MODULE - 1

1.1 WHY SHOULD YOU LEARN TO WRITE PROGRAMS

Programs are generally written to solve the real-time arithmetic/logical problems. Nowadays, computational devices like personal computer, laptop, and cell phones are embedded with operating system, memory and processing unit. Using such devices one can write a program in the language (which a computer can understand) of one's choice to solve various types of problems. Humans are tend get bored by doing computational tasks multiple times. Hence, the computer can act as a personal assistant for people for doing their job!! To make a computer to solve the required problem, one has to feed the proper program to it. Hence, one should know how to write a program!!

There are many programming languages that suit several situations. The programmer must be able to choose the suitable programming language for solving the required problem based on the factors like computational ability of the device, data structures that are supported in the language, complexity involved in implementing the algorithm in that language etc.

1.1.1 Creativity and Motivation

When a person starts programming, he himself will be both the programmer and the end-user. Because, he will be learning to solve the problems. But, later, he may become a proficient programmer. A programmer should have logical thinking ability to solve a given problem. He/she should be creative in analyzing the given problems, finding the possible solutions, optimizing the resources available and delivering the best possible results to the end-user. Motivation behind programming may be a job-requirement and such other prospects. But, the programmer should follow certain ethics in delivering the best possible output to his/her clients. The responsibilities of a programmer includes developing a feasible, user-friendly software with very less or no hassles. The user is expected to have only the abstract knowledge about the working of software, but not the implementation details. Hence, the programmer should strive hard towards developing most effective software.

1.1.2 Computer Hardware Architecture

To understand the art programming, it is better to know the basic architecture of computer hardware. The computer system involves some of the important parts as shown in Figure 1.1. These parts are as explained below:

- **Central Processing Unit (CPU):** It performs basic arithmetic, logical, control and I/O operations specified by the program instructions. CPU will perform the given tasks with a tremendous speed. Hence, the good programmer has to keep the CPU busy by providing enough tasks to it.
- **Main Memory:** It is the storage area to which the CPU has a direct access. Usually, the programs stored in the secondary storage are brought into main memory before the execution. The processor (CPU) will pick a job from the main memory and performs the tasks. Usually, information stored in the main memory will be vanished when the computer is turned-off.

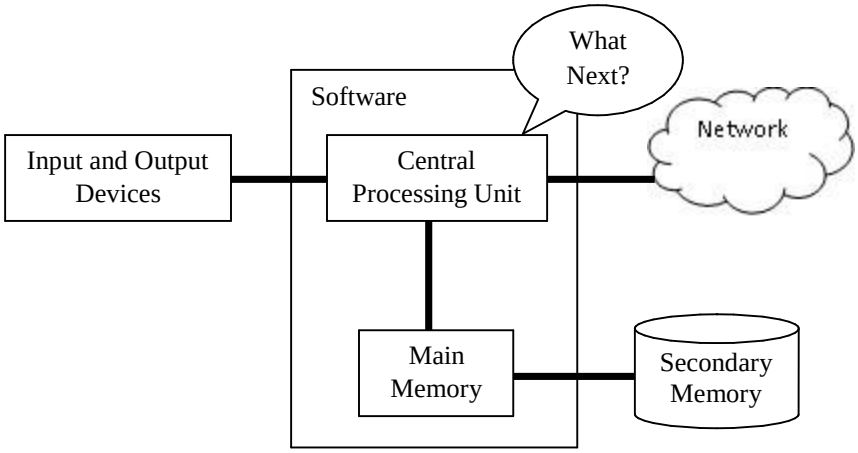


Figure 1.1 Computer Hardware Architecture

- **Secondary Memory:** The secondary memory is the permanent storage of computer. Usually, the size of secondary memory will be considerably larger than that of main memory. Hard disk, USB drive etc can be considered as secondary memory storage.
- **I/O Devices:** These are the medium of communication between the user and the computer. Keyboard, mouse, monitor, printer etc. are the examples of I/O devices.
- **Network Connection:** Nowadays, most of the computers are connected to network and hence they can communicate with other computers in a network. Retrieving the information from other computers via network will be slower compared to accessing the secondary memory. Moreover, network is not reliable always due to problem in connection.

The programmer has to use above resources sensibly to solve the problem. Usually, a programmer will be communicating with CPU by telling it 'what to do next'. The usage of main memory, secondary memory, I/O devices also can be controlled by the programmer.

To communicate with the CPU for solving a specific problem, one has to write a set of instructions. Such a set of instructions is called as a program.

1.1.3 Understanding Programming

A programmer must have skills to look at the data/information available about a problem, analyze it and then to build a program to solve the problem. The skills to be possessed by a good programmer includes -

- **Thorough knowledge of programming language**: One needs to know the vocabulary and grammar (technically known as syntax) of the programming language. This will help in constructing proper instructions in the program.
- **Skill of implementing an idea**: A programmer should be like a 'story teller'. That is, he must be capable of conveying something effectively. He/she must be able to solve the problem by designing suitable algorithm and implementing it. And, the program must provide appropriate output as expected.

Thus, the art of programming requires the knowledge about the problem's requirement and the strength/weakness of the programming language chosen for the implementation. It is always advisable to choose appropriate programming language that can cater the complexity of the problem to be solved.

1.1.4 Words and Sentences

Every programming language has its own constructs to form syntax of the language. Basic constructs of a programming language includes set of characters and keywords that it supports. The keywords have special meaning in any language and they are intended for doing specific task. Python has a finite set of keywords as given in Table 1.1.

Table 1.1 Keywords in Python

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

A programmer may use *variables* to store the values in a program. Unlike many other programming languages, a variable in Python need not be declared before its use.

1.1.5 Python Editors and Installing Python

Before getting into details of the programming language Python, it is better to learn how to install the software. Python is freely downloadable from the internet. There are multiple IDEs (Integrated Development Environment) available for working with Python. Some of them are PyCharm, LiClipse, IDLE etc. When you install Python, the IDLE editor will be available automatically. Apart from all these editors, Python program can be run on command prompt also. One has to install suitable IDE depending on their need and the Operating System they are using. Because, there are separate set of editors (IDE) available for different OS like Window, UNIX, Ubuntu, Soloaris, Mac, etc. The basic Python can be downloaded from the link:

https://www.python.org/downloads/

Python has rich set of libraries for various purposes like large-scale data processing, predictive analytics, scientific computing etc. Based on one's need, the required packages can be downloaded. But, there is a free open source distribution ***Anaconda***, which simplifies package management and deployment. Hence, it is suggested for the readers to install *Anaconda* from the below given link, rather than just installing a simple Python.

https://anaconda.org/anaconda/python

Successful installation of *anaconda* provides you Python in a command prompt, the default editor IDLE and also a browser-based interactive computing environment known as ***jupyter notebook***.

The jupyter notebook allows the programmer to create notebook documents including live code, interactive widgets, plots, equations, images etc. To code in Python using jupyter notebook, search for *jupyter notebook* in windows search (at *Start* menu). Now, a browser window will be opened similar to the one shown in Figure 1.2.

