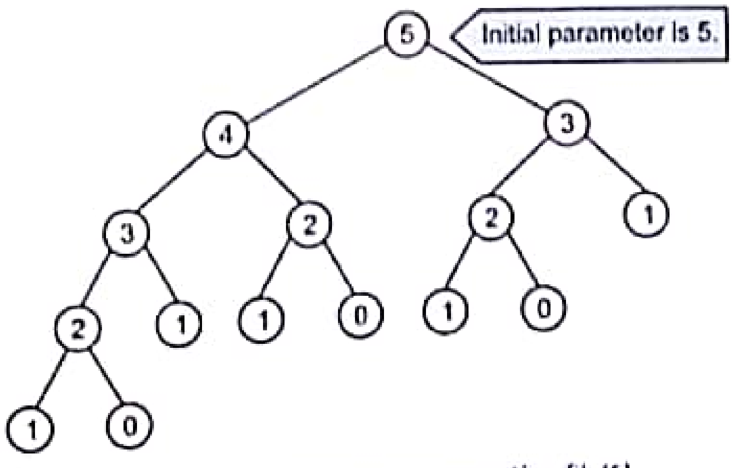


Figure 6.3 Recursion tree for computing *fib* (5).

The recursion tree has the original parameter (5 in this case) at the top, representing the original method invocation. In the case of *fib*(5), there would be two recursive calls, to *fib*(4) and *fib*(3), so we include 4 and 3 in our diagram below 5 and draw a line connecting them. Of course, *fib*(4) has two recursive calls itself, diagrammed in the recursion tree, as does *fib*(3). The complete diagram in Fig. 6.3 depicts all the recursive invocation of *fib* made in the course of computing *fib*(5). The bottom of the recursion tree depicts those cases when there are no recursive calls – in this case, when *n* <= 1.

Following program lists the complete code of using recursive fib( ) function given above.

Program 6.7 Program using a recursive function to print fibonacci series upto nth term.

```
def fib(n) :
    if n == 1 :                          # 1st term is 0
        return 0
    elif n == 2 :                        # 2nd term is 1
        return 1
    else :
        return fib( n -1 ) + fib( n -2 )
# __main__
n = int(input("Enter last term required :"))
for i in range(1, n + 1 ) :              # list with values 1..n
    print(fib(i), end = ', ')
print("...")
```

The output produced by above program is :

```
Enter last term required : 8
0, 1, 1, 2, 3, 5, 8, 13, ...
```

6.4.2 Binary Search

Another popular algorithm that uses recursion successfully is binary search algorithm. But hey, we have not talked about binary search before. No worries! We shall first discuss what *binary search* is, how it works, write its normal iterative code, and then use it recursively too.

Binary Search Technique

This popular search technique searches the given *ITEM* in minimum possible comparisons. The *binary search* requires **the array**, to be scanned, **must be sorted in any order** (for instance, say ascending order). In binary search, the *ITEM* is searched for in a smaller *segment* (nearly half the previous segment) after every stage. For the first stage, the segment contains the entire array.

To search for *ITEM* in a sorted array (in *ascending order*), the *ITEM* is compared with *middle element* of the segment (*i.e.*, in the entire array for the first time). If the *ITEM* is more than the *middle element*, latter part of the segment becomes new segment to be scanned ; if the *ITEM* is less than the *middle element*, former part of the segment becomes new segment to be scanned. The same process is repeated for the new segment(s) until either the *ITEM* is found (search successful) or the segment is reduced to the single element and still the ITEM is not found (search unsuccessful).

NOTE

Binary search can work for only sorted arrays whereas linear search can work for both sorted as well as unsorted arrays.

Following figure illustrates the process of binary search in a sorted array.

Search item is Key in a sorted array with N elements.

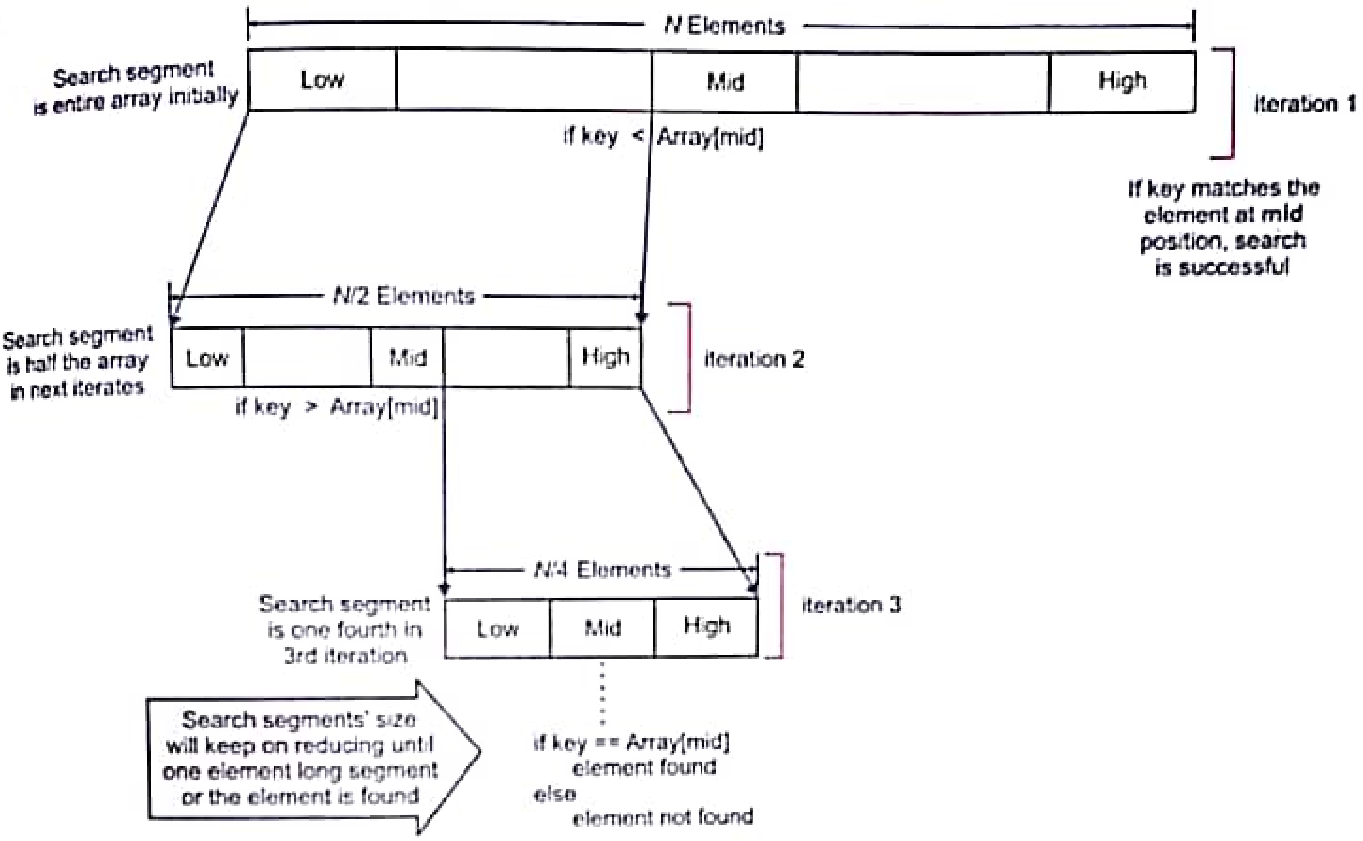

N Elements
Search segment is entire array initially
Low
Mid
High
Iteration 1
if key < Array[mid]
If key matches the element at mid position, search is successful
N/2 Elements
Search segment is half the array in next iterates
Low
Mid
High
Iteration 2
if key > Array[mid]
N/4 Elements
Search segment is one fourth in 3rd iteration
Low
Mid
High
Iteration 3
Search segments' size will keep on reducing until one element long segment or the element is found
if key == Array[mid]
element found
else
element not found

Figure 6.4 Working of Binary Search Algorithm.

Algorithm *Binary Search in Linear List (Array AR[L : U])*

Case I : AR in *ascending* order

```
#Initialise segment variables
1.  Set beg = L, last = U    # L is 0, U is size-1
2.  while beg <= last, perform steps 3 to 6
3.        mid = (beg + last)/2
4.        if AR [mid] == ITEM then
          {   print("Search Successful")
              print(ITEM, "found at", mid)
              break
          }
5.        if AR[mid] < ITEM then
              beg = mid + 1
6.        if AR[mid] > ITEM then
              last = mid - 1
     # End of while
7.   if beg ≠ last
          print("Unsuccessful Search")
8.   END.
```

Case II : AR in *descending order*

```
# Initialise
:
if AR[mid] = = ITEM then
{     :
}
if AR[mid] < ITEM then
     last = mid - 1
if AR[mid] > ITEM then
     beg = mid + 1
:
# Rest is similar to the algorithm in Case I.
```

NOTE

beg, last signify the limits of the search segment. Sometimes beg, last are also termed as low and high to signify lower and higher limit of the search segment.

Program 6.8 Binary Searching in an array (a sorted list).

```
def binsearch(ar, key) :
    low = 0                       # initially low end is at 0
    high = len(ar) - 1            # initially high end is at size -1
    while low <= high :
        mid = int((low + high) / 2)
        if key == ar[mid] :       #if key matches the middle element
            return mid            # then send its inde in array
        elif key < ar[mid] :
            high = mid - 1        # now the segment should be first half
        else:
            low = mid + 1         # now the segment should be latter half
    else:   # loop's else
        return -999

#__main__
ar = [12, 15, 21, 25, 28, 32, 33, 36, 43, 45]
item = int(input("Enter search item :"))
res = binsearch(ar, item)
if res >= 0 :        # if res holds a 0..n value
    print(item,"FOUND at index", res )
else:
    print("Sorry!", item, "NOT FOUND in array")
```

```
Enter search item: 32
32 FOUND at index 5
================================
Enter search item: 15
15 FOUND at index 1
================================
Enter search item: 10
Sorry! 10 NOT FOUND in array
================================
Enter search item: 45
45 FOUND at index 9
================================
Enter search item: 12
12 FOUND at index 0
================================
Enter search item: 35
Sorry! 35 NOT FOUND in array
```

Some sample runs of the above program as shown on the right.

Pre-requisits of Binary Search

In order to implement binary search on an array, following conditions must be fulfilled.

(*i*) the given array or sequence must be sorted.

(*ii*) its sort-order and size must be known.

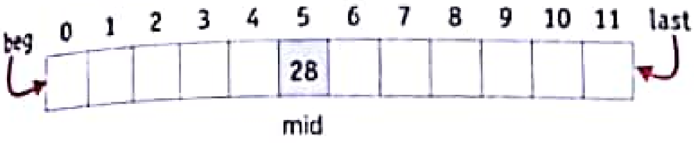
NOTE

Binary search is very fast compared to linear search on one-dimensional arrays. However, linear search can work on both sorted and unsorted arrays while binary search can work with sorted arrays only. Also, binary search algorithm works with single-dimensional array where linear search algorithm can work with single and multi-dimensional arrays.

Example 6.1 *Write the steps to search 44 and 36 using binary search in the following array DATA :*

10	12	14	21	23	28	31	37	42	44	49	53
0	1	2	3	4	5	6	7	8	9	10	11

Solution. (*i*) **Search for 44.**



Step I :

beg = 0 ; last = 11

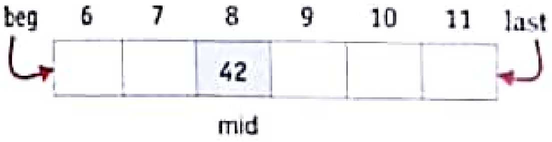
$$\text{mid} = \text{INT}\ \frac{(0+11)}{2} = \text{Int}\ (5.5) = 5$$

Step II :

Data[mid] *i.e.*, Data[5] is 28

28 < 44 then

beg = mid + 1 *i.e.*, beg = 5 + 1 = 6

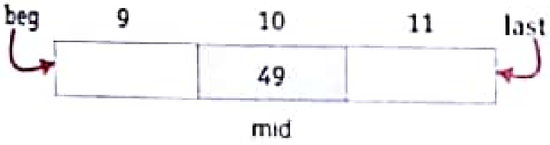


Step III :

$$\text{mid} = \text{INT}\ ((\text{beg} + \text{last})/2) = \text{INT}\left(\frac{(6+11)}{2} = 8\right)$$

Data[8] *i.e.*, 42 < 44 then

beg = mid + 1 *i.e.*, beg = 8 + 1 = 9



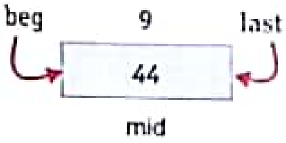
Step IV :

$$\text{mid} = \text{INT}\left(\frac{9+11}{2}\right) = 10$$

Data[10] *i.e.*, 49 > 44 then

last = mid - 1

last = 10 − 1 = 9



Step V :

$$\text{mid} = \text{INT}\left(\frac{(\text{beg} + \text{last})}{2}\right) = \frac{(9+9)}{2} = 9$$

(beg = last = 9)

Data [9] *i.e.*, 44 = 44

SEARCH SUCCESSFUL !! At location number 10.

(*ii*) **Search for 36**

Step I : beg = 0 ; last = 11

$$\text{mid} = \text{INT}\left(\frac{0+11}{2}\right) = 5$$

Step II : Data [mid] *i.e.*, Data [5] is 28

28 < 36 then beg

= mid + 1 = 5 + 1 = 6

Step III :

$$\text{mid} = \text{INT}\left(\frac{6+11}{2}\right) = 8$$

Data[8] *i.e.*, 42

42 > 36 then

last = mid − 1 = 8 − 1 = 7

Step IV :

$$\text{mid} = \text{INT}\left(\frac{6+7}{2}\right) = 6$$

Data [6] is 31

31 < 36 then

beg = mid + 1 = 6 + 1 = 7

Step V :

$$\text{mid} = \text{INT}\left(\frac{7+7}{2}\right) = 7$$

(beg = last = 7)

Data [7] is 37 ⇒ 37 ≠ 36

SARCH UNSUCCESSFUL !!

6.4.3 Recursive Binary Search

Now that you have fair idea about Binary search algorithm and how it works, let us see how it can be implemented recursively.

As you can see that process of finding the element is same in all search-segments, only the lower limit *low* and higher limit *high* of a segment changes if the element is not found at the middle (*mid*) position of the search-segment. Thus, a recursive call can be made to the binary search function by changing the limits. The search stops when either the element is found and its index is returned OR lower limit becomes more than higher limit–this will happen when the search- segment reduces to size of single element and search is still unsuccessful, in this case both mid + 1 or mid – 1 will make lower limit more than higher limit - which means search is unsuccessful.

Following program lists the recursive version of binary search algorithm.

Program 6.9 Write a recursive function to implement binary search algorithm.

```
# binary recursive search
def binsearch(ar, key, low, high) :
    if low > high :                          # search unsuccessful
        return -999
                                              BASE CASES
    mid = int((low + high) / 2)
    if key == ar[mid] :
        return mid                           # if key matches the middle element
                                             # then send its index in array
    elif key <ar[mid] :
        high = mid -1
                                             # now the segment should be first half
        return binsearch(ar, key, low, high)
    else:                                           RECURSIVE CASES
        low = mid + 1
                                             # now the segment should be latter half
        return binsearch(ar, key, low, high)

#__main__
ary = [12, 15, 21, 25, 28, 32, 33, 36, 43, 45]
                             # sorted array
item = int(input("Enter search item:"))
res = binsearch(ary, item, 0, len(ary)-1)
if res >= 0 :   # if res holds a 0..n value,
    print(item,"FOUND at index", res)
else:
    print("Sorry!", item, "NOT FOUND in array")
```

Some sample runs of the above program as shown on the right .

```
Enter search item: 32
32 FOUND at index 5
===========================
Enter search item: 15
15 FOUND at index 1
===========================
Enter search item: 10
Sorry! 10 NOT FOUND in array
===========================
Enter search item: 45
45 FOUND at index 9
===========================
Enter search item: 12
12 FOUND at index 0
===========================
Enter search item: 35
Sorry! 35 NOT FOUND in array
```

RECURSION PRACTICALLY

PriP — Progress In Python 6.1

This 'PriP' session is aimed at practice of recursion in Python.

:

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 6.1 under Chapter 6 after practically doing it on the computer.

>>>❖<<<

Check Point 6.1

Which of the following are examples of recursive functions ?

(a)
```
def Print(ch) :
    if ch != ' ' :
        print(ch + 1)
Print('k')
```

(b)
```
def recur(p) :
    if p == 0 :
        print("##")
    else:
        recur(p)
        p = p - 1
    recur(5)
```

(c)
```
def recur(p) :
    if p == 0 :
        print("#")
    else:
        p = p - 1
        recur1(p)
def recur1(n) :
    if n % 2 == 0:
        return n
    else:
        return (n-5)
```

(d)
```
def Check(c) :
    Mate(c+1)
def Mate(d) :
    Check(d-1)
```

(e)
```
def PrnNum(n) :
    if n == 1 :
        return
    else :
        print(n)
        PrnNum(n-2)
```

6.5 RECURSION VS ITERATION

Recursion and loops are actually related concepts. Generally, anything you can do with a loop, you can do with recursion, and vice versa. Sometimes one way is simpler to write, and sometimes the other is, but in principle they are inter- changeable. Although loop and recursion are interchangeable, yet there are some examples where recursion is indeed the best way to approach a problem.

Even for problems that can be treated equally well through iteration and recursion, there is a subtle difference which is because of the way loops and method calls are treated by a programming language compiler.

When a loop repeats, it uses same memory locations for variables and repeats the same unit of code. Whereas in recursion, instead of repeating the same unit of code and using the same memory locations for variables, fresh memory space is allocated for each recursive call.

As it happens, any problem that can be solved via iteration can be solved using recursion and any problem that can be solved via recursion can be solved using iteration. Iteration is preferred by programmers for most recurring events, reserving recursion for instances where the programming solution would be greatly simplified. In a programming language, recursion involves an additional cost in terms of the space used in RAM by each recursive call to a function and in time used by the function call.

Because of extra memory stack manipulation, recursive versions of functions often run slower and use more memory than their iterative counterparts. But this is not always the case, and recursion can sometimes make code easier to understand.

So, you can say that :

- recursion makes a solution look shorter, closer in spirit to abstract mathematical entity.
- iteration makes a solution less costly to implement but all logic is part of loop, so lengthier it appears.

It always depends on the problem being solved which approach is better for it – iteration or recursion.

Let Us Revise

- *A function is said to be recursive if it calls itself.*
- *There are two cases in each recursive functions ; the recursive case and the base case.*
- *The base case is the case whose solution is pre-known and is used without computation.*
- *The recursive case is more general case of problem, which is being solved.*
- *An infinite recursion is when a recursive function calls itself endlessly.*
- *If there is no base case, or if the base case is never executed, infinite recursion occurs.*
- *Iteration uses same memory space for each pass contrary to recursion where fresh memory is allocated for each successive call.*
- *Recursive functions are relatively slower than their iterative counterparts.*
- *Some commonly used recursive algorithms are factorial, gcd, fibonacci series printing, binary search etc.*

Solved Problems

1. *What is recursion ?*

Solution. In a program, if a function calls itself (whether directly or indirectly), it is known as recursion. And the function calling itself is called recursive function *e.g.*, following are two examples of recursion :

(*i*)
```
def A( ) :
      A( )
```

(*ii*)
```
def B( ) :
      C( )
def C( ) :
      B( )
```

2. *What are base case and recursive case? What is their role in a recursive program?*

Solution. In a recursive solution, the **Base cases** are predetermined solutions for the simplest versions of the problem : if the given problem is a base case, no further computation is necessary to get the result.

The **recursive case** is the one that calls the function again with a new set of values. The recursive step is a set of rules that eventually reduces all versions of the problem to one of the base cases when applied repeatedly.

3. *Which of the following is correct skeleton for a recursive function?*

(*a*)
```
def solution (N) :
      if base_case_condition :
            return something easily computed or directly available
      else :
            divide problem into pieces
            return something calculated using solution(some number)
```

(*b*)
```
def solution (N) :
      if base_case_condition :
            return something easily computed or directly available
      else :
            divide problem into pieces
            return something calculated using solution(N)
```

(c)

```
def solution (N) :
        divide problem into pieces
        return something calculated using solution(N)
```

(d)

```
def solution (N) :
        if base_case_condition :
            return something easily computed or directly available
        else :
            divide problem into pieces
            return something calculated using solution(some number other than N)
```

Solution. (*d*)

4. *Why is base case so important in a recursive function ?*

   Solution. The base case, in a recursive case, represents a pre-known case whose solution is also preknown.

   This case is very important because upon reaching at base case, the termination of recursive function occurs as base case does not invoke the function again, rather it returns a pre-known result. In the absence of base case, the recursive function executes endlessly. Therefore, the execution of base case is necessary for the termination of the recursive function.

5. *When does infinite recursion occur ?*

   Solution. Infinite recursion is when a recursive function executes itself again and again, endlessly. This happens when either the base case is missing or it is not reachable.

6. *Compare iteration and recursion.*

   Solution. In iteration, the code is executed repeatedly using the same memory space. That is, the memory space allocated once, is used for each pass of the loop.

   On the other hand in recursion, since it involves function call at each step, fresh memory is allocated for each recursive call. For this reason *i.e.*, because of function call overheads, the recursive function runs slower than its iterative counterpart.

7. *State one advantage and one disadvantage of using recursion over iteration.*

   Solution. **Advantage.** Recursion makes the code short and simple while iteration makes the code longer comparatively.

   Disadvantage. Recursion is slower than iteration due to overhead of multiple function calls and maintaining a stack for it.

8. *Consider the following function that takes two positive integer parameters x and y. Answer the following questions based on the code below.*

```
def compute (x, y) :
    if x > 1:
        if x % y == 0:
            print(y, end = ' ')
            compute (int(x/y), y)
        else:
            compute (x, y + 1)
```

(*a*) *What will be printed by the function call* **compute (24, 2)** *?*

(*b*) *What will be printed by the function call* **compute (84, 2)** *?*

(*c*) *State in one line what is* **compute( )** *trying to calculate ?*

Solution. (*a*) 2 2 2 3 (*b*) 2 2 3 7 (*c*) Finding factors of x which are >= y

9. *What will the following function Check( ) return when the values of both 'm' and 'n' are equal to 5 ? Show the working.*

```
def Check(m, n) :
    if n == 1 :
        return -m
    else:
        return (m + 1) + Check(m + 1, n - 1)
```

Solution.

Start	m	n	Statement executed	Internal work and Stack
Call 1	5	5	Check (m, n) *i.e.,* Check (5, 5) n == 1	5 == 1 False
Call 2	6	4	return (m + 1) + Check(m, n−1) Check (6, 4) n == 1	return 6 + Check (6, 4) 6 + Check (6, 4) 4 == 1 = False

Call 3	7	3	return (m + 1) + Check(m, n−1)
			Check(7, 3)
			n == 1
				return 7 + Check(7, 3)
				7 + Check(7, 3)
				6 + Check (6, 4)
				3 == 1 = False

Call 4	8	2	return (m + 1) + Check (m, n−1)
			Check(8, 2)
			n == 1
				return 8 + Check(8, 2)
				8 + Check(8, 2)
				7 + Check(7, 3)
				6 + Check (6, 4)
				2 == 1 = False

Call 5	9	1	return (m + 1) + Check(m, n−1)
			Check(9, 1)
				return 9 + Check(9, 1)
				9 + Check(9, 1)
				8 + Check(8, 2)
				7 + Check(7, 3)
				6 + Check (6, 4)

Call 6	8		n == 1 return − m ;	1 == 1 = True − 9 and returns to call 5 statement *i.e.,* Check (9, 1) −9 9 + (−9) = 0 8 + Check(8, 2) 7 + Check(7, 3) 6 + Check (6, 4)
Call 5			return 9 + (− 9)	returns 0 to Call 4 *i.e.,* Check(8, 2) 0 8 + (0) = 8 7 + Check(7, 3) 6 + Check (6, 4)
Call 4			return 8 + 0	returns 8 to Call 3 *i.e.,* Check(7, 3) 8 7 + (8) = 15 6 + Check (6, 4)
Call 3			return 7 + 8	returns 15 to Call 2 *i.e.,* Check(6, 4) 15 6 + (15) = 21
Call 2			return 6 + 15	returns 21 to Call 1
Call 1			21 = Ans	

10. *Consider the following recursive function which has a base case defined and recursive case too. But when run with an odd number as parameter, this always gives* ***RecursionError.*** *Figure out the reason and suggest the solution.*

```
def skip_prod(n):
    """Return the product of n * (n - 2) * (n - 4) * ... """
    if n == 0 :                          # the base case
        return 1
    else:
        return n * skip_prod(n - 2)      # recursive case
```

Solution. In a recursive code, the base case should always be available and reachable too. That means, it must get executed for some value of the parameter passed.

Let us consider what happens when we choose an odd number for *n, e.g.,*

```
skip_prod(3) will return 3 * skip_prod(1).
skip_prod(1) will return 1 * skip_prod(-1).
```

Here arises the problem. Since parameter *n* decreases by 2 at a time, for odd numbers, it will completely miss **the base case**, as from 1, the value of *n* will decrease to –1 completely missing ***n* == 0, the base case** ; and the function will end up recursing indefinitely.

The solution to this case is that the values less than 1 should also be considered. Thus the corrected code may be like : (there may be other solutions as well)

```
def skip_prod(n):
    if n <= 0 : ←————————————— Now, base case is reachable for all values
        return 1
    else:
        return n * skip_prod(n - 2)
```

11. *Figure out the error in the following recursive code of factorial :*

```
def factorial(n) :
    if n == 0:
        return 1
    else:
        n * factorial(n-1)
#__main__
print(factorial(4) )
```

Solution. The error in above code is that the recursive case is calling function factorial( ) with a changed value but not returning the result of the call.

The corrected version will be :

```
def factorial(n) :
    if n == 0:
        return 1
    else:
        return n * factorial(-1)
#__main__
print(factorial(4))
```

12. *Why is following code printing 1 endlessly ?*

```
def Out_upto(n):
    i = 1
    if i > n:
        return
    else:
        print(i)
        i += 1
        Out_upto(i)
Out_upto(4)
```

Solution. The above code is printing 1 endlessly because the base case **i > n** is never reachable. The reason is that value of *i* becomes 1 for each call to function *Out_upto( )*.

13. *Write your own version of code so that the problem in previous question gets solved.*

Solution.

```
def Out_upto(n):
    if n == 0 :
        return
    else:
        Out_upto(n-1)
        print(n)
#__main__
Out_upto(4)
```

14. *Write recursive code to compute and print sum of squares of n numbers. Value of n is passed as parameter.*

Solution.

```
def sqsum(n):
    if n == 1:
        return 1
    return n * n + sqsum(n - 1)
#__main__
n = int(input("Enter value of n :"))
print(sqsum(n))
```

15. *Write recursive code to compute greatest common divisor of two numbers.*

Solution.

```
def gcd(a,b):
    if(b == 0):
        return a
    else:
        return gcd(b, a % b )
n1 = int(input("Enter first number:"))
n2 = int(input("Enter second number:"))
d = gcd(n1, n2)
print("GCD of", n1, "and", n2, "is:", d)
```

7

Idea of Algorithmic Efficiency

7.1 INTRODUCTION

An algorithm is a method or procedure for accomplishing a specific task, and which is sufficiently precise and that can be programmed on computer. In Computer Science, it is important to measure **efficiency of algorithms** before applying them on a large scale *i.e.*, on bulk of data. The quality of an algorithm or shall we say, performance of an algorithm depends on many *internal* and *external* factors.

Internal Factors specify algorithm's efficiency in terms of :

- *Time* required to run
- *Space* (or Memory) required to run.

External Factors affect the algorithm's performance. These include :

- *Size of the input* to the algorithm
- *Speed of the computer* on which it is run
- *Quality of the compiler.*

Since, external factors are controllable to some extent, mainly *internal factors* are studied and measured in order to determine an algorithm's efficiency or we can say *complexity*. *Complexity* of an algorithm is determined by studying and measuring internal factors affecting the algorithm.

This chapter is going to introduce how you can determine or measure efficiency of an algorithm in terms of computational complexity. Let us begin our discussion with '*What is computational complexity ?*'

7.2 WHAT IS COMPUTATIONAL COMPLEXITY ?

The term *'Computational complexity'* is made of *two* words : *'Computation'* and *'Complexity'*.

The **Computation** involves the *problems* to be solved and the algorithms to solve them. **Complexity** involves the study of factors to determine how much *resource* is sufficient/necessary for this algorithm to run efficiently (performance).

The *resources* generally include :

- The *time* to run the algorithm (*Temporal complexity*).
- The space (or the memory/storage) needed to run the algorithm (*Space Complexity*)[1].

COMPLEXITY

Complexity refers to the measure of the efficiency of an algorithm.

The first thing to take into account is the difference between **efficiency** and **effectiveness**. ***Effectiveness*** means that the algorithm carries out its intended function *correctly*. But ***efficiency*** means the algorithm should be *correct with the best possible performance*. And to measure *efficiency*, we determine complexity. Complexity of an algorithm quantifies the resources needed as a function of the amount of data processed.

Complexity is not the absolute measure, but rather a bounding function characterizing the behaviour of the algorithm as the size of the data set increases. It allows the comparison of algorithm for efficiency, and predicts their behaviour as data size increases.

7.3 ESTIMATING COMPLEXITY OF ALGORITHMS

As mentioned, algorithms are usually compared along *two* dimensions : ***amount of space*** (that is memory) used and the ***time taken***. Of the two, the *time taken* is usually considered the more important. The motivation to study time complexity is to compare different algorithms and use the one that is the most efficient in a particular situation.

Actual run time on a particular computer (external factors) is not a good basis for comparison since it depends heavily on the speed of the computer, the total amount of RAM in the computer, the OS running on the system and the quality of the compiler used. So we need a more abstract way to compare the time complexity of algorithms.

Informally, we can define *time complexity* of a program (for a given input) is the number of elementary instructions that this program executes. This number is computed with respect to the size *n* of the input data.

7.3.1 Big-O Notation

The Big-O notation is used to depict an algorithm's growth rate. *The growth rate determines the **algorithm's performance when its input size grows***. Through big-O, the upper bound of an algorithm's performance is specified *e.g.*, if we say an algorithm takes $O(n^2)$ time ; this means that this algorithm will carry out its task taking **at the most N^2 steps for input size N.**

The Big-O notation is very useful for comparing the performance of two or more algorithms. For instance, if we say that we have two algorithms for solving a problem ; one has complexity $O(n^2)$, and the other has complexity $O(2^n)$. We can now, compare the number of steps needed to solve the problem for different sizes as depicted in following table :

[1] **These days, for web specific algorithms, *Network Traffic* may also be considered.**

Comparing Number of Steps for O(n^2) and O(2^n) Algorithms

Size N Complexity	10	20	40	100	400
N^2	100	400	1600	10000	160000
2^n	1024	1048576	10^{12}	1.26×10^{30}	Very Big...

More the number of steps, more is the time taken and lesser is the performance. As for $N = 100$, (from the above table) the time taken by an algorithm with $O(2^n)$ is 1.26×10^{30} compared to algorithm with $O(N^2)$ which is 10000. As 1.26×10^{30} is way more than 10000 *i.e.*, 10^5. Thus, you can say that performance of algorithms is inversely proportional to the wall clock time it records for a given input size. Size $O(2^n)$'s $1.26 \times 10^{30} >> O(N^2)$'s 10^5. $O(N^2)$ is better algorithm than $O(2^n)$ algorithm as it clocks lesser time comparatively. It other words, we can say that the algorithm with complexity $O(n^2)$ is better than the algorithm with complexity $O(2^n)$ for solving the same problem.

NOTE

Performance of algorithms is inversely proportional to the wall clock time it records for a given input size. Programs with a bigger *O* run slower than programs with a smaller *O*.

Dominant Term

Big-O notation indicates the growth rate. It is the class of mathematical formula that best describes an algorithm's performance, and is discovered by looking inside the algorithm.

Big-*O* is a function with parameter *N*, where *N* is usually the *size of the input* to the algorithm. More the input size, more impact it can have on the growth rate of the algorithm. However, while describing the growth rate of an algorithm, we simply consider the term, which is going to affect the most on the algorithm's performance. This term is known as the **dominant term.**

For example, if an algorithm depending on the value *n* has performance $an^2 + bn + c$ (for constants *a, b, c*) then we can say that the maximum impact on the algorithm's performance will be of the term an^2. So while describing the complexity of the algorithm, we can ignore the rest of the terms and simply say that the algorithm has performance $O(N^2)$. In other words, for large *N*, the N^2 term dominates. Only the dominant term is included in big-O.

Common Growth Rates

Some Growth rates of algorithms are as shown in following table :

Time complexity		Example
O(1)	*constant*	Adding to the front of a list
O(log N)	*log*	Finding an entry in a sorted array
O(N)	*linear*	Finding an entry in an unsorted array
O(N log N)	*n log n*	Sorting *n* items by 'divide-and-conquer'
O(N^2)	*quadratic*	Shortest path between two nodes in a graph
O(N^3)	*cubic*	Simultaneous linear equations
O(2^N)	*exponential*	The Towers of Hanoi problem

7.3.2 Guidelines for Computing Complexity

After talking about some basic terms, let us now discuss how you can actually compute complexity of an algorithm.

The basic steps for computing complexity of an algorithm are :

1. Select the computational resource you want to measure. Normally we have two options : *time* and *memory*. But other studies can be undertaken like, *e.g.*, network traffic. But here, we are mainly interested in measuring time complexity.
2. Look to the algorithm and pay attention to loops or recursion. Try to see the variables and conditions that make the algorithm work more or less. Sometimes it's one variable, some times several. This is going to be our **size of input.** Remember that with complexity analysis, we are interested in getting a function that relates the size of input with the computational resource.
3. Once we have the *size of input* of the algorithm, try to see if there are different cases inside it, such as when the algorithm gives best performance *i.e.*, takes shortest possible time (*best case*); when the algorithm gives worst performance *i.e.*, takes maximum possible time (*worst case*); and when the algorithm performs in between the two cases *i.e.*, performs better than the worst case but does not give best performance (*average case*).

Calculating Complexity

Five guidelines for finding out the time complexity of a piece of code are: **(Assumption: All** steps take precisely same time to execute.)

- Loops
- Nested loops
- Consecutive statements
- If-then-else statements
- Logarithmic complexity

1. Loops

The running time of a loop is, at most, equal to the running time of the statements inside the loop (including tests) multiplied by the number of iterations. *For instance,* consider the following loop :

Loop executed n times

```
for i in range(n) :
    m = m + 2
```

All the steps in this loop take constant time, say c

So, the total time taken by the above loop to execute itself is :

Total time = c * n = *cn*   i.e.,   O(n)

Time taken by one step

No of steps in the loop

Total Time taken by the loop to execute all its steps

In a loop executing *N* times, the body of the loop gets executed *N* – 1 times, the condition evaluates to *false* and loop terminates without executing the body.

2. Nested loops

To compute complexity of nested loops, analyze inside out. For nested loops, total running time is the product of the sizes of all the loops.

For instance, consider the following code :

```
for i in range(n) :

    for j in range(n):
        K = K + 1
```

Outer loop executes n times

Inner loop executes n times

All these steps take constant time, say c

So the total time taken by the above shown nested loops to execute is :

$$\text{Total time} = c * n * n = cn^2 \qquad \text{i.e.,} \qquad O(n^2)$$

Time taken by one step

No of steps in the outer loop

No of steps in the inner loop

Total Time taken by the nested loops to execute all steps

3. Consecutive statements

To compute the complexity of consecutive statements, simply add the time complexities of each statement. For instance,

```
x = x + 1

for i in range (n):
    m = m + 2

for j in range (n) :

    for k in range (n):
        a = a + 1
```

takes constant time, say c_0

loop1 executes n times

takes constant times say c_1

Outer loop executes n times

inner loop executes n times

takes constant time, say c_2

So the total time taken by the above shown code to execute is :

$$\text{Total time} = c_0 + c_1n + c_2n^2 \qquad \text{i.e.,} \qquad O(n^2)$$

Time taken by statement1

Time taken by loop1

Time taken by nested loop

Considering only the dominant term which is n^2

NOTE

Simple programs can be analyzed by counting the nested loops of the program. A single loop over *n* items yields $f(n) = n$. A loop within a loop yields $f(n) = n^2$. A loop within another loop inside a third loop yields $f(n) = n^3$.

4. If-then-else statements

To compute time complexity of if-then-else statement, we consider worst-case running time, which means, we consider ***the time taken by the test, plus time taken by*** either ***the then part*** or ***the else part*** (*whichever is larger*).

For instance, consider the following code :

```
if  len(list 1) != len(list2) :
    return False
else :
    for i in range(n) :
        if  list1[i] != list2[i]:
            return False
```

Comparison test takes constant time say c_0

The true part's step takes constant time, say c_1

Loop repeats n times

Test condition takes constant time say c_3

Takes constant time say c_2

The else part has a loop (n times) which in total takes times as $(c_2 + c_3) * n$

So the total time taken by the above shown code to execute is :

$$\text{Total time} = c_0 + c_1 + (c_2 + c_3) * n \quad \text{i.e.,} \quad O(N)$$

Time taken by *test*

Time taken by then part

Time taken by else part

Considering only the dominant term which is N.

5. Logarithmic Complexity

The logarithmic complexity means that an algorithms' performance time has logarithmic factor *e.g.* an algorithm is O(log *N*) if it takes a constant time to cut the problem size by a fraction (usually by ½).

An example algorithm having logarithmic complexity is *binary search* algorithm.

But, the best performance isn't always desirable. There can be a tradeoff between :

- Ease of understanding, writing and debugging
- Efficient use of time and space

NOTE

Given a series of for loops that are sequential, the slowest of them determines the asymptotic behavior of the program. Two nested loops followed by a single loop is asymptotically the same as the nested loops alone, because the nested loops dominate the simple loop.

So, maximum performance is not always desirable. However, it is still useful to compare the performance of different algorithms, even if the optimal algorithm may not be adopted.

Some Useful Rules for Algorithm Analysis

To analyze a function analyze each statement in the function. The statement that has the greatest complexity determines the order of the function. Some helpful rules are being given for your reference :

1. Simple statements such as assignment statements are $O(1)$, since their execution time is not dependent on the amount of data. The exception would be assigning one array's elements to another which would be $O(n)$, where n represents the size of the array.
2. The running time for a function call is just the running time for the function's body, plus the time to set up the parameters. Setting up the parameters is $O(1)$, unless a large structure is passed as a value parameter, in which case that structure must be copied into the parameter which is $O(n)$.
3. "In an if/else statement, the test (which is usually $O(1)$) and one of the conditional statements are executed. Since you generally cannot determine the result of the test ahead of time, you should assume the worst case, *i.e.*, the *maximum time* and thus the running time of the if/else will be the sum of the running time of the test and the running time of the worst-case statement."
4. **Addition Rule** "*You can combine sequences of statements by using the addition rule, which states that the running time of a sequence of statements is just the maximum of the running times of each individual statement.*"
5. **Multiplication Rule** This is used to determine the running time for a loop. "*You need to determine the running time for the loop's body and the number of times the loop iterates. Frequently you can just multiply the running time of the body by the number of iterations.*"

7.4 BEST, AVERAGE AND WORST CASE COMPLEXITY

We have discussed in the previous section that we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. However, to really understand how good or bad an algorithm is, we must know how it works over *all* instances, which means :

- the best possible performance of the algorithm (best case)
- the worst possible performance of the algorithm (worst case)
- the average performance of the algorithm (average case)

To understand the notions of the *best, worst,* and *average-case complexity,* one must think about running an algorithm on all possible instances of data that can be fed to it. For the problem of sorting, the set of possible input instances consists of all the possible arrangements of all the possible numbers of keys. We can represent every input instance as a point on a graph, where the *x*-axis is the *size of the problem* (for sorting, the number of items to sort) and the *y*-axis is the *number of steps taken* by the algorithm on this instance. Here we assume, quite reasonably, that it

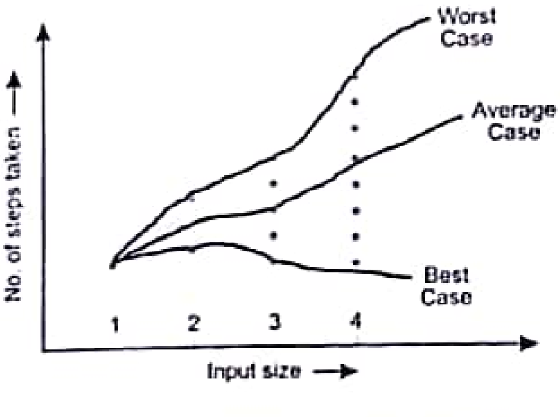


Figure 7.1 Best, worst, and average-case complexity.

doesn't matter what the values of the keys are, just how many of them there are and how they are ordered. It should not take longer to sort 1,000 English names than it does to sort 1,000 French names, for example.

Once we have these points, we can define *three* different functions over them :

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.
- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

In practice, the most useful of these three measures proves to be the *worst-case complexity*, which many people find counterintuitive. *Worst case*

- Provides an upper bound on running time of an algorithm.
- An absolute guarantee that no matter what, this algorithm won't take time more than this time.

Let us understand the efficiency calculation in terms of complexity by taking up two different programs for the same problem.

Example 7.1 *Determine the complexity of a program that checks if a number n is prime.*

Solution.

Option 1 : (Linear approach for prime test)

```
# check whether n is a prime or not using the linear approach
n = int(input("Enter a number :"))   ←——— Constant time c0

# prime flag 0 indicates that n is a prime and the value 1 indicates that it is not a prime

prime_flag = 0   ←——— Constant time c1
                                                      
repeats
nearly
n times
            for i in range(2, n):                 Constant time c2
      ——→       if n%i == 0:   ←——         # if i is a factor of n, n is not a prime
                    prime_flag = 1   ←——— Constant time c3
                    break

if prime_flag :   ←——— Constant time c4 (checks if prime_flag is true)

    print(n, "is a prime number")   ←——— Constant time c5
else:
    print(n, "is not a prime number")   ←——— Constant time c6
```

Total time taken by option 1 (the linear approach)

$$= c_0 + c_1 + (c_2 + c_3)n + (c_4 + c_5 + c_6)$$

$$= C_0 + C_1 n \qquad (C_0 = c_0 + c_1 + c_4 + c_5 + c_6,\ C_1 = c_2 + c_3)$$

$= O(n)$ considering the dominant term, which is n.

Option 2 : ($\sqrt{N}$ approach for prime test)

```
#check whether n is a prime or not using the sqrt n approach
n = int(input("Enter a number :"))   ← Constant time C0

#check whether n is a prime or not using the sqrt n approach
prime_flag = 0   ← Constant time c1
i = 2   ← Constant time c2
                                      Condition takes constant time c3
while (i * i <= n):      # notice that if n does not have a factor less that sqrt n, it will
                         # never have a factor between 2 and n - 1
    if n % i == 0:   ← Constant time c4
        prime_flag = 1   ← Constant time c5
        break   ← Constant time c6
    i = i + 1   ← Constant time c7

if prime_flag :   ← Constant time c8
    print(n, "is a prime number")   ← Constant time c9
else:
    print(n, "is not a prime number")   ← Constant time c10
```

Loop repeats $\sqrt{N}$ times

Total time taken for option 2 ($\sqrt{N}$ approach)

$$= c_0 + c_1 + c_2 + \sqrt{N}(c_3 + c_4 + c_5 + c_6 + c_7) + c_8 + c_9 + c_{10}$$

$$= C_1 + \sqrt{N}C_2$$

$$= C_1 + C_2\sqrt{N} \qquad [C_1 = c_0 + c_1 + c_2 + c_8 + c_4 + c_{10},\ C_2 = c_3 + c_4 + c_5 + c_6 + c_7 + c_8]$$

$= O(\sqrt{N})$ Considering the dominant term.

Comparing complexity of option 1 and option 2, we can say that **option 2 is better** as $\sqrt{N} < N$.

Example 7.2 *Determine the complexity of a program thats searches for an element in an array.*

Solution.

Option 1 : (Linear search)

```
def linearSearch(arr, x) :
    i = 0          ]
    n = len(arr)   ]  ← Constant time c0
```

Loop repeats maximum n times

```
while( i<n  and  x != arr[i] ):     ← Constant time c2
                                     ← Constant time c1
    i = i+1                         ← Constant time c3

if i<n :                            ← Constant time c4

    return i        # Element x found at index i    ← Constant time c5

else:
    return None     # element x not found           ← Constant time c6
```

Total time taken by option 1 (the linear search)

Total time taken $= c_0 + n(c_1 + c_2 + c_3) + c_4 + c_5 + c_6$

$= c_0 + c_4 + c_5 + c_6 + n(c_1 + c_2 + c_3)$

$= C_0 + nC_1 \qquad [C_0 = c_0 + c_4 + c_5 + c_6,\ C_1 = c_1 + c_2 + c_3]$

$= O(n)$ considering the dominant term.

Option 2 : (binary search)

Loop repeats max $\log_2 N$ times because every time segment becomes half in size (explanation below)

```
def binsearch(ar, key) :                              ...... Constant time c0
    low = 0          # initialy low end is at 0
    high = len(ar) - 1    # initialy high end is at size - 1
                                                      ...... Constant time c1
    while  low <= high :         ← Constant time c2
        mid = int((low + high) / 2)    ← Constant time c3
        if key == ar[mid] :      ← Constant time c4
            return mid           ← Constant time c5
        elif key <ar[mid] :      ← Constant time c6
            high = mid - 1       ← Constant time c7
        else:
            low = mid + 1        ← Constant time c8
    else :                       # loop's else; reaches here when key not matched
        return None              ← Constant time c9
```

Total time taken by option 2 (the binary search)

Total time taken $= c_0 + c_1 + \log_2 n\,(c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8) + c_9$

$= C_0 + \log_2 nC_1 \qquad (c_0 + c_1 + c_9 = c_0 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 = C_1)$

$= O(\log_2 n)$ considering the dominant term.

How Many Times above While Loop Executes

The while loop of above given binary search algorithm executes how many times ? To determine this you need to answer the question :

> **How many times can you divide *N* by 2 until you have 1 ?** This is because in binary search, the search segment's size begins with *N* and reduces by half in every iteration and stops when the search segment's size reduces to 1 element.

So if loop repeats *k* times (until the segment reduces to size 1), then in a formula this would be :

$$N/2^k = 1$$
$$2^k = N$$

Taking the $\log_2$ on both sides :

$$\log_2(2^k) = \log_2 N$$
$$k * \log_2(2) = \log_2 N$$
$$k * 1 = \log_2 N$$
$$k = \log_2 N$$

This means you can divide log *N times* until you have everything divided. That means the above loop repeats at max log *N* times.

Comparing complexity of *option 1* and *option 2*, we can say that option 2 is better as $\log_2 N < N$.

With this, we have come to the end of our chapter. Let us quickly revise what we have learnt so far.

Let us Revise

- *An algorithm is a sufficiently precise method or procedure for accomplishing a specific task, which can be programmed on computer.*
- *Complexity refers to the measure of the performance of an algorithm.*
- *Complexity can be related to time (temporal complexity) or to space (space complexity).*
- *Big-O notation is used to depict an algorithm's growth rate i.e., change in algorithm performance when its input size grows.*
- *Dominant term is the one which affects the most, an algorithm's performance.*
- *Only the dominant term is included in Big-O notation.*
- *The Worst case complexity provides an upper-bound on running time.*
- *Average-Case complexity provides expected running time.*
- *Best-Case complexity provides the time of optimal performance.*

Solved Problems

1. *Define Big 'O' notation. State the two factors which determine the complexity of an algorithm.*

Solution. Big O notation is a particular tool for assessing algorithm efficiency. It describes the performance or complexity of an algorithm denoted *via* Big O, *e.g.,* O(1), O(*N*), O(*N* log *N*) etc.

Performance of an algorithm depends on many *internal* and *external* factors.

Internal Factors specify algorithm's efficiency in terms of :

- *Time* required to run
- *Space* (or Memory) required to run.

External Factors affect the algorithm's performance. These include :

- *Size of the input* to the algorithm
- *Speed of the computer* on which it is run
- *Quality of the compiler.*

2. *Consider the following three algorithms for determining whether anyone in the room has the same birthday as you.*

Algorithm 1 : You say your birthday, and ask whether anyone in the room has the same birthday. If anyone does have the same birthday, they answer yes.

Algorithm 2 : You tell the second person your birthday and ask whether they have the same birthday ; and so forth, for each person in the room.

Algorithm 3 : You only ask questions of person 1, who only asks questions of person 2, who only asks questions of person 3, etc. You tell person 1 your birthday, and ask if they have the same birthday; if they say no, you ask them to find out about person 2. Person 1 asks person 2 and tells you the answer. If it is no, you ask person 1 to find out about person 3. Person 1 asks person 2 to find out about person 3, etc.

(*i*) *What is the factor that can affect the number of questions asked (the "problem size") ?*

(*ii*) *In the worst case, how many questions will be asked for each of the three algorithms ?*

(*iii*) *For each algorithm, say whether it is constant, linear, or quadratic in the problem size in the worst case.*

Solution.

(*i*) The problem size is the number of people in the room.

(*ii*) Assume there are N people in the room. In algorithm 1 you always ask 1 question. In algorithm 2, the worst case is if no one has your birthday. Here you have to ask every person to figure this out. This means you have to ask N questions. In algorithm 3, the worst case is the same as algorithm 2. The number of questions is 1 + 2 + 3 + ... + N – 1 + N. This sum is N(N+1)/2, using formula for sum of first N natural numbers.

(*iii*) Given the number of questions you can see that algorithm 1 is constant time, algorithm 2 is linear time, and algorithm 3 is quadratic time in the problem size.

3. *Distinguish between worst-case and best case complexity of an algorithm.*

Solution. The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.

The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.

4. *(i) Give the meaning of the following common expression in Big O notation :*

$$O(N) ; \quad O(N^2)$$

(ii) List any two cases to analyse algorithm complexities.

Solution. (*i*) **O(N)** describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set.

(*ii*) The efficiency of an algorithm is determined through algorithm complexity. Two common cases to analyse algorithm complexities are :

- Sorting algorithms
- Searching Algorithms

5. *What is the worst case complexity of the following code fragment ?*

```
for i in range(n) :
    a = i + (i + 1)
    print(a)
for j in range(m):
    b = i * (i + 1)
    print(b)
```

Solution. Time taken in the execution of given code :

loop1 repeats n times

```
for i in range(n) :
    a = i + (i + 1)
    print(a)
```

takes constant time c_0

takes constant time c_1

Loop 2 repeats m times

```
for j in range(m) :
    b = i * (i + 1)
    print(b)
```

takes constant time c_2

takes constant time c_3

$$
\begin{aligned}
\text{Total time taken} &= n * (c_0 + c_1) + m * (c_2 + c_3) \\
&= nC_1 + mC_2 \qquad [C_1 = c_0 + c_1,\ \ C_2 = c_2 + c_3] \\
&= O(n + m)
\end{aligned}
$$

6. *What is the worst-case complexity of the following code fragment having a nested loop followed by a single loop :*

```
for i in range(n) :
    for j in range(n):
        sequence of statements
for k in range(n):
        sequence of statements
```

Solution. Time taken in the execution of given code :

Outer loop repeats n times

Inner loop repeats n times

```
for i in range(n) :
    for j in range(n) :
        sequence of statements
```

Take up constant time say C_0

Loop 3 repeats n times

```
for k in range(n) :
    sequence of statements
```

Take up constant time say C_1

$$
\begin{aligned}
\text{Total time taken} &= n * (n + C_0) + nC_1 \\
&= C_0 n^2 + C_1 n \\
&= O(n^2) \text{ considering the dominant term only.}
\end{aligned}
$$

7. *(a) What is the worst case complexity of the following code fragment ?*

```
for x in range(a):
    statements

for y in range(b):
    for z in range(c):
        statements
```

(b) How would the complexity change if all the three loops repeated N times instead of a, b and c times respectively ?

Solution. Assuming that each statement has $O(1)$ complexity, then : *(a)* $O(a + bc)$ *(b)* $O(N^2)$

8. *Compute the complexity of following algorithm (it is not written in any programming language) :*

```
# Finding the largest and second largest elements of a list
# 1.  Find the largest element by scanning from left to right
# 2.  Go back through the list and find the second largest element.

big1(array a of  integers) :
    max1 ← $a_1$
    for i ← 2 to n do
        if $a_i$ > max1 then max1 ← $a_i$
            if max1 = $a_1$ then max2 ← $a_2$
            else max2 ← $a_1$
            for i ← 2 to n do
                if ($a_i$ > max2 and $a_i \neq$ max1) then
                    max2 ← $a_i$
    return(max1, max2)
```

Solution. Analysing the given code.

```
max1 ← a1                                    1 assignment
for i ← 2 to n do                            n - 1 iterations
    if ai > max1 then max1 ← $a_i$               1 assignment and one comparison
        if max1 = $a_1$ then max2 ← $a_2$        1 assignment and one comparison
        else max2 ← $a_1$
        for i ← 2 to n do                    n - 1 iterations
            if ($a_1$ > max2 and $a_i \neq$ max1) then   2 comparisons and 1 assignment each time
                max2 ← $a_i$                     Total for loop 2(n - 1) comparisons and
                                             at most n - 1 assignments
return(max1, max2)                           O(1)–constant complexity
```

This gives us the following totals :

Total number of comparisons

n – 1	first for loop
1	if statement
2n – 2	second for loop
3n - 2	

Number of assignments

1 + n - 1 + 1 + n - 1 = 2n

For the first for loop, we have $\sum_{i=2}^{n} 1 = n-1$ and in the second we have $\sum_{i=2}^{n} 2 = 2\,(n-1)$

That is $n - 1 + 2(n - 1) = 3(n - 1) = O\,(n)$.

9. *(i) Reorder the following efficiencies from the smallest to the largest :*

(*a*) 2^n (*b*) n! (*c*) n^5 (*d*) 10,000 (*e*) $n\log_2(n)$

(ii) Reorder the following efficiencies from the smallest to the largest :

(*a*) $n\log_2(n)$ (*b*) $n + n^2 + n^3$ (*c*) 2^4 (*d*) $n^{0.5}$

Solution.

(*a*) $10{,}000 < n\ \log_2(n) < n^5 < 2^n < n!$

(*b*) $2^4 < n^{0.5} < n\log_2(n) < n + n^2 + n^3$

10. *Calculate the run-time efficiency of the following program segment :*

```
i = 1
while i <= n :
    j = 1
    while j <= n :
        k = 1
        while k <= n :
            print(i, j, k)
            k = k + 1
        j = j + 1
    i = i + 1
```

Solution. There are *three* nested loops, each loop is executed in *n* times, so run-time efficiency is

$$n \times n \times n = n^3$$

11. *Calculate the run time efficiency of the following program segments.*

(*a*) *If the function* **doIt** *has an efficiency factor of 5n.*

```
i = 1
while i <= n :
    doIt(...)
    i = i + 1
```

(*b*) *If the efficiency of the algorithm* **doIt** *can be expressed as* $O(n) = n^2$.

```
i = 1
while i <= n :
    j = 1
    while j < n :
        doIt(...)
        j = j + 1
    i = i + 1
```

(*c*) *If the efficiency of the algorithm* **doIt** *can be expressed as* $O(n) = n^2$.

```
i = 1
while i < n :
    doIt(...)
    i = i * 2
```

Solution.

(*a*) Function **doIt** is executed in *n* times, so the run-time efficiency is $5n^2$ *i.e.*, $O(n^2)$.

(*b*) The iteration of variable *i* is executed in *n* times while the iteration of variable *j* is executed in *n-1* times. Therefore, the run-time efficiency is

$$n \times (n-1) \times n^2 = n^4 - n^3 = O(n^4)$$

(*c*) The function *doIt* is executed inside a logarithmic loop (notice *i* is updated as $i = i*2$ *i.e.*, *i* doubles every time), with size *n*, so the run time efficiency of the program is :

$$\log_2(n) \times n^2 = O(n^2 \log_2 n)$$

12. *(a) Given that the efficiency of an algorithm is* $5n^2$*, if a step in this algorithm takes 1 nanosecond* (10^{-9})*, how long does it take the algorithm to process an input of size 1000 ?*

(b) Given that the efficiency of an algorithm is n^3*, if a step in this algorithm takes 1 nanosecond* (10^{-9})*, how long does it take the algorithm to process an input of size 1000 ?*

(c) Given that the efficiency of an algorithm is $5n\log_2(n)$ *if a step in this algorithm takes 1 nanosecond* (10^{-9})*, how long does it take the algorithm to process an input of size 1000 ?*

Solution.

(*a*) Given size is $n = 1000$, hence the time taken is : $5 \times 1000^2 \times 10^{-9} = 5\,\text{ms}$

(*b*) Given size is $n = 1000$, hence the time taken is : $1000^3 \times 10^{-9} = 1\,\text{s}$

(*c*) Given size is $n = 1000$, hence the time taken is : $5 \times 1000 \times \log_2(1000) \times 10^{-9} \approx \mu\text{s}$.

[Recall that 10^{-3} is *milli* (m) ; 10^{-6} is *micro* (μ) ; 10^{-9} is *nano* (n) ; 10^{-12} is *pico* (p)]

GLOSSARY

Algorithm	Sufficiency finite and precise method for accomplishing a task.
Average-Case Complexity	Expected running time of an algorithm.
Best-Case Complexity	Running time of an algorithm in case of optimal performance.
Complexity	Measure of an algorithm's performance.
Worst-Case Complexity	Upper bound on running time of an algorithm.

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is an algorithm? What do you understand by algorithm performance ?
2. What is computational complexity ?
3. Which factors affect an algorithm's performance ?
4. What are different types of complexities that are considered ?
5. What do you understand by Big-O notation? What is its significance ?
6. What do you understand by best-case, worst-case and average case complexities ? When are they considered ?
7. Determine the big-O notation for the following calculated complexity :

   (*a*) $5n^{5/2} + n^{2/5}$ (*b*) $6\log_2(n) + 9n$ (*c*) $3n^4 + n\log_2(n)$ (*d*) $5n^2 + n^{3/2}$
8. Reorder the following efficiencies from the smallest to the largest :

   (*a*) 2^n (*b*) $n!$ (*c*) n^5 (*d*) 10,000 (*e*) $n\log_2(n)$
9. Reorder the following efficiencies from the smallest to the largest :

   (*a*) $n\log_2(n)$ (*b*) $n + n^2 + n^3$ (*c*) 2^4 (*d*) $n^{0.5}$
10. Determine the big-O notation for the following :

    (*a*) $5n^{5/2} + n^{2/5}$ (*b*) $6\log_2(n) + 9n$ (*c*) $3n^4 + n\log_2(n)$ (*d*) $5n^2 + n^{3/2}$

8

Data Visualization using Pyplot

In This Chapter

8.1 WHAT IS DATA VISUALIZATION ?

Modern age has become very fast and competitive with the pace of the development in all fields of life. The reach of all businesses and facilities have gone global because of the modern technologies. This has made things easier and competitive but at the same time data have increased multifold. In fact, data have grown so big that a specific term has been coined, *'big data'*.

As you all are aware that the role of data is to empower decision makers to make decisions based on *facts, trends* and *statistical numbers* available in the form of data. But since data is so huge these days that the decision makers must be able to sift through the unnecessary, unwanted data and get the right information presented in compact and apt way, so that they can make the best decisions. For this purpose, *data visualization techniques* have gained popularity. Confused? What is this *data visualization*? Well, read on.

Data visualization basically refers to the graphical or visual representation of information and data using visual elements like *charts, graphs,* and *maps* etc. Data visualization is immensely useful in decision making. Data visualization unveils *patterns, trends, outliers, correlations* etc. in the data, and thereby helps decision makers understand the meaning of data to drive business decisions.

For instance, a company has to decide about, *'which advertising solution should it invest in to promote its new product?' Data visualization* is here to help – just pull out data of company's previous campaigns, and plot a bar chart comparing the performance of different platforms and bingo! the right decision is right in front.

This chapter is dedicated to get you started with data visualization using Python's useful tool **PyPlot** where you shall learn to represent data visually in various forms such as *line, bar* and *pie charts* etc. So, let's get started.

DATA VISUALIZATION

Data visualization basically refers to the graphical or visual representation of information and data using visual elements like *charts, graphs,* and *maps* etc.

8.2 USING PYPLOT OF MATPLOTLIB LIBRARY

For data visualization in Python, the Matplotlib library's Pyplot interface is used.

The **matplotlib** is a Python library that provides many interfaces and functionality for 2D-graphics similar to MATLAB[1]'s in various forms. In short, you can call *matplotlib* as a high quality plotting library of Python. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats. The **matplotlib** library offers many different named collections of methods; **PyPlot** is one such interface.

NOTE

PyPlot is a collection of methods within matplotlib library (of Python) which allow user to construct 2D plots easily and interactively.

PyPlot is a collection of methods within **matplotlib** which allow user to construct 2D plots easily and interactively. **PyPlot** essentially reproduces plotting functions and behavior of MATLAB.

8.2.1 Installing and Importing *matplotlib*

You shall learn to plot data using **PyPlot** but before that make sure that **matplotlib** library is installed on your computer.

- If you have installed Python using Anaconda, then **matplotlib** library is already installed on your computers. You can check it yourself by starting **Anaconda Navigator.** From Navigator window, click **Environments** and then scroll down in the alphabetically sorted list of installed packages on your computer. You will find **matplotlib** there.

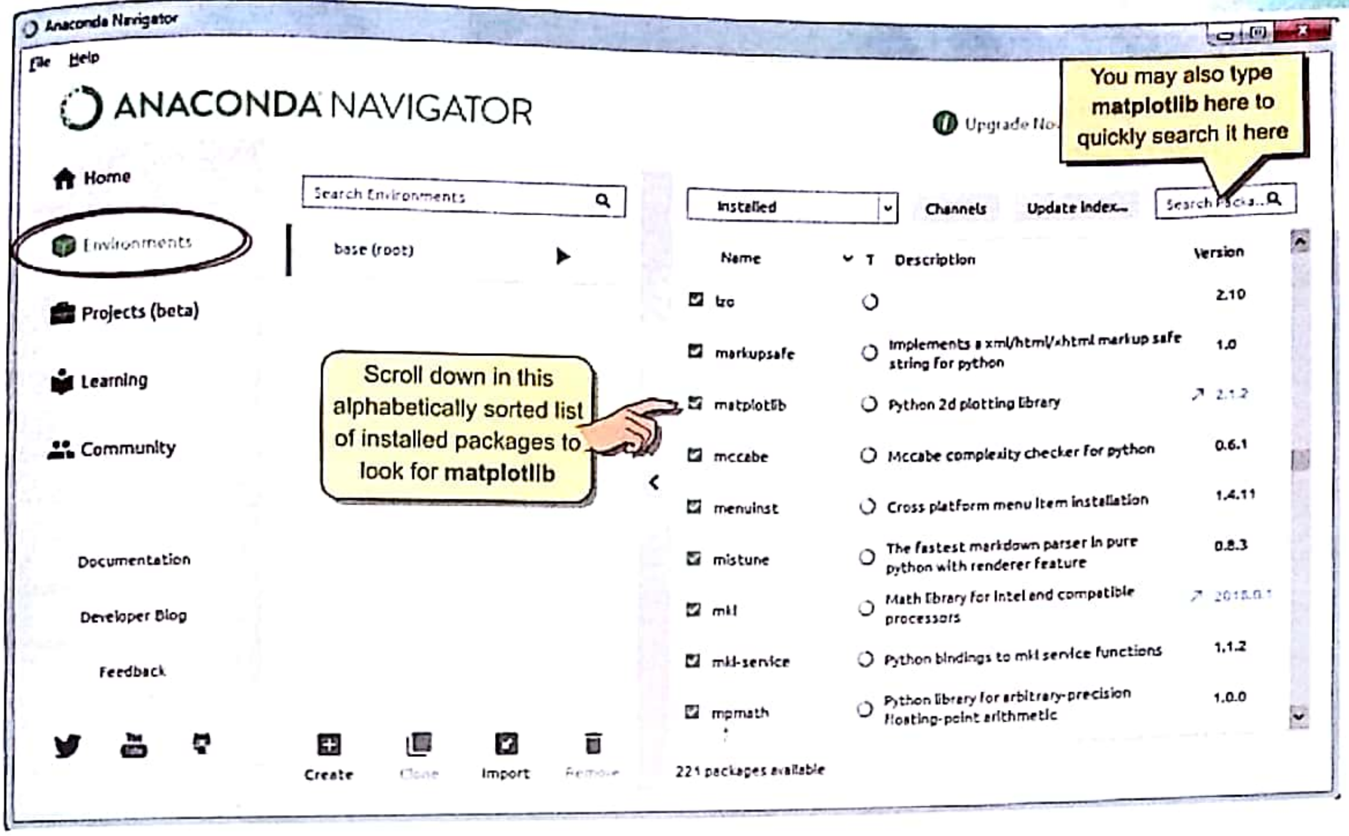
NOTE

The matplotlib library is preinstalled with Anaconda distribution; you need not install it separately if you have installed Python using Anaconda distribution.

- If you have installed Python using standard official distribution, you may need to install **matplotlib** separately as explained here.
  - First you will need to download wheel package of **matplotlib** as per Python's version installed and the platform (*MacOs or Windows or Linux*) on which it is installed. For this go to the link **https://pypi.org/project/matplotlib/#files** and download required wheel package as per your platform.
  - Next you need to install it by giving following commands :

```
python -m pip install -U pip
python -m pip install -U matplotlib
```

[1] **MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.**



You may also type **matplotlib** here to quickly search it here

Scroll down in this alphabetically sorted list of installed packages to look for **matplotlib**

	Name	T	Description	Version
☑	lzo	○		2.10
☑	markupsafe	○	Implements a xml/html/xhtml markup safe string for python	1.0
☑	matplotlib	○	Python 2d plotting library	↗ 2.1.2
☑	mccabe	○	Mccabe complexity checker for python	0.6.1
☑	menuinst	○	Cross platform menu item installation	1.4.11
☑	mistune	○	The fastest markdown parser in pure python with renderer feature	0.8.3
☑	mkl	○	Math library for Intel and compatible processors	↗ 2018.0.1
☑	mkl-service	○	Python bindings to mkl service functions	1.1.2
☑	mpmath	○	Python library for arbitrary-precision floating-point arithmetic	1.0.0

221 packages available

Importing PyPlot

In order to use **pyplot** on your computers for data visualization, you need to first import it in your Python environment by issuing one of the following commands :

```
import matplotlib.pyplot
```

This would require you to refer to every command of ***pyplot*** *as* ***matplotlib.pyplot.<command>***

```
import matplotlib.pyplot as pl
```

With this, you can refer to every command of ***pyplot*** *as* ***pl.<command>***
as you have given an alias name to ***matplotlib.pyplot*** *as* ***pl***

With the *first command* above, you will need to issue every **pyplot** command as per following syntax :

```
matplotlib.pyplot.<command>
```

But with the *second command* above, you have provided **pl** as the shorthand for **matplotlib.pyplot** and thus now you can invoke PyPlot's methods as this :

```
pl.plot(X,Y)
```

You can choose any legal identifier in place of **pl** above.

8.2.2 Working with PyPlot Methods

The PyPlot interface provides many methods for 2D plotting of data. The *matplotlib's* Pyplot interface lets one plot data in multiple ways such as *line chart, bar chart, pie chart, scatter chart* etc. Let us talk about how you can plot sequence or array of data using PyPlot methods.

But before we proceed with it, I shall recommend you to have an idea of using NumPy library and some of its functions, for these two reasons, specifically :

(i) NumPy offers some other useful functions to create arrays of data, which prove useful while plotting data.

(ii) NumPy offers many useful functions and routines.

(iii) NumPy arrays which are like lists, support ***vectorized operations***, *i.e.*, if you apply a function, it is performed on every item (element by element) in the array unlike lists *e.g.*,

> *Let's suppose you want to add the number 2 to every item in the list or array and if you give an expression such as* **List +2** *(where List is a Python list), Python will give error but not with NumPy array. For a NumPy array, it will simply add 2 to each element of the NumPy array.*
>
> *This feature may prove very useful in plotting, e.g., if you have a NumPy array namely* **X** *(say., ([1,2,3, 4])), and you want to compute* **sin( )** *for each of the values of this array for plotting purpose, you just need to write is* ***numpy.sin***(X)

For these reasons, I recommend that you should know how to create NumPy arrays and use NumPy functions on them. Following section briefly introduces how to create and use NumPy arrays.

8.2.2A Using NumPy Arrays

NumPy ('*Numerical Python*' or '*Numeric Python*', pronounced as *Num Pie*) is an open source module of Python that offers functions and routines for fast mathematical computation on arrays and matrices. In order to use NumPy, you must import in your module by using a statement like :

```
import numpy as np
```

You can use any identifier name in place of np but np has been a preferred choice, generally

The above import statement has given **np** as an alias name for NumPy module. Once imported with as <name>, you use both names *i.e.*, **numpy** or **np** for functions *e.g.*, **numpy.array( )** is same as **np.array( )**.

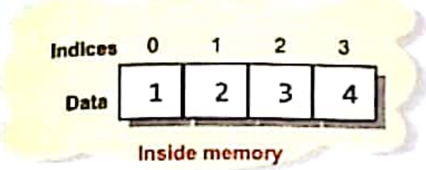
Array in general refers to a named group of homogeneous (of same type) elements. For example, if you store the similar details of all sections together such as if you store *number of students* in each section of class X in a school, *e.g.*, ***Students*** array containing 5 entries as [34, 37, 36, 41, 40] then ***Students*** is an array that represents number of students in each section of class X. Like lists, you can access individual elements by giving *index* with array name *e.g.*, **Students[1]** will give details about 2nd section, **Students[3]** will give you details about 4th section and so on.

NOTE

Please note that if you have installed Python through Anaconda installer then *NumPy, SciPy, Pandas, Matplotlib* etc. are by default installed with Python, otherwise you need to install these separately through pip installer of Python.

A NumPy array is simply a grid that contains values of the same/homogeneous type. You can think of a NumPy array like a *Python list* having all elements of similar types (but NumPy arrays are different from Python lists in functionality, which will be clear to you in coming lines).