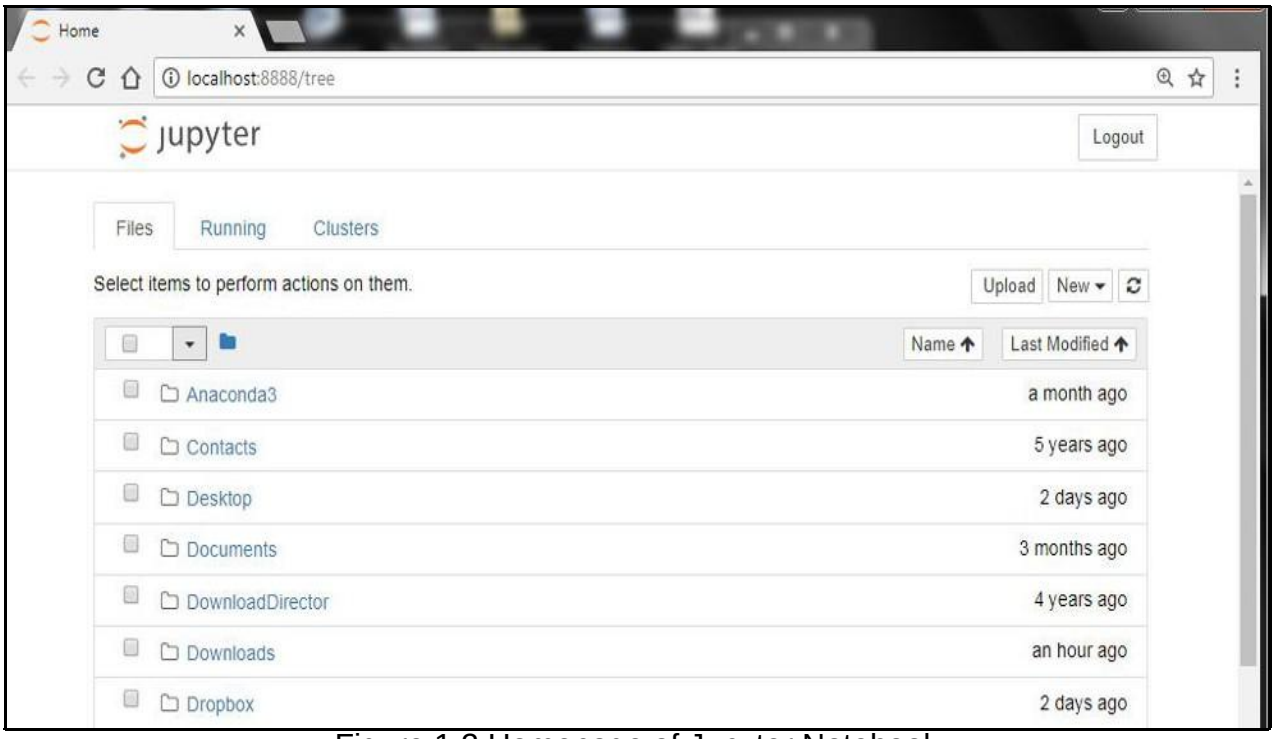


Figure 1.2 Homepage of Jupyter Notebook

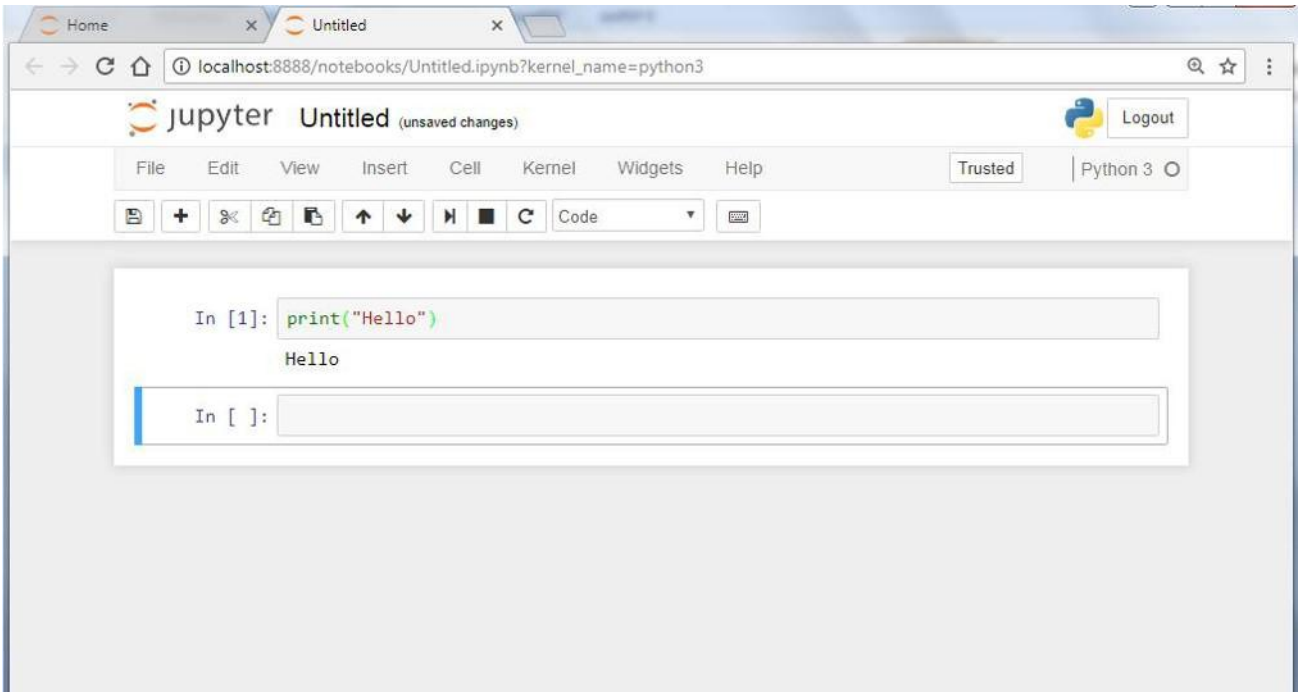


Figure 1.3 IDE of Jupyter Notebook

You can choose the working directory of your choice for storing your work. To open a notebook for Python programming, click on *New* button at the right-side of the screen. Now select *Python 3* from the drop-down list. A new notebook (or workbook will be created as shown in Figure 1.3. Type a command of your choice and press *Ctrl+Enter* to run that command. One can give headings/subheadings etc for the commands being typed, store the entire workbook for future reference etc. Readers are advised to try and experience various options/menu's available.

1.1.6 Conversing with Python

Once Python is installed, one can go ahead with working with Python. Use the IDE of your choice for doing programs in Python. After installing Python (or Anaconda distribution), if you just type 'python' in the command prompt, you will get the message as shown in Figure 1.4. The prompt >>> (usually called as ***chevron***) indicates the system is ready to take Python instructions. If you would like to use the default IDE of Python, that is, the IDLE, then you can just run IDLE and you will get the editor as shown in Figure 1.5.

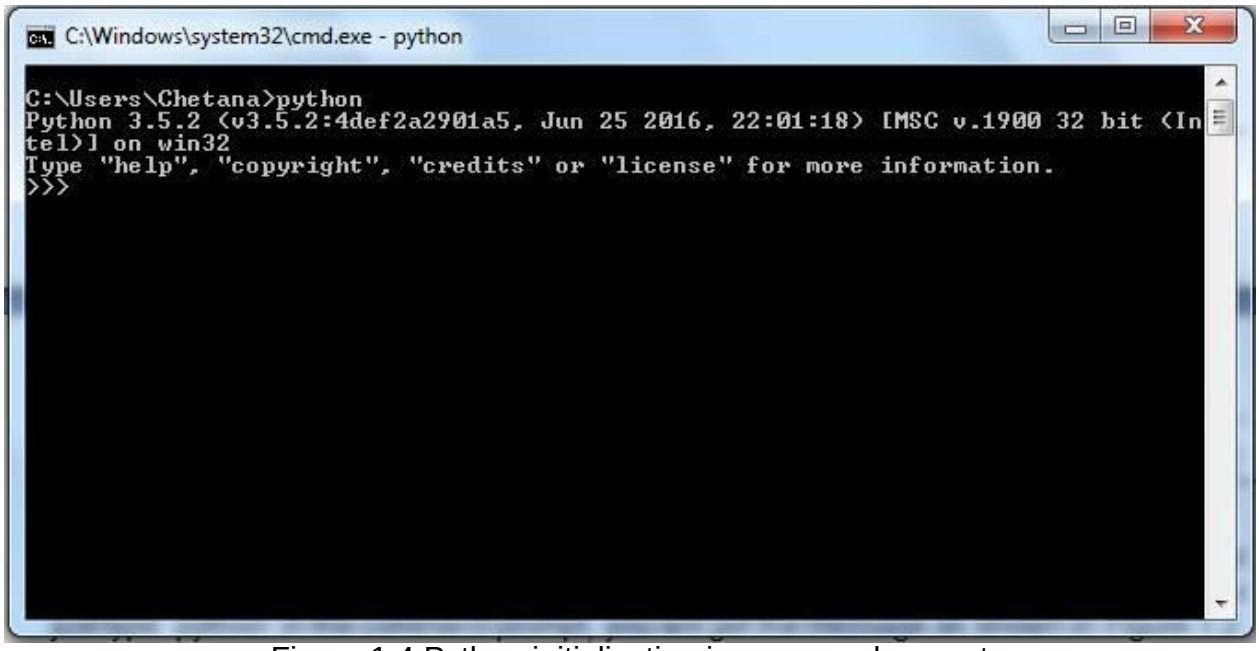


Figure 1.4 Python initialization in command prompt

After understanding the basics of few editors of Python, let us start our communication with Python, by saying *Hello World*. The Python uses *print()* function for displaying the contents. Consider the following code -

```
>>> print("Hello World")       #type this and press enter key
Hello World                    #output displayed
>>>                            #prompt returns again
```

Here, after typing the first line of code and pressing the enter key, we could able to get the output of that line immediately. Then the prompt (>>>) is returned on the screen. This indicates, Python is ready to take next instruction as input for processing.