Consider following code :



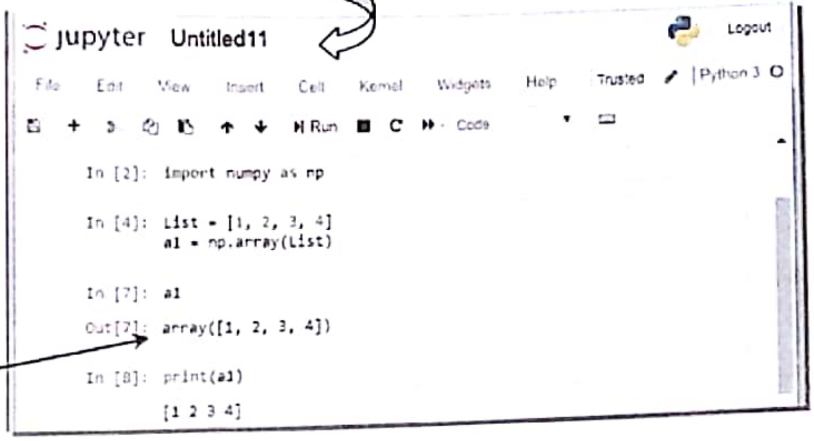
Code

```
import numpy as np
List = [1, 2, 3, 4]
a1 = np.array(List)
print(a1)
```

Now, you can use numpy functions either as ***numPy.functionName*** *or* ***np.functionName*** *.* *e.g., numpy.array( ) or np.array( )*

It will create a NumPy array from the given list

```
In [13]: import numpy as np

In [14]: List = [1, 2, 3, 4]

In [15]: a1 = np.array(List)

In [16]: a1
Out[16]: array([1, 2, 3, 4])

In [17]: print(a1)
[1 2 3 4]
```



```
In [2]: import numpy as np

In [4]: List = [1, 2, 3, 4]
        a1 = np.array(List)

In [7]: a1
Out[7]: array([1, 2, 3, 4])

In [8]: print(a1)
        [1 2 3 4]
```

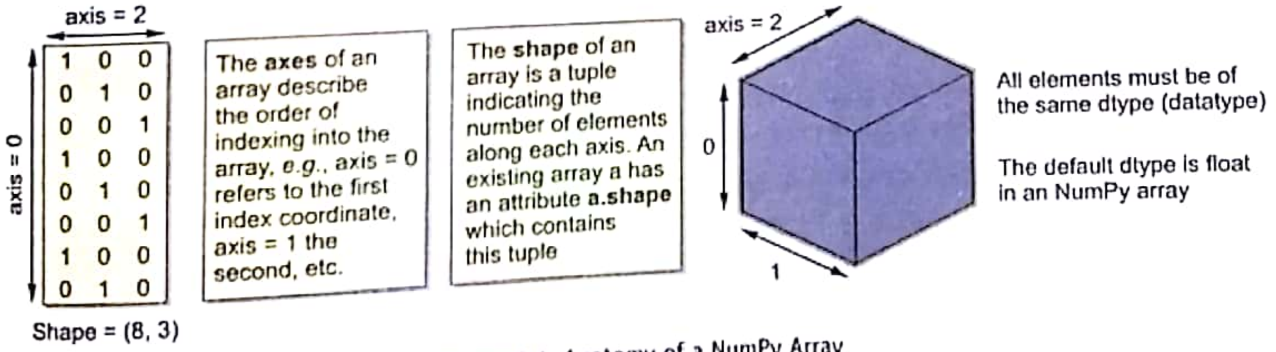
Notice how it displays array and how it prints it with print( )

Individual elements of above array can be accessed just like you access a list's, *i.e.*,

`<array-name>[<index>]`

That is, ***a1[0]*** will give you ***1***, ***a1[1]*** will give you **2**, ...***a1[3]*** will give you **4**.

Following Fig. 8.1 illustrates the basic terms associated with a NumPy array :



```
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
```

The **axes** of an array describe the order of indexing into the array, *e.g.*, axis = 0 refers to the first index coordinate, axis = 1 the second, etc.

The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array a has an attribute **a.shape** which contains this tuple

All elements must be of the same dtype (datatype)

The default dtype is float in an NumPy array

Figure 8.1 Anatomy of a NumPy Array

You can check *type* of a NumPy array in the same way you check the type of objects in Python, *i.e.*, using type( ). Also, there are associated attributes : (*i*) shape that returns dimensions of a NumPy array and (*ii*) itemsize that returns the length of each element of array in bytes. (see below)

```
In [25]: type (a1), type(a2)
Out[25]: (numpy.ndarray, numpy.ndarray)

In [26]: a1.shape
Out[26]: (4,)

In [27]: a2.shape
Out[27]: (2, 3)

In [28]: a2.itemsize
Out[28]: 4
```

See the type it returned for NumPy arrays you created

The shape attribute gives the dimensions of a NumPy array. For 1D array, see it returned as (4,) i.e., 4 elements , single index only For 2D arrays, it returned both dimensions.

The itemsize attribute returns the length of each element of array in bytes

```
In [11]: a1.dtype
Out[11]: dtype('int32')
```

You can check data type of a NumPy array's elements using **<arrayname>.dtype** *e.g.*, (see on the right)

Some common NumPy data types (used as numpy.<datatype>) are being given on Table 8.1.

NOTE

Numpy arrays are also called **ndarrays.**

Table 8.1 *NumPy Data Types*

S.No.	Data Type	Description	Size
1.	`numpy.int8`	Stores signed integers in range –128 to 127	1 byte
2.	`numpy.int16`	Stores signed integers in range –32768 to 32767	2 bytes
3.	`numpy.int32`	Stores signed integers in range -2^{16} to $2^{16}-1$	4 bytes
4.	`numpy.int64`	Stores signed integers in range -2^{32} to $2^{32}-1$	8 bytes
5.	`numpy.float_`	Default type to store floating point (*float64*)	8 bytes
6.	`numpy.float16`	Stores **half precision floating point values** (*5 bits* exponent, *10 bit* mantissa)	2 bytes
7.	`numpy.float32`	Stores **single precision floating point values** (*8 bits* exponent, *23 bit* mantissa)	4 bytes
8.	`numpy.float64`	Stores **double precision floating point values** (*11 bits* exponent, *52 bit* mantissa)	8 bytes

Some useful ways of creating NumPy arrays

(*i*) Creating arrays with a numerical range using arange( ). The arange( ) function is similar to Python's **range( )** function but it returns an *ndarray* in place of Python list returned by **range( )** of Python. In other words, the arange( ) creates a NumPy array with evenly spaced values within a specified numerical range. It is used as :

```
<arrayname> = numpy.arange([start,] stop [, step ] [, dtype ])
```

- ✧ The **start**, **stop** and **step** attribute provide the values for *starting value, stopping value* and *step value* for a numerical range. *Start* and *step values* are optional. When only *stop value* is given, the numerical range is generated from *zero* to *stop value* with *step* 1.
- ✧ The **dtype** specifies the datatype for the NumPy array.

Consider the following statements :

```
In [73]: arr5 = np.arange(7)

In [74]: arr5
Out[74]: array([0, 1, 2, 3, 4, 5, 6])

In [75]: arr5.dtype
Out[75]: dtype('int32')
```

Start value *Stop value* *Step value*

```
In [77]: arr6 = np.arange(1,7, 2, np.float32)

In [78]: arr6
Out[78]: array([1., 3., 5.], dtype=float32)
```

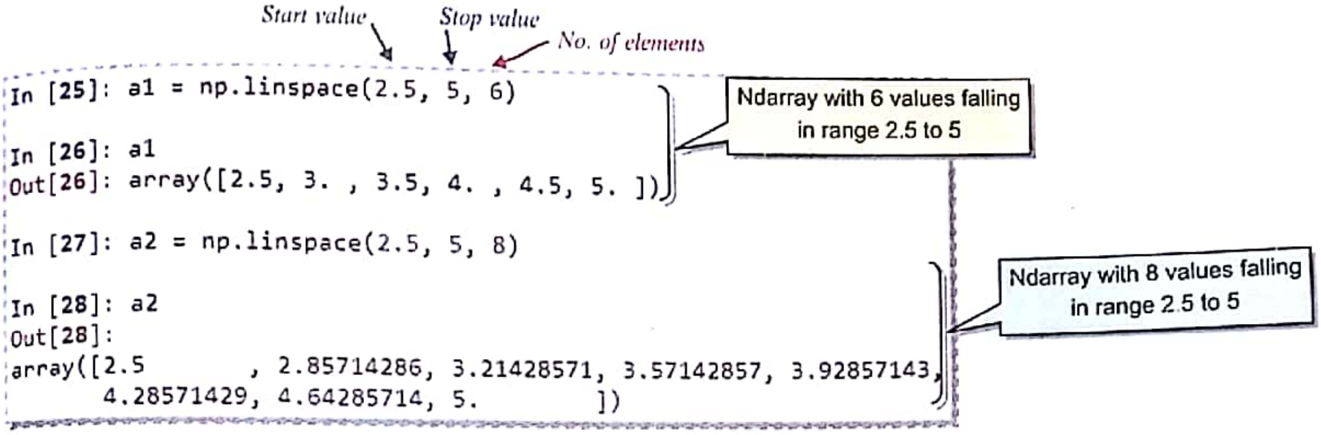
(*ii*) Creating arrays with a numerical range using linspace( ). Sometimes, you need evenly spaced elements between two given limits. For this purpose, NumPy provides linspace( ) function to which you provide, *the start value, the end value* and number of elements to be generated for the *ndarray*. The syntax for using linspace( ) function is :

```
<arrayname> = numpy.linspace(<start>, <stop>, <number of values to be generated>)
```

For example, following code will create an ndarray with 6 values falling in the range 2 to 3 :

```
Arr1 = np.linspace(2, 3, 6)
```

Consider some more examples :



```
In [25]: a1 = np.linspace(2.5, 5, 6)

In [26]: a1
Out[26]: array([2.5, 3. , 3.5, 4. , 4.5, 5. ])

In [27]: a2 = np.linspace(2.5, 5, 8)

In [28]: a2
Out[28]:
array([2.5       , 2.85714286, 3.21428571, 3.57142857, 3.92857143,
       4.28571429, 4.64285714, 5.        ])
```

Ndarray with 6 values falling in range 2.5 to 5

Ndarray with 8 values falling in range 2.5 to 5

Some useful NumPy functions

We are giving some NumPy functions (without much details) that may prove useful while plotting data.

Trigonometric functions	
`sin(x, /[, out])`	Trigonometric sine, element-wise.
`cos(x, /[, out])`	Cosine element-wise.
`tan(x, /[, out])`	Compute tangent element-wise.
`arcsin(x, /[, out])`	Inverse sine, element-wise.
`arccos(x, /[, out])`	Trigonometric inverse cosine, element-wise.
`arctan(x, /[, out])`	Trigonometric inverse tangent, element-wise.
Hyperbolic functions	
`sinh(x, /[, out])`	Hyperbolic sine, element-wise.
`cosh(x, /[, out])`	Hyperbolic cosine, element-wise.
`tanh(x, /[, out])`	Compute hyperbolic tangent element-wise.
`arcsinh(x, /[, out])`	Inverse hyperbolic sine element-wise.
`arccosh(x, /[, out])`	Inverse hyperbolic cosine, element-wise.
`arctanh(x, /[, out])`	Inverse hyperbolic tangent element-wise.
Rounding Functions	
`round_(a[, decimals, out])`	Round an array to the given number of *decimals*.
`floor(x, /[, out])`	Return the floor of the input, element-wise.
`ceil(x, /[, out])`	Return the ceiling of the input, element-wise.
`trunc(x, /[, out])`	Return the truncated value of the input, element-wise.
Exponents and logarithms	
`exp(x, /[, out])`	Calculate the exponential of all elements in the input array.
`exp2(x, /[, out])`	Calculate 2***p* for all *p* in the input array.
`log(x, /[, out])`	Natural logarithm, element-wise.
`log10(x, /[, out])`	Return the base 10 logarithm of the input array, element-wise.
`log2(x, /[, out])`	Base-2 logarithm of x.
`log1p(x, /[, out ...])`	Return the natural logarithm of one plus the input array, element-wise.

Parameters used in the above functions :

x : *array_like sequence*

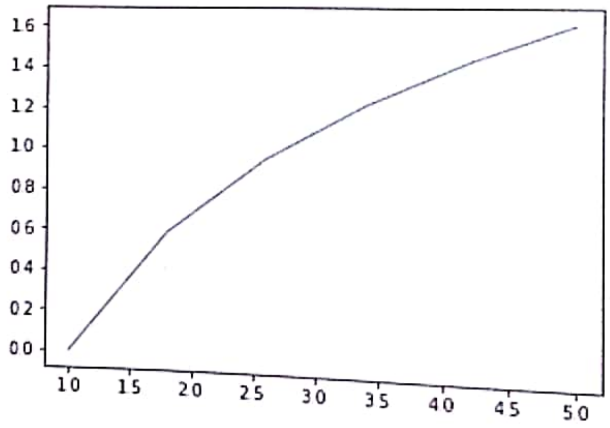
out : *ndarray, None, or tuple of ndarray and None, optional argument*

it is a location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned.

See below how useful the numpy functions are while plotting. Although you shall learn about all components of plotting in the coming section, yet following code lines make you appreciate the use of numpy arrays and functions. (Don't worry if things are not clear right now, they will be, soon. I promise ☺)

```
In [39]: import numpy as np

In [40]: import matplotlib.pyplot as pl

In [41]: x = np.linspace(1,5, 6)

In [42]: y = np.log(x)

In [43]: pl.plot(x,y)
Out[43]: [<matplotlib.lines.Line2D at 0x8ef79d0>]
```

Generate an array in range 1..5, having 6 elements

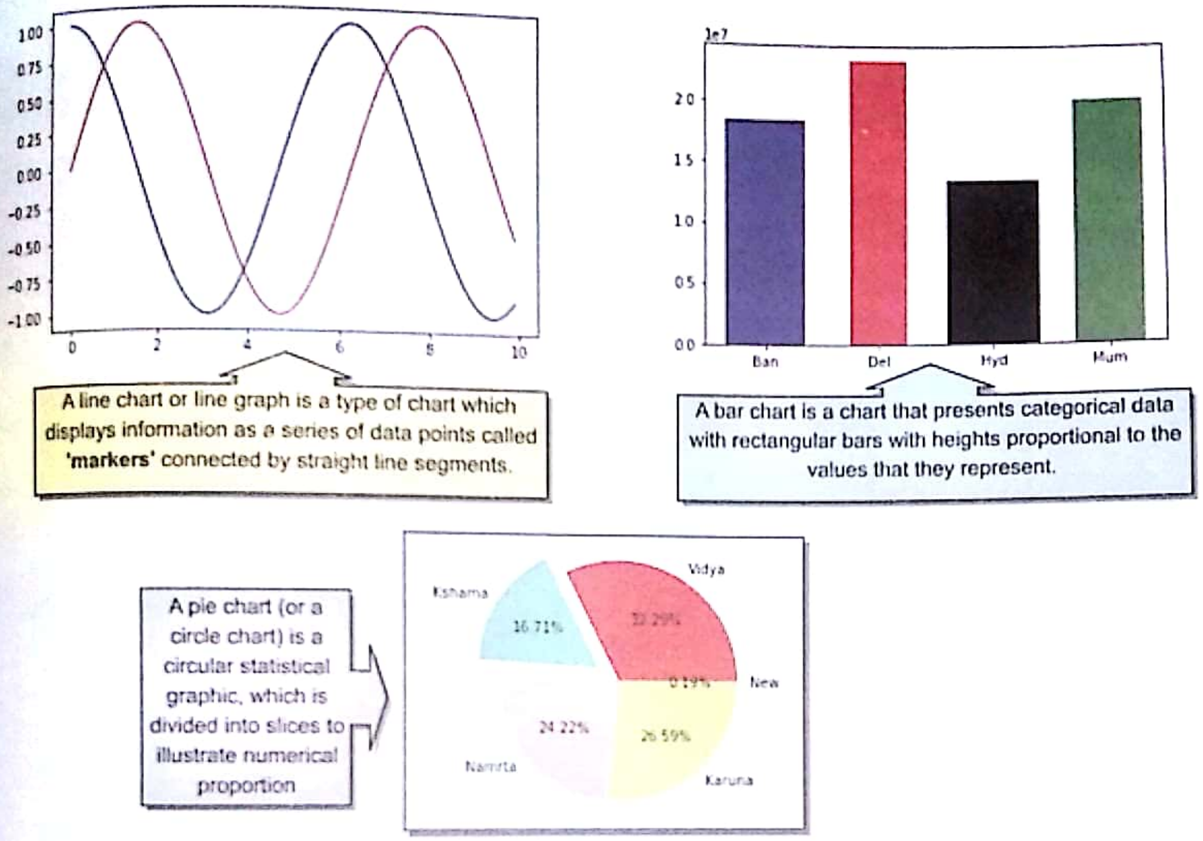


8.2.2B Basics of Simple Plotting

Data visualization essentially means graphical representation of compiled data. Thus, graphs and charts are very effective tools for data visualization. You can create many different types of graphs and charts using PyPlot but we shall stick to just a few of them as per your syllabus. Some commonly used chart types are :

- **Line Chart.** A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.
- **Bar Chart.** A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally.
- **Pie Chart.** A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion.

Some commonly used chart types are shown in figure below.



A line chart or line graph is a type of chart which displays information as a series of data points called **'markers'** connected by straight line segments.

A bar chart is a chart that presents categorical data with rectangular bars with heights proportional to the values that they represent.

A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion

Figure 8.2 Some commonly used chart types.

Following section talks about how you can create various chart types using pyplot methods.

8.3 CREATING CHARTS WITH MATPLOTLIB LIBRARY'S PYPLOT INTERFACE

You know that graphs and charts play a big and an important role in data visualization and decision making. Every chart type has a different utility and serves a different purpose.

Let us talk about how you can create *line, bar* and *pie charts* using *matplotlib.pyplot* library of Python.

As stated earlier make sure to **import matplotlib.pyplot** library interface and other required libraries before you start using any of their functions.

8.3.1 Line Chart

A line chart or line graph is a type of chart which displays information as a series of data points called *'markers'* connected by straight line segments. The PyPlot interface offers **plot( )** function for creating a line graph. Carefully go through the example codes given below to understand the working of **plot( )**.

LINE CHART

A line chart or line graph is a type of chart which displays information as a series of data points called *'markers'* connected by straight line segments.

Say we have following three lists, namely *a*, *b* and *c* :

```
a = [1, 2, 3, 4]
b = [2, 4, 6, 8]
c = [1, 4, 9, 16]
```

List b containing values as double of values in list a

List c containing values squares of values in list a

Now, if you want to plot a line chart for all values of *list a* vs values of *list b*, then, in simplest form you may write :

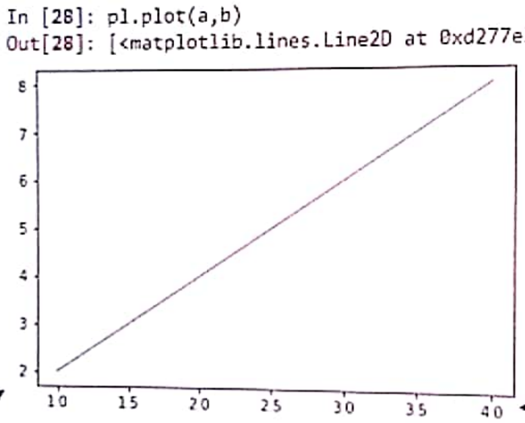
```
import matplotlib.pyplot as pl
pl.plot(a, b)
```

The import statement is to be given just once

NOTE

Data points are called ***markers***

And python will show you result as :



```
In [28]: pl.plot(a,b)
Out[28]: [<matplotlib.lines.Line2D at 0xd277e30>]
```

The values of ***list b*** are plotted on vertical axis

The values of ***list a*** are plotted on horizontal axis

But are you satisfied with the result? Shouldn't the axes show the labels of the axes ? Let us say we want to label the x-axis, the **horizontal axis**, as **'Some values'** and the y-axis, the vertical axis as **'Doubled values'**. You can set *x*-axis' and *y*-axis' labels using functions **xlabel( )** and **ylabel( )** respectively, *i.e.*, :

```
<matplotlib.pyplot or its alias>.xlabel(<str>)
```

and

```
<matplotlib.pyplot or its alias>.ylabel(<str>)
```

NOTE

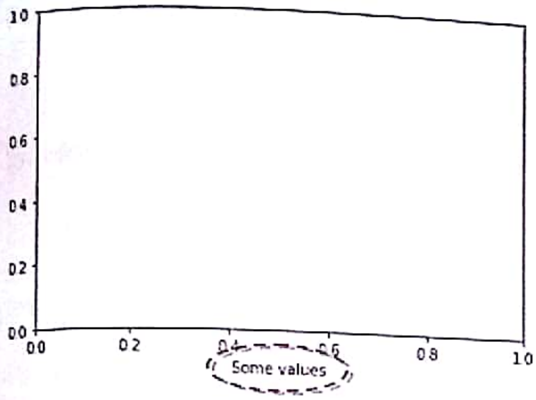
Before using any PyPlot function or numpy function, make sure that these libraries are imported beforehand.

So you may write (since we used identifier **pl** as alias for **matplotlib.pyplot** in **import** statement) :

```
pl.xlabel("Some Values")
pl.ylabel("Doubled Values")
pl.plot(a, b)
```

Let us see, what Python does.

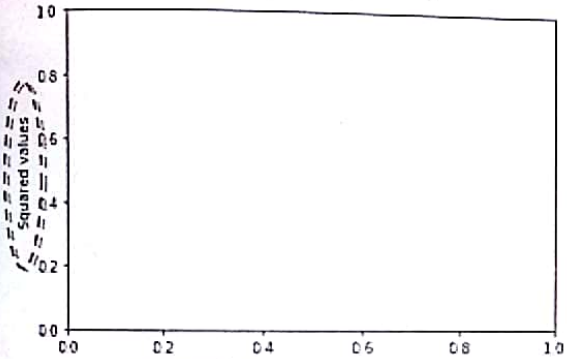
```
In [30]: pl.xlabel('Some values')
Out[30]: Text(0.5,0,'Some values')
```



See, on IPython console's prompt, the three statements have produced three charts, even if you wanted a single chart

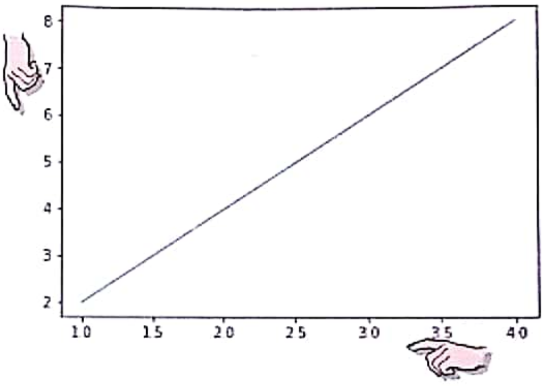
No axes titles because Python has not considered any of the previous statements

```
In [31]: pl.ylabel('Squared values')
Out[31]: Text(0,0.5,'Squared values')
```



No X-axis title only Y-axis title as only current statement is considered

```
In [32]: pl.plot(a,b)
Out[32]: [<matplotlib.lines.Line2D at 0xd42b390>]
```



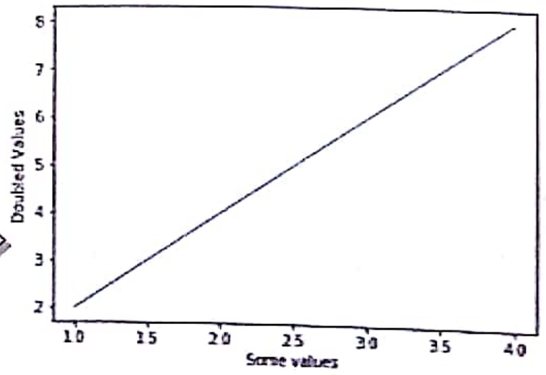
See, Python has shown us *three* different charts, one for each statement we issued. But this is not what we wanted. We wanted all these three statements to take effect in a single plot.

On the prompt, if you give these *three* statements, Python interactively draws plot for each statement separately. **To apply multiple statements on a single plot**, do one of the following :

(*i*) Either write a script, store it with **.py** extension and then run it

(*ii*) Or, on the *IPython console* prompt, press **CTRL + ENTER** to end each statement and **ENTER** in the end. This will group multiple statements and execute them as one unit, *i.e.*,

```
In [33]: pl.xlabel('Some values') ← Press CTRL+ENTER here
    ...: pl.ylabel('Doubled Values') ←
    ...: pl.plot(a, b)  ← Press ENTER in the end
    ...:
Out[33]: [<matplotlib.lines.Line2D at 0xd464b70>]
```



Now all these statements have impacted one plot as you can see the labels on both X and Y axes and graph plotted in the same plot

But the first method of writing plotting commands in a script(.py extension) is often a preferred choice for many. Also, in a script, use **<matplotlib.pyplot>.show( )** command in the end to show a plot as per given specifications.

8.3.1A Changing Line Style, Line Width and Line Color in a Line Chart

While plotting in a line chart, you can control the look of the plotted line by specifying its features like :

- Line width, Line color, line style
- Line marker type, size
- etc.

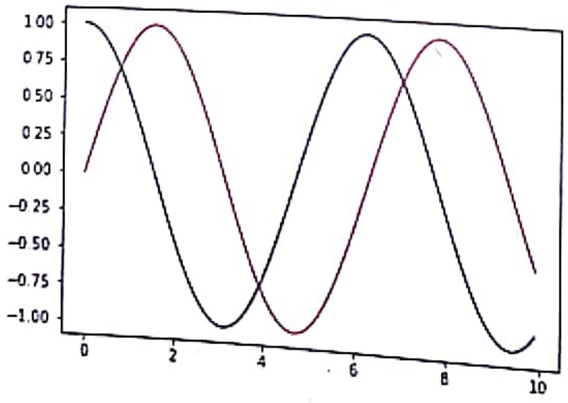
To change line color, you can specify the color code next to the data being plotting in plot() function as shown below :

```
<matplotlib>.plot(<data1>, [,data2], <color code> )
```

You can use color codes as: 'r' for red, 'y' for yellow, 'g' for green, 'b' for blue etc. (Complete color codes list has been given in Table 8.2), *e.g.*,

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0., 10, 0.1)
a = np.cos(x)
b = np.sin(x)
plt.plot(x, a, 'b')  ←——— Color code blue for ndarray a
plt.plot(x, b, 'r')  ←——— Color code red for ndarray b
plt.show()
```

The output produced by above script is as shown below :



Please note that even if you skip the color information in *plot( )*, Python will plot multiple lines in the same plot with different colors but these colors are decided internally by Python.

NOTE

Even if you skip the color information in plot( ), Python will plot multiple lines in the same plot with different colors but these colors are decided internally by Python.

Table 8.2 *Different Color Codes*

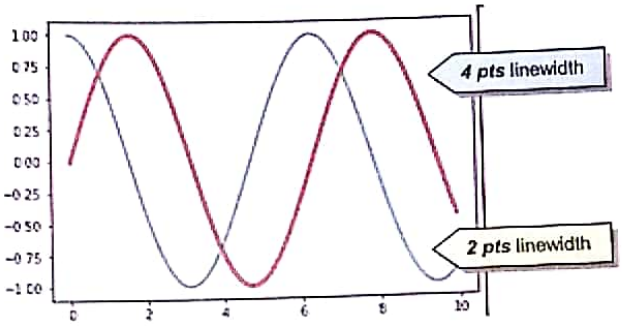
character	color	character	color	character	color
'b'	blue	'm'	magenta	'c'	cyan
'g'	green	'y'	yellow	'w'	white
'r'	red	'k'	black		

In addition, you can also specify colors in many other ways, including full color names (*e.g.*, 'red', 'green' etc.), hex strings ('#008000') etc.

To change Line width, you can give additional argument in *plot( )* as **linewidth = <width>** where you have to specify the width value in points[2], *e.g.*,

```
: #x, a, b are same as earlier code
plt.plot(x, a, linewidth = 2)
plt.plot(x, b, linewidth = 4)
```

The result produced by above code is as shown here :



To change the line style, you can add following additional optional argument in *plot( )* function :

```
linestyle or ls = ['solid' | 'dashed', 'dashdot', 'dotted']
```

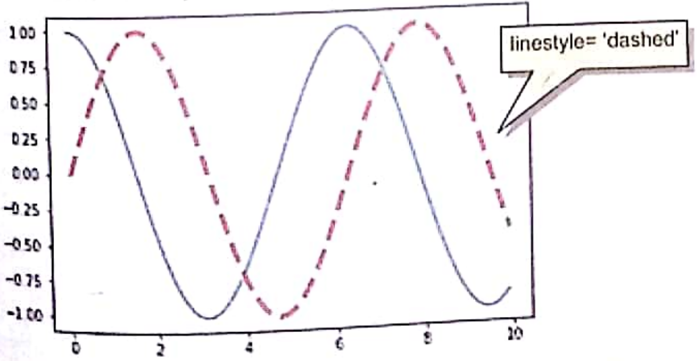
e.g.,

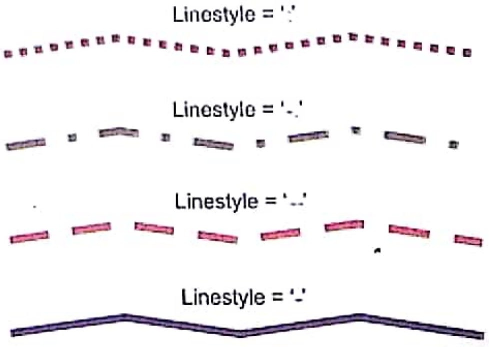
```
: #x, a, b are same as earlier code
plt.plot(x, a, linewidth = 2)
plt.plot(x, b, linewidth = 4, linestyle = 'dashed')
```

NOTE

Possible line styles are : **'solid'** for *solid line*, **'dashed'** for *dashed line*, **'dotted'** for *dotted line* and **'dashdot'** for *dashdotted line.*

The result produced by above code is as shown here :





8.3.1B Changing Marker Type, Size and Color

Recall that *data points* being plotted are called **markers**. To change *marker type*, its *size* and color, you can give following additional optional arguments in *plot( )* function :

```
marker = <valid marker type> , markersize = <in points>, markeredgecolor = <valid color>
```

[2] Line width or thickness is measured in points e.g., 0.75 points, 2.0 points etc.

You can control the *type of marker i.e., dots or crosses or diamonds* etc. by specifying desired marker type from the table below. If you do not specify marker type, then data points will not be marked specifically on the line chart and its default type will be same as that of the line type.

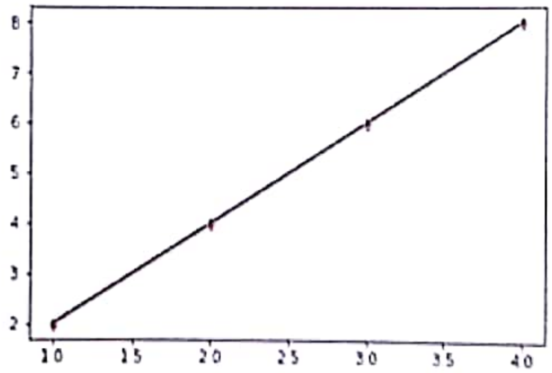
Also, the ***markersize*** is to be specified in points and ***markeredgecolor*** must be a valid color.

Table 8.3 *Marker Types for Plotting*

marker	description	marker	description	marker	description
'.'	point marker	's'	square marker	'3'	tri_left marker
','	pixel marker	'p'	pentagon marker	'4'	tri_right marker
'o'	circle marker	'*'	star marker	'v'	triangle_down marker
'+'	plus marker	'h'	hexagon1 marker	'^'	triangle_up marker
'x'	x marker	'H'	hexagon2 marker	'<'	triangle_left marker
'D'	diamond marker	'1'	tri_down marker	'>'	triangle_right marker
'd'	thin_diamond marker	'2'	tri_up marker	'|', '_'	vline, hline markers

Now consider following example where p = [1, 2, 3, 4] and q = [2, 4, 6, 8]

```
In [61]: plt.plot(p, q, 'k', marker='d', markersize = 5, markeredgecolor ='red')
Out[61]: [<matplotlib.lines.Line2D at 0xd09c310>]
```

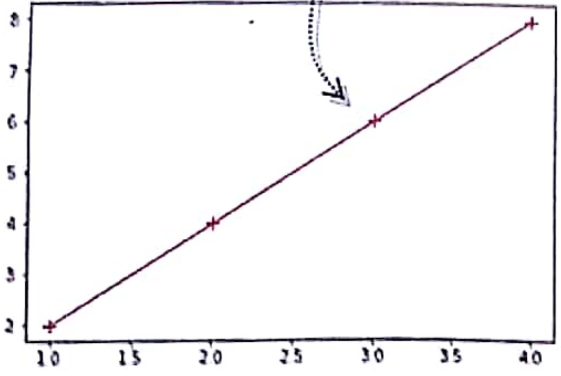


Line color is black (color 'k')
Marker type = small diamond ('d')
Marker size = 5 points
Marker color = 'red'

Note you can combine the marker type with color code, *e.g.*, 'r+' when given for ***line color*** marks the *color* as **'red'** and ***markertype*** as plus('+'). Consider two following examples combining these :

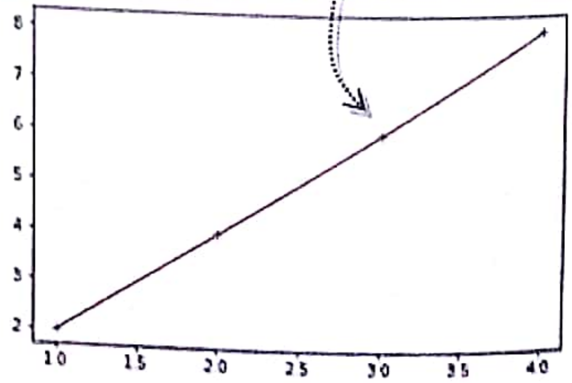
Line color and *marker style* combined so marker takes same color as line

```
In [75]: plt.plot(p, q, 'r+', linestyle = 'solid')
Out[75]: [<matplotlib.lines.Line2D at 0xbdfc470>]
```



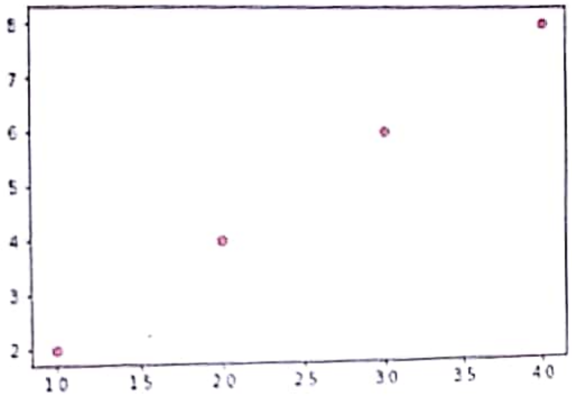
Here *marker color* separately specified

```
In [76]: plt.plot(p, q, 'r+', linestyle = 'solid',
    ...: markeredgecolor='b')
Out[76]: [<matplotlib.lines.Line2D at 0xbd46c50>]
```



See, when you do not specify **markeredgecolor** separately in **plot( )**, the marker takes the same color as the line. Also, if you do not specify the **linestyle** separately along with *linecolor- and-marketstyle-combination-string* (*e.g.*, 'r+' above), Python will only plot the markers and not the line, (see below). To get the line, specify **linestyle** argument also as shown above.

```
In [3]: plt.plot(p,q, 'ro')
Out[3]: [<matplotlib.lines.Line2D at 0x8834770>]
```



if you do not specify the **linestyle** argument separately along with linecolor-&-marketstyle-string (e.g., 'r+' or 'bo' etc.), Python will only plot the markers and not the line

So full useful syntax of **plot( )** can be summarized as :

```
plot( <data1> [,<data2>] [, ... <colorcode and marker type>] [, <linewidth>]
        [, <linestyle>] [,<marker>][, <markersize>] [,<markeredgecolor>] )
```

where,

- ***data1, data2*** are sequences of values to be plotted on x-axis and y-axis respectively
- Rest of the arguments affect the look and format of the line chart as given below :
  - **color code with markertype** color and marker symbol for the line chart
  - **linewidth** width of the line in points (a float value)
  - **linestyle** can be ['solid' | 'dashed', 'dashdot', 'dotted' or take markertype string]
  - **marker** a symbol denoting valid marker style such as '–', 'o', 'x', '+', 'd', and others etc.
  - **markersize** size of marker in points (a float value)
  - **markeredgecolor** color for markers; should be a valid color identified by PyPlot

* *please note that this is still not the complete and full syntax of plot( ). Complete syntax coverage of plot( ) is beyond the scope of the book.*

NOTE

If you are working on a Jupyter Notebook, you can make the plots appear inline in the notebook by using following magic function :

```
%matplotlib inline
```

If you don't use above function, the plots will appear in a pop up window as with other methods.

Example 8.1 *Create an array in the range 1 to 20 with values 1.25 apart. Another array contains the log values of the elements in first array.*

(*a*) *Simply plot the two arrays first vs second in a line chart.*

(*b*) *In the plot of first vs second array, specify the x-axis (containing first array's values) title as* ***'Random Values'*** *and y-axis title as* ***'Logarithm Values'.***

(c) *Create a third array that stores the COS values of first array and then plot both the second and third arrays vs first array. The Cos values should be plotted with a* **dashdotted line.**

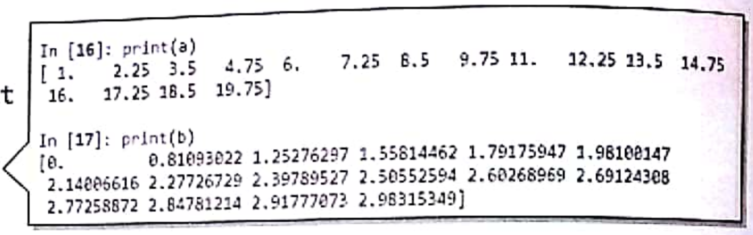
(d) *Change the marker type as a circle with blue color in second array.*

(e) *Only mark the data points as this : second array data points as* **blue small diamonds,** *third array data points as* **black circles.**

Note : Show commands and their results.

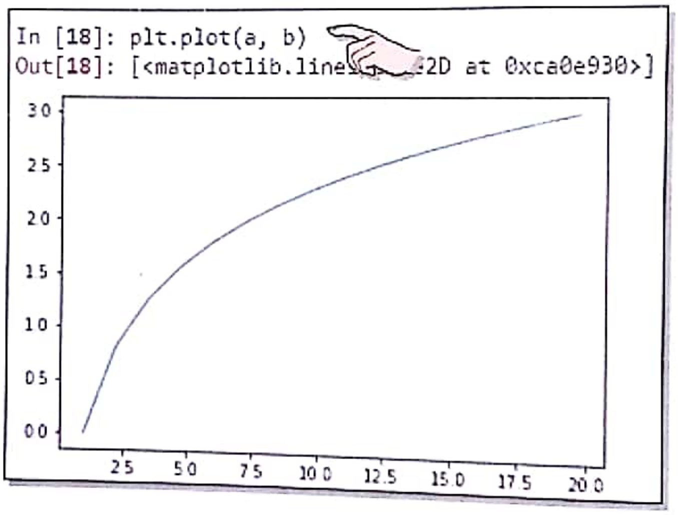
Solution.

```
import numpy as np
import matplotlib.pyplot as plt
a = np.arange(1, 20, 1.25)
b = np.log(a)
```



```
In [16]: print(a)
[ 1.     2.25  3.5   4.75  6.    7.25  8.5   9.75 11.   12.25 13.5  14.75
 16.   17.25 18.5  19.75]

In [17]: print(b)
[0.         0.81093022 1.25276297 1.55814462 1.79175947 1.98100147
 2.14006616 2.27726729 2.39789527 2.50552594 2.60268969 2.69124308
 2.77258872 2.84781214 2.91777073 2.98315349]
```

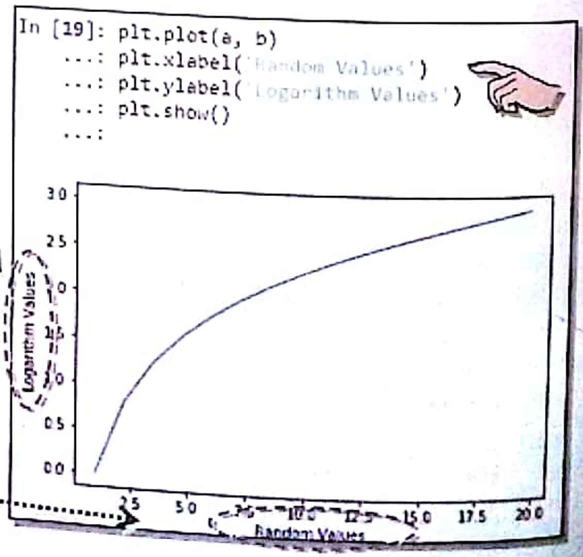
(*a*) `plt.plot(a, b)`



```
In [18]: plt.plot(a, b)
Out[18]: [<matplotlib.line[illegible]2D at 0xca0e930>]
```

(*b*)

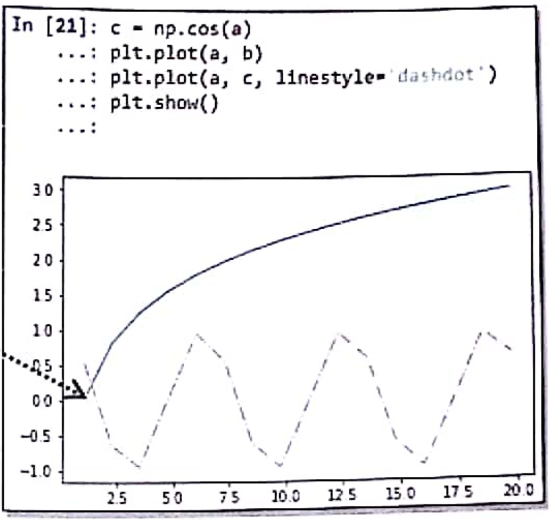
```
plt.plot(a, b)
plt.xlabel('Random Values')
plt.ylabel('Logarithm Values')
plt.show()
```



```
In [19]: plt.plot(a, b)
    ...: plt.xlabel('Random Values')
    ...: plt.ylabel('Logarithm Values')
    ...: plt.show()
    ...:
```

(c)

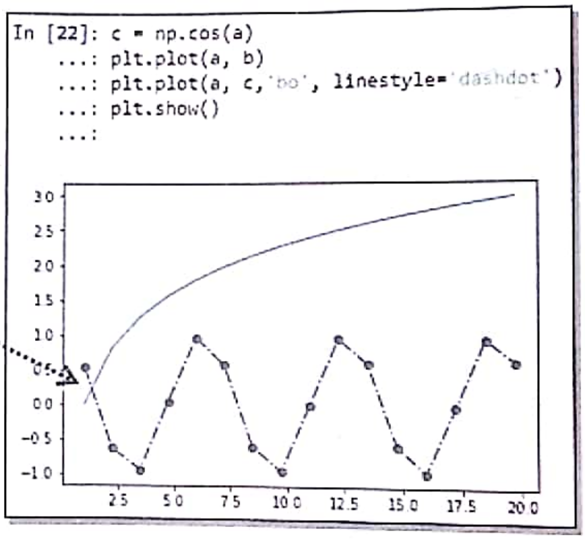
```
c = np.cos(a)
plt.plot(a, b)
plt.plot(a, c, linestyle = ‘dashdot’)
plt.show()
```



```
In [21]: c = np.cos(a)
    ...: plt.plot(a, b)
    ...: plt.plot(a, c, linestyle='dashdot')
    ...: plt.show()
    ...:
```

(d)

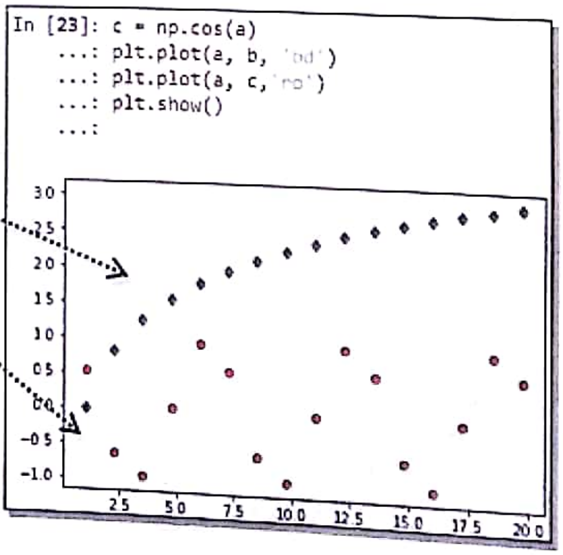
```
c = np.cos(a)
plt.plot(a, b)
plt.plot(a, c, ‘bo’, linestyle = ‘dashdot’)
plt.show()
```



```
In [22]: c = np.cos(a)
    ...: plt.plot(a, b)
    ...: plt.plot(a, c, 'bo', linestyle='dashdot')
    ...: plt.show()
    ...:
```

(e)

```
c = np.cos(a)
plt.plot(a, b, ‘bd’)
plt.plot(a, c, ‘ro’)
plt.show()
```



```
In [23]: c = np.cos(a)
    ...: plt.plot(a, b, 'bd')
    ...: plt.plot(a, c, 'ro')
    ...: plt.show()
    ...:
```

8.3.2 Bar Chart

A *Bar Graph* or a *Bar Chart* is a graphical display of data using bars of different heights. A bar chart can be drawn *vertically* or *horizontally* using rectangles or bars of different heights /widths PyPlot offers **bar( )** function to create a bar chart where you can specify the sequences for *x*-axis and corresponding sequence to be plotted on *y*-axis. Each *y* value is plotted as bar on corresponding *x*-value on *x*-axis.

BAR GRAPH / CHART

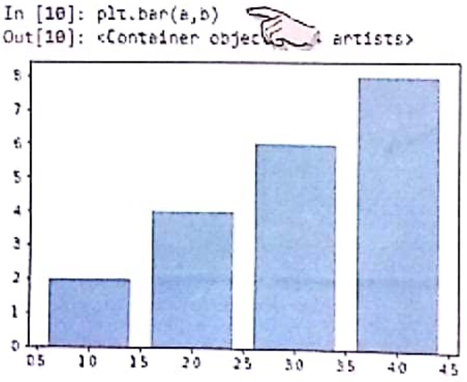
A Bar Graph / Chart is a graphical display of data using bars of different heights.

But before creating any bar chart, like you did in line charts, make sure to *import matplotlib.pyplot and also numpy, if you are using any numpy function. Likewise, if you want that multiple commands affect a common bar chart, then either store all the related statements in a Python script(.py file) with last statement being <matplotlib.pyplot>.show( ), or on iPython console, group related commands by pressing CTRL+ENTER; press ENTER only after the last command.*

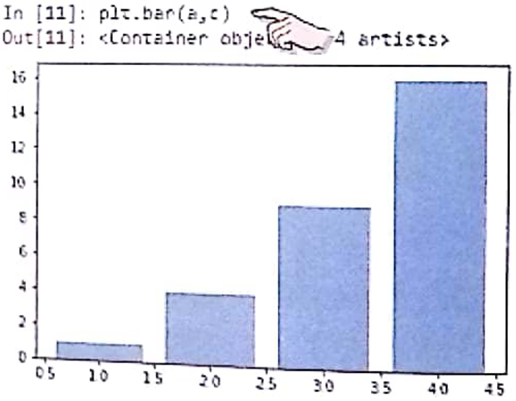
Now consider the following code. If we have *three* sequences **a**, **b**, and **c** as :

```
a, b, c = [1, 2, 3, 4], [2, 4, 6, 8], [1, 4, 9, 16]
```

Then commands **matplotlib.pyplot.bar(a.b)** and **matplotlib.pyplot.bar(a,c)** will produce result as :



```
In [10]: plt.bar(a,b)
Out[10]: <Container object of 4 artists>
```



```
In [11]: plt.bar(a,c)
Out[11]: <Container object of 4 artists>
```

Notice, first sequence given in the **bar( )** forms the *x-axis* and the second sequence values are plotted on y-axis.

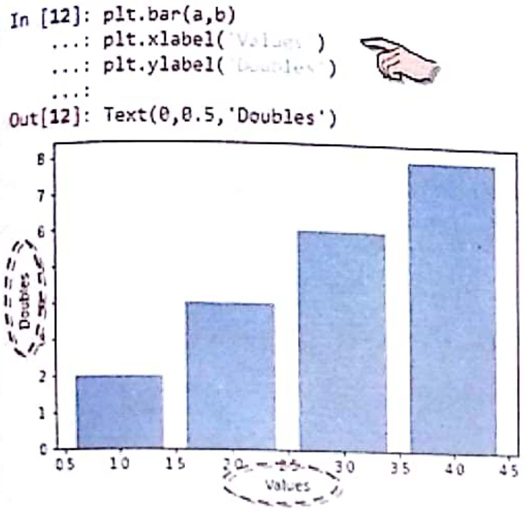
As earlier, if you want to specify *x-axis label* and *y-axis label*, then you need to give commands :

```
matplotlib.pyplot.xlabel(<label string>)
matplotlib.pyplot.ulabel(<label string>)
```

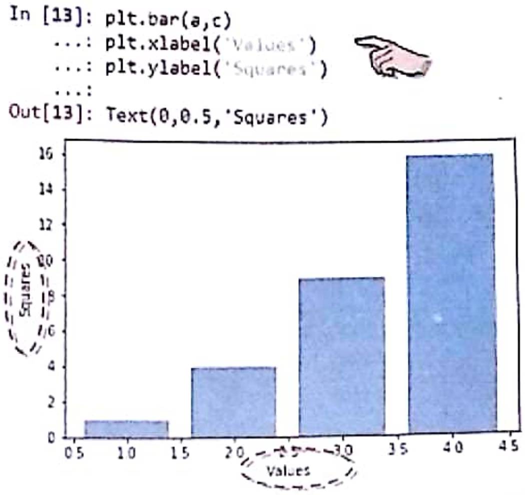
Also, you need to group the commands either in a script file (*.py* file) or by pressing CTRL+ENTER on the prompt for all commands except for the last command of the group (ENTER for the last command), as you can see yourself in the figures below.

NOTE

The simple bar plot is best used when there is just one level of grouping to your variable.



```
In [12]: plt.bar(a,b)
    ...: plt.xlabel('Values')
    ...: plt.ylabel('Doubles')
    ...:
Out[12]: Text(0,0.5,'Doubles')
```



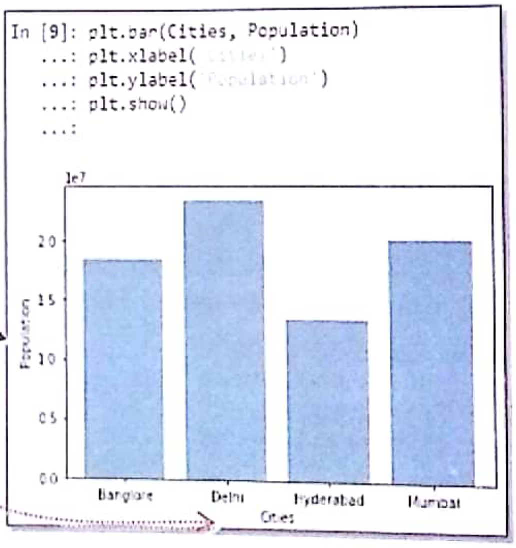
```
In [13]: plt.bar(a,c)
    ...: plt.xlabel('Values')
    ...: plt.ylabel('Squares')
    ...:
Out[13]: Text(0,0.5,'Squares')
```

NOTE

The first sequence given in the **bar( )** forms the *x-axis* and the second sequence values are plotted on *y-axis*.

Consider another example :

```
Cities = ['Delhi', 'Mumbai', 'Banglore', 'Hyderabad']
Population = [23456123, 20083104, 18456123, 13411093]
plt.bar(Cities, Population)
plt.xlabel('Cities')
plt.ylabel('Population')
plt.show()
```



```
In [9]: plt.bar(Cities, Population)
   ...: plt.xlabel('Cities')
   ...: plt.ylabel('Population')
   ...: plt.show()
   ...:
```

NOTE

The order of bars plotted may be different from the order in actual data sequence.

Note that the order of bars plotted is different from the order in actual data sequence *e.g.*, the **Cities** sequence being plotted contains values in the order ("Delhi", "Mumbai", "Bangalore", "Hyderabad") but when plotted their order is different, *i.e.*, "Bangalore", "Delhi", "Hyderabad" and "Mumbai" – the sorted order.

8.3.2A Changing Widths of the Bars in a Bar Chart

By default, bar chart draws bars with equal widths and having a default width of 0.8 units on a bar chart. That is, all bars have the width same as the *default width*. But you can always change the widths of the bars.

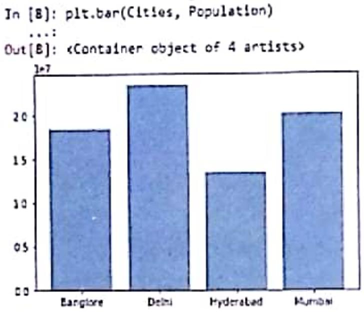
- You can specify a different width (other than the default width) for all the bars of a bar chart
- You can also specify different widths for different bars of a bar chart

(i) To specify common width (other than the default width) for all bars, you can specify ***width* argument** having a scalar float value in the **bar( )** function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, width = <float value>)
```

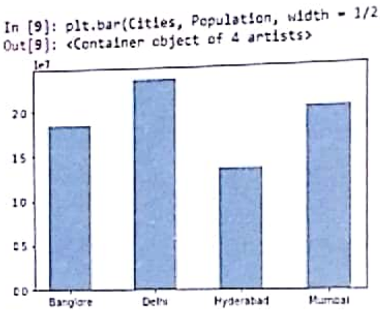
Consider the following three bar charts where the first chart has the bars with default *widths*, second chart with a *width as per expression 1/2* (*i.e.*, 0.5 units) and the third chart has *width specified as 0.3 units.*

Default width for all bars
plt.bar(Cities, Population)



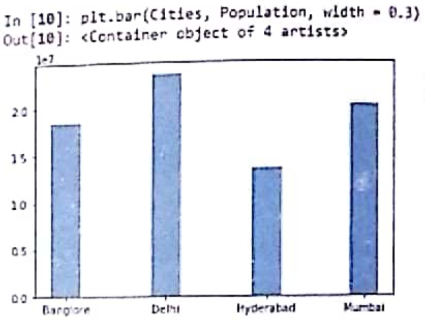
```
In [8]: plt.bar(Cities, Population)
   ...:
Out[8]: <Container object of 4 artists>
```

Width as 0.5 units for all bars
plt.bar(Cities, Population, width = 1/2)



```
In [9]: plt.bar(Cities, Population, width = 1/2)
Out[9]: <Container object of 4 artists>
```

Width as 0.3 units for all bars
plt.bar(Cities, Population, width = 0.3)



```
In [10]: plt.bar(Cities, Population, width = 0.3)
Out[10]: <Container object of 4 artists>
```

So, when you specify a scalar value *i.e.*, a single value for **width** argument, then that width is applied to all the bars of the bar chart.

NOTE

If you specify a scalar value (a single value) for **width** argument, then that width is applied to all the bars of the bar chart.

(ii) To specify different widths for different bars of a bar chart, you can specify *width* **argument** having a sequence (such as lists or tuples) containing widths for each of the bars, in the **bar( )** function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, width = <width values sequence>)
```

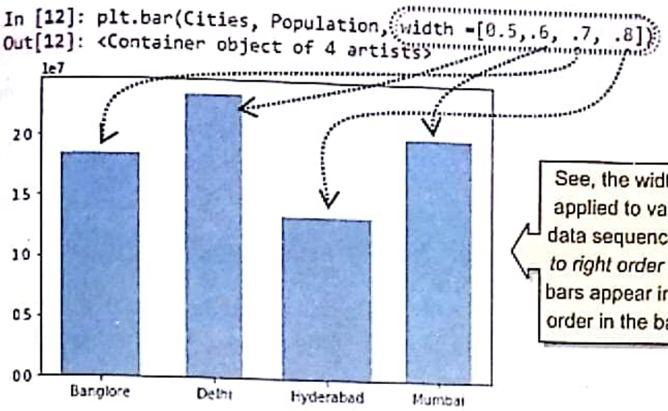
The widths given in the sequence are applied from left to right, *i.e.*, the first value of the sequence specifies the width of first value of data sequence, second value specifies the width for the second value of data sequence, and so on. The width values are *float* values. Please note **that the width sequence must have widths for all the bars** (*i.e.*, its length must match the length of data sequences being plotted) otherwise Python will report an error, the [*ValueError: shape mismatch error*].

Consider the following bar chart :

```
plt.bar(Cities, Population, width =[0.5,.6, .7, .8])
```

So as per above **bar( )**, the widths 0.5, .6, .7, .8 are applicable to **Cities** values in order *i.e.*, "**Delhi**"'s bar will be 0.5 units wide; "**Mumbai**"'s bar will be 0.6 units wide ; "**Bangalore**"'s bar will be 0.7 units wide ; and "**Hyderabad**"'s bar will be 0.8 units wide.

The output produced by above command will be as shown below :



```
In [12]: plt.bar(Cities, Population, width =[0.5,.6, .7, .8])
Out[12]: <Container object of 4 artists>
```

See, the widths are applied to values of data sequence in *left to right order* but the bars appear in sorted order in the bar chart

NOTE

The width values' sequence in a bar( ) must have widths for all the bars, *i.e.*, its length must match the length of data sequences being plotted, otherwise Python will report an error.

8.3.2B Changing Colors of the Bars in a Bar Chart

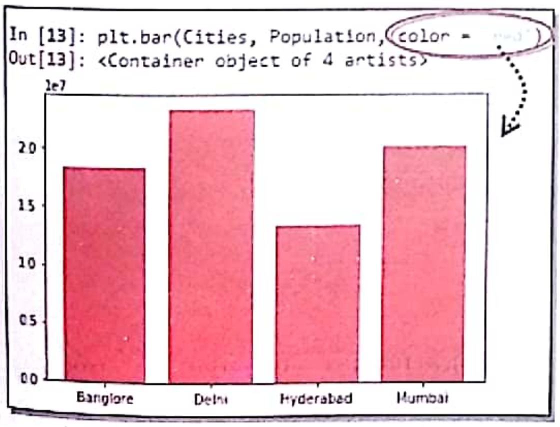
By default, a bar chart draws bars with same default color. But you can always change the color of the bars.

- You can specify a different color for all the bars of a bar chart.
- You can also specify different colors for different bars of a bar chart.

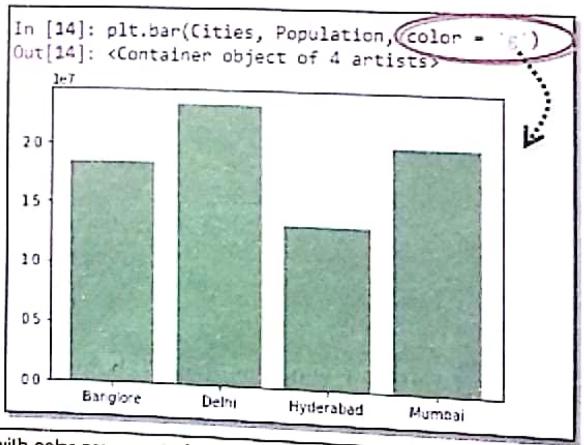
(*i*) **To specify common color (other than the default color) for all bars, you can specify** *color* **argument** having a valid color code/name in the bar( ) function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, color = <color code/name>)
```

The *color* given with **color** argument will be applied to all the bars, *e.g.*, consider this :



```
In [13]: plt.bar(Cities, Population, color = 'red')
Out[13]: <Container object of 4 artists>
```



```
In [14]: plt.bar(Cities, Population, color = 'g')
Out[14]: <Container object of 4 artists>
```

When you specify single color name or single color code with cobr argument of the bar() function, the specified color is applied to all the bars of the bar chart, i.e., all bars of the bar chart have the same common color.

(*ii*) **To specify different colors for different bars of a bar chart,** you can specify *color* argument having a sequence (such as lists or tuples) containing colors for each of the bars, in the **bar( )** function, *i.e.*, as :

```
<matplotlib.pyplot>.bar(<x-sequence>, <y-sequence>, color = <color names/codes sequence>)
```

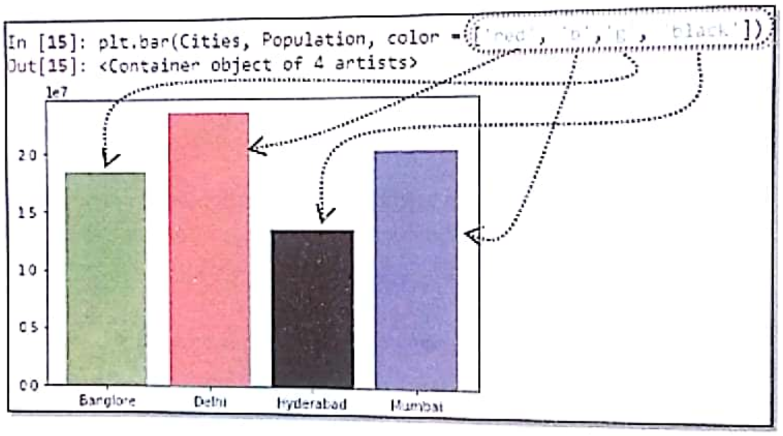
The colors given in the sequence are applied from left to right, *i.e.*, the first value of the sequence specifies the color of first value of data sequence, second value specifies the color for the second value of data sequence, and so on. The color values are *valid color codes or names*. Please note that **the color sequence must have colors for all the bars** (*i.e.*, its length must match the length of data sequences being plotted) otherwise Python will report an error, the [*ValueError: shape mismatch error*].

Consider the following bar chart :

```
plt.bar(Cities, Population, color = ['red', 'b', 'g', 'black'])
```

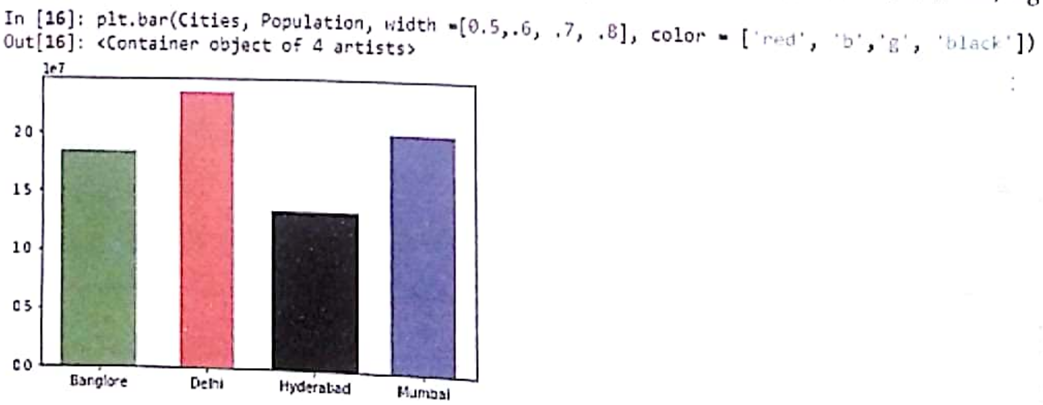
So as per above **bar( )**, the colors *'red', 'b', 'g', 'black'* are applicable to **Cities** values in order *i.e.*, "**Delhi**"'s bar will have 'red' color; "**Mumbai**"'s bar will have blue color (color code'b'); "**Bangalore**"'s bar will have 'green' color (color code 'g') ; and "**Hyderabad**"'s bar will have 'black' color.

The output produced by above command will be as shown below :



```
In [15]: plt.bar(Cities, Population, color = ['red', 'b','g', 'black'])
Out[15]: <Container object of 4 artists>
```

Both *width* and *color* arguments are optional and you can combine them both also, *e.g.*,



```
In [16]: plt.bar(Cities, Population, width =[0.5,.6, .7, .8], color = ['red', 'b','g', 'black'])
Out[16]: <Container object of 4 artists>
```

8.3.2C Creating Multiple Bars chart

Often in real life, you may need to plot multiple data ranges on the same bar chart creating multiple bars. As such PyPlot does not provide a specific function for this, BUT you can always create one exploiting the *width* and *color* arguments of **bar( )** that you learnt above.

Let's see how.

(*i*) **Decide number of X points.** Firstly, you need to determine how many X points you will need. For this, calculate the number of entries in the ranges being plotted. Say, you are plotting two sequences **A** and **B**, so length of sequences **A** or **B** being plotted will determine the number of X points in our case. You can create a sequence with this length of A or B using *range( )* or **numpy.arange( )** functions.

Do note that both the sequences A and B must have similar shape *i.e.*, same length (number of elements).

(*ii*) **Decide thickness of each bar and accordingly adjust X points on X-axis.** So, if you want to plot two bars per X point, then carefully think of the thickness of each bar. Say you want the thickness of each bar as .3 , then for first range keep it as X and for the second data range, make the X point as X + 0.3. (see below)

(*iii*) **Give different color to different data ranges using color argument of bar( ) function.**

(*iv*) **The *width* argument remains the same for all ranges being plotted.**

(*v*) **Plot using bar( ) for each range separately,** *i.e.*, the number of bars decide the number of **bar( )** functions you will be using to plot different bars on same plot.

Following example will make it clear.

Say we want to plot ranges $A = [2, 4, 6, 8]$ and $B = [2.8, 3.5, 6.5, 7.7]$

(*i*) Deciding X points and thickness. Say, we want the thickness of each bar as 0.35, then **for the first range, X point will be** X, and for the second range, the X will shift by first bar's thickness, *i.e.*, X + 0.35. (If there were three bars, then X points would have been X, X + 0.35 *i.e.*, X + thick of one bar on its left, and X + 0.70 , *i.e.*, X + thickness of two bars on its left.)

(*ii*) **Deciding colors.** Say we want *red* color for the first range and *blue* color for the second range.

(*iii*) The **width** argument will take value as 0.35 in this case.

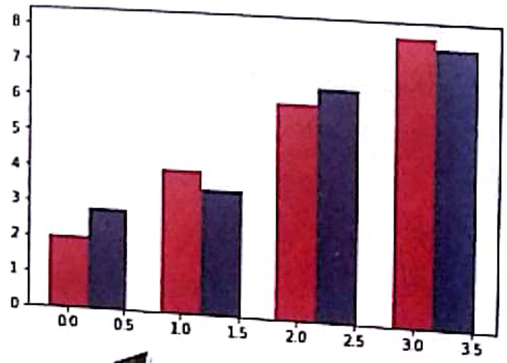
(*iv*) Plot using multiple bar( ) functions.

As there are two ranges to be plotted, we shall need to use two bar( ) functions one for plotting X *vs.* A and the second one for plotting X + width *vs.* B, as depicted below :

```
In [1]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...:

In [2]: A = [2, 4, 6, 8]
   ...: B = [2.8, 3.5, 6.5, 7.7]
   ...:

In [3]: X = np.arange( len(A) )

In [4]: plt.bar(X, A, color='red', width = 0.35)
   ...: plt.bar(X+0.35, B, color= 'blue', width = 0.35)
   ...: plt.show()
   ...:
```

Sequence created for X points using lengths of sequences A or B being plotted

See, X in first bar() is X and in second bar(), it is X +0.35. Also, first bar() plots X vs A and second bar() plots X+width vs B



Following example also creates a multi-bar chart plotting three different data ranges.

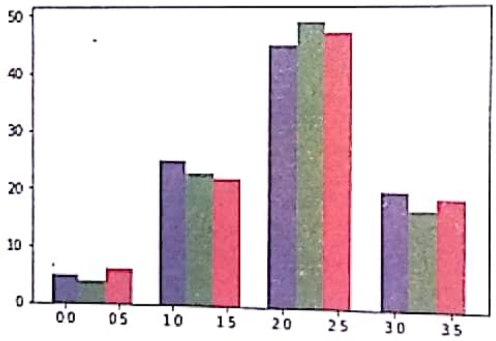
Example 8.2 *Val is a list having three lists inside it. It contains summarized data of three different trials conducted by company A. Create a bar chart that plots these three sublists of Val in a single chart. Keep the width of each bar as 0.25.*

Solution.

```
import numpy as np
import matplotlib.pyplot as plt

Val = [[5., 25., 45., 20.], [4., 23., 49., 17.], [6., 22., 47., 19.]]

X = np.arange(4)
plt.bar(X + 0.00, Val[0], color = 'b', width = 0.25)
plt.bar(X + 0.25, Val[1], color = 'g', width = 0.25)
plt.bar(X + 0.50, Val[2], color = 'r', width = 0.25)

plt.show()
```

See, the X range for the third bar(), is deviated with X+ widths of two bars on the left of it, i.e., X+0.50 as width of one bar is 0.25

The output produced by above code is :

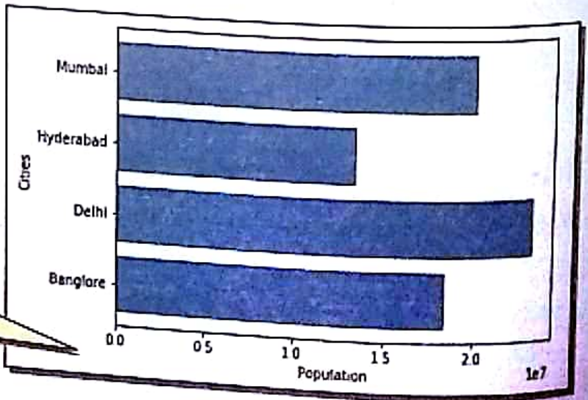


8.3.2D Creating a Horizontal Bar Chart

To create a horizontal bar chart, you need to use **barh( )** function (bar horizontal), in place of bar( ). Also, you need to give x and y axis labels carefully – the label that you gave to x axis in bar( ), will become y-axis' label in barh( ) and vice versa.