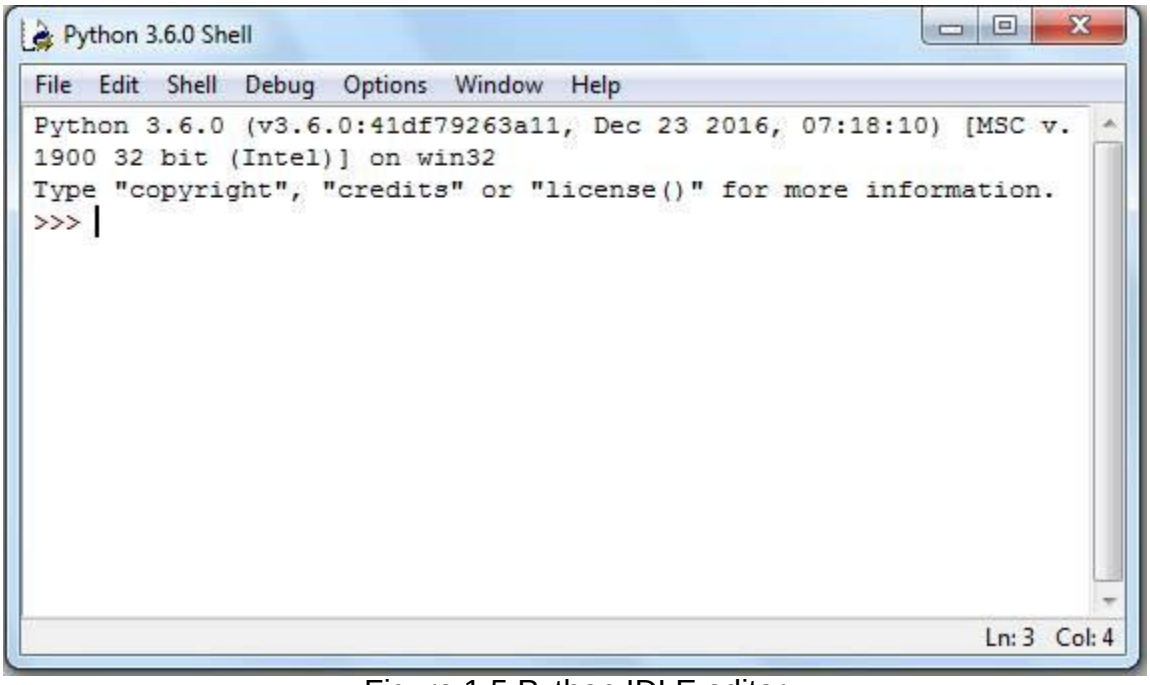


Figure 1.5 Python IDLE editor

Once we are done with the program, we can close or terminate Python by giving *quit()* command as shown -

```
>>> quit()      #Python terminates
```

1.1.7 Terminology: Interpreter and Compiler

All digital computers can understand only the machine language written in terms of zeros and ones. But, for the programmer, it is difficult to code in machine language. Hence, we generally use high level programming languages like Java, C++, PHP, Perl, JavaScript etc. Python is also one of the high level programming languages. The programs written in high level languages are then translated to machine level instruction so as to be executed by CPU. How this translation behaves depending on the type of translators viz. ***compilers*** and ***interpreters***.

A compiler translates the source code of high-level programming language into machine level language. For this purpose, the source code must be a complete program stored in a file (with extension, say, .java, .c, .cpp etc). The compiler generates executable files (usually with extensions .exe, .dll etc) that are in machine language. Later, these executable files are executed to give the output of the program.

On the other hand, interpreter performs the instructions directly, without requiring them to be pre-compiled. Interpreter parses (syntactic analysis) the source code ant interprets it immediately. Hence, every line of code can generate the output immediately, and the source code as a complete set, need not be stored in a file. That is why, in the previous section, the usage of single line `print("Hello World")` could able to generate the output immediately.

Consider an example of adding two numbers -

```
>>> x=10
>>> y=20
>>> z= x+y
>>>  print(z)
30
```

Here, `x, y` and `z` are variables storing respective values. As each line of code above is processed immediately after the line, the variables are storing the given values. Observe that, though each line is treated independently, the knowledge (or information) gained in the previous line will be retained by Python and hence, the further lines can make use of previously used variables. Thus, each line that we write at the Python prompt are logically related, though they look independent.

NOTE that, Python do not require variable declaration (unlike in C, C++, Java etc) before its use. One can use any valid variable name for storing the values. Depending on the type (like number, string etc) of the value being assigned, the type and behavior of the variable name is judged by Python.

1.1.8 Writing a Program

As Python is interpreted language, one can keep typing every line of code one after the other (and immediately getting the output of each line) as shown in previous section. But, in real-time scenario, typing a big program is not a good idea. It is not easy to logically debug such lines. Hence, Python programs can be stored in a file with extension ***.py*** and then can be run. Programs written within a file are obviously reusable and can be run whenever we want. Also, they are transferrable from one machine to other machine via pen-drive, CD etc.

1.1.9 What is a Program?

A program is a sequence of instructions intended to do some task. For example, if we need to count the number of occurrences of each word in a text document, we can write a program to do so. Writing a program will make the task easier compared to manually counting the words in a document. Moreover, most of the times, the program is a generic solution. Hence, the same program may be used to count the frequency of words in another file. The person who does not know anything about the programming also can run this program to count the words.

Programming languages like Python will act as an intermediary between the computer and the programmer. The end-user can request the programmer to write a program to solve one's problem.

1.1.10 The Building Blocks of Programs

There are certain low-level conceptual structures to construct a program in any programming language. They are called as building-blocks of a program and listed below -

- **Input:** Every program may take some inputs from outside. The input may be through keyboard, mouse, disk-file etc. or even through some sensors like microphone, GPS etc.
- **Output:** Purpose of a program itself is to find the solution to a problem. Hence, every program must generate at least one output. Output may be displayed on a monitor or can be stored in a file. Output of a program may even be a music/voice message.
- **Sequential Execution:** In general, the instructions in the program are sequentially executed from the top.
- **Conditional Execution:** In some situations, a set of instructions have to be executed based on the truth-value of a variable or expression. Then conditional constructs (like *if*) have to be used. If the condition is true, one set of instructions will be executed and if the condition is false, the true-block is skipped.
- **Repeated Execution:** Some of the problems require a set of instructions to be repeated multiple times. Such statements can be written with the help of looping structures like *for, while* etc.
- **Reuse:** When we write the programs for general-purpose utility tasks, it is better to write them with a separate name, so that they can be used multiple times whenever/wherever required. This is possible with the help of *functions*.

The art of programming involves thorough understanding of the above constructs and using them legibly.

1.1.11 What Could Possibly Go Wrong?

It is obvious that one can do mistakes while writing a program. The possible mistakes are categorized as below -

- **Syntax Errors:** The statements which are not following the grammar (or syntax) of the programming language are tend to result in syntax errors. Python is a case-sensitive language. Hence, there is a chance that a beginner may do some syntactical mistakes while writing a program. The lines involving such mistakes are encountered by the Python when you run the program and the errors are thrown by specifying possible reasons for the error. The programmer has to correct them and then proceed further.
- **Logical Errors:** Logical error occurs due to poor understanding of the problem. Syntactically, the program will be correct. But, it may not give the expected output. For example, you are intended to find a%b, but, by mistake you have typed a/b. Then it is a logical error.
- **Semantic Errors:** A semantic error may happen due to wrong use of variables, wrong operations or in wrong order. For example, trying to modify un-initialized variable etc.

Note that, some of textbooks/authors refer logical and semantic error both as same, as the distinction between these two is very small.

NOTE: There is one more type of error - runtime error, usually called as *exceptions*. It may occur due to wrong input (like trying to divide a number by zero), problem in database connectivity etc. When a run-time error occurs, the program throws some error, which may not be understood by the normal user. And he/she may not understand how to overcome such errors. Hence, suspicious lines of code have to be treated by the programmer himself by the procedure known as *exception handling*. Python provides mechanism for handling various possible exceptions like *ArithmeticError, FloatingpointError, EOFError, MemoryError* etc. A brief idea about exception handling is there in Section 1.3.7 later in this Module. For more details, interested readers can go through the links -

https://docs.python.org/3/tutorial/errors.html and
https://docs.python.org/2/library/exceptions.html

1.2 VARIABLES, EXPRESSIONS AND STATEMENTS

After understanding some important concepts about programming and programming languages, we will now move on to learn Python as a programming language with its syntax and constructs.

1.2.1 Values and Types

A *value* is one of the basic things in a program. It may be like 2, 10.5, “Hello” etc. Each value in Python has a type. Type of 2 is integer; type of 10.5 is floating point number; “Hello” is string etc. The type of a value can be checked using ***type*** function as shown below -

```
>>> type("hello")
     <class 'str'>        #output
>>> type(3)
     <class 'int'>
>>> type(10.5)
     <class 'float'>
>>> type("15")
     <class 'str'>
```

In the above four examples, one can make out various types *str, int* and *float.* Observe the 4th example - it clearly indicates that whatever enclosed within a double quote is a string.

1.2.2 Variables

A variable is a named-literal which helps to store a value in the program. Variables may take value that can be modified wherever required in the program. Note that, in Python, a variable need not be declared with a specific type before its usage. Whenever you want a variable, just use it. The type of it will be decided by the value assigned to it. A value can be assigned to a variable using *assignment operator* (=). Consider the example given below-

```
>>> x=10
>>> print(x)
     10              #output
```

```
>>> type(x)
     <class  'int'>  #type of x is integer
>>> y="hi"
>>> print(y)
     hi              #output
>>> type(y)
     <class 'str'>   #type of y is string
```

It is observed from above examples that the value assigned to variable determines the type of that variable.

1.2.3 Variable Names and Keywords

It is a good programming practice to name the variable such that its name indicates its purpose in the program. There are certain rules to be followed while naming a variable -

- Variable name must not be a keyword
- They can contain alphabets (lowercase and uppercase) and numbers, but should not start with a number.
- It may contain a special character underscore(_), which is usually used to combine variables with two words like *my_salary, student_name* etc. No other special characters like @, $ etc. are allowed.
- As Python is case-sensitive, variable name *sum* is different from *SUM, Sum* etc.

Examples:

```
>>> 3a=5                        #starting with a number
SyntaxError: invalid syntax
>>> a$=10                       #contains $
SyntaxError: invalid syntax
>>> if=15                       #if is a keyword
SyntaxError: invalid syntax
```

1.2.4 Statements

A *statement* is a small unit of code that can be executed by the Python interpreter. It indicates some action to be carried out. In fact, a program is a sequence of such statements. Following are the examples of statements -

```
>>>  print("hello")       #printing statement
hello
>>> x=5                   #assignment  statement
>>> print(x)              #printing statement
```

1.2.5 Operators and Operands

Special symbols used to indicate specific tasks are called as *operators*. An operator may work on single operand (unary operator) or two operands (binary operator). There are several types of operators like arithmetic operators, relational operators, logical operators etc. in Python.

Arithmetic Operators are used to perform basic operations as listed in Table 1.2.

Table 1.2 Arithmetic Operators

Operator	Meaning	Example
+	Addition	Sum= a+b
-	Subtraction	Diff= a-b
*	Multiplication	Pro= a*b
/	Division	Q = a/b X = 5/3 (X will get a value 1.666666667)
//	Floor Division - returns only integral part after division	F = a//b X= 5//3 (X will get a value 1)
%	Modulus - remainder after division	R = a %b (Remainder after dividing a by b)
**	Exponent	E = x** y (means x to the powder of y)

Relational or Comparison Operators are used to check the relationship (like less than, greater than etc) between two operands. These operators return a Boolean value - either *True* or *False*.

Assignment Operators: Apart from simple assignment operator = which is used for assigning values to variables, Python provides compound assignment operators. For example,

x= x+y

can be written as -

x+=y

Now, += is compound assignment operator. Similarly, one can use most of the arithmetic and bitwise operators (only binary operators, but not unary) like *, /, %, //, &, ^ etc. as compound assignment operators. For example,

```
>>> x=3
>>> y=5
>>> x+=y        #x=x+y
>>> print(x)
     8
>>> y//=2       #y=y//2
>>> print(y)
     2          #only integer part will be printed
```

NOTE:

1. Python has a special feature - one can assign values of different types to multiple variables in a single statement. For example,

```
>>> x, y, st=3, 4.2, "Hello"
>>> print("x= ", x, " y= ",y, " st= ", st)
     x=3  y=4.2  st=Hello
```

2. Python supports bitwise operators like &(AND), | (OR), ~(NOT), ^(XOR), >>(right shift) and <<(left shift). These operators will operate on every bit of the operands. Working procedure of these operators is same as that in other languages like C and C++.
3. There are some special operators in Python viz. *Identity operator* (`is` and `is not`) and *membership operator* (`in` and `not in`). These will be discussed in further Modules.

1.2.6 Expressions

A combination of values, variables and operators is known as expression. Following are few examples of expression -

```
x=5
y=x+10
z= x-y*3
```

The Python interpreter evaluates simple expressions and gives results even without *print()*. For example,

```
>>> 5
5                #displayed as it is
>>> 1+2
3                #displayed the sum
```

But, such expressions do not have any impact when written into Python script file.

1.2.7 Order of Operations

When an expression contains more than one operator, the evaluation of operators depends on the *precedence of operators*. The Python operators follow the precedence rule (which can be remembered as *PEMDAS*) as given below -

- **Parenthesis** have the highest precedence in any expression. The operations within parenthesis will be evaluated first. For example, in the expression (a+b)*c, the addition has to be done first and then the sum is multiplied with c.
- **Exponentiation** has the 2nd precedence. But, it is right associative. That is, if there are two exponentiation operations continuously, it will be evaluated from right to left (unlike most of other operators which are evaluated from left to right). For example,

```
>>> print(2**3)   #It is 2^3
   8
>>> print(2**3**2) #It is 2^(3^2), so to be evaluated from right
   512
```

- **Multiplication and Division** are the next priority. Out of these two operations, whichever comes first in the expression is evaluated.

```
>>> print(5*2/4)       #multiplication and then division
   2.5
>>> print(5/4*2)       #division and then multiplication
   2.5
```

- **Addition and Subtraction** are the least priority. Out of these two operations, whichever appears first in the expression is evaluated.

1.2.8 String Operations

String concatenation can be done using + operator as shown below -

```
>>> x="32"
>>> y="45"
>>>  print(x+y)
   3245
```

Observe the output: here, the value of y (a string “45”, but not a number 45) is placed just in front of value of x(a string “32”). Hence the result would be “3245” and its type would be *string*.

NOTE: One can use single quotes to enclose a string value, instead of double quotes.

1.2.9 Asking the User for Input

Python uses the built-in function ***input()*** to read the data from the keyboard. When this function is invoked, the user-input is expected. The input is read till the user presses enterkey. For example:

```
>>> str1=input()
Hello how are you?        #user input
>>> print(“String is “,str1)
String is Hello how are you?        #printing str1
```

When *input()* function is used, the curser will be blinking to receive the data. For a better understanding, it is better to have a prompt message for the user informing what needs to be entered as input. The *input()* function itself can be used to do so, as shown below -

```
>>> str1=input("Enter a string: ")
Enter a string: Hello
>>> print("You have entered: ",str1)
You have entered:  Hello
```

One can use new-line character \n in the function *input()* to make the cursor to appear in the next line of prompt message -

```
>>> str1=input("Enter a string:\n")
Enter a string:
Hello                    #cursor is pushed here
```

The key-board input received using *input()* function is always treated as a string type. If you need an integer, you need to convert it using the function ***int()***. Observe the following example -

```
>>> x=input("Enter x:")
Enter x:10          #x takes the value “10”, but not 10
>>> type(x)         #So, type of x would be str
<class 'str'>
```

```
>>> x=int(input("Enter x:"))        #use  int()
Enter x:10
>>> type(x)          #Now, type of x is int
<class 'int'>
```

A function ***float()*** is used to convert a valid value enclosed within quotes into float number as shown below -

```
>>> f=input("Enter a float value:")
Enter a float value: 3.5
>>> type(f)
<class 'str'>        #f is actually a string "3.5"
>>> f=float(f)       #converting "3.5" into float value 3.5
>>> type(f)
<class 'float'>
```

A function ***chr()*** is used to convert an integer input into equivalent ASCII character.

```
>>> a=int(input("Enter an integer:"))
Enter an integer:65
>>> ch=chr(a)
>>> print("Character Equivalent of ", a, "is ",ch)
Character Equivalent of 65 is A
```

There are several such other utility functions in Python, which will be discussed later.