For instance, consider the following code where we have replaced bar( ) with barh( ) in an earlier code :

```
In [2]: Cities =['Delhi', 'Mumbai', 'Banglore', 'Hyderabad']
   ...: Population= [23456123, 20083104, 18456123, 13411093]
   ...: plt.barh(Cities, Population)
   ...: plt.ylabel('Cities')
   ...: plt.xlabel('Population')
   ...: plt.show()
```



Also, notice that the labels of x and y axis have been swapped for **barh()** when compared to **bar()**

8.3.3 The Pie Chart

Recall that a pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. Typically, a Pie Chart is used to show parts to the whole, and often a % share. The PyPlot interface offers **pie( )** function for creating a pie chart.

PIE CHART

The **pie chart** is a type of graph in which a circle is divided into sectors that each represent a proportion of the whole.

Before we proceed with pie( ) function's examples, it is important to know two things :

(*i*) The pie( ) function, **plots a single data range only**. It will calculate the share of individual elements of the data range being plotted *vs.* the whole of the data range.

(*ii*) The default shape of a pie chart is oval but you can always change to circle by using **axis( )** of pyplot, sending "equal" as argument to it. That is, issue following command before you use **pie( )** function :

```
matplotlib.pyplot.axis("equal")
```

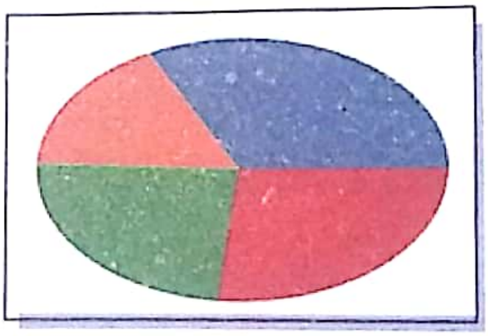
In place of ***matplotlib.pyplot****, you may also use its alias name, the name you have given to it while importing*

Now consider following example code to understand working of pie( ).

Your school houses have collected amount for PM charity fund. To plot a pie chart from this data, you can do the following :

```
contri = [17, 8.8, 12.75, 14]          # contribution in thousands
plt.pie(contri)
```

The data range being plotted in pie( )



The output produced by above code will be :

The adjacent chart shows the values of **contri** list as slices of a pie.

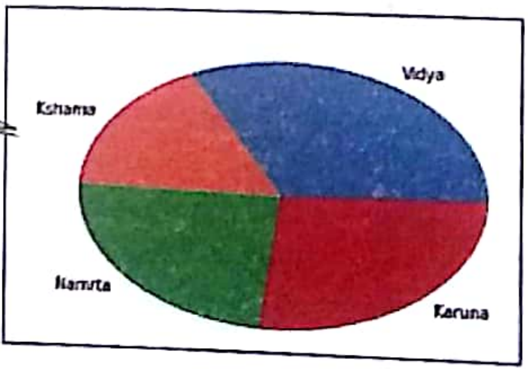
8.3.3A Labels of Slices of Pie

The above shown pie chart is just incomplete. You cannot make out which slice belongs to what. So, there must be labels next to the slices to make more readable and understandable. For this purpose, you need to create a sequence containing the labels and then specify this sequence as value for labels argument of pie() function. The first label is given to first value; second label to second value and so on, *e.g.*, see below :

```
contri = [17, 8.8, 12.75, 14]          # contribution in thousands
houses = ['Vidya', 'Kshama', 'Namrta', 'Karuna']
plt.pie(contri, labels = houses)
```

Now the labels for the slices will be taken from the sequence namely houses.

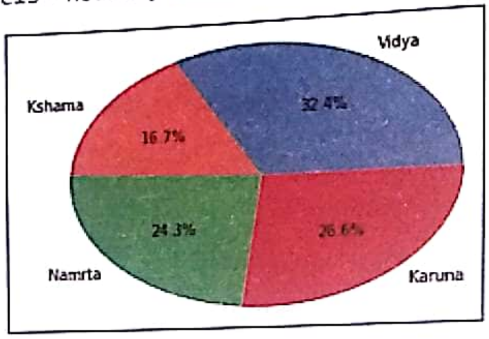
Now the output looks like :



8.3.3B Adding Formatted Slice Percentages to Pie

To view percentage of share in a pie chart, you need to add an argument **autopct** with a format string, such as "%1.1F%%". It will show the percent share of each slice to the whole, formatted in a certain way, *e.g.*, see below :

```
plt.pie(contri, labels = houses, autopct ="%1.1f%%")
```



The percentage of each value plotted is calculated as : *single value/(sum of all values)* * 100 and this also determines the size of the slice, *i.e.*, more the value bigger the slice.

For instance, if a list being plotted contains values as [30, 50, 10, 6], then

(*i*) First of internally sum of all these is calculated which is 30 + 50 + 10 + 6 = **96.**

(*ii*) Next each value's percentage share and its slice size is calculated as :

Slice 1 : 30/96 = 31.25%
Slice 2 : 50/96 = 52.08%
Slice 3 : 10/96 = 10.42%
Slice 4 : 6/96 = 6.25%

The format string used with **autopct** will determine the format of the percentage being displayed. The format string begins with the "%" operator, which specifies the format of the percentage value being displayed.

The general syntax for a format placeholder is

```
"%[flags][width][.precision]type"
```

- The first % symbol is a special character that signifies that it is a special string which will determine the format of the values to be displayed.
- The **width** determines the total number of characters to be displayed (digits before and after decimal point + 1 for decimal point). If the value being displayed has lesser number of digits than the width specified, then the value is padded as per the *flag* specified. If however, the value has more digits than the *width* specifies, then the full value is displayed. The examples below will make it clear.
- In pie charts, the most useful **flag** is 0 (zero), which when specified will pad the value being displayed with preceding zeros if the digits of value is less the *width.*
- The *precision* is specified with digits and it specifies the number of digits after decimal point.

- The *type* specifies the type of value : *d* or *i* means integer type and *f* or F means float type.
- To print a % sign, given %% (two percentage signs) in the format string. The 2 %% signs are needed to print a percentage sign, otherwise single % is interpreted as a special character with a special meaning and thus used accordingly. By doubling %, *i.e.*, %%, you suppress its special meaning and thus a percentage sign is printed.

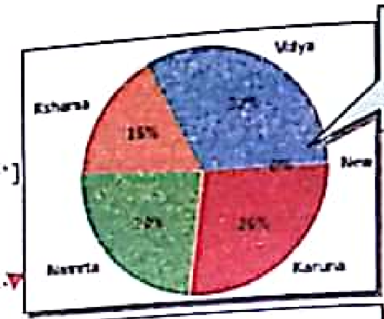
Now consider following examples of format strings as given below.

<table>
<tr><th>Format string</th><th>Description</th></tr>
<tr><td>"%5d" , "%5i"</td><td><code>width = 5, type = d or i (integer type)</code>
Print the value with 5 characters e.g., if the value being printed is 123 then it will be printed as __123, because 123 consists only of 3 digits, the output is padded with 2 leading blanks.</td></tr>
<tr><td>"%05d", "%05i"</td><td><code>flag = 0, width = 5, type = d or i (integer type i.e., no decimal point)</code>
Print the value with 5 characters and pad with leading zeros if necessary. <i>E.g.</i>, if the value being printed is 123, then it will be printed as 00123, because value 123 consists only of 3 digits, the output is padded with 2 leading zeros to make it have width of 5 digits.</td></tr>
<tr><td>"%03d %%",
"%03i %%"</td><td><code>width = 3, type = d or i (integer type), percentage sign in the end</code>
Print the value with 3 characters, pad with leading zeros if necessary and end with a % sign. <i>E.g.</i>, if the value being printed is 123 then it will be printed as 123 % ; no leading zeros as the width 3 is fully filled.</td></tr>
<tr><td>"%6.2f",
"%6.2F"</td><td><code>width = 6, precision = 2, type = f (float type)</code>
Print the value with space of 6 characters; after decimal point, keep the precision of 2 digits (rounding off may take place for this precision); pad with leading blank if necessary. Consider following examples :
<table>
<tr><th>Value</th><th>Formatted As</th><th>Remarks</th></tr>
<tr><td>12.679</td><td>_12.68</td><td>Rounded off for 2 precise digits ; 1 leading blank</td></tr>
<tr><td>0.018</td><td>_ _0.02</td><td>Rounded off for 2 precise digits ; 2 leading blanks</td></tr>
<tr><td>211.2</td><td>211.20</td><td>1 trailing zero added for 2 precise digits.</td></tr>
<tr><td>7</td><td>_ _7.00</td><td>2 trailing zeros added for 2 precise digits ; 2 leading blanks.</td></tr>
<tr><td>1009.27</td><td>1009.27</td><td>This time 7 characters will get printed (more than the width) as the number before the decimal point cannot be reduced and also, precision digits cannot be reduced as .27 is already precise to 2 digits.</td></tr>
<tr><td>1009.277</td><td>1009.28</td><td>7 characters will get printed as the number before the decimal point cannot be reduced and for 2 precision digits, 0.277 is rounded off as .28.</td></tr>
</table></td></tr>
<tr><td>"%6.2f%%",
"%6.2F%%"</td><td>Same as previous but with a percentage sign in the end.</td></tr>
</table>

Now consider some examples below where we added one new value each to lists **contri** and **houses** used above.

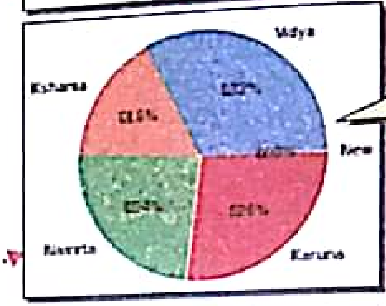
```
In [30]: contri
Out[30]: [17, 8.8, 12.75, 14, 0.1]

In [31]: houses
Out[31]: ['Vidya', 'Kshama', 'Namrta', 'Karuna', 'New']
```

```
plt.axis("equal")      # for circle shape
plt.pie(contri, labels = houses, autopct ="%3d%%")
```



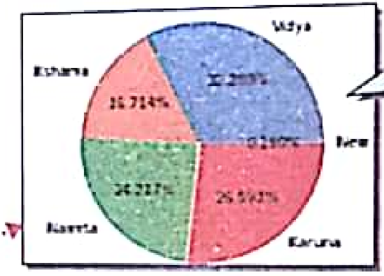
See, 'New' house's contri value 0.1 is depicted as 0% in d (integer format)

```
plt.axis("equal")      # for circle shape
plt.pie(contri, labels = houses, autopct ="%03d%%")
```



See, leading zeros have been padded this time to make it 3 characters wide

```
plt.axis("equal")      # for circle shape
plt.pie(contri, labels = houses, autopct ="%05.3f%%")
```



See, trailing zeros have been added this time to make it 3 digits precise

8.3.3C Changing Colors of the Slices

By default, the **pie( )** function shows each slice with a different color. If you are not satisfied with the default colors, you can specify own colors for the slices. For this purpose, you need to create a sequence containing the color codes or names for each slice and then specify this sequence as a value for **colors** argument of **pie( )** function. The first color is given to first value; second color to second value and so on, *e.g.*, see below (considering the same **contri** and **houses** lists containing five values each).

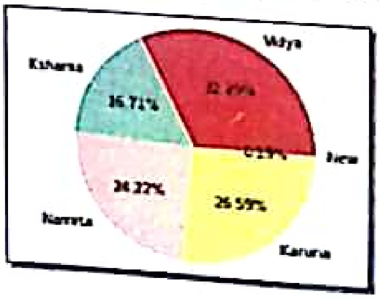
We need a sequence containing five colors for five values of list **contri**, so we created a sequence **colr** containing five color names (you may also give color codes such as 'r', 'b', 'g', etc.)

```
colr = ['red', 'cyan', 'pink', 'yellow', 'silver']
```

Now we issued followed command :

```
plt.pie(contri, labels = houses, colors = colr, autopct ="%2.2f%%" )
```

And the result we got was :



See, the colors to each slices have been given from colr sequence in order (1st color to 1st value,, 2nd color to 2nd value and so on)

Data Visualization with matplotlib's Pyplot Interface

PriP Progress In Python 8.1

The pyplot interface is a popular procedure interface of matplotlib library that is used for mostly 2d plotting with MATLAB like functions. In this PriP session, you shall work with pyplot to explore the plotting capabilities of matplotlib.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 8.1 under Chapter 8 after practically doing it on the computer.

>>>❖<<<

Let us Revise

- *Data visualization basically refers to the graphical or visual representation of information and data using visual elements like charts, graphs, and maps etc.*
- *The **matplotlib** is a Python library that provides many interfaces and functionality for 2D-graphics similar to MATLAB's in various forms.*
- *PyPlot is a collection of methods within matplotlib which allow user to construct 2D plots easily and interactively. PyPlot essentially reproduces plotting functions and behavior of MATLAB.*
- *In order to use pyplot on your computers for data visualization, you need to first import it in your Python environment by issuing **import** command for **matplotlib.pyplot**.*
- *NumPy is a Python library that offer many functions for creating and manipulating arrays, which come handy while plotting.*
- *You need to import numpy before using any of its functions.*
- *Numpy arrays are also called ndarrays.*
- *Some commonly used chart types are line chart, bar chart, pie chart, scatter chart etc.*
- *A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.*
- *You can create line charts by using pyplot's **plot( )** function.*
- *You can change line color, width, line-style, marker-type, marker-color, marker-size in plot( ) function.*
- *Possible line styles are : 'solid' for solid line, 'dashed' for dashed line, 'dotted' for dotted line and 'dashdot' for dashdotted line.*
- *A Bar Graph/Chart is a graphical display of data using bars of different heights.*
- *You can create bar chart using pyplot's bar( ) function.*
- *You can change colors of the bars, widths of the bars in bar( ) function.*
- *Use barh( ) function to create horizontal bar chart.*
- *The pie chart is a type of graph in which a circle is divided into sectors that each represent a proportion of the whole.*
- *You can create a pie chart using pie( ) function. The pie( ) chart plots a single data range.*
- *By default, the pie chart is oval in shape but you can change to circular shape by giving command **<matplotlib.pyplot>.axis("equal")**.*
- *The plot area is known as figure and every other element of chart is contained in it.*

- *The axes can be labelled using **xlabel( )** and **ylabel( )** functions.*
- *The limits of axes can be defined using **xlim( )** and **ylim( )** functions.*
- *The tick marks for axes values can be defined using **xticks( )** and **yticks( )** functions.*
- *The **title( )** function adds title to the plot.*
- *Using **legend( )** function, one can add legends to a plot where multiple data ranges have been plotted, but before that the data ranges must have their **label** argument defined in plot( ) or bar( ) function.*
- *The **loc** argument of **legend( )** provides the location for legend, which by default is 1 or "upper right".*

Solved Problems

1. *What is data visualization ? What is its significance ?*

   Solution. Data visualization is a general term that describes any effort to help people understand the significance of data by placing it in a visual context. In simple words, Data visualization is the process of displaying data/information in graphical charts, figures and bars.

   Patterns, trends and correlations that might go undetected in text-based data can be exposed and recognized easier with data visualization techniques or tools such as line chart, bar chart, pie chart, histogram, scatter chart etc. Thus with data visualization tools, information can be processed in efficient manner and hence better decisions can be made.

2. *What is Python's support for Data visualization ?*

   Solution. Python support data visualizations by providing some useful libraries for visualization. Most commonly used data visualization library is **matplotlib.**

   Matplotlib is a Python library, also sometimes known as the plotting library. The matplotlib library offers very extensive range of 2D plot types and output formats. It offers complete 2D support along with limited 3D graphic support. It is useful in producing publication quality figures in interactive environment across platforms. It can also be used for animations as well.

   There are many other libraries of Python that can be used for data visualization but matplotlib is very popular for 2D plotting.

3. *What is pyplot ? Is it a Python library ?*

   Solution. The *pyplot* is one of the interfaces of *matplotlib* library of Python. This interface offers simple MATLAB style functions that can be used for plotting various types of charts using underlying *matplotlib* library's functionality.

   Pyplot is an interface, *i.e.*, a collection of methods for accessing and using underlying functionality of a library, not a library. The *matplotlib* library has may other interfaces too, along with *pyplot* interface.

4. *Name some commonly used chart types.*

   Solution. Some commonly used chart types are line chart, bar chart, pie chart, scatter chart etc.

5. *Name the functions you will use to create a (i) line chart, (ii) bar chart, (iii) pie chart ?*

   Solution.

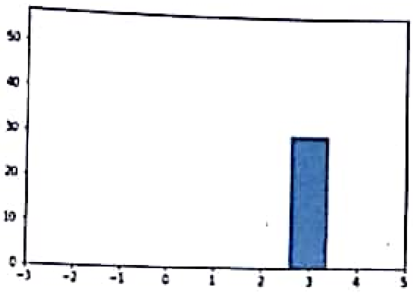
   (*i*) matplotlib.pyplot.plot( )

   (*ii*) matplotlib.pyplot.bar( )

   (*iii*) matplotlib.pyplot.pie( )

6. *Consider the code given below (all required libraries are imported) and the output produced by it. Why is the chart showing one bar only while we are plotting four values on the chart ?*

```
:
a = [3, 6, 9, 12]
b = [30, 48, 54, 48]
plt.xlim(-3, 5)
plt.bar(a,b)
plt.show()
```

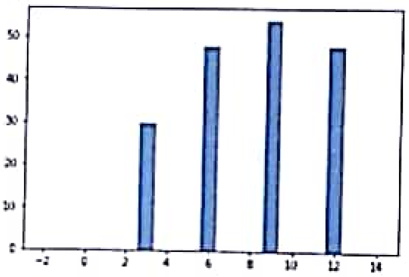


Solution. The given chart is showing a single bar as the limits of x axis have been set as –3 to 5. On this range, only one value from the data range being plotted falls *i.e.*, only a[0] and b[0] fall on this range. Thus only a single value b[0] *i.e.*, 30 is plotted against a[0] *i.e.*, 3.

7. *What changes will you make to the code of previous question so that the bars are visible for all four points. But do keep in mind that the x axis must begin from the point –3.*

Solution. If we change the limits of X-axis so that all the points being plotted fall in the range of limits, all values will show. Thus, we have changes the limits from –3 to 15, in place of –3 to 5.

```
plt.xlim(-3, 15)
plt.bar(a,b)
plt.show()
```



8. *Why is following code not producing any result ? Why is it giving errors ?*

(Note. *all required libraries have been imported and are available)*

```
a = range(10, 50, 12)
b = range(90, 200, 20)
matplotlib.pyplot.plot(a, b)
```

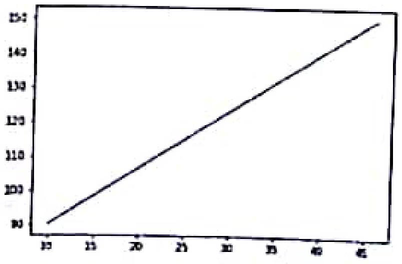
Solution. The above code is producing error because the two sequences being plotted *i.e.*, **a** and **b** do not match in shape. While sequence 'a' contains 4 elements, sequence 'b' contains 6 elements. For plotting, it is necessary that the two sequences being plotted must match in their shape.

9. *What changes will you recommend to rectify the error in previous question's code ?*

Solution. Since both the sequences being plotted must match in their shape, we can achieve this either by adding two elements to sequence a so that it has the same shape as sequence **b** (*i.e.*, 6 elements) or by removing two elements from sequence b so that it matches the shape of sequence a (*i.e.*, 4 elements).

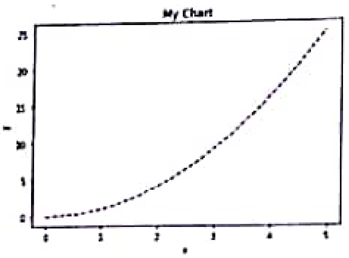
For instance,

```
a = range(10, 50, 12)
b = range(90, 160, 20)
matplotlib.pyplot.plot(a, b)
```

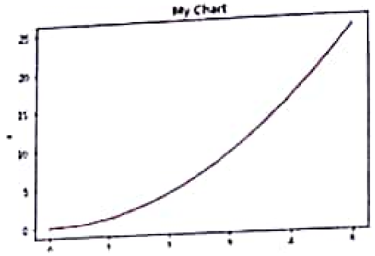


10. *Consider the two charts given below :*

(a)



(b)



Which of these charts has been produced by following code ?

```
plt.plot(x, y, 'r--')
plt.xlabel('x')
plt.ylabel('y')
plt.title('My Chart')
```

Solution. Chart (*a*) has been produced by above code as the linestyle is "--"

11. *Write code to produce the chart shown in q 10 b.*

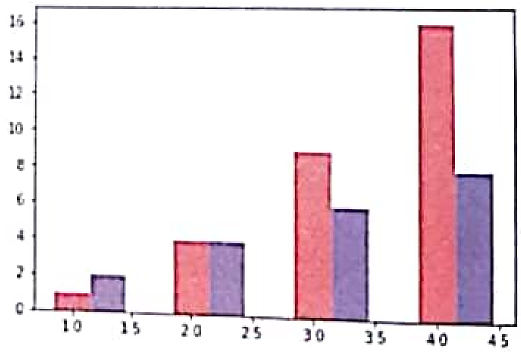
Solution.

```
plt.plot(x, y, 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.title('My Chart')
```

12. *Given an ndarray p as ([1, 2, 3, 4]). Write code to plot a bar chart having bars for p and p**2 (with red color) and another bar for p vs p*2 (with blue color). (assume that libraries have been imported)*

Solution.

```
plt.bar(p, p**2, color = 'r', width = 0.3)
plt.bar(p+0.3, p*2, color = 'b', width = 0.3)
```

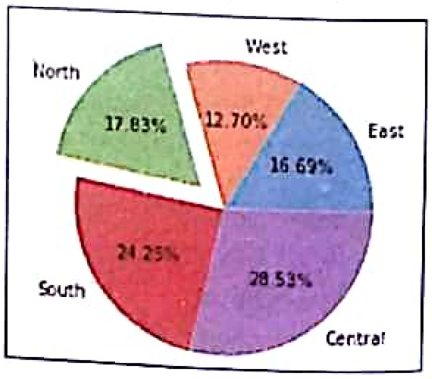


13. *Write to create a pie for sequence con = [23.4, 17.8, 25, 34, 40] for Zones = ['East', 'West', 'North', 'South', 'Central'].*

- Show North zone's value exploded
- Show % contribution for each zone
- The pie chart should be circular.

Solution.

```
import matplotlib.pyplot as plt
con = [23.4, 17.8, 25, 34, 40]
Zones = ['East', 'West', 'North', 'South', 'Central']
plt.axis("equal")
plt.pie(con, labels = Zones, explode = [0, 0, 0.2, 0, 0], autopct = "%1.2f%%")
plt.show()
```



9

Data Structures – I : Linear Lists

In This Chapter

9.1 INTRODUCTION

The computer system is used essentially as data manipulation system where '*Data*' are very important thing for it. The data can be referred to in many ways *viz.* data, data items, data structures etc. The *data* being an active participant in the organizations' operations and planning, data are aggregated and summarized in various meaningful ways to form *information*. The *data* must be *represented, stored, organized, processed* and *managed* so as to support the user environment. All these factors very much depend upon the way data are aggregated. The *Data structures* are an effective and reliable way to achieve this.

First part of this chapter introduces data structures, basic terminology used for them, and different commonly used data structures.

The *second* part of this chapter talks about list data structures. You have learnt about list data type in your previous class, but here you shall see them in data structure avatar.

9.2 ELEMENTARY DATA REPRESENTATION

Elementary representation of data can be in forms of *raw data, data item, data structures.*

- **Raw data** are raw facts. These are simply values or set of values.
- **Data item** represents single unit of values of certain type.

Data items are used as per their associated data types.

DATA STRUCTURE

A *Data Structure* is a named group of data of different data types which is stored in a specific way and can be processed as a single unit. A data structure has well-defined operations, behaviour and properties.

While designing data structures, one must determine the logical picture of the data in a particular program, choose the representation of the data, and develop the operations that will be applied on.

Data Type vs. Data Structure

A *Data type* defines a set of values along with well-defined operations stating its input-output behaviour *e.g.*, you cannot put decimal point in an integer or two strings cannot not be multiplied etc.

On the other hand, a *Data structure* is a physical implementation that clearly defines a way of storing, accessing, manipulating data stored in a data structure. The data stored in a data structure has a specific work pattern *e.g.*, in a stack, all insertions and deletions take place at one end only.

For example, Python supports **lists** as a collection data type which can store heterogeous types of elements, but when we implement *List as a data structure*, its behaviour is clearly predecided *e.g.*, it will store all elements of same type ; for searching, list elements will be presorted and so forth.

9.3 DIFFERENT DATA STRUCTURES

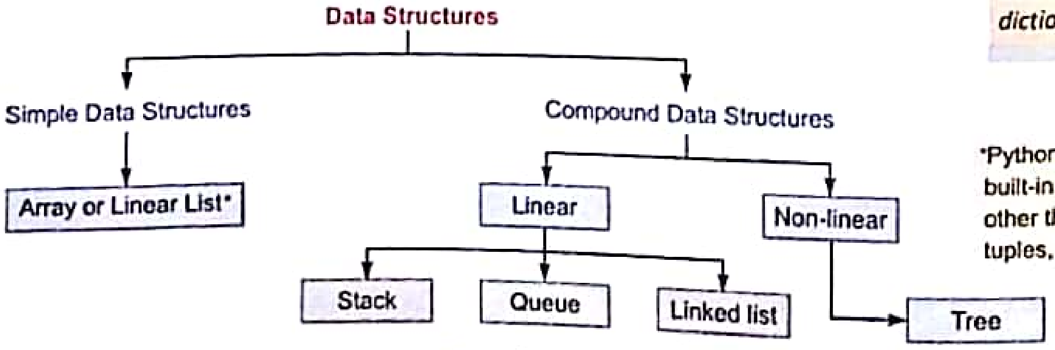
Data Structures are very important in a computer system, as these not only allow the user to combine various data types in a group but also allow processing of the group as a single unit thereby making things much simpler and easier. The data structures can be classified into following *two* types :

1. ***Simple Data Structures***. These data structures are normally built from primitive data types like integers, reals, characters, boolean. Following data structures can be termed as simple data structures :
   - Array or Linear Lists
2. ***Compound Data Structures***. Simple data structures can be combined in various ways to form more complex structures called *compound data structures*. Compound data structures are classified into following *two* types :
   - *Linear data structures*. These data structures are single level data structures. A data structure is said to be linear if its elements form a sequence. Following are the examples of linear data structures : (*a*) Stack (*b*) Queue (*c*) Linked List
   - *Non-Linear data structures*. These are multilevel data structures. Example of non-linear data structure is *Tree*.

NOTE

Python's built-in linear data structures are : *lists, tuples, dictionaries, sets.*

Following figure (Fig. 9.1) shows all the data structures.



*Python offers many linear built-in data structures other than lists – tuples, sets and dictionaries

Figure 9.1 Different Data Structures

9.3.1 Linear Lists Arrays

Linear Lists or Arrays refer to a named list of a finite number *n* of **similar data elements.** Each of the data elements can be referenced respectively by a set of consecutive numbers, usually 0, 1, 2, 3,...*n*. If the name of a linear list of 10 elements is LIL, then its elements will be referenced as shown : LIL[0] , LIL[1] , LIL[2] , LIL[3] , ... LIL[9]

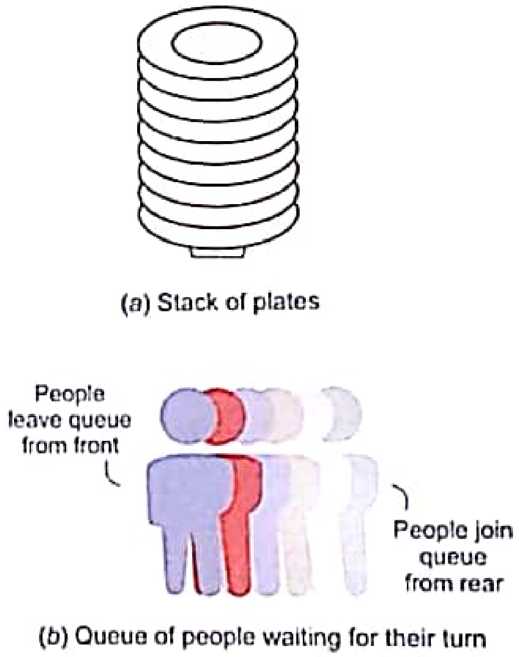
Arrays can be *one dimensional, two dimensional* or *multi dimensional*. In Python, arrays are implemented through List data types as *Linear Lists* or through NumPy arrays.

9.3.2 Stacks

Stacks data structures refer to the lists stored and accessed in a special way, where LIFO (*Last In First Out*) technique is followed. In Stacks, insertions and deletions take place only at one end, called the top. Stack is similar to a stack of plates as shown in Fig. 9.2(*a*). Note that plates are inserted or removed only from the top of the stack.

9.3.3 Queues

Queues data structures are FIFO (*First In First Out*) lists, where insertions take place at the "rear" end of the queues and deletions take place at the "front" end of the queues. Queue is much the same as a line of people [shown in Fig. 9.2(*b*)] waiting for their turn to vote. First person will be the first to vote and a new person can join the queue, at the rear end of it.



(*a*) Stack of plates

(*b*) Queue of people waiting for their turn

Figure 9.2 A stack and a queue.

9.3.4 Linked Lists

Linked lists are special lists of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a *node*, which has two parts. The *INFO* part which stores the information and the reference-pointer part, which points to *i.e.*, stores the reference of the next element. Figure 9.3 shows both types of lists (singly linked lists and doubly linked lists).

In singly linked lists, nodes have one reference-pointer (*next*) pointing to the next node, whereas nodes of doubly linked lists have two reference-pointers (*prev* and *next*). *Prev* points to the previous node and *next* points to the next node in the list.

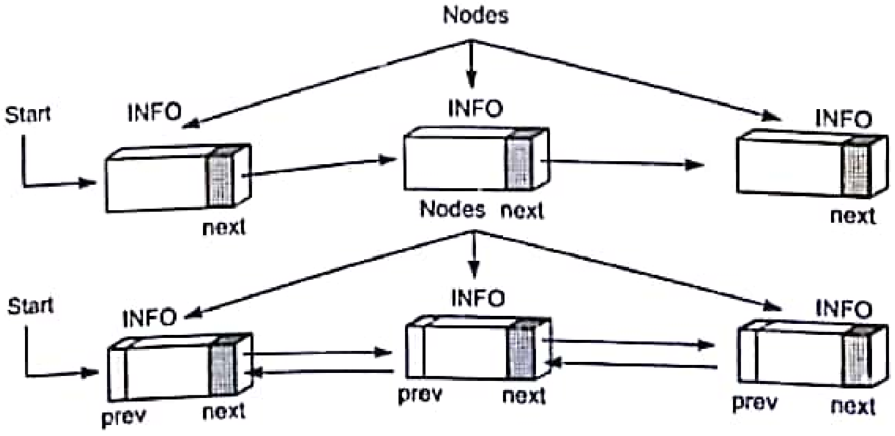


Figure 9.3 Singly and doubly linked lists.

9.3.5 Trees

Trees are multilevel data structures having a hierarchical relationship among its elements called *nodes. Topmost node is called the root of the tree and bottommost nodes are called leaves of the tree.*

Each of the nodes has some reference-pointers, pointing to (*i.e.*, storing the reference of) the nodes below it.

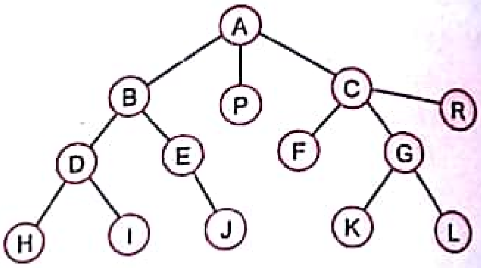


Figure 9.4 Trees.

9.4 OPERATIONS ON DATA STRUCTURES

The basic operations that are performed on data structures are as follows :

1. *Insertion.* Insertion means addition of a new data element in a data structure.
2. *Deletion.* Deletion means removal of a data element from a data structure. The data element is searched for before its removal.
3. *Searching.* Searching involves searching for the specified data element in a data structure.
4. *Traversal.* Traversal of a data structure means processing all the data elements of it, one by one.
5. *Sorting.* Arranging data elements of a data structure in a specified order is called sorting.
6. *Merging.* Combining elements of two similar data structures to form a new data structure of same type, is called *merging.*

9.5 LINEAR LISTS

A linear data structure is that whose elements form a sequence. When elements of linear structures are homogeneous and are represented in memory by means of sequential memory locations, these linear structures are called *arrays*. Linear Lists or **Arrays** are one of the simplest data structures and are very easy to traverse, search, sort etc. *An array stores a list of finite number (n) of homogeneous data elements (i.e., data elements of the same type)*. The number *n* is called length or size or range of a linear list. When upper bound and lower bound of a linear list are given, its size is calculated as follows :

Linear list size (length) = *UB* – *LB* + 1

(UB - *Upper Bound*, LB - *Lower Bound*)

For instance, if a linear list or an array has elements numbered as –7, – 6, – 5.....0, 1, 2, ...15 then its UB is 15 and LB is –7 and array length is

$$= 15 - (-7) + 1$$
$$= 15 + 7 + 1 = 23$$

LINEAR LIST

A linear list is a sequence of $n \geq 0$ elements, E_1, E_2, ... E_n where each element E_i has the same data type T.

9.6 LINEAR LIST DATA STRUCTURE

You have learnt about list datatype in class XI, but when we talk about lists as data structures, it means that all its implementation functions along with storage details (*e.g.*, size) are clearly defined at one place. In the following lines, we shall first talk about various operations performed on linear list and then finally have a complete program implementing linear lists. Here, please note that we shall implement linear list data structure based on what you have learnt in lists in class XI.

9.6.1 Searching in a Linear List

There are many different searching algorithms : *linear search* and *binary search.*

In linear search, each element of the array/linear list is compared with the given *Item* to be searched for, one by one.

Linear Search

In linear search, each element of the array/linear list is compared with the given *Item* to be searched for, one by one. This method, which traverses the array/linear list sequentially to locate the given *Item*, is called *linear search* or *sequential search.*

Algorithm *Linear Search in Linear List*

```
#Initialise counter by assigning lower bound value of the linear list
Step 1.  Set ctr = L                          # L (Lower bound) is 0 (zero).
            # Now search for the ITEM
Step 2.     Repeat steps 3 through 4 until ctr > U.   #U (Upper bound) is size-1
Step 3.     IF AR[ctr] == ITEM then
            {   print("Search Successful")
                print(ctr, "is the location of", ITEM)
                break                          # go out of loop
            }
Step 4.     ctr = ctr + 1
            # End of Repeat
Step 5.     IF ctr > U then
                print("Search Unsuccessful !")
Step 6.     END
```

The above algorithm searches for ITEM in linear list AR with lower bound *L* and upper bound *U*. As soon as the search is successful, it jumps out of the loop (*break* statement), otherwise continues till the last element.