1.2.10 Comments

It is a good programming practice to add comments to the program wherever required. This will help someone to understand the logic of the program. Comment may be in a single line or spread into multiple lines. A single-line comment in Python starts with the symbol **#**. Multiline comments are enclosed within a pair of 3-single quotes.

Ex1.

```
#This is a single-line comment
```

Ex2.

```
''' This
is a
multiline
comment '''
```

Python (and all programming languages) ignores the text written as comment lines. They are only for the programmer's (or any reader's) reference.

1.2.11 Choosing Mnemonic Variable Names

Choosing an appropriate name for variables in the program is always at stake. Consider the following examples -

Ex1.

```
a=10000
b=0.3*a
c=a+b
print(c)        #output is 13000
```

Ex2.

```
basic=10000
da=0.3*basic
gross_sal=basic+da
print("Gross Sal = ",gross_sal)     #output is 13000
```

One can observe that both of these two examples are performing same task. But, compared to Ex1, the variables in Ex2 are indicating what is being calculated. That is, variable names in Ex2 are indicating the purpose for which they are being used in the program. Such variable names are known as ***mnemonic variable names***. The word *mnemonic* means *memory aid*. The mnemonic variables are created to help the programmer to remember the purpose for which they have been created.

Python can understand the set of reserved words (or keywords), and hence it flashes an error when such words are used as variable names by the programmer. Moreover, most of the Python editors have a mechanism to show keywords in a different color. Hence, programmer can easily make out the keyword immediately when he/she types that word.

1.2.12 Debugging

Some of the common errors a beginner programmer may make are syntax errors. Though Python flashes the error with a message, sometimes it may become hard to understand the cause of errors. Some of the examples are given here -

Ex1.

```
>>> avg sal=10000
SyntaxError: invalid syntax
```

Here, there is a space between the terms `avg` and `sal`, which is not allowed.

Ex2.

```
>>> m=09
SyntaxError: invalid token
```

Python does not allow preceding zeros for numeric values.

Ex3.

```
>>> basic=2000
>>> da=0.3*Basic
NameError: name 'Basic' is not defined
```

As Python is case sensitive, `basic` is different from `Basic`.

As shown in above examples, the syntax errors will be alerted by Python. But, programmer is responsible for logical errors or semantic errors. Because, if the program does not yield into expected output, it is due to mistake done by the programmer, about which Python is unaware of.

1.3 CONDITIONAL EXECUTION

In general, the statements in a program will be executed sequentially. But, sometimes we need a set of statements to be executed based on some conditions. Such situations are discussed in this section.

1.3.1 Boolean Expressions

A *Boolean Expression* is an expression which results in *True* or *False*. The *True* and *False* are special values that belong to class ***bool*.** Check the following -

```
>>> type(True)
<class   'bool'>
>>>  type(False)
<class 'bool'>
```

Boolean expression may be as below -

```
>>> 10==12
False
>>>  x=10
>>>  y=10
>>>  x==y
True
```

Various comparison operations are shown in Table 1.3.

Table 1.3 Relational (Comparison) Operators

Operator	Meaning	Example
>	Greater than	a>b
<	Less than	a<b
>=	Greater than or equal to	a>=b
<=	Less than or equal to	a<=b
==	Comparison	a==b
!=	Not equal to	a !=b
is	Is same as	a is b
is not	Is not same as	a is not b

Few Examples:

```
>>> a=10
>>> b=20
>>> x= a>b
>>> print(x)
     False
>>> print(a==b)
     False
```

```
>>> print("a<b is ", a<b)
     a<b is True
>>> print("a!=b is", a!=b)
     a!=b is True
>>> 10 is 20
     False
>>> 10 is 10
     True
```

NOTE: For a first look, the operators *==* and *is* look same. Similarly, the operators *!=* and *is not* look the same. But, the operators *==* and *!=* does the ***equality test*.** That is, they will compare the values stored in the variables. Whereas, the operators *is* and *is not* does the ***identity test***. That is, they will compare whether two objects are same. Usually, two objects are same when their memory locations are same. This concept will be more clear when we take up classes and objects in Python.

1.3.2 Logical Operators

There are 3 logical operators in Python as shown in Table 1.4. (NOTE that symbols like &&, || are not used in Python for representing logical operators)

Table 1.4 Logical Operators

Operator	Meaning	Example
and	Returns true, if both operands are true	a and b
or	Returns true, if any one of two operands is true	a or b
not	Return true, if the operand is false (it is a unary operator)	not a

NOTE:

1. Logical operators treat the operands as Boolean (True or False).
2. Python treats any non-zero number as True and zero as False.
3. While using *and* operator, if the first operand is False, then the second operand is not evaluated by Python. Because *False and'*ed with anything is False.
4. In case of *or* operator, if the first operand is True, the second operand is not evaluated. Because True *or'*ed with anything is True.

Example 1 (with Boolean Operands)**:**

```
>>> x= True
>>> y= False
>>> print('x and y is', x and y)
     x and y is False
>>> print('x or y is', x or y)
     x or y is True
>>> print('Complement of x is ', not x)
     Complement of x is False
```

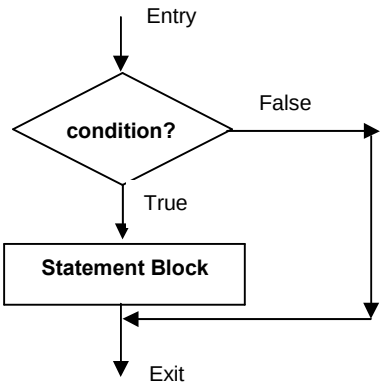
Example 2 (With numeric Operands)**:**

```
>>> a=-3
>>> b=10
>>> print(a and b)        #and operation
     10         #a is true, hence b is evaluated and printed

>>> print(a or b)         #or operation
     -3                   #a is true, hence b is not evaluated #0
>>> print(0 and 5)       is false, so printed
     0
```

1.3.3 Conditional Execution

The basic level of conditional execution can be achieved in Python by using *if* statement. The syntax and flowcharts are as below -

```
if condition:
      Statement block
```



Observe the colon symbol after *condition*. When the *condition* is true, the *Statement block* will be executed. Otherwise, it is skipped. A set (block) of statements to be executed under *if* is decided by the indentation (tab space) given.

Consider an example -

```
>>> x=10
>>> if x<40:
          print("Fail")        #observe indentation after if

Fail           #output
```

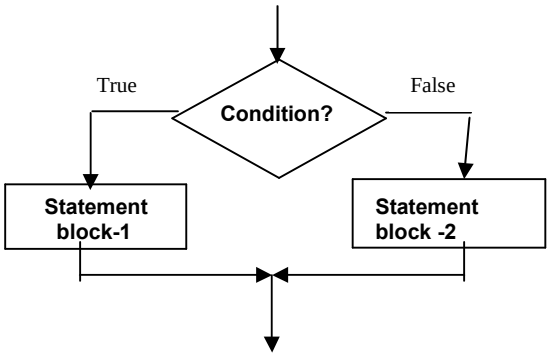
Usually, the *if* conditions have a statement block. In any case, the programmer feels to do nothing when the condition is true, the statement block can be skipped by just typing ***pass*** statement as shown below -

```
>>> if x<0:
          pass           #do nothing when x is negative
```

1.3.4 Alternative Execution

A second form of *if* statement is *alternative execution*. Here, when the condition is true, one set of statements will be executed and when the condition is false, another set of statements will be executed. The syntax and flowchart are as given below -

```
if condition:
   Statement block -1
else:
   Statement block -2
```



As the *condition* will be either true or false, only one among *Statement block-1* and *Statement block-2* will be get executed. These two alternatives are known as ***branches*.**

Example:

```
x=int(input("Enter  x:"))
if x%2==0:
    print("x  is  even")
else:
     print("x is odd")
```

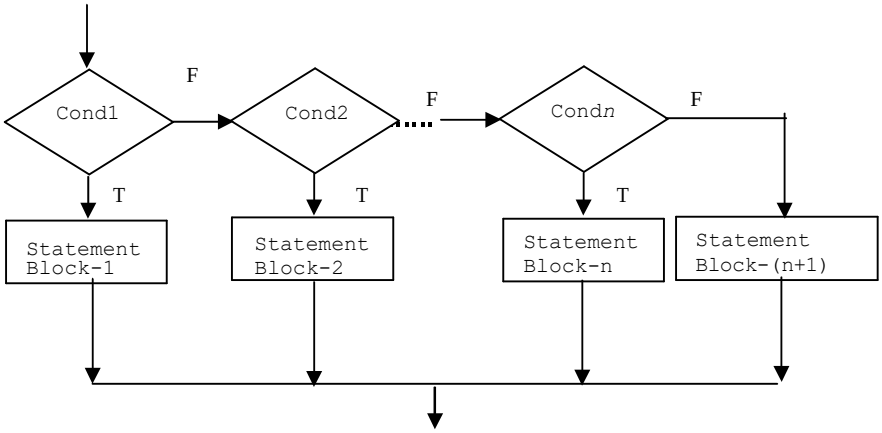
Sample output:

```
Enter  x:  13
x is odd
```

1.3.5 Chained Conditionals

Some of the programs require more than one possibility to be checked for executing a set of statements. That means, we may have more than one branch. This is solved with the help of *chained conditionals*. The syntax and flowchart is given below -

```
if condition1:
      Statement Block-1
elif condition2:
      Statement Block-2
      |
      |
      |
      |
elif condition_n:
      Statement Block-n
else:
      Statement Block-(n+1)
```



The conditions are checked one by one sequentially. If any condition is satisfied, the respective statement block will be executed and further conditions are not checked. Note that, the last *else* block is not necessary always.

Example:

```
marks=float(input("Enter marks:"))
if marks >= 80:
    print("First Class with Distinction")
elif marks >= 60 and marks < 80:
    print("First Class")
elif marks >= 50 and marks < 60:
    print("Second Class")
elif marks >= 35 and marks < 50:
    print("Third Class")
else:
    print("Fail")
```

Sample Output:

Enter marks: 78
First Class

1.3.6 Nested Conditionals

The conditional statements can be nested. That is, one set of conditional statements can be nested inside the other. It can be done in multiple ways depending on programmer's requirements. Examples are given below -

Ex1.

```
marks=float(input("Enter marks:"))
if marks>=60:
    if marks<70:
        print("First  Class")
    else:
        print("Distinction")
```

Sample Output:

Enter marks:68
First Class

Here, the outer condition `marks>=60` is checked first. If it is true, then there are two branches for the inner conditional. If the outer condition is false, the above code does nothing.

Ex2.

```
gender=input("Enter gender:")
age=int(input("Enter age:"))

if gender == "M" :
    if age >= 21:
        print("Boy, Eligible for Marriage")
    else:
        print("Boy, Not Eligible for Marriage")
elif gender == "F":
    if age >= 18:
```

```
        print("Girl, Eligible for Marriage")
    else:
        print("Girl, Not Eligible for Marriage")
```

Sample Output:

Enter gender: F
Enter age: 17
Girl, Not Eligible for Marriage

NOTE: Nested conditionals make the code difficult to read, even though there are proper indentations. Hence, it is advised to use logical operators like *and* to simplify the nested conditionals. For example, the outer and inner conditions in **Ex1** above can be joined as -

```
if marks>=60 and marks<70:
     #do something
```

1.3.7 Catching Exceptions using try and except

As discussed in Section 1.1.11, there is a chance of runtime error while doing some program. One of the possible reasons is wrong input. For example, consider the following code segment -

```
a=int(input("Enter  a:"))
b=int(input("Enter  b:"))
c=a/b
print(c)
```

When you run the above code, one of the possible situations would be -

```
Enter a:12
Enter b:0
Traceback (most recent call last):
  File "C:\Users\Chetana\Dropbox\PythonNotes\p1.py", line 154, in
<module>
    c=a/b
ZeroDivisionError: division by zero
```

For the end-user, such type of system-generated error messages is difficult to handle. So the code which is prone to runtime error must be executed conditionally within *try* block. The ***try*** block contains the statements involving suspicious code and the ***except*** block contains the possible remedy (or instructions to user informing what went wrong and what could be the way to get out of it). If something goes wrong with the statements inside ***try*** block, the ***except*** block will be executed. Otherwise, the except-block will be skipped. Consider the example -

```
a=int(input("Enter  a:"))
b=int(input("Enter  b:"))
try:
    c=a/b
    print(c)
```

```
except:
    print("Division by zero is not possible")
```

Output:

```
Enter  a:12
Enter b:0
Division by zero is not possible
```

Handling an exception using *try* is called as ***catching*** an exception. In general, catching an exception gives the programmer to fix the probable problem, or to try again or at least to end the program gracefully.

1.3.8 Short-Circuit Evaluation of Logical Expressions

When a logical expression (expression involving operands *and, or, not*) is being evaluated, it will be processed from left to right. For example, consider the statements -

```
x= 10
y=20
if x<10 and x+y>25:
     #do something
```

Here, the expression `x<10 and x+y>25` involves the logical operator `and`. Now, `x<10` is evaluated first, which results to be *False.* As there is an `and` operator, irrespective of the result of `x+y>25`, the whole expression will be *False*. In such situations, Python ignores the remaining part of the expression. This is known as ***short-circuiting*** the evaluation. When the first part of logical expression results in *True*, then the second part has to be evaluated to know the overall result.

The short-circuiting not only saves the computational time, but it also leads to a technique known as ***guardian pattern*.** Consider following sequence of statements -

```
>>> x=5
>>> y=0
>>>  x>=10  and  (x/y)>2
False

>>> x>=2 and (x/y)>2
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x>=2 and (x/y)>2
ZeroDivisionError: division by zero
```

Here, when we executed the statement `x>=10 and (x/y)>2`, the first half of logical expression itself was *False* and hence by applying short-circuit rule, the remaining part was not executed at all. Whereas, in the statement `x>=2 and (x/y)>2`, the first half is *True* and the second half is resulted in runtime-error. Thus, in the expression `x>=10 and (x/y)>2`, short-circuit rule acted as a *guardian* by preventing an error.

One can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:

```
>>> x=5
>>> y=0
>>>  x>=2  and  y!=0  and(x/y)>2
False
```

Here, `x>=2` results in *True*, but `y!=0` evaluates to be False. Hence, the expression `(x/y)>2` is never reached and possible error is being prevented from happening.

1.3.9 Debugging

One can observe from previous few examples that when a runtime error occurs, it displays a term *Traceback* followed by few indications about errors. A *traceback* is a stack trace from the point of error-occurrence down to the call-sequence till the point of call. This is helpful when we start using functions and when there is a sequence of multiple function calls from one to other. Then, traceback will help the programmer to identify the exact position where the error occurred. Most useful part of error message in traceback are -

- What kind of error it is
- Where it occurred

Compared to runtime errors, syntax errors are easy to find, most of the times. But, *whitespace* errors in syntax are quite tricky because spaces and tabs are invisible. For example -

```
>>>    x=10
>>>  y=15
 SyntaxError: unexpected indent
```

The error here is because of additional space given before y. As Python has a different meaning (separate block of code) for indentation, one cannot give extra spaces as shown above.

In general, error messages indicate where the problem has occurred. But, the actual error may be before that point, or even in previous line of code.

1.4 FUNCTIONS

Functions are the building blocks of any programming language. A sequence of instructions intended to perform a specific independent task is known as a *function*. In this section, we will discuss various types of built-in functions, user-defined functions, applications/uses of functions etc.

1.4.1 Function Calls

A function is a named sequence of instructions for performing a task. When we define a function we will give a valid name to it, and then specify the instructions for performing

required task. Later, whenever we want to do that task, a function is *called* by its name. Consider an example -

```
>>> type(15)
<class 'int'>
```

Here *type* is a function name, 15 is the argument to a function and `<class 'int'>` is the result of the function. Usually, a function *takes* zero or more arguments and *returns* the result.

1.4.2 Built-in Functions

Python provides a rich set of built-in functions for doing various tasks. The programmer/user need not know the internal working of these functions; instead, they need to know only the purpose of such functions. Some of the built in functions are given below -

- **max():** This function is used to find maximum value among the arguments. It can be used for numeric values or even to strings.
  - `max(10, 20, 14, 12)  #maximum of 4 integers`
    `20`
  - `max("hello world")`
    `'w'  #character having maximum ASCII code`
  - `max(3.5, -2.1, 4.8, 15.3, 0.2)`
    `15.3  #maximum of 5 floating point values`

- **min():** As the name suggests, it is used to find minimum of arguments.
  - `min(10, 20, 14, 12)  #minimum of 4 integers`
    `10`
  - `min("hello world")`
    `' '  #space has least ASCII code here`
  - `min(3.5, -2.1, 4.8, 15.3, 0.2)`
    `-2.1  #minimum of 5 floating point values`

- **len():** This function takes a single argument and finds its length. The argument can be a string, list, tuple etc.
  - len(“hello how are you?”)
    18

There are many other built-in functions available in Python. They are discussed in further Modules, wherever they are relevant.