Program 9.1 Linear Searching in an array (linear list)

```
# linear search
def Lsearch(AR, ITEM) :
    i = 0
    while i < len(AR) and AR[i] != ITEM :
        i += 1
    if i < len(AR) :
        return i
    else :
        return False

# ---- main ----
N = int(input("Enter desired linear-list size (max. 50)... "))
print("\nEnter elements for Linear List\n")
AR = [0] * N                          # initialize List of size N with zeros
```

```
for i in range(N) :
    AR[i] = int(input("Element" + str(i)+ ":"))
ITEM = int(input("\nEnter Element to be searched for ..."))
index = Lsearch(AR, ITEM)
if index :
    print("\nElement found at index :", index, ", Position : ",(index + 1))
else :
    print("\nSorry!! Given element could not be found.\n")
```

Sample run of above code :

```
Enter desired linear-list size (max. 50) ... 7
Enter elements for Linear List
Element 0 :    88
Element 1 :    77
Element 2 :    44
Element 3 :    33
Element 4 :    22
Element 5 :    11
Element 6 :    10
Enter Element to be searched for ... 11
Element found at index : 5, Position : 6
Enter Element to be searched for ... 78
Sorry!! Given element could not be found.
```

NOTE

In this chapter, we are alternatively using words linear-lists or arrays. All linear-lists being considered here are assumed to store homogenous elements, just like arrays.

The above program reads a linear-list and asks for the item to be searched for. Then it calls a function called **Lsearch( )** which receives linear-list and search-item as parameters. It then searches for given item in the passed linear-list. If the item is found, then it returns the *index* of found element otherwise it returns *False*.

The above search technique will prove the worst, if the element to be searched is one of the last elements of the linear list as so many comparisons would take place and the entire process would be time-consuming. *To save on time and number of comparisons, binary search is very useful.*

Binary Search

This popular search technique searches the given *ITEM* in minimum possible comparisons. The *binary search* requires the array, to be scanned, must be sorted in any order (for instance, say ascending order). In binary search, the *ITEM* is searched for in smaller *segment* (nearly half the previous segment) after every stage. For the first stage, the segment contains the entire array.

To search for *ITEM* in a sorted array (in *ascending order*), the *ITEM* is compared with *middle element* of the segment (*i.e.*, in the entire array for the first time). If the *ITEM* is more than the middle element, latter part of the segment becomes new segment to be scanned ; if the *ITEM* is less than the *middle element*, former part of the segment becomes new segment to be scanned. The same process is repeated for the new segment(s) until either the *ITEM* is found (search successful) or the segment is reduced to the single element and still the ITEM is not found (search unsuccessful).

NOTE

Binary search can work for only sorted arrays whereas linear search can work for both sorted as well as unsorted arrays.

Algorithm *Binary Search in Linear List*

Case I : Array AR[L : U] is stored in *ascending* order

```
# Initialise segment variables
1.  Set beg = L, last = U                    # L is 0 and U is size-1.
2.  while beg <= last, perform steps 3 to 6  # INT( ) is used to extract integer part
3.      mid = INT ( (beg + last)/2)
4.      if AR [mid] == ITEM then
            print("Search Successful")
            print(ITEM, "found at", mid)
            break                            # go out of the loop
5.      if AR[mid] < ITEM then
            beg = mid + 1
6.      if AR[mid] > ITEM then
            last = mid – 1
    # End of while
7.  if beg ≠ last
        print("Unsuccessful Search")
8.  END.
```

Case II : Array AR[L : U] is stored in *descending* order

```
# Initialize
:
if AR[mid] = = ITEM then
:
if AR[mid] < ITEM then
    last = mid – 1
if AR[mid] > ITEM then
    beg = mid + 1
:   # Rest is similar to the algorithm in Case I.
```

Program 9.2 Binary Searching in an array.

```
def Bsearch( AR, ITEM) :
    beg = 0
    last = len(AR) - 1
    while (beg <= last) :
        mid = (beg + last )/2
        if (ITEM == AR[mid]) :
            return mid
        elif (ITEM > AR[mid]) :
            beg = mid + 1
        else :
            last = mid-1
    else :
        return False                 #when ITEM not found
```

```
# __main__
N = int(input("Enter desired linear-list size (max. 50)..."))
print("\nEnter elements for Linear List in ASCENDING ORDER\n")
AR = [0] * N                    # initialize List of size N with zeros
for i in range(N) :
    AR[i] = int (input("Element" + str(i)+" :"))
ITEM = int(input("\nEnter Element to be searched for ..."))
index = Bsearch(AR, ITEM)
if index :
    print("\nElement found at index : ", index, ", Position : ", (index + 1))
else :
    print("\nSorry!! Given element could not be found.\n")
```

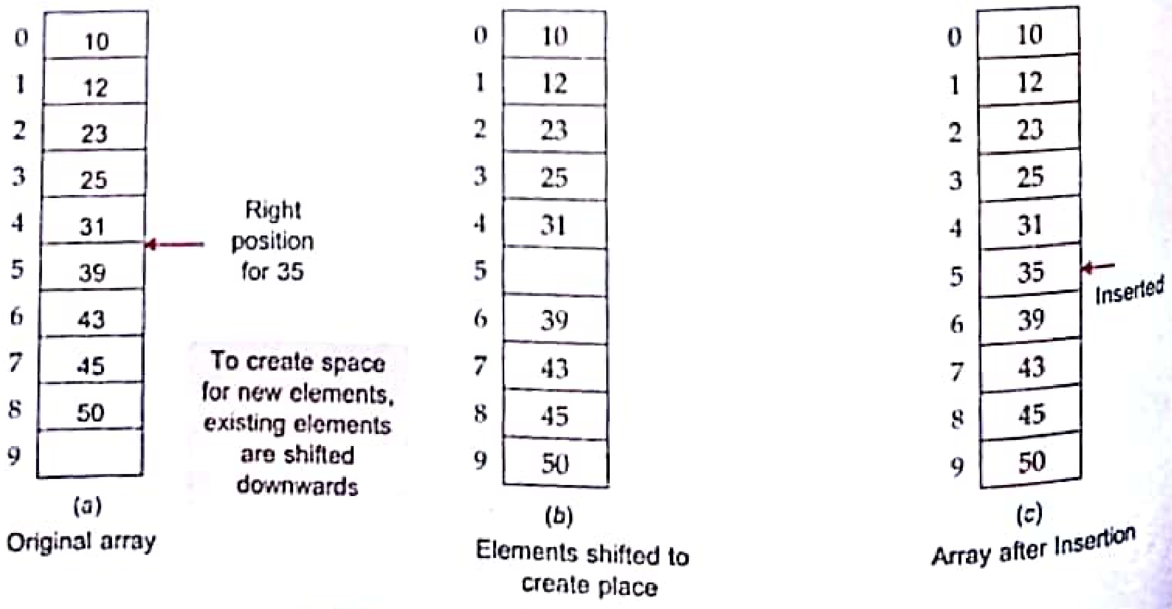
Sample run of the above code as shown on the right.

```
Enter desired linear-list size (max. 50)... 8
Enter elements for Linear List in ASCENDING ORDER
Element 0 :   11
Element 1 :   15
Element 2 :   18
Element 3 :   21
Element 4 :   23
Element 5 :   25
Element 6 :   27
Element 7 :   29
Enter Element to be searched for ... 29
Element found at index : 7, Position :  8
```

9.6.2 Insertion in a Linear List

Insertion of new element in array can be done in *two* ways : (*i*) if the array is unordered, the new element is inserted at the end of the array, (*ii*) if the array is sorted then new element is added at appropriate position without altering the order and to achieve this, rest of the elements are shifted.

For instance, 35 is to be added in an array as shown in Fig. 9.5(*a*).



(*a*) Original array

(*b*) Elements shifted to create place

(*c*) Array after Insertion

Figure 9.5 Insertion in a sorted array.

If the array is already full, then insertion of an element into its results into *OVERFLOW*.

Algorithm *Insertion in Linear List*

```
# First the appropriate position for ITEM is to be determined i.e., if the
appropriate position is I+ 1 then AR[ I ] <= ITEM <= AR[I + 1], LST specifies
maximum possible index in array, u specifies index of last element in Array
1.    ctr = L                                        # Initialise the counter
2.    If LST = U then
          Print("Overflow :")
          Exit from program
3.    if AR[ctr] > ITEM then
          pos = 1
      else
      {
4.        while ctr < U perform steps 5 and 6
5.            if AR[ctr] <= ITEM and ITEM <= AR[ctr + 1] then
                  pos = ctr + 1
                  break
6.            ctr = ctr + 1
7.        if ctr = U then
              pos = U + 1
      }                                              # end of if step 3
      # shift the elements to create space
8.    ctr = U                                        # Initialise the counter
9.    while ctr >= pos perform steps 10 through 11
10.   {   AR[ctr + 1 ] = AR[ctr]
11.       ctr = ctr - 1
      }
12.   AR[pos] = ITEM                                 # Insert the element
13.   END.
```

You have just now read the traditional algorithm of inserting an element in a sorted list, given above. If you notice, the above algorithm involves shifting of elements. Shifting of elements is an expensive operation in a programming language. Expensive in the sense that it consumes much of CPU time.

To understand this, consider the following example :

Given array is :

0	1	2	3	4	5	6
22	33	55	77	88	99	100

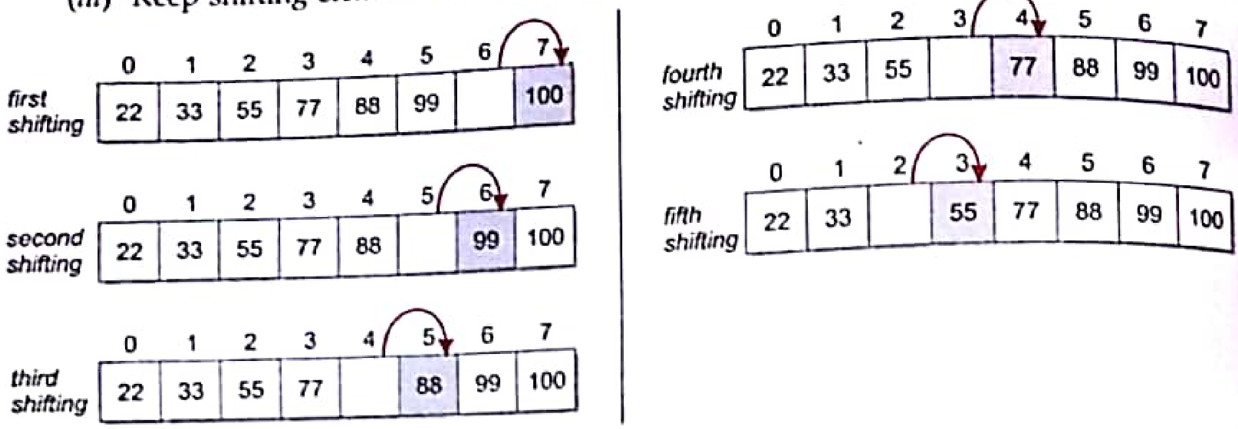
Element to be inserted : 44

(i) Determine the appropriate position for new element : it is **index 2**, after element 33

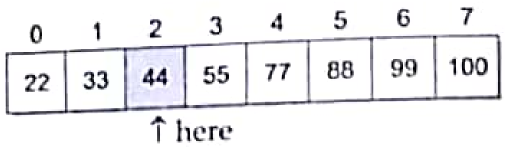
(ii) Create space in the end of array for one element.

0	1	2	3	4	5	6	7
22	33	55	77	88	99	100	

(*iii*) Keep shifting elements from the right end till index 2 *i.e.*,



(*iv*) Now insert the new element :



As you can make out that insertion of a new element requires the shifting of elements to make room for new element.

Shifting of elements is an expensive process and should be avoided if a better algorithm is available. Python makes available a better algorithm called **bisect**, available in **bisect** module. So, if you use following statement after importing bisect module :

```
bisect.insort(list, <newelement>)
```

The **insort( )** function of bisect module inserts an item in the sorted sequence, keeping it sorted. The above function offers a better algorithm to insert in a sorted sequence.

But one thing that you must remember is that the **bisect** module works on a sequence **arranged in ascending order only**. The **bisect** module also offers another function **bisect( )** that returns the appropriate *index* where the new item can be inserted to keep the order maintained, *e.g.*,

```
bisect.bisect(list, <element>)
```

will give you the index where the new element should be inserted.

In the following lines we are giving two programs for insertion of an element — **program 9.3** that uses traditional algorithm given above by finding position and shifting elements and **program 9.4** that uses efficient **bisect** algorithm of Python **bisect** module.

Program **9.3** Inserting an element in a sorted array using traditional algorithm.

```
def FindPos (AR, item) :
    size = len(AR)
    if item < AR[0] :
        return 0
    else :
        pos = -1
        for i in range(size -1) :
            if (AR[i] <= item and item < AR[i + 1] ) :
                pos = i + 1
                break
        if (pos == -1 and i <= size -1 ):
            pos = size
        return pos

def Shift( AR, pos ):
    AR.append(None)                  # add an empty element at the end
    size = len (AR)

    i = size -1
    while i >= pos :
        AR[i] = AR[i-1]
        i = i-1
# __main__
myList = [10, 20, 30, 40, 50, 60, 70]
print("The list in sorted order is")
print(myList)
ITEM = int(input("Enter new element to be inserted :"))
position = FindPos( myList, ITEM)
Shift (myList, position)
myList[position] = ITEM
print("The list after inserting", ITEM, "is")
print(myList)
```

The output produced by above program is :

```
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter new element to be inserted : 80
The list after inserting 80 is
[10, 20, 30, 40, 50, 60, 70, 80]
```

Following program uses bisect module of Python for insertion of new element in a sorted array.

Program 9.4 Insertion in sorted array using bisect module.

```
import bisect
myList = [10, 20, 30, 40, 50, 60, 70]
print("The list in sorted order is")
print(myList)
```

```
ITEM = int(input("Enter new element to be inserted :"))

ind = bisect.bisect( myList, ITEM )
bisect.insort (myList, ITEM )
```

The bisect( ) function would return the correct index value for ITEM. And the insort( ) would insert the ITEM in the list myList maintaining the order of the elements.

```
print(ITEM, "inserted at index", ind)
print("The list after inserting new element
is")
print(myList)
```

The output produced by above program as shown on the right.

```
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter new element to be inserted : 45
45 inserted at index 4
The list after inserting new element is
[10, 20, 30, 40, 45, 50, 60, 70]
```

IMPORTANT

But one thing, you must be aware of, which is that the **bisect module's functions work with sequences arranged in ascending order** of their elements. In order to insert an element in a descending order sequence, you may tweak it as listed below :

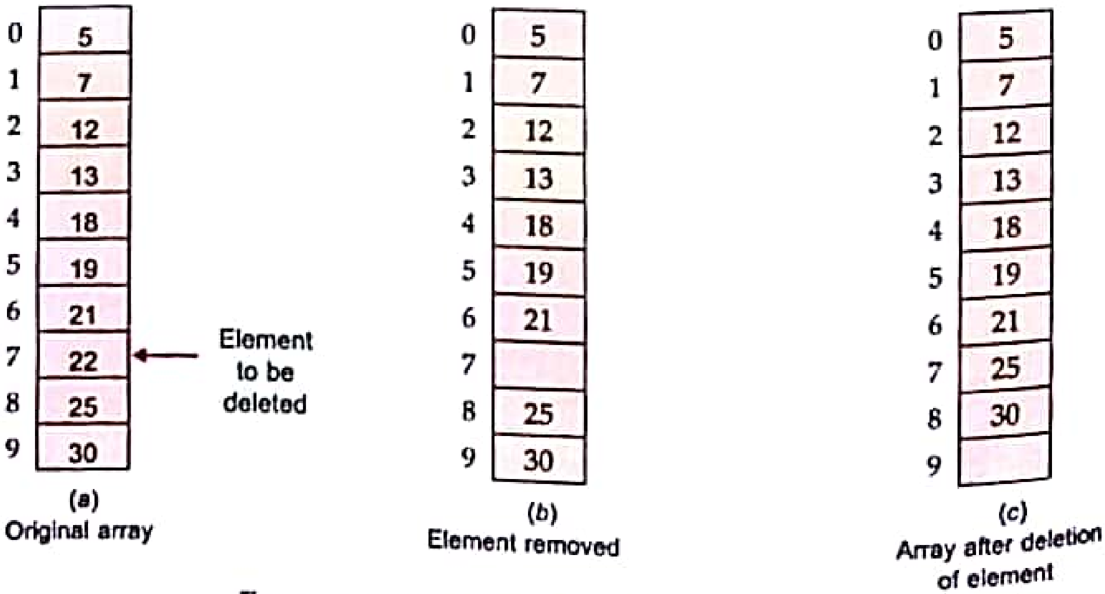
- firstly reverse the list as follows to change it to *ascending order* list :

  `<list>.reverse( )`
- Add the new element using bisect module as explained above.
- Reverse the list again so that it is back to descending order.

```
>>> l2 = [80, 70, 60, 50, 40, 30, 20, 10]
>>> l2.reverse()
>>> l2
[10, 20, 30, 40, 50, 60, 70, 80]
>>> bisect.insort(l2, 45)
>>> l2.reverse()
>>> l2
[80, 70, 60, 50, 45, 40, 30, 20, 10]
>>>
```

9.6.3 Deletion of an Element from a Sorted Linear List

The element to be deleted is first searched for in the array using one of the search techniques *i.e.,* either *linear search* or *binary search*. If the search is successful, the element is removed and rest of the elements are shifted so as to keep the order of array undisturbed. Let us consider element 22 is to be deleted from array X shown in Fig. 9.6(*a*).



(a) Original array

(b) Element removed

(c) Array after deletion of element

Figure 9.6 Deletion of an element in array *X*.

If the elements are shifted downwards or towards right side, then unused elements (free spaces) are available in the beginning of the fixed-size array otherwise free space is available at the end of the array. However, if you are implementing arrays through lists, then there would not be any such issue as Python lists are very flexible and can grow/shrink easily.

Algorithm *Deletion in Linear List*

Considering that *ITEM*'s search in array AR is successful at location pos

Case I *Shifting upwards (or in left side)*

```
# Initialise counter
1.  ctr = pos
2.  while ctr < U perform steps 3 and 4
3.      AR[ctr] = AR[ctr + 1]
4.      ctr = ctr + 1
# End of while
```

Case II *Shifting downwards (or in right side)*

```
1.  ctr = pos
2.  while ctr > 1 perform steps 3 and 4
3.      AR[ctr] = AR[ctr – 1]
4.      ctr = ctr – 1
# End of while
```

Python provides a remove( ) method that itself takes care of shifting of elements. You have worked with remove( ) method and del<item> statements of lists in class XI. The same method/statement you can use for removing an element from a sorted array/list.

Following program deletes an element from a sorted array.

Program 9.5 Deletion of an element from a sorted linear list.

```
def Bsearch( AR, ITEM) :
    beg = 0
    last = len(AR) - 1
    while (beg <= last) :
        mid = (beg + last)/2
        if(ITEM == AR[mid]) :
            return mid
        elif (ITEM > AR[mid] ) :
            beg = mid + 1
        else :
            last = mid - 1
    else :                          # else of loop
        return False                # when ITEM not found
```

```
# __main__
myList = [10, 20, 30, 40, 50, 60, 70]
print("The list in sorted order is")
print(myList)
ITEM = int(input("Enter element to be deleted :"))
position = Bsearch(myList, ITEM)
if position :
    del myList[position]
    print("The list after deleting", ITEM, "is")
    print(myList)
else :
    print("SORRY! No such element in the list")
```

Determining the position of element to be deleted

Statement deleting the element at index position

The output produced by above code is :

```
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter element to be deleted : 45
SORRY! No such element in the list
>>> ====================== RESTART ======================
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter element to be deleted : 40
The list after deleting 40 is
[10, 20, 30, 50, 60, 70]
```

The above program firstly reads an array and then asks for the element to be deleted. It then searches for the element's position in the array, using Binary search. And if found, then the element at determined position is deleted.

9.6.4 Traversal of a Linear List

Traversal in one-dimensional arrays involves processing of all elements (*i.e.*, from very first to last element) one by one. For instance, traversal of array shown in Fig. 9.6(*a*) would process in the following order :

X[0], X[1], X[2], X[3].......X[10]

Algorithm *Traversal in Linear List*

```
# Considering that all elements have to be printed
1.  ctr = L                          # Initialising the counter
2.  Repeat steps 3 through 4 until ctr > U
3.      print(AR[ctr])
4.      ctr = ctr + 1
    # End of Repeat
5.  END.
```

Program 9.6 Traversing a linear list.

```
def traverse(AR) :
    size = len(AR)

    for i in range(size) :
        print(AR[i], end = ' ')
# __main__
size = int (input("Enter the size of Linear list to be input :"))
AR = [None] * size                    # create an empty list of the given size
print("Enter elements for the Linear List")
for i in range(size) :
    AR[i] = int (input("Element" + str(i) + ":"))
print("Traversing the list :")
traverse(AR)
```

Loop traversing the elements in the linear list

traversal function invoked

```
Enter the size of Linear list to be input : 6
Enter elements for the Linear List
Element 0 : 12
Element 1 : 23
Element 2 : 34
Element 3 : 45
Element 4 : 56
Element 5 : 67
Traversing the list :
12 23 34 45 56 67
```

PriP **SEARCHING, INSERTION, DELETION IN THE LIST** Progress In Python 9.1

This PriP session aims at practice of these concepts : searching, insertion and deletion of elements in an array.

⋮

>>>❖<<<

9.6.5 Sorting a Linear List

Sorting refers to arranging elements of a list in ascending or descending order. There are many sorting algorithms that you can apply to sort elements of a linear list. You have learnt about two sorting algorithms, *bubble sort* and *insertion sort*, in your previous class. You can choose any of these algorithms to sort your linear list.

We are not giving below any program to sort a linear list as you already know this. In fact, we expect you to add a sorting function to following program (create a need) that implements a linear list data structure. Before we do that, let us talk about an efficient way of creating lists– **List comprehensions.** Data structures always demand efficient implementation methods, thus, it is important for you to know **List comprehensions** – an efficient way of creating lists.

9.6.6 List Comprehensions

You know already how to create lists. You have been creating lists either by directly assigning comma separated values to a name or by employing a for loop to do this, *e.g.*,

```
lst = [2, 4, 6, 8, 10]     Or     lst = [ ]
                                  for i in range(1,6) :
                                  lst.append(i * 2) # lst is now = [ 2, 4, 6, 8, 10]
```

Both above codes create the lists with the same elements.

There is another way of creating lists – in fact, a concise way called **list comprehensions.**

A **list comprehension** is a concise description of a list that shorthands *the list creating for loop* in the form of a single statement.

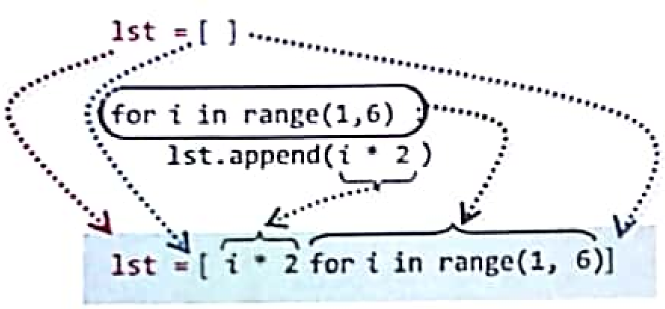
Let us see how it works. The above list creating *for loop* can be written as :

```
lst2 = [ i * 2   for i in range(1, 6) ]          # it is a list comprehension
```

Now if you print *lst2*, it will also be containing the same elements *i.e.*, [2, 4, 6, 8, 10]

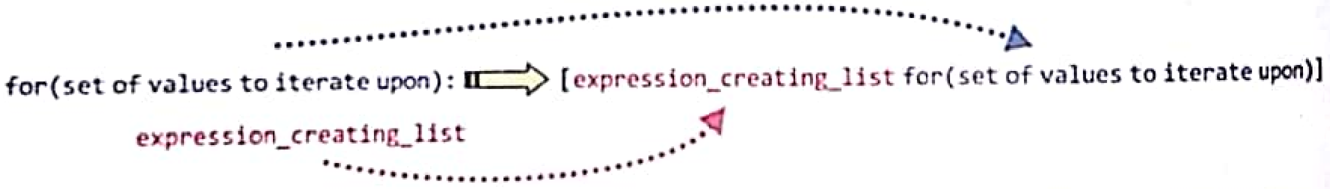
> **NOTE**
>
> A list comprehension is short-hand for a loop that creates a list.
>
> A list comprehension is usually shorter, more readable, and more efficient.

Carefully look at following figure that illustrates how list comprehension is created from above loop :



```
lst = [ ]
for i in range(1,6) :
    lst.append(i * 2 )

lst = [ i * 2 for i in range(1, 6)]
```

The above conversion of for loop to list comprehension can be summarized as :

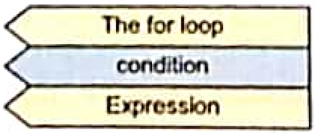


for(set of values to iterate upon): ⟹ [expression_creating_list for(set of values to iterate upon)]

expression_creating_list

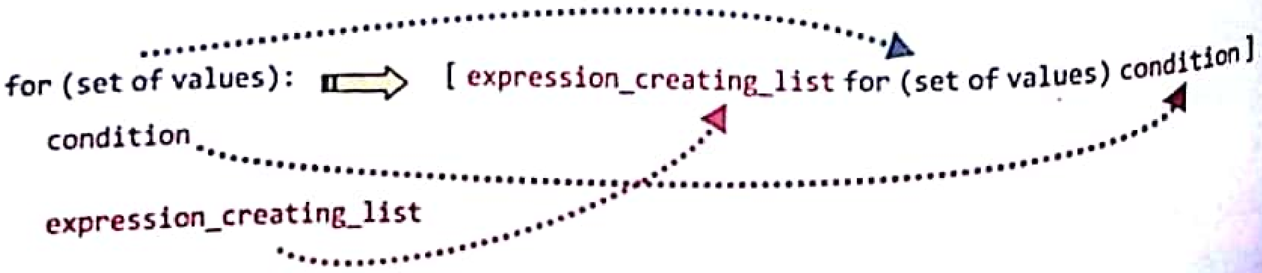
Now let us consider another for loop (more detailed) that is creating a list based on a condition.

```
lst3 = []
for num in range(1, 50):
    if num % 7 == 0 :
        lst3.append(num)
```



As you can see that above loop carries out expression_creating_list only if the given condition is true. The list produced by above code is [7, 14, 21, 28, 35, 42, 49].

To create a list comprehension from above for loop, you need to use following conversion rule:



for (set of values): ⟹ [expression_creating_list for (set of values) condition]

condition

expression_creating_list

So, our list comprehension for above for loop will be :

```
lst3 = [num for num in range(1,50) if num % 7 == 0]
```

Please note that colon (:) is not required with if and else keywords in List Comprehension

Consider another list comprehension.

```
A = [num if num < 5 else num*2 for num in range(2, 9)]
```

The above list comprehension will produce list A as

```
[ 2, 3, 4,  10, 12, 14, 16 ]
```

Values < 5 are stored as it is *Values > 5 are stored through expression num*2*

For more clarity, you can enclose entire if else part in parenthesis, *e.g.*,

```
A = [(num if num < 5 else num * 2) for num in range (2, 9)]
```

Notice again that with if and else we do not give colon in list comprehensions otherwise Python gives error, *e.g.*, following code will give an error.

```
A = [num if num < 5 :    else : num * 2 for num in range(2, 9)]
```

Not to be given in list comprehensions

Consider some more examples given below :

LIST COMPREHENSION

A List Comprehension is a concise description of a list that shorthands *the list creating for loop* in the form of a single statement.

Example 9.1 *Create a list myList with these elements (i) using for loop, (ii) using list comprehension.*

Solution (*i*)

```
myList = []
for i in range(11):
    myList.append( 2 ** i)
```

(*ii*)

```
myList = [2 ** i for i in range(11)]
```

Example 9.2 *Given an input list Vals below, produce a list namely Mul3, using a list comprehension having the numbers from Vals that are multiples of 3.*

```
Vals = [31, 15, 42, 12, 5, 39, 21, 61, 25]
```

Solution

```
Mul3 = [num for num in Vals if num % 3 == 0]
```

The list, *Mul3*, will store [15, 42, 12, 39, 21]

Example 9.3 *Consider the code below. What will the list NL be storing ?*

```
Lst = [('a', 11), ('b', 12), ('c', 13)]
NL = [ n * 3 for (x, n) in Lst if x == 'b' or x == 'c']
```

Solution

```
[36, 39]
```

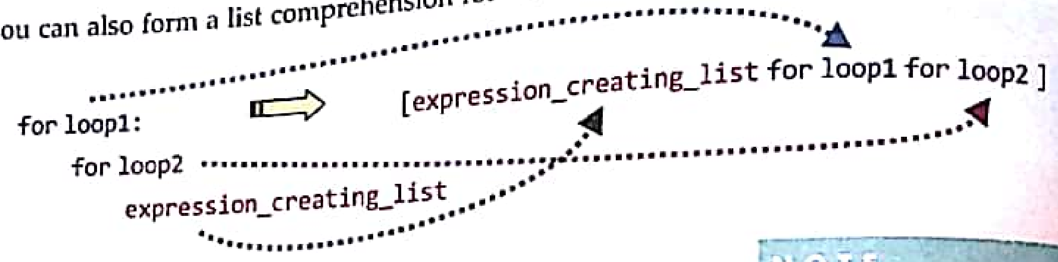
Example 9.4 *Consider the following code. What will the list Res be storing ?*

```
Res = ["Ev"  if i % 2 == 0 else "Od" for i in range(10,20)]
print(Res)
```

Solution

```
['Ev', 'Od', 'Ev', 'Od', 'Ev', 'Od', 'Ev', 'Od', 'Ev', 'Od']
```

You can also form a list comprehension for a nested for loop.



```
for loop1:
    for loop2
        expression_creating_list
```

```
[expression_creating_list for loop1 for loop2 ]
```

For example, consider the following nested loop :

```
result= []
for x in [10, 5, 2]:
    for y in [2, 3, 4]:
        result.append( x ** y)
print(result)
```

It produced result as : [100, 1000, 10000, 25, 125, 625, 4, 8, 16]

NOTE

Please note that list comprehensions work with square brackets only. Try using it with ( ) and Python will give you an error. An assignment question is based on this, check *Type B.*

The list comprehension for above nested loop will be like :

```
result = [ x ** y for x in [10, 5, 2] for y in [2, 3, 4]]
```

So the equivalent code for above code will be :

```
result= []
result = [ x ** y for x in [10, 5, 2] for y in [2, 3, 4]]
print(result)
```

And the result will just be the same.

The nested for loops may have optional condition(s) and the list comprehension will include optional condition just in the same way as you have included conditions in single for loop.

For instance, look at the following list comprehension :

```
[(x,y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
```

Can you write its equivalent nested for loop ? You surely can, I bet ! Please check if mine (given below) is correct ? ☺

NOTE

List comprehensions are although useful but you should not use them when you need to check multiple conditions.

```
L1 = []
for x in range(5) :
    if x % 2 == 0 :
        for y in range(5) :
            if y % 2 == 1 :
                L1.append((x, y))
print(L1)
```

Advantages of List Comprehensions

List comprehensions are considered more Pythonic as they truly represent Python coding style and also offer these advantages :

(*i*) **Code reduction.** A code of 3 or more lines (for loop with or without a condition) gets reduced to a single line of code.

(*ii*) **Faster code processing.** List comprehensions are executed faster than their equivalent for loops for these *two* reasons :

(*a*) Python will allocate the list's memory first, before adding the elements to it, instead of having to resize on runtime.

(*b*) Also, calls *to append( )* function get avoided, reducing function overhead time (*i.e.*, additional time taken to call and return from a function) which may be cheap but add up.

Program 9.5 **Linear List Implementation.** A charity organization conducts camps in various locations of a city. Each camp is located at a location on a specific date. The organization maintains a monthly list of camps planned in this format :

03 New Cly, 18 T Nagar, 25 K Pura

Where first two digits of every entry signify the date on which the camp is to be conducted. Write a program to implement this along with following points :

(*i*) A linear list, planned, stores the camps that are to be conducted. A camp's details are added to **planned** list once NOC is obtained.

(*ii*) As soon as a camp is conducted, its details are moved to list **conducted** and removed from **planned** list.

(*iii*) Each camp servers some people.

(*iv*) The program should be able to provide options for adding to planned list, getting conducted camp's details, searching for a camp, a report of how many camps were conducted so far and how many people were served and display the linear lists **planned** and **conducted**.

```
def addloc(cmp):
    dd = cmp[0:2]
    ln = len(planned)
    if ln == 0:
        planned.append(cmp)
    else:
        last = planned[ln-1]
        if int(dd) >= int(last[0:2]):
            planned.append(cmp)
        else:
            for i in range(ln):
                cp = planned[i]
                if int(dd) <= int(cp[0:2]):
                    planned.insert(i,cmp)
                    break
```

```
def conductCamp(cmp):
    conducted.append(cmp)
    planned.remove(cmp)                  # linear search technique
def search(cmp, lst):
    ln = len(lst)
    for i in range(ln):
        if cmp in lst[i]:
            return lst[i]
    else:
        return False
def Report() :
    lenp = len(planned)
    lenc = len(conducted)
    print("\t R E P O R T")
    print("————————————")
    print("Camps conducted so far:", lenc)
    print("People served so far", ppl)
    print("Camps to be conducted:", lenp)
    print("————————————")
def display():
    print("\nCamps Planned : ", end = ' ')
    for i in planned:
        print(i, end=', ')
    print("...!!")
    print("\nCamps Conducted so far : ", end = ' ')
    for i in conducted:
        print(i, end=', ')
    print("...!!")
#__main__
planned = []
conducted = []
ppl = 0
ch = 0
while (ch != 6):
    print("\t—")
    print("\tMENU")
    print("\t—")
    print("1. Add Camp Location")
    print("2. Camp Conducted")
    print("3. Look for a Camp")
    print("4. Report")
    print("5. Display List")
    print("6. Exit")
    ch = int(input("Enter your choice (1-6):"))
```

```
        if ch == 1 :
            cm = input("Enter Camp location : ")
            dd = input("Enter date of the month (only dd) : ")
            cmp = dd + cm
            addloc(cmp)
        elif ch == 2 :
            cm = input("Camp conducted at location?")
            p = int(input("How many people are served at this camp?"))
            ppl  = ppl + p
            result = search(cm, planned)
            if result == False :
                print("Sorry no such camp in the list")
            else :
                conductCamp(result)
        elif ch == 3:
            cm = input("Enter camp location : ")
            r1 = search(cm, planned)   # result1
            if r1 == False :
                r2 = search(cm, conducted) # result2
                if r2 == False:
                    print("Sorry no such camp in our list")
                else:
                    dd = r2[0:2]
                    print(cm, "was conducted on date", dd, "of this month")
            else :
                dd = r1[0:2]
                print(cm, "camp is to be conducted on date", dd, "of this month")
        elif ch == 4 :
            Report()
        elif ch == 5:
            display()
        elif ch != 6 :
            print("Wrong choice! Enter choice from 1 to 6 only")
    else:
        print("THANK YOU")
```

• Please note above program is based on certain assumptions such that :

(i) dd part is always 2 digits long *i.e.*, enter single digit dates preceeded with a zero *e.g.*, 03, 06 etc.

(ii) there is no check to ensure that location name is not repeated. If you want, you can add this functionality.