1.4.3 Type Conversion Functions

As we have seen earlier (while discussing *input()* function), the type of the variable/value can be converted using functions ***int()***, ***float()***, ***str()***. Consider following few examples -

- `int('20')  #integer enclosed within single quotes`
  `20  #converted to integer type`
- `int("20")  #integer enclosed within double quotes`
  `20`

```
int("hello")    #actual string cannot be converted to int
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    int("hello")
ValueError: invalid literal for int() with base 10: 'hello'

int(3.8)        #float value being converted to integer
3               #round-off will not happen, fraction is ignored
int(-5.6)
-5
float('3.5')    #float enclosed within single quotes
3.5             #converted to float type
float(42)       #integer is converted to float
42.0
str(4.5)        #float converted to string
'4.5'
str(21)         #integer converted to string
'21'
```

1.4.4 Random Numbers

Most of the programs that we write are *deterministic*. That is, the input (or range of inputs) to the program is pre-defined and the output of the program is one of the expected values. But, for some of the real-time applications in science and technology, we need randomly generated output. This will help in simulating certain scenario. Radom number generation has important applications in games, noise detection in electronic communication, statistical sampling theory, cryptography, political and business prediction etc. These applications require the program to be *nondeterministic.* There are several algorithms to generate random numbers. But, as making a program completely *nondeterministic* is difficult and may lead to several other consequences, we generate ***pseudo-random numbers***. That is, the type (integer, float etc) and range (between 0 and 1, between 1 and 100 etc) of the random numbers are decided by the programmer, but the actual numbers are unknown. Moreover, the algorithm to generate the random number is also known to the programmer. Thus, the random numbers are generated using deterministic computation and hence, they are known as pseudo-random numbers!!

Python has a module ***random*** for the generation of random numbers. One has to *import* this module in the program. The function used is also ***random()***. By default, this function generates a random number between 0 and 1 (excluding 1). For example -

```
import random              #module random is imported
print(random.random())     #random() function is invoked
0.7430852580883088         #a random number generated
print(random.random())
0.5287778188896328         #one more random number
```

Importing a module creates an object. Using this object, one can access various functions and/or variables defined in that module. Functions are invoked using a dot operator.

There are several other functions in the module *random* apart from the function *random()*. (Do not get confused with module name and function name. Observe the parentheses while referring a function name). Few are discussed hereunder:

- **randint():** It takes two arguments *low* and *high* and returns a random integer between these two arguments (both *low* and *high* are inclusive). For example,

```
>>>random.randint(2,20)
   14                    #integer between 2 and 20 generated
>>>  random.randint(2,20)
   10
```

- **choice():** This function takes a sequence (a *list* type in Python) of numbers as an argument and returns one of these numbers as a random number. For example,

```
>>> t=[1,2, -3, 45, 12, 7, 31, 22]   #create a list t
>>> random.choice(t)      #t is argument to choice()
    12                    #one of the elements in t
>>> random.choice(t)
    1                     #one of the elements in t
```

Various other functions available in *random* module can be used to generate random numbers following several probability distributions like Gaussian, Triangular, Uniform, Exponential, Weibull, Normal etc.

1.4.5 Math Functions

Python provides a rich set of mathematical functions through the module ***math***. To use these functions, the ***math*** module has to be imported in our code. Some of the important functions available in *math* are given hereunder -

- **sqrt():** This function takes one numeric argument and finds the square root of that argument.

```
>>> math.sqrt(34)          #integer argument
     5.830951894845301
>>> math.sqrt(21.5)        #floating point argument
     4.636809247747852
```

- **pi:** The constant value ***pi*** can be used directly whenever we require.

```
>>>print (math.pi)
     3.141592653589793
```

- **log10():** This function is used to find logarithm of the given argument, to the base 10.

```
>>> math.log10(2)
     0.3010299956639812
```

- **log():** This is used to compute natural logarithm (base e) of a given number.

```
>>> math.log(2)
    0.6931471805599453
```

- **sin():** As the name suggests, it is used to find *sine* value of a given argument. Note that, the argument must be in radians (not degrees). One can convert the number of degrees into radians by multiplying pi/180 as shown below -

```
>>>math.sin(90*math.pi/180)    #sin(90) is 1
    1.0
```

- **cos():** Used to find *cosine* value -

```
>>>math.cos(45*math.pi/180)
    0.7071067811865476
```

- **tan():** Function to find tangent of an angle, given as argument.

```
>>> math.tan(45*math.pi/180)
    0.9999999999999999
```

- **pow():** This function takes two arguments x and y, then finds x to the power of y.

```
>>> math.pow(3,4)
81.0
```

1.4.6 Adding New Functions (User-defined Functions)

Python facilitates programmer to define his/her own functions. The function written once can be used wherever and whenever required. The syntax of user-defined function would be -

```
def fname(arg_list):
    statement_1
    statement_2
    ……
    Statement_n
    return value
```

Here *def* is a keyword indicating it as a function definition.
fname is any valid name given to the function
arg_list is list of arguments taken by a function. These are treated as inputs to the function from the position of function call. There may be zero or more arguments to a function.
statements are the list of instructions to perform required task.
return is a keyword used to return the output *value*. This statement is optional

The first line in the function ***def*** `fname(arg_list)` is known as ***function header***. The remaining lines constitute ***function body***. The function header is terminated by a colon and the function body must be indented. To come out of the function, indentation must be terminated. Unlike few other programming languages like C, C++ etc, there is no *main()*

function or specific location where a user-defined function has to be called. The programmer has to invoke (call) the function wherever required.

Consider a simple example of user-defined function -

Observe indentation

Statements outside the function without indentation. `myfun()` is called here.

```
def myfun():
    print("Hello")
    print("Inside the function")


print("Example of function")
myfun()
print("Example over")
```

The output of above program would be -

```
Example of function
Hello
Inside the function
Example over
```

The function definition creates an object of type *function*. In the above example, `myfun` is internally an object. This can be verified by using the statement -

```
>>>print(myfun)                # myfun without parenthesis
     <function myfun at 0x0219BFA8>
>>> type(myfun)               # myfun without parenthesis
     <class 'function'>
```

Here, the first output indicates that `myfun` is an object which is being stored at the memory address `0x0219BFA8` (0x indicates octal number). The second output clearly shows `myfun` is of type `function`.

(**NOTE:** In fact, in Python every type is in the form of class. Hence, when we apply *type* on any variable/object, it displays respective class name. The detailed study of classes will be done in Module 4.)

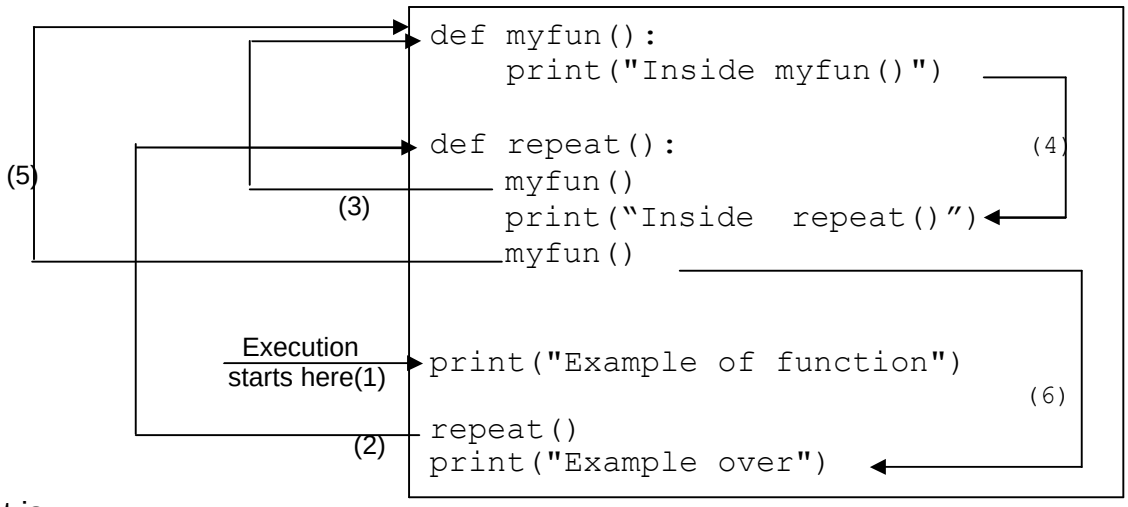
The *flow of execution* of every program is sequential from top to bottom, a function can be invoked only after defining it. Usage of function name before its definition will generate error. Observe the following code:

```
print("Example of function")
myfun()                       #function call before definition
print("Example over")

def myfun():                  #function definition is here
    print("Hello")
    print("Inside the function")
```