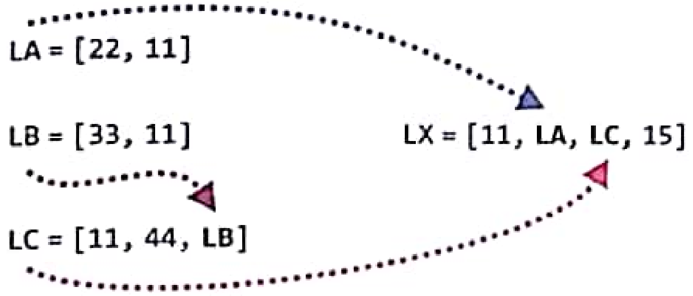
Sample run of above program is as shown below :

```
          MENU
1. Add Camp Location
2. Camp Conducted
3. Look for a Camp
4. Report
5. Display List
6. Exit
Enter your choice (1-6): 1
Enter Camp location : K Pur
Enter date of the month (only dd) : 09

          MENU
:
Enter your choice (1-6): 1
Enter Camp location : D Pur
Enter date of the month (only dd) : 27

          MENU
:
Enter your choice (1-6): 1
Enter Camp location : B Nagar
Enter date of the month (only dd) : 03

          MENU
:
Enter your choice (1-6): 1
Enter Camp location : Y Vihar
Enter date of the month (only dd) : 15

          MENU
:
Enter your choice (1-6): 5
Camps Planned : 03 B Nagar, 09 K Pur, 15 Y
Vihar, 27 D Pur, ...!!
Camps Conducted so far : ...!!

          MENU
:
Enter your choice (1-6): 3
Enter camp location : D Pur
D Pur camp is to be conducted on date 27
of this month
```

```
          MENU
:
Enter your choice (1-6): 4
      R E P O R T
--------------------------
Camps conducted so far: 0
People served so far 0
Camps to be conducted: 4
--------------------------

          MENU
:
Enter your choice (1-6): 2
Camp conducted at location? B Nagar
How many people are served at this camp? 320

          MENU
:
Enter your choice (1-6): 2
Camp conducted at location? K Pur
How many people are served at this camp? 412

          MENU
:
Enter your choice (1-6): 4
      R E P O R T
--------------------------
Camps conducted so far: 2
People served so far 732
Camps to be conducted: 2
--------------------------

          MENU
:
Enter your choice (1-6):6
THANK YOU
```

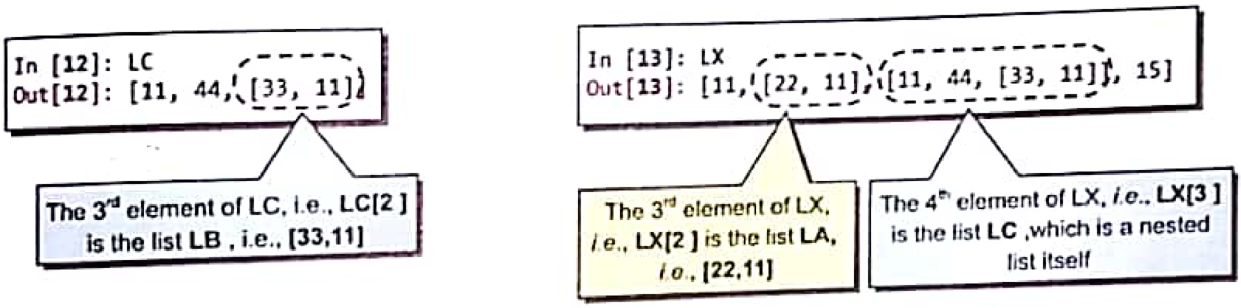
9.7 NESTED/TWO DIMENSIONAL LISTS IN PYTHON

You know that lists are objects that can hold objects of any other types as its elements. Since *lists* are objects themselves, they can also hold other list(s) as its element(s). Carefully go through the code shown below that creates four lists namely LA, LB, LC and LX :



```
LA = [22, 11]

LB = [33, 11]                  LX = [11, LA, LC, 15]

LC = [11, 44, LB]
```

Out of the four lists created above, two are nested lists, *i.e., LC* and *LX* are nested lists as they contain one or more lists as their elements. So when you display or print the contents of lists *LC* and *LX*, it will be like :



```
In [12]: LC
Out[12]: [11, 44, [33, 11]]
```

The 3rd element of LC, i.e., LC[2] is the list LB , i.e., [33,11]

```
In [13]: LX
Out[13]: [11, [22, 11], [11, 44, [33, 11]], 15]
```

The 3rd element of LX, *i.e.*, LX[2] is the list LA, *i.e.*, [22,11]

The 4th element of LX, *i.e.*, LX[3] is the list LC ,which is a nested list itself

So with *LX* as :

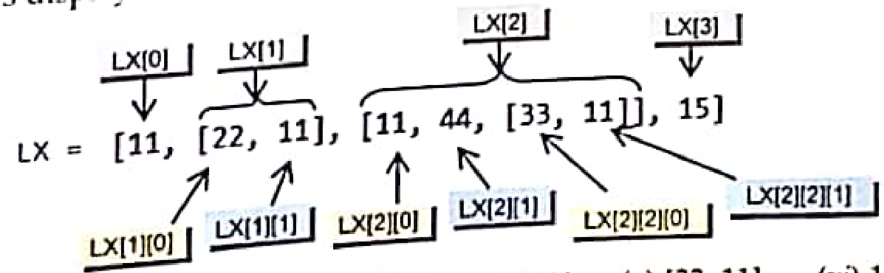
```
LX = [11, [22, 11], [11, 44, [33, 11]], 15]
```

What will be shown if you display :

(*i*) LX[1], (*ii*) LX[2], (*iii*) LX[1][1], (*iv*) LX[2][0], (*v*) LX[2][2], (*vi*) LX[2][2][1] ?

Well, I already knew that you knew it . 😉

So the values displayed are like :



```
LX = [11, [22, 11], [11, 44, [33, 11]], 15]
```

(*i*) [22, 11] (*ii*) [11, 44, [33, 11]] (*iii*) 11 (*iv*) 11 (*v*) [33, 11] (*vi*) 11

NESTED LIST

A list that has one or more lists as its elements is a nested list.

So, you can say that a list that has one or more lists as its elements is a nested list. A two dimensional list is also a nested list. Let us see how.

9.7.1 Two Dimensional Lists

A two dimensional list is a list having all its elements as lists of same shapes, *i.e.*, a two dimensional list is ***a list of lists***, *e.g.*,

```
L1 = [ [1, 2], [9, 8], [3, 4] ]
```

L1 is a two dimensional list as it contains *three lists*, each having same shape, *i.e.*, all are singular lists (*i.e.*, non-nested) with a length of 2 elements. Following representation will make it clearer.

```
L1 = [ [1, 2],
       [9, 8],
       [3, 4] ]
```

You can visualize it as:

	0	1
0	1	2
1	9	8
2	3	4

Each row, column has a value, which is singular value

Regular two-dimensional lists are the nested lists with these properties :

(*i*) All elements of a 2D list have same shape (*i.e.*, same dimension and length)

(*ii*) The length of a 2D list tells about **Number of Rows** in it (*i.e.*, **len(list)**)

(*iii*) The length of single row gives the **Number of Columns** in it (*i.e.*, **len(list[n])**)

Ragged list. A list that has lists with different shapes as its elements is also a 2d list but it is an irregular 2d list, also known as a *ragged list*.

For instance, following list ( ) is a ragged list :

```
L2 = [ [1, 2, 3], [5, 6] ]
```

as its one element has a length as 3 while its second element is a list with a length of 2 elements.

Let us quickly learn how you can use a two dimensional list. Please note that in the following examples, we shall mostly be using regular 2D lists unless specified explicitly.

REGULAR TWO DIMENSIONAL LIST

A regular two dimensional list is a list having lists as its elements and each element-list has the same shape i.e., same number of elements (length).

9.7.1A Creating a 2D List

To create a 2D list by inputting element by element, you can employ a **nested loop** as shown below: one loop for rows and the other loop for reading individual elements of each row.

```
Lst = []
r = int(input("How many rows?"))
c = int(input("How many columns?"))
for i in range(r) :
    row = []                  ← Each row is initialized as an empty list and then
                                elements are added to this row list
    for j in range(c) :
        elem = int(input("Element"+ str(i) +", "+str(j)+ ": "))
        row.append(elem)      ← Integers being added to row list
    Lst.append(row)           ← row list is added as an element of 2d list
print("List created is :", Lst)
```

The sample run of above program is given below :

```
How many rows? 3
How many columns? 3
Element 0, 0: 2
Element 0, 1: 3
Element 0, 2: 4
Element 1, 0: 22
Element 1, 1: 32
Element 1, 2: 43
Element 2, 0: 50
Element 2, 1: 60
Element 2, 2: 70
List created is : [[2, 3, 4], [22, 32, 43], [50, 60, 70]]
```

As you can see that above code has successfully created a regular 2D list having a size of 3×3.

9.7.1B Traversing a 2D List

To traverse a 2D list element by element, you can employ a nested loop as earlier : one loop for rows and another for traversing individual elements of each row (see below).

```
# We have added following nested loop to previous code of 2d list creation
print("List created is :")
print("Lst = [ ")
for i in range(r) :
    print("\t[", end = " ")
    for j in range(c) :
        print( Lst[i][j], end = "  " )
    print("]")
print("\t]")
```

The output produced is like :

```
List created is :
Lst = [
            [ 2   3   4   ]
            [ 22  32  43 ]
            [ 50  60  70 ]
      ]
```

9.7.1C Accessing/Changing Individual Elements in a 2D List

You can access individual elements in a 2D list by specifying its indexes in square brackets, *e.g.*,

To access 3rd row's 3rd element from the 2D list namely *Lst* that we created in previous sections, you will write :

```
Lst[2][2]
```

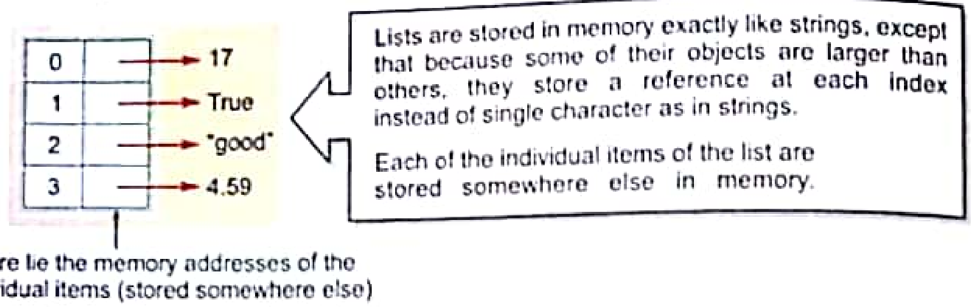
Indexes begin with *0* and go till *n* – 1.

Using the same syntax, you can also change a specific value in a 2D list as lists are mutable types, *i.e.*, following statement will change the 3rd element of 2nd row to 345 :

```
Lst[1][2] = 345
```

9.7.1D How a Two-dimensional List is Stored

You know that Python variables are not like storage containers rather they store or point to the address where a certain value is located. You have read in class XI that lists also store reference (memory location) of each individual item stored in them.



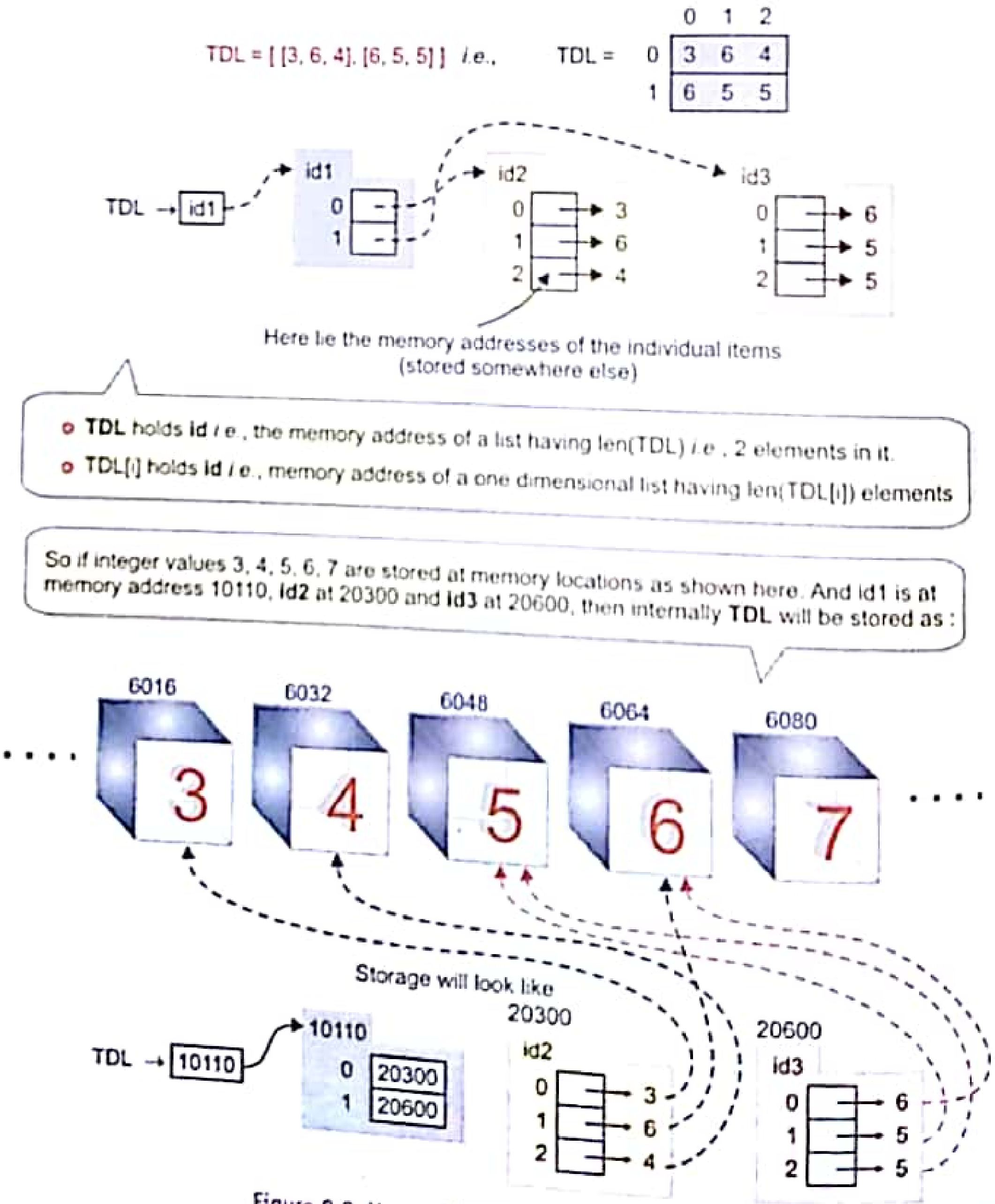
Lists are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.

Each of the individual items of the list are stored somewhere else in memory.

Here lie the memory addresses of the individual items (stored somewhere else)

Figure 9.7 How lists are internally stored.

On the same lines, a 2D list stores references of its element lists at its indexes. Following figure 9.8 will make it clearer.



Here lie the memory addresses of the individual items (stored somewhere else)

- TDL holds id i.e., the memory address of a list having len(TDL) i.e., 2 elements in it.
- TDL[i] holds id i.e., memory address of a one dimensional list having len(TDL[i]) elements

So if integer values 3, 4, 5, 6, 7 are stored at memory locations as shown here. And id1 is at memory address 10110, id2 at 20300 and id3 at 20600, then internally TDL will be stored as :

Figure 9.8 How 2D lists are internally stored.

In the same way as regular 2D lists, ragged lists are also stored (*see* Fig. 9.9)

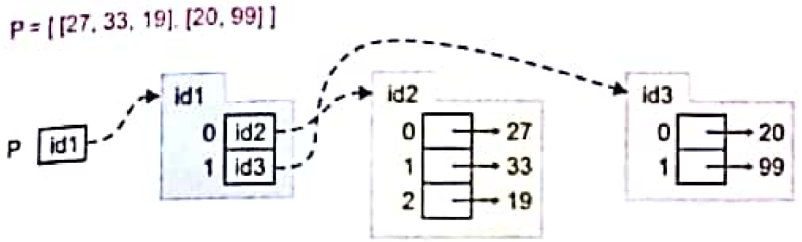


Figure 9.9 Internal Storing of Ragged lists.

NOTE

The regular 2D lists are the lists having lists of same sizes as its elements and that means its rows have equal sizes. On the other hand, the ragged lists are also nested lists with rows of different sizes.

9.7.1E Slices of Two-dimensional Lists

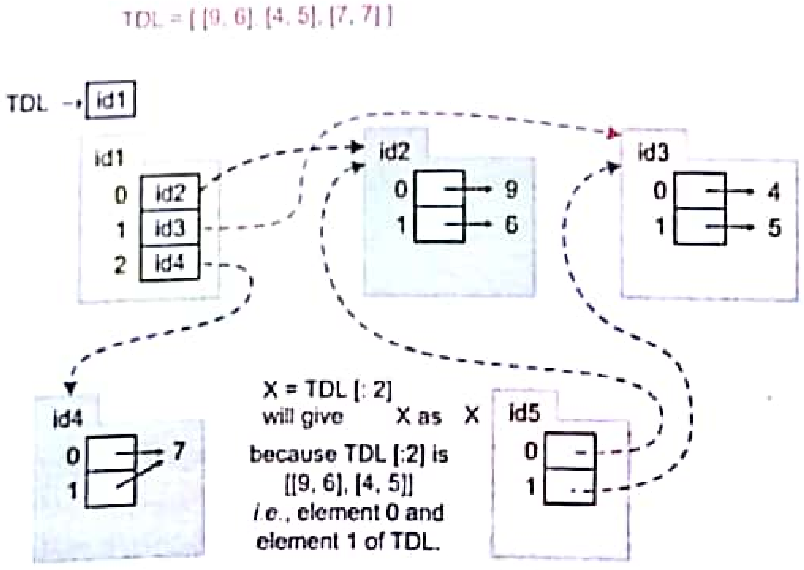
The slicing rules of Python are applicable to 2D lists in the same way as you have applying to lists, tuples and other sequences. In a 2D array, you just need to be always aware that the top level list's elements are lists themselves. Following examples will make it clearer.

```
>>> Lst = [[2, 3, 4], [22, 32, 43], [50, 60, 70], [9, 10, 11]]
>>> Lst[ : 2] ←———————————————— Elements of Lst with index < 2
[[2, 3, 4], [22, 32, 43]]
>>> Lst[2: ] ←———————————————— Elements of Lst with index >= 2
[[50, 60, 70], [9, 10, 11]]
```

Since the slice itself is a list, you can further apply slicing on it :

```
>>> Lst[2: ][ :1]
[[50, 60, 70]]
```

Following figure (Fig. 9.10) illustrates it.



X = TDL [: 2] will give X as X

because TDL [:2] is [[9, 6], [4, 5]] *i.e.*, element 0 and element 1 of TDL.

Figure 9.10 Slicing a Multi-dimensional list.

Check Point 9.1

1. What do you mean by the following terms ?
   (*i*) raw data (*ii*) data item
   (*iii*) data type (*iv*) data structure
2. What do you understand by the following :
   (*i*) simple data structures
   (*ii*) compound data structures
   (*iii*) linear data structures
   (*iv*) non-linear data structures ?
3. When would you go for linear search in an array and why ?
4. State condition(s) when binary search is applicable.
5. Name the efficient algorithm offered by Python to insert element in a sorted sequence.

   State condition(s) when binary search is applicable.
6. What is a list comprehension ?
7. What is a 2D list ?
8. What is a nested list ?
9. Is Ragged list a nested list ?

LET US REVISE

- *A data structure is a named group of data of different data types which can be processed as a single unit.*
- *Simple data structures are normally built from primitive data types.*
- *Simple data structures can be combined in various ways to form compound data structures. The compound data structures may be linear (whose elements form a sequence) and non-linear (which are multi-level).*
- *A Linear list or an array refers to a named list of a finite number n of similar data elements whereas a structure refers to a named collection of variables of different data types.*
- *Stacks are LIFO (Last In First Out) lists where insertions and deletions take place only at one end.*
- *Queues are FIFO (First In First Out) lists where insertions take place at " rear" end and deletions take place at the " front" end.*
- *In linear search, each element of the array is compared with the given Item to be searched for, one by one.*
- *A **List Comprehension** is a concise description of a list that shorthands the list creating for loop in the form of a single statement.*
- *A list that has one or more lists as its elements is a nested list.*
- *A regular two dimensional list is a list having lists as its elements and each element-list has the same shape i.e., same number of elements (length).*
- *A list that has lists with different shapes as its elements is also a 2d list but it is an irregular 2d list, also known as a ragged list.*
- *Internally a 2D list stores references of its element lists at its indexes.*

Solved Problems

1. *What is a Data Structure ?*

   Solution. A data structure is a logical way of organizing data that makes them efficient to use. Different data structures are suited to different types of applications, and some are highly specialized to specific tasks. For instance, stacks are best suited for reversal applications or recursive applications.

2. *Compare a data type with a Data structure.*

   Solution. A Data type defines a set of values along with well-defined operations stating its input-output behavior *e.g.*, you cannot put decimal point in an integer or two strings cannot not be multiplied etc.

   On the other hand, a Data structure is a physical implementation that clearly defines way of storing, accessing, manipulating data stored in a data structure. The data stored in a data structure has a specific data type *e.g.*, in a stack, all insertions and deletions take place at one end only.

3. *What are linear and nonlinear data Structures ?*

   Solution. **Linear Data structure.** A data structure is said to be linear if its elements form a sequence or a linear list, *e.g.*, Arrays or linear lists, Stacks and Queues etc.

   Non-Linear Data structure. A data structure is said to be non-linear if traversal of nodes is nonlinear in nature, *e.g.*, Graphs, Trees etc.

4. *In general, what common operations are performed on different Data Structures ?*

Solution. Some commonly performed operations on data structures are :

(i) **Insertion.** To add a new data item in the given collection of data items.

(ii) **Deletion.** To delete an existing data item from the given collection of data items.

(iii) **raversal.** To access each data item exactly once so that it can be processed.

(iv) **Searching.** To find out the location of the data item if it exists in the given collection of data items.

(v) **Sorting.** To arrange the data items in some order *i.e.*, in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

5. *What purpose Linear lists data structures are mostly used for ?*

Solution. Linear lists data structures are used to store and process elements that are similar in type and are to be processed in the same way. For example, to maintain a shopping list, a linear list may be used where items to be shopped are inserted to it one by one and as soon as an item is shopped, it is removed from the list.

6. *Consider the following similar codes (carefully go through these) and predict their outputs.*

(i)
```
NList = [ 60, 32, 13, 'hello' ]
print(NList[1], NList[-2])
NList.append( 15 )
print( len(NList) )
print( len(NList[3]) )
NList.pop(3)
NList.sort()
NList.insert(2, [14, 15] )
NList[3] += NList[4]
NList[3] += NList[2][1]
print(NList[3])
NList.pop()
NList[2].remove(14)
print(NList)
```

(ii)
```
NList = [ 60, 32, 13, 'hello' ]
print(NList[1], NList[-2])
NList.append( 15 )
print( len(NList) )
print( len(NList[3]) )
NList.pop(3)
NList.insert(2, [14, 15] )
NList[3] += NList[4]
NList[3] += NList[2][1]
print(NList[3])
NList[2].remove(14)
print(NList)
```

Solution.

(i)
```
32 13
5
5
107
[13, 15, [15], 107]
```

(ii)
```
32 13
5
5
43
[60, 32, [15], 43, 15]
```

7. *What will be the output produced by following code ?*

```
text = ['h', 'e', 'l', 'l', 'o']
print(text)
vowels = "aeiou"
newText= [ x.upper() for x in text if x not in vowels ]
print(newText)
```

Solution.

```
['h', 'e', 'l', 'l', 'o']
['H', 'L', 'L']
```

8. *What is a list comprehension ? How is it useful ?*

Solution. A List Comprehension is a concise description of a list that shorthands the list creating for loop in the form of a single statement.

List comprehensions make the code compact and are faster to execute.

9. *Predict the output.*

(*i*)
```
LA = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
LB = [num/3 for num in LA if num % 3 == 0]
print(LB)
```

(*ii*)
```
[x + y for x in 'ball' for y in 'boy']
```

(*iii*)
```
li = [1, 2, 3, 4, 5, 6, 7, 8, 9]
k = [elem1*elem2 for elem1 in li if (elem1 - 4) >1 for elem2 in li[:4] ]
print(k)
```

Solution.

(*i*) `[3.0, 12.0, 27.0]`

(*ii*) `['bb', 'bo', 'by', 'ab', 'ao', 'ay', 'lb', 'lo', 'ly', 'lb', 'lo', 'ly']`

(*iii*) `[6, 12, 18, 24, 7, 14, 21, 28, 8, 16, 24, 32, 9, 18, 27, 36]`

10. *What is the difference between a regular 2D list and a ragged List ?*

Solution. A regular two dimensional list is a list having lists as its elements and each element-list has the same shape *i.e.*, same number of elements (length).

On the other hand, a list that contains lists with different shapes as its elements is also a 2D list but it is an irregular 2D list, also known as a ragged list.

For instance, in the example code below *List1* is a regular 2D list while *List2* is a ragged list :

```
List1 = [ [1, 2, 3], [7, 9, 8] ]
List2 = [ [4, 5, 6], [1, 7] ]
```

11. *The Investiture Ceremony is a prestigious event in every school's calendar wherein the school formally entrusts responsibilities on the 'young student leaders'.*

At MySchool, the list STL *stores the names of all the students of class XI who have applied for the various School Leaders posts. Out of these, school leaders will be selected.*

Students need to register their name through an online application form available on school's website. Ideally the name should be in the form such that First name and Last name have their first letter capitalized and rest of the letters in lowercase.

But not all students are careful when entering their names, so the names can appear with incorrectly capitalized letters. For example :

```
STL = ['Meesha Jain', 'khushi khan', 'Raman Singh', 'yashi Sheril']
```

Write a program that provides functions for

(*i*) selecting only those correctly entered entries where the first letters of the first name and last name are capitalized.

(*ii*) selecting only the incorrectly entered names.

(*iii*) returning a list with corrected names.

Solution.

```
def select_errors(STL):
      newList = []
      for record in STL:
          name_surname = record.split(' ')
          name = name_surname[0]
          surname = name_surname[1]
          if name[0].islower() or surname[0].islower():
              newList.append(record)
      return newList

def select_correct(STL):
      newList = []
      for record in STL:
          name_surname = record.split(' ')
          name = name_surname[0]
          surname = name_surname[1]
          if not name[0].islower() and not surname[0].islower():
              newList.append(record)
      return newList

def correct_entries(STL):
      newList = []
      for record in STL:
          name_surname = record.split(' ')
          name = name_surname[0]
          surname = name_surname[1]
          newList.append(name.capitalize() + ' ' + surname.capitalize())
      return newList


#__main__
STL = []
ch = 0
while (ch != 4):
   print("\t----")
   print("\tMENU")
   print("\t----")
   print("1. Apply for the School Post")
   print("2. List of all applicants")
   print("3. Correct the Incorrect Entries")
   print("4. Exit")
   ch = int(input("Enter your choice (1-4):"))
   if ch == 1 :
       name = input("Enter your name : ")
       STL.append (name)
   elif ch == 2 :
       print("Students applied so far:")
       print(STL)
```

```
    elif ch == 3:
        ok_entries = select_correct(STL)
        error_entries = select_errors(STL)
        corrected_entries = correct_entries(STL)
        print("Correctly entered names:", ok_entries)
        print("Incorrectly entered names:", error_entries)
        print("Corrected names :", corrected_entries)
    elif ch != 4 :
        print("valid choices are 1..4:")
else:
        print("THANK YOU")
```

Sample run of above program is :

```
-----
MENU
-----
1. Apply for the School Post
2. List of all applicants
3. Correct the Incorrect Entries
4. Exit
Enter your choice (1-4): 1
Enter your name : Meesha Jain

-----
MENU
-----
:
Enter your choice (1-4): 1
Enter your name : khushi khan

-----
MENU
-----
:
Enter your choice (1-4): 1
Enter your name : Raman Singh

-----
MENU
-----
1. Apply for the School Post
2. List of all applicants
3. Correct the Incorrect Entries
4. Exit
Enter your choice (1-4): 1
Enter your name : yashi Sheril

-----
MENU
-----
:
Enter your choice (1-4): 2
Students applied so far:
['Meesha Jain', 'khushi khan', 'Raman Singh', 'yashi Sheril']

-----
MENU
-----
:
Enter your choice (1-4): 3
Correctly entered names: ['Meesha Jain', 'Raman Singh']
Incorrectly entered names: ['khushi khan', 'yashi Sheril']
Corrected names : ['Meesha Jain', 'Khushi Khan', 'Raman Singh', 'Yashi Sheril']
```

10

Data Structures – II : Stacks and Queues

10.1 INTRODUCTION

Any good programming language course does cover data structures. A *data structure*, in general, refers to a particular way of storing and organizing data in a computer so that it can be used most efficiently to give optimal performance. Different kinds of *data structures* are designed and used for different kinds of applications. The significance of data structures lies in the fact that they help a computer perform tasks in most efficient and productive manner.

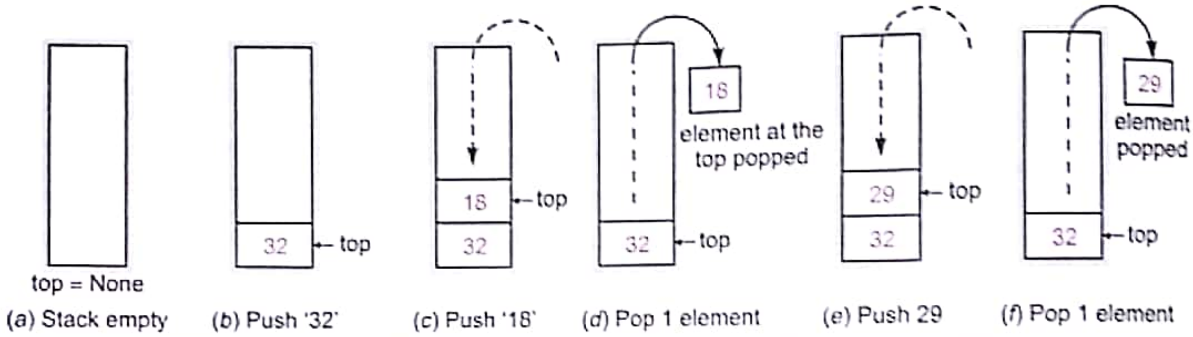
Some data structures are highly specialized to specific tasks. **Stacks** and **Queues** are such data structures. In this chapter, we shall be talking about the basics of these data structures and how these can be implemented in Python.

10.2 STACKS

A stack is a linear structure implemented in LIFO (**Last In First Out**) manner where insertions and deletions are restricted to occur only at one end – ***Stack's top***. LIFO means element last inserted would be the first one to be deleted. Thus, we can say that a stack is a list of data that follows these rules :

1. Data can only be removed from the top (*pop*), *i.e.*, the element at the *top of the stack*. The removal of element from a stack is technically called **POP operation**.
2. A new data element can only be added to the *top of the stack* (*push*). The insertion of element in a stack is technically called **PUSH operation**.

Consider Fig. 10.1 that illustrates the operations (**Push** and **Pop**) on a stack.



top = None

(*a*) Stack empty (*b*) Push '32' (*c*) Push '18' (*d*) Pop 1 element (*e*) Push 29 (*f*) Pop 1 element

Notice, all stack operations - **Push** or **Pop-take** place at one end-the ***stack's top***

Figure 10.1 Stack operations – *Push* and *Pop*.

The stack is a *dynamic data structure* as it can grow (with increase in number of elements) or shrink (with decrease in number of elements). A *static data structure*, on the other hand, is the one that has fixed size.

STACK

A Stack is a linear list implemented in LIFO – Last In First Out manner where insertions and deletions are restricted to occur only at one end – *Stack's top*.

Other Stack Terms

There are *some* other terms related to stacks, such as *Peek*, *Overflow* and *Underflow*.

Peek Refers to inspecting the value at the ***stack's top*** without removing it. It is also sometimes referred as **inspection**.

Overflow Refers to situation (ERROR) when one tries to push an item in stack that is full. This situation occurs when the size of the stack is fixed and cannot grow further or there is no memory left to accommodate new item.

Underflow Refers to situation (ERROR) when one tries to pop/delete an item from an empty stack. That is, stack is currently having no item and still one tries to pop an item.

NOTE

The technical terms for insertion-in-a-stack and deletion-from-stack are Push and Pop respectively.

Consider some examples illustrating stack-functioning in limited-size stack. (Please note, we have bound fixed the capacity of the stack for understanding purposes.)

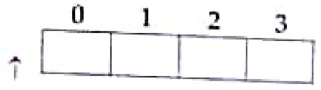
EXAMPLE 10.1 *Given a Bounded Stack of capacity 4 which is initially empty, draw pictures of the stack after each of the following steps. Initially the Stack is empty.*

(*i*) Stack is empty (*ii*) Push 'a' (*iii*) Push 'b' (*iv*) Push 'c'
(*v*) Pop (*vi*) Push 'd' (*vii*) Pop (*viii*) Push 'e'
(*ix*) Push 'f' (*x*) Push (*xi*) Pop (*xii*) Pop
(*xiii*) Pop (*xiv*) Pop (*xv*) Pop

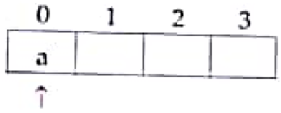
Solution.

top = ↑

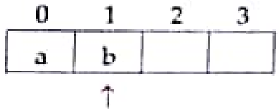
(*i*) Stack is empty (top = None)



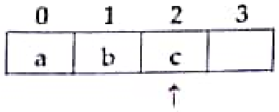
(*ii*) Push 'a' *top = 0*



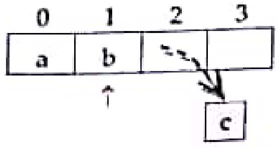
(*iii*) Push 'b' *top = 1*



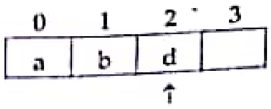
(*iv*) Push 'c' *top = 2*



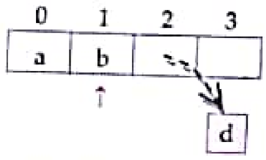
(*v*) Pop *top = 1*



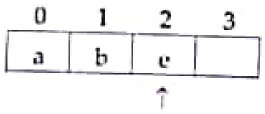
(*vi*) Push 'd' *top = 2*



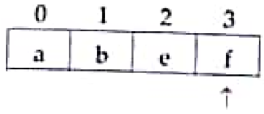
(*vii*) Pop *top = 1*



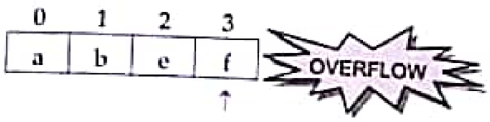
(*viii*) Push 'e' *top = 2*



(*ix*) Push 'f' *top = 3*



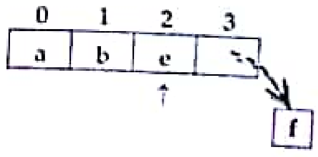
(*x*) Push 'g' *top = 3*



[OVERFLOW because the stack is bounded, it cannot grow. If it could grow, then there would have been no OVERFLOW until no memory is left.

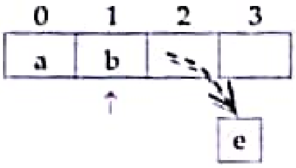
In Python, (for stacks implemented through lists) since Lists can grow, OVERFLOW condition does not arise until all the memory is exhausted.]

(*xi*) Pop *top = 2*

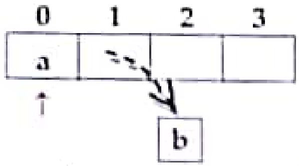


top = ↑

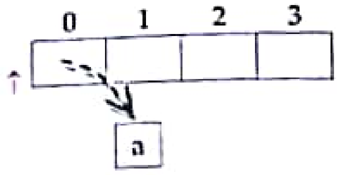
(*xii*) Pop *top* = 1



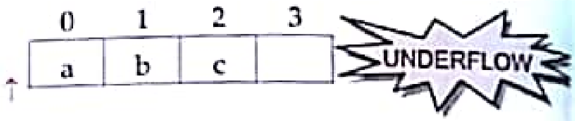
(*xiii*) Pop top = 0



(*xiv*) Pop top = None



(*xv*) Pop top = None



10.2.1 Implementing Stack in Python

In Python, you can use Lists to implement stacks. Python offers us a convenient set of methods to operate lists as stacks.

For various stack operations, we can use a list say *Stack* and use Python code as described below :

Peek We can use : `<Stack> [top]`

where **<Stack>** is a list ; *top* is an integer having value equal to **len(<Stack>) - 1.**

Push We can use : `<Stack>.append(<item>)`

where **<item>** is the item being pushed in the Stack.

Pop We can use : `<Stack>.pop()`

it removes the last value from the *Stack* and returns it.

Let us now implement a stack of numbers through a program.

```
def Push(stk, item) :
```

Program 10.1 Python program to implement stack operations.

```
#############     STACK IMPLEMENTATION     ##############
"""
    Stack: implemented as a list
    top : integer having position of topmost element in Stack
"""
def isEmpty( stk ) :
    if stk == [] :
        return True
    else :
        return False
```

```
def Push(stk, item) :
    stk.append(item)
    top = len(stk) - 1
def Pop(stk) :
    if isEmpty(stk) :
        return "Underflow"
    else :
        item = stk.pop()
        if len (stk) == 0:
            top = None
        else :
            top = len(stk) - 1
        return item
def Peek(stk) :
    if is Empty(stk) :
        return "Underflow"
    else :
        top = len(stk) - 1
        return stk[top]
def Display(stk) :
    if isEmpty(stk) :
        print("Stack empty")
    else :
        top = len(stk) - 1
        print(stk[top], "<– top")
        for a in range(top-1, -1 , -1 ) :
            print(stk[a])
#__main__
Stack = []                        # initially stack is empty
top = None
while True :
    print("STACK OPERATIONS")
    print("1. Push")
    print("2. Pop")
    print("3. Peek")
    print("4. Display stack")
    print("5. Exit")
    ch = int(input("Enter your choice (1-5) :"))
    if ch == 1 :
        item = int(input("Enter item :"))
        Push(Stack, item)
    elif ch == 2 :
        item = Pop(Stack)
        if item == "Underflow" :
            print("Underflow! Stack is empty!")
        else :
            print("Popped item is", item)
    elif ch == 3 :
        item = Peek(Stack)
        if item == "Underflow" :
            print("Underflow! Stack is empty!")
        else :
            print("Topmost item is", item)
    elif ch == 4 :
        Display(Stack)
    elif ch == 5 :
        break
    else :
        print("Invalid choice!")
```

Sample run of the above program is as shown below :

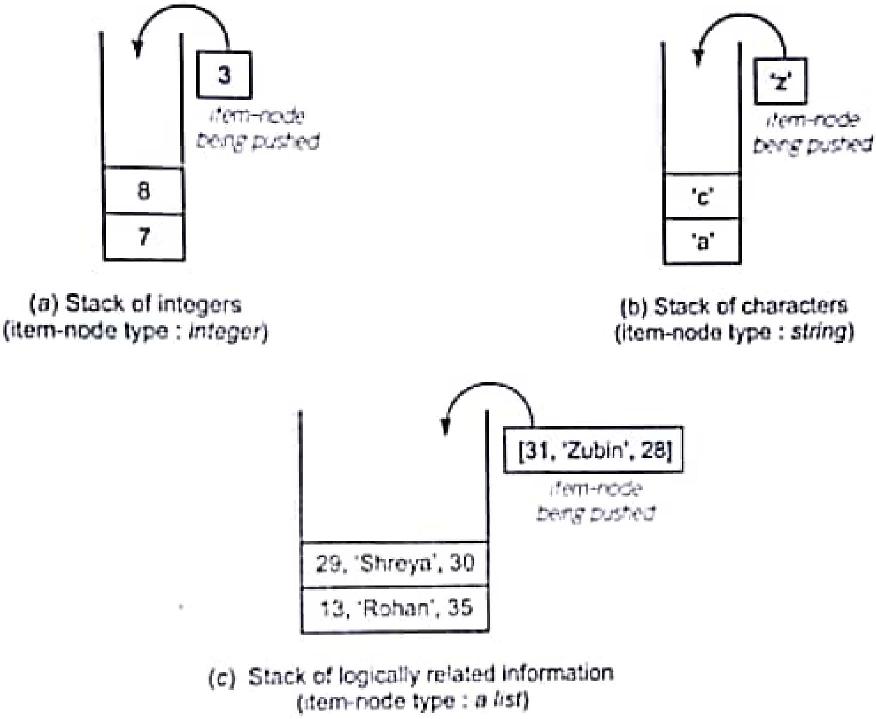
```
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :6
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :8
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :2
_______________________
```

```
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :4
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :4
4 <- top
2
8
6
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice ( 1-5) :3
Topmost item is  4
_______________________
```

```
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :4
4 <- top
2
8
6
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :5
```

Types of Stack–Itemnode

An item stored in a stack is also called item-node sometimes. In the above implemented stack, the stack contained item-nodes containing just integers. If you want to create stack that may contain logically group information such as member details like : *member no, member name, age* etc. For such a stack the item-node will be a list containing the member details and then this list will be entered as an item to the stack. (See figure 10.2 below).



(a) Stack of integers (item-node type : *integer*)

(b) Stack of characters (item-node type : *string*)

(c) Stack of logically related information (item-node type : *a list*)

Figure 10.2 Different types of stack item-nodes.

- For stack of Fig. 10.2(*a*), the stack will be implemented as ***Stack of integers*** as *item-node* is of integer type.
- For stack of Fig. 10.2(*b*), the stack will be implemented as ***Stack of strings*** as *item-node* is of string type.
- For stack of Fig. 10.2(*c*), the stack will be implemented as ***Stack of lists*** as *item-node* is of list type. Solved problem 20 implements such a stack.

10.2.2 Stack Applications[1]

There are several applications and uses of stacks. The stacks are basically applied where LIFO (Last In First Out) scheme is required.

10.2.2A Reversing a Line

A simple example of stack application is reversal of a given line. We can accomplish this task by pushing each character on to a stack as it is read. When the line is finished, characters are then

[1] Some other applications of stacks include :
(*a*) The compilers use stacks to store the previous state of a program when a function is called, or during recursion.
(*b*) One of the most important applications of Stacks is *backtracking*. Backtracking is used in large number of puzzles like n-Queen problem, *Sudoku* etc and optimization problems such as *knapsack problem*.

popped off the stack, and they will come off in the reverse order as shown in Fig. 10.3. The given line is : *Stack*

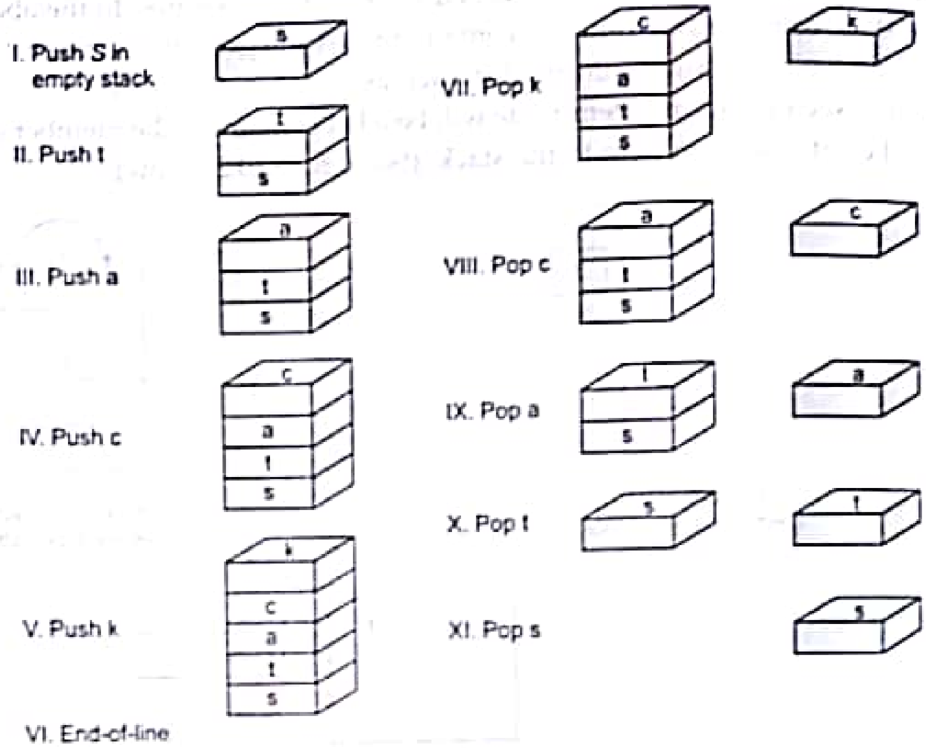


Figure 10.3
Reversal of a line using stack.

10.2.2B Polish Strings

Another application of stacks is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only *e.g.*, *A* + *B*, C × *D*, *D*/*A* etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands *e.g.*, (*A* + *B*) × C (*D*/(*J* + *D*)). These complex arithmetic operations can be converted into *polish strings* using stacks which then can be executed in *two operands and a operator* form.

Polish string, named after a polish mathematician, *Jan Lukasiewicz*, refers to the notation in which the operator symbol is placed either before its operands (*prefix notation*) or after its operands (*postfix notation*) in contrast to usual form where operator is placed in between the operands (*infix notation*).

Following table shows the three *types* of notations :

Table 10.1 *Expressions in infix, prefix, postfix notations*

Infix notation	Prefix notation	Postfix notation
A + *B*	+ *AB*	*AB* +
(*A* – C) × *B*	× – *ACB*	*AC* – *B*×
A + (*B* × C)	+ *A* × *BC*	*ABC* × +
(*A* + *B*)/(C – *D*)	/+ *AB*–*CD*	*AB*+*CD*–/
(*A* + (*B* × C))/(C – (*D* × *B*))	/+ *A*× *BC*–C× *DB*	*ABC*× + *CDB*× –/

Conversion of Infix Expression to Postfix (Suffix) Expression

While evaluating an infix expression, there is an evaluation order according to which

I Brackets or Parenthesis,
II Exponentiation,
III Multiplication or Division,
IV Addition or Subtraction

take place in the above specified order. The operators with the same priority (*e.g.*, × and/) are evaluated from left to right.

To convert an infix expression into a postfix expression, this evaluation order is taken into consideration.

An infix expression may be converted into postfix form either manually or using a stack. The manual conversion requires two passes : one for inserting braces and another for conversion. However, the conversion through stack requires single pass only.

The steps to convert an infix expression into a postfix expression manually are given below :

(*i*) Determine the actual evaluation order by inserting braces.

(*ii*) Convert the expression in the innermost braces into postfix notation by putting *the operator after the operands.*

(*iii*) Repeat step (*ii*) until entire expression is converted into postfix notation.

Example 10.2 *Convert $(A+B) \times C / D$ into postfix notation.*

Solution.

Step I : Determine the actual evaluation order by putting braces

$$= ((A+B) \times C) / D$$

Step II : Converting expressions into innermost braces

$$= ((AB+) \times C) / D = (AB + C \times) / D = AB + C \times D /$$

Example 10.3 *Convert $((A+B) * C / D + E \wedge F) / G$ into postfix notation.*

Solution. The evaluation order of given expression will be

$$= ((((A+B) * C) / D) + (E \wedge F)) / G$$

Converting expressions in the braces, we get

$$= ((((AB+) * C) / D) + (EF \wedge)) / G$$

$$= (((AB + C *) / D) + EF \wedge) / G$$

$$= ((AB + C * D/) + EF \wedge) / G = (AB + C * D / EF \wedge +) / G$$

$$= AB + C * D / EF \wedge + G /$$

Example 10.4 *Give postfix form of the following expression*

$$A * (B + (C + D) * (E + F) / G) * H$$

Solution. Evaluation order is

$$(A * (B + ((C + D) * (E + F)) / G)) * H$$

Converting expressions in the braces, we get

$$= (A * (B + [(CD +) * (EF +)] / G)) * H = A * (B + (CD + EF + *) / G) * H$$
$$= A * (B + (CD + EF + * G/)) * H = (A * (BCD + EF + * G / +)) * H$$
$$= (ABCD + EF + * G / + *) * H = \mathbf{ABCD + EF + * G / + * H *}$$

Example 10.5 *Give postfix form for* $A + [(B + C) + (D + E) * F] / G$

Solution. Evaluation order is : $A + [\{(B + C) + ((D + E) * F)\} / G]$

Converting expressions in braces, we get

$$= A + [\{(BC +) + (DE +) * F\} / G] = A + [\{(BC +) + (DE + F *)\} / G]$$
$$= A + [\{BC + DE + F * +\} / G] = A + [BC + DE + F * + G/]$$
$$= ABC + DE + F * + G / +$$

Example 10.6 *Give postfix form of expression for the following : NOT A OR NOT B NOT C*

Solution. The order of evaluation will be

((NOT A) OR ((NOT B) AND (NOT C)))

(As priority order is NOT, AND, OR)

= ((A NOT) OR ((B NOT) AND (C NOT)))

= ((A NOT) OR ((B NOT C NOT AND))

= A NOT B NOT C NOT AND OR

While converting from *infix to prefix form,* operators are put before the operands. Rest of the conversion procedure is similar to that of *infix to postfix* conversion.

Algorithm to Convert Infix Expression to Postfix Form

The following algorithm transforms the infix expression X into its equivalent postfix expression Y. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression Y will be constructed from left to right using the operands from X and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of X. The algorithm is completed when STACK is empty.

Algorithm *Infix to Postfix Conversion using Stack*

Suppose X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

```
1.  Push "(" onto STACK, and add ")" to the end of X.
2.  Scan X from left to right and REPEAT Steps 3 to 6 for each element of
    X UNTIL the STACK is empty :
3.  If an operand is encountered, add it to Y.
4.  If a left parenthesis is encountered, push it onto STACK.
5.  If an operator is encountered, then :
      (a) Repeatedly pop from STACK and add to Y each operator (on the top
          of STACK) which has the same precedence as or higher precedence
          than operator.
```

```
            (b) Add operator to STACK.
                '''End of If structure'''
        6.  If a right parenthesis is encountered, then :
            (a) Repeatedly pop from STACK and add to Y each operator (on the top
                of STACK) until a left parenthesis is encountered.
            (b) Remove the left parenthesis. [Do not add the left parenthesis to Y].
                '''End of If structure'''
                '''End of Step 2 Loop'''
        7. END.
```

EXAMPLE 10.7 *Convert X: A + (B * C – (D / E ^ F) * G) * H into postfix form showing stack status after every step in tabular form.*

Solution.

Symbol Scanned		Stack	Expression Y
1.	A	(	A
2.	+	(+	A
3.	(	(+ (	A
4.	B	(+ (	A B
5.	*	(+ (*	A B
6.	C	(+ (*	A B C
7.	–	(+ (–	A B C *
8.	(	(+ (– (	A B C *
9.	D	(+ (– (	A B C * D
10.	/	(+ (– (/	A B C * D
11.	E	(+ (– (/	A B C * D E
12.	^	(+ (– (/ ^	A B C * D E
13.	F	(+ (– (/ ^	A B C * D E F
14.	)	(+ (–	A B C * D E F ^ /
15.	*	(+ (– *	A B C * D E F ^ /
16.	G	(+ (– *	A B C * D E F ^ / G
17.	)	(+	A B C * D E F ^ / G * –
18.	*	(+ *	A B C * D E F ^ / G * –
19.	H	(+ *	A B C * D E F ^ / G * – H
20.	)		A B C * D E F ^ / G * – H * +

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence). Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

WORKING WITH STACKS

PriP — Progress In Python 10.1

This PriP session is dedicated to the practice of data structure Stack's concepts and provides practical assignment for the same.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 10.1 under Chapter 10 after practically doing it on the computer.

>>>❖<<<

10.3 QUEUES

Queues are similar to stacks in that a queue also consists of a sequence of items (a linear list), and there are restrictions about how items can be added to and removed from the list. However, a queue has two ends, called the **front-end** and the **back-end** or **rear-end** of the queue. Items are always added to the queue at the ***rear-end*** and removed from the queue at the *front-end*. The operations of adding and removing items are called **enqueue** and **dequeue**. An item that is added to the back of the queue will remain on the queue until all the items in front of it have been removed.

QUEUE

A Queue is a linear list implemented in FIFO – First In First Out manner where insertions take place at the rear-end and deletions are restricted to occur only at front end of the queue.

A queue is like a "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive on the queue. Thus a queue is also called a **FIFO list** *i.e.*, **First In First Out** list where the item first inserted is the first to get removed. So, we can say that a queue is a list of data that follows these rules :

1. Data can only be removed from the front end, *i.e.*, the element at the *front-end of the queue*. The removal of element from a queue is technically called DEQUEUE operation.
2. A new data element can only be added to the *rear of the queue*. The insertion of element in a stack is technically called **ENQUEUE operation**.

NOTE

The Enqueue operation adds item at the back of the queue (at rear-end) and the Dequeue operation removes the item from the front of the queue (at front-end) and returns it.

Other Queue Terms

There are *some* other terms related to queues, such as *Overflow* and *Underflow*.

Peek Refers to inspecting the value at the *queue's front* without removing it. It is also sometimes referred as **inspection**.

Overflow Refers to situation (ERROR) when one tries to enqueue an item in a queue that is full. This situation occurs when the size of the queue is fixed and cannot grow further or there is no memory left to accommodate new item.

Underflow Refers to situation (ERROR) when one tries to Dequeue/delete an item from an empty queue. That is, queue is currently having no item and still one tries to dequeue an item.

EXAMPLE 10.12 *Given a Bounded Queue of capacity 4 which is initially empty, draw pictures of the queue after each of the following steps :*

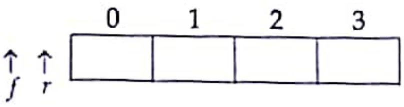
(*i*) Queue empty (*ii*) enqueue 'a' (*iii*) enqueue 'b'
(*iv*) enqueue 'c' (*v*) dequeue (*vi*) enqueue 'd'
(*vii*) enqueue 'e' (*viii*) dequeue (*ix*) dequeue
(*x*) dequeue (*xi*) dequeue

Solution.

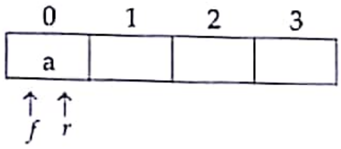
Front = $\underset{f}{\uparrow}$; Rear = $\underset{r}{\uparrow}$

(*i*) Queue is empty

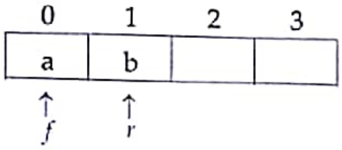
(front = rear = None)



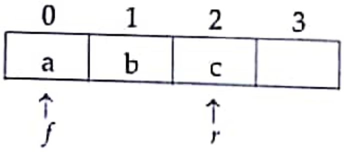
(*ii*) enqueue 'a' *(front = 0, rear = 0)*



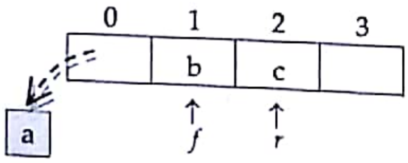
(*iii*) enqueue 'b' *(front = 0, rear = 1)*



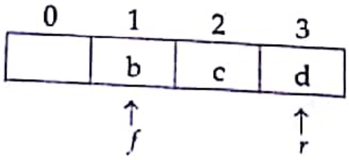
(*iv*) enqueue 'c' *(front = 0, rear = 2)*



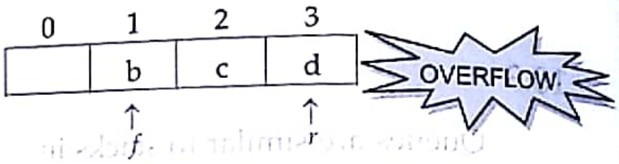
(*v*) dequeue *(front = 1, rear = 2)*



(*vi*) enqueue 'd' *(front = 1, rear = 3)*

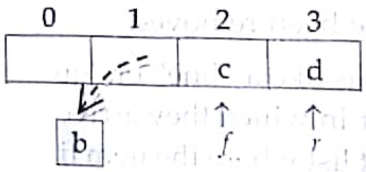


(*vii*) enqueue 'e' *(front = 1, rear = 3)*

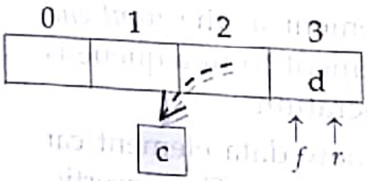


[OVERFLOW because the queue is bounded, it cannot grow. If it could grow, then there would have been no OVERFLOW until no memory is left. In Python, (for queues implemented through lists) since Lists can grow, OVERFLOW condition does not arise until all the memory is exhausted.]

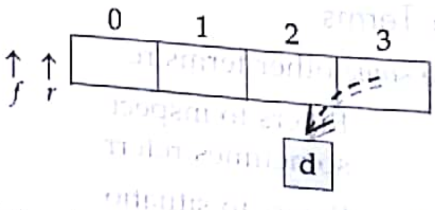
(*viii*) dequeue *(front = 2, rear = 3)*



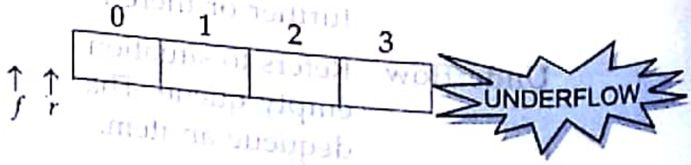
(*ix*) dequeue *(front = 3, rear = 3)*



(*x*) dequeue *(front = None, rear = None)*



(*xi*) dequeue *(front = None, rear = None)*



10.3.1 Implementing Queues in Python

In Python, you can use *Lists* to implement queues. Python offers us a convenient set of methods to operate lists as queues. For various Queue operations, we can use a list say *Queue* and use Python code as described below :

Peek We can use : `<Queue>[front]`

where **<Queue>** is a list ; ***front*** is an integer storing the position of first value in the queue.

Enqueue We can use : `<Queue>.append(<item>)`

where **<item>** is the item being pushed in the *Queue*. The item will be added at the rear-end of the queue.

Dequeue We can use : `<Queue>.pop(0)`

it removes the first value from the *Queue* (*i.e.*, the item at the *front-end*) and returns it.

Number of Elements in Queue

We can determine the size of a queue using the formula:

`Number of elements in queue = rear - front + 1`

In Python queues, implemented through lists, the *front* and *rear* are :

`front = 0 and rear = len(<queue>) - 1`

You can also use Python function **len(<queue>)** to get the size of the queue.

Let us now implement a Queue of numbers through a program.

Program 10.2 Program to implement Queue Operations

```
##############   queue IMPLEMENTATION   ##############
"""
queue: implemented as a list
front : integer having position of first (frontmost) element in queue
rear : integer having position of last element in queue
"""
def cls():
    print("\n" * 100)

def isEmpty( Qu ) :
    if Qu == [] :
        return True
    else :
        return False

def Enqueue(Qu, item) :
    Qu.append(item)
    if len(Qu) == 1 :
        front = rear = 0
    else :
        rear = len(Qu) - 1
```

```
def Dequeue(Qu) :
    if isEmpty(Qu) :
        return "Underflow"
    else :
        item = Qu.pop(0)
    if len(Qu) == 0 :            #if it was single-element queue
        front = rear = None
    return item

def Peek(Qu) :
    if isEmpty(Qu) :
        return "Underflow"
    else :
        front = 0
    return Qu[front]

def Display(Qu) :
    if isEmpty(Qu) :
        print("Queue Empty!")
    elif len(Qu) == 1:
        print(Qu[0], "<== front, rear")
    else :
        front = 0
        rear = len(Qu) - 1
        print(Qu[front], "<- front")
        for a in range(1, rear ) :
            print(Qu[a])
        print(Qu[rear], "<- rear")

# __main__ program
queue = []                  # initially queue is empty
front = None
while True :
    cls()
    print("QUEUE OPERATIONS")
    print("1. Enqueue")
    print("2. Dequeue")
    print("3. Peek")
    print("4. Display queue")
    print("5. Exit")
    ch = int(input("Enter your choice ( 1-5) : "))
    if ch == 1 :
        item = int(input("Enter item :"))
        Enqueue(queue, item)
        input("Press Enter to continue...")
    elif ch == 2 :
        item = Dequeue(queue)
        if item == "Underflow" :
            print("Underflow! Queue is empty!")
        else :
            print("Dequeue-ed item is", item)
        input("Press Enter to continue...")
        elif ch == 3 :
            item = Peek(queue)
            if item == "Underflow" :
                print("Queue is empty!")
            else :
                print("Frontmost item is", item)
            input("Press Enter to continue...")
        elif ch == 4 :
            Display(queue)
            input("Press Enter to continue...")
        elif ch == 5 :
            break
        else :
            print("Invalid choice!")
            input("Press Enter to continue...")
```

```
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 1
Enter item :5
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 4
5 <== front, rear
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 1
Enter item :7
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 3
Frontmost item is  5
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 1
Enter item :9
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 4
5 <- front
7
9 <- rear
Press Enter to continue...
__________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Dequeue-ed item is  5
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Dequeue-ed item is  7
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Dequeue-ed item is  9
Press Enter to continue...
______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Underflow! Queue is empty!
Press Enter to continue...
```

Solved Problems

1. *What is a Data-Structure ?*

   Solution. A data-structure is a logical way for organizing data in memory that considers not only the items stored but also the relationship among the items so as to give efficient and optimal performance. In other words, it is a way of organizing data such that it can be used efficiently.

2. *What is a stack ? What basic operations can be performed on them ?*

   Solution. Stack is a basic data-structure where insertion and deletion of data takes place at one end called *the top of the stack i.e.*, it follows the **Last In-First Out(LIFO)** principle.

   In Python, a stack is generally implemented with lists.

   Following basic operations can be performed on stacks:

   (*a*) ***Push*** *i.e.*, Insertion of element in the stack

   (*b*) ***Pop*** *i.e.*, Deletion of an element from the stack

   (*c*) ***Peek*** *i.e.*, viewing topmost element without removing it

   (*d*) ***Display*** or view all the elements in the stack.

3. *Enlist some applications of stacks.*

   Solution. Because of LIFO property of stacks, these are used in various applications like :

   ◆ reversal of a sequence,

   ◆ Infix to Postfix conversion,

   ◆ Postfix and prefix expression evaluation,

   and many more.

4. *What are queues? What all operations can be performed on queues?*

   Solution. Queues are the data structures where data is entered into the queue at one end – *the rear end* and deleted from the other end – *the front end, i.e.*, these follow **First In-First Out (FIFO)** principle. Following basic operations can be performed on queues :

   (*i*) ***Enqueue*** *i.e.*, Insertion of an element in the queue

   (*ii*) ***Dequeue*** *i.e.*, Deletion of an element from the queue

   (*iii*) ***Peek*** *i.e.*, viewing *frontmost* element without removing it

   (*iv*) ***Display*** or view all the elements in the queue.

5. *Enlist some applications of queues.*

   Solution. Applications of queues include the situations where FIFO property is exploited. Some common applications of queues include :

   (*i*) Sharing of one resource among multiple users or seekers such as shared printer among multiple computers; Call center executive's response among waiting callers etc.

   (*ii*) Airport authorities make use of queues in situation of sharing a single runway of airport for both landing and take-off of flights.

   (*iii*) CPU uses queues to implement round-robin scheduling among waiting processes.

   (*iv*) Queues are used in many computer algorithms also.

6. *Write the equivalent infix expression for :* 10, 3, *, 7, 1, –, *, 23, +

Solution. 10 * 3 * (7 – 1) + 23

7. *Consider the following stack of characters implemented as an array of 8 elements :*

$$\text{STACK : A, J, P, } \underset{\uparrow\ top}{N}$$

Describe the stack as the following operations take place :

(*a*) POP (STACK, ITEM) (*b*) PUSH (STACK, K) (*c*) PUSH (STACK, S)

(*d*) POP (STACK, ITEM) (*e*) PUSH (STACK, G)

Solution.

(*a*) **STACK :** $A, J, \underset{top}{P}$ as *N* is Popped

(*b*) **STACK :** $A, J, P, \underset{top}{K}$ as *K* is Pushed

(*c*) **STACK :** $A, J, P, K, \underset{top}{S}$ as *S* is Pushed

(*d*) **STACK :** $A, J, P, \underset{top}{K}$ as *S* is Popped

(*e*) **STACK :** $A, J, P, K, \underset{top}{G}$ as G is Pushed

8. *Given a stack as an array of 7 elements STACK : K, P, S, -, -, -,*

(*a*) *When will overflow and underflow occur ?*

(*b*) *Can K be deleted before S ? Why ?*

Solution.

(*a*) *Overflow* will occur when stack will be having 7 elements and there would be no space to insert a new element.

Underflow will occur when all the elements will have been deleted from the stack and no more element could be deleted.

(*b*) *K* can not be deleted before *S*. As S has been inserted after *K* and stack follows the LIFO rule *i.e., Last In First Out*, therefore *S* will be deleted before *K*.

9. *Consider the following infix expression* $P = -a + b * c \uparrow (d * e) / f$ *where* ↑ *denotes exponentiation, and / integer division. Translate P into its equivalent postfix expression using the following set of priorities :*

(	0
+ –)	1
unary – × / +	2
↑	3

Solution. $P = -a + b * c \uparrow (d * e) / f$

The order of evaluation will be (according to given priorities) as given below

$$= ((-a) + ((b * (c \uparrow (d * e))) / f)) = (a - + ((b * (c \uparrow (de *))) / f))$$

$$=((a-)+((b*(cde*\uparrow))/f))$$

$$=((a-)+((bcde*\uparrow*/f))=(a-)+((bcde*\uparrow*f/))$$

$$=(a-)+(bcde*\uparrow*f/)=a-bcde*\uparrow*f/+$$

10. *Give the algorithm for converting an infix arithmetic expression into a postfix arithmetic expression. Use the algorithm for the following expression, showing in a tabular form the changing status of the stack :*

Q : (A – B) * (C / D) + E

Solution. The simulation of this algorithm for the above expression is shown in the following table :

*Conversion of (A – B) * (C / D) + E into postfix expression*

Step	Input element / symbol taken	Action	Stack status	Output to be printed
			# (empty)	
1.	'('	PUSH	'('	
2.	'A'	Print	'('	A
3.	'–'	PUSH	'('-	A
4.	B	Print	'(' -	A
5.	')'	POP & Print POP	'(' # #	AB- AB-
6.	*	PUSH	#	AB-
7.	'('	PUSH	* '('	AB-
8.	C	Print	*'('	AB-C
9.	/	PUSH	*'(/	AB-C
10.	D	Print	*'(/	AB-CD
11.	')'	POP & Print POP	*'(*	AB-CD/ AB-CD/
12.	+	POP & Print PUSH	* + +	AB-CD/*
13.	E	Print	+	AB-CD/*E
14.	;	POP & Print STOP	# (empty)	AB-CD/*E+

The resultant postfix expression is AB – CD/ *E + .

11. *Evaluate following arithmetic expression A which is in postfix notation. Show the contents of the stack during the execution of the algorithm using the following :*

A = 30, 5, 2, ^ 12, 6, /, +, – .

Solution. *Push '(' to the stack and add ')' at the end of expression i.e.,* the expression becomes

30, 5, 2, **, 12, 6, /, +, −)

Step	Input Element/Symbol	Action taken	Stack Status	Output
1.			(	
2.	30	Push	(, 30	
3.	5	Push	(, 30, 5	
4.	2	Push	(, 30, 5, 2	
5.	**	POP (2 elements)	(, 30	5 ** 2 =25
6.		Push the result (25)	(, 30, 25	
7.	12	Push	(, 30, 25, 12	
8.	6	Push	(, 30, 25, 12, 6	
9.	/	POP (2 elements)	(, 30, 25	12/6 = 2
10.		Push result (2)	(, 30, 25, 2	
11.	+	POP (2 elements)	(, 30	25 + 2 =27
12.		Push the result (27)	(, 30, 27	
13.	−	POP (2 elements)	(	30 − 27 =3
14.		Push the result (3)	(3	
15.	)	POP everything	#	

Result = 3.

12. *Convert the expression (TRUE and FALSE) or not (FALSE or TRUE) to postfix expression. Show the contents of the stack at every step.*

Solution. (TRUE and FALSE) or not (FALSE or TRUE)]

Adding] to the end of expression and inserting [to the beginning of stack.

Scanning from Left to Right

S.No	Symbol	Stack	Postfix Expression Y
0.		[	
1.	(	[(	
2.	TRUE	—	TRUE
3.	and	[(and	—
4.	FALSE	—	TRUE FALSE
5.	)	[	TRUE FALSE and
6.	or	[or	—
7.	not	[or not	—
8.	(	[or not (	—
9.	FALSE	—	TRUE FALSE and FALSE
10.	or	[or not (or	—
11.	TRUE	—	TRUE FALSE and FALSE TRUE
12.	)	[or not	TRUE FALSE and FALSE TRUE or
13.	]	End of Expression	TRUE FALSE and FALSE TRUE or not or

20. *Describe the similarities and differences between queues and stacks.*

Solution.

Similarities :

1. Both queues and stacks are special cases of linear lists.
2. Both can be implemented as arrays or lists.

Differences :

1. A stack is a LIFO list, a queue is a FIFO list.
2. There are no variations of stack, a queue, however, may be circular or deque.

21. *What is the difference between an array and a stack housed in an array ? Why is stack called a LIFO data structure ? Explain how push and pop operations are implemented on a stack.*

Solution. An array is a group of homogeneous elements stored in contiguous memory locations. The elements in an array can be processed from anywhere in the array.

A stack implemented as an array also has elements stored in contiguous memory locations. But a stack is always processed in a LIFO manner *i.e., Last In First Out* manner wherein the elements can be added or removed from the top end of the stack. That is why a stack is also called a LIFO data structure.

An addition to the stack is known as PUSH. The new element is added at the top and the top (variable or pointer) is made to refer to the new element.

A removal of an element from a stack is known as POP. The elements (being pointed to by top) is removed and the top is made to point to the next element in the row.