The above code would generate error saying

```
NameError: name 'myfun' is not defined
```

Functions are meant for code-reusability. That is, a set of instructions written as a function need not be repeated. Instead, they can be called multiple times whenever required. Consider the enhanced version of previous program as below -



The output is -

```
Example of function
Inside myfun()
Inside  repeat()
Inside   myfun()
Example over
```

Observe the output of the program to understand the flow of execution of the program. Initially, we have two function definitions `myfun()` and `repeat()` one after the other. But, functions are not executed unless they are called (or invoked). Hence, the first line to execute in the above program is -

```
print("Example of function")
```

Then, there is a function call `repeat()`. So, the program control jumps to this function. Inside `repeat()`, there is a call for `myfun()`. Now, program control jumps to `myfun()` and executes the statements inside and returns back to `repeat()` function. The statement `print("Inside repeat()")` is executed. Once again there is a call for `myfun()` function and hence, program control jumps there. The function `myfun()` is executed and returns to `repeat()`. As there are no more statements in `repeat()`, the control returns to the original position of its call. Now there is a statement `print("Example over")` to execute, and program is terminated.

1.4.7 Parameters and Arguments

In the previous section, we have seen simple example of a user-defined function, where the function was without any argument. But, a function may take arguments as an input from

the calling function. Consider an example of a function which takes a single argument as below -

```
def test(var):
    print("Inside test()")
    print("Argument is ",var)


print("Example of function with arguments")
x="hello"
test(x)
y=20
test(y)
print("Over!!")
```

The output would be -

```
Example of function with arguments
Inside test()
Argument is hello
Inside test()
Argument is 20
Over!!
```

In the above program, `var` is called as ***parameter*** and `x` and `y` are called as ***arguments.*** The argument is being passed when a function `test()` is invoked. The parameter receives the argument as an input and statements inside the function are executed. As Python variables are not of specific data types in general, one can pass any type of value to the function as an argument.

Python has a special feature of applying multiplication operation on arguments while passing them to a function. Consider the modified version of above program -

```
def test(var):
    print("Inside test()")
    print("Argument is ",var)


print("Example of function with arguments")
x="hello"
test(x*3)
y=20
test(y*3)
print("Over!!")
```

The output would be -

```
Example of function with arguments
Inside test()
Argument is hellohellohello          #observe repetition
Inside test()
Argument is 60                       #observe multiplication
Over!!
```

One can observe that, when the argument is of type *string*, then multiplication indicates that string is repeated 3 times. Whereas, when the argument is of numeric type (here, integer), then the value of that argument is literally multiplied by 3.

1.4.8 Fruitful Functions and void Functions

A function that performs some task, but do not return any value to the calling function is known as ***void function***. The examples of user-defined functions considered till now are void functions. The function which returns some result to the calling function after performing a task is known as ***fruitful function.*** The built-in functions like mathematical functions, random number generating functions etc. that have been considered earlier are examples for fruitful functions. One can write a user-defined function so as to return a value to the calling function as shown in the following example -

```
def sum(a,b):
    return a+b

x=int(input("Enter a number:"))
y=int(input("Enter another number:"))

s=sum(x,y)
print("Sum of two numbers:",s)
```

The sample output would be -

```
Enter a number:3
Enter another number:4
Sum of two numbers: 7
```

In the above example, The function `sum()` take two arguments and returns their sum to the receiving variable `s`.

When a function returns something and if it is not received using a LHS variable, then the return value will not be available. For instance, in the above example if we just use the statement `sum(x,y)` instead of `s=sum(x,y)`, then the value returned from the function is of no use. On the other hand, if we use a variable at LHS while calling void functions, it will receive `None`. For example,

```
p= test(var)    #function used in previous example
print(p)
```

Now, the value of `p` would be printed as `None`. Note that, `None` is not a string, instead it is of type `class 'NoneType'`. This type of object indicates *no value*.

1.4.9 Why Functions?

Functions are essential part of programming because of following reasons -

- Creating a new function gives the programmer an opportunity to name a group of statements, which makes the program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. If any modification is required, it can be done only at one place.
- Dividing a long program into functions allows the programmer to debug the independent functions separately and then combine all functions to get the solution of original problem.
- Well-designed functions are often useful for many programs. The functions written once for a specific purpose can be re-used in any other program.

For the Curious Minds (Something beyond the syllabus)

Special parameters of print() - *sep* and *end* :

Consider an example of printing two values using *print()* as below -

```
>>> x=10
>>> y=20
>>> print(x,y)
10 20          #space is added between two values
```

Observe that the two values are separated by a space without mentioning anything specific. This is possible because of the existence of an argument ***sep*** in the *print()* function whose default value is white space. This argument makes sure that various values to be printed are separated by a space for a better representation of output.

The programmer has a liberty in Python to give any other character(or string) as a separator by explicitly mentioning it in *print()* as shown below -

```
>>> print("18","2","2018",sep='-')
     18-2-2018
```

We can observe that the values have been separated by hyphen, which is given as a value for the argument *sep*. Consider one more example -

```
>>> college="RNSIT"
>>> address="Channasandra"
>>> print(college, address, sep='@')
     RNSIT@Channasandra
```

If you want to deliberately suppress any separator, then the value of *sep* can be set with empty string as shown below -

```
>>> print("Hello","World", sep='')
     HelloWorld
```

You might have observed that in Python program, the *print()* adds a new line after printing the data. In a Python script file, if you have two statements like -

```
print("Hello")
print("World")
```

then, the output would be

```
Hello
World
```

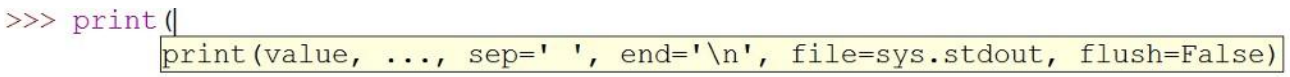
This may be quite unusual for those who have experienced programming languages like C, C++ etc. In these languages, one has to specifically insert a new-line character (\n) to get the output in different lines. But, in Python without programmer's intervention, a new line will be inserted. This is possible because, the *print()* function in Python has one more special argument ***end*** whose default value itself is new-line. Again, the default value of this argument can be changed by the programmer as shown below (Run these lines using a script file, but not in the terminal/command prompt) -

```
print("Hello", end= '@')
print("World")
```

The output would be -

```
Hello@World
```

In fact, when you just type *print* and open a parentheses in any Python IDE, the intelliSense (the context-aware code completion feature of a programming language which helps the programmer with certain suggestions using a pale-yellow box) of *print()* will show the existence of ***sep*** and ***end*** arguments as below -



The above figure clearly indicates that the ***sep*** and ***end*** have the default values space and new-line respectively.

(NOTE: You can see two more arguments *file* and *flush* here. The default value *sys.stdout* of the argument *file* indicates that *print()* will send the data to standard output, which is usually keyboard. When you are willing to print the data into a specific file, the file-object can be given as a value for *file* argument. The *flush* argument with *True* value makes sure that operations are successfully completed and the values are flushed into the memory from the buffer. The default value of *flush* is False, because in most of the cases we need not check whether the data is really got flushed or not - as it would be happening even otherwise. While printing the data into a file (that is, when a file is open for write purpose), we may need to make sure whether the data got flushed or not. Because, someone else in the network trying to read the same file (trying to open a file for read purpose) when write operation is under progress may result in file corruption. In such situations, we need to set *flush* argument as True. Indeed, this is just a basic vague explanation of *flush* argument and it has much more meaning in real.)

Formatting the output:
There are various ways of formatting the output and displaying the variables with a required number of space-width in Python. We will discuss few of them with the help of examples.

- **Ex1:** When multiple variables have to be displayed embedded within a string, the *format()* function is useful as shown below -

```
>>> x=10
>>> y=20
>>> print("x={0}, y={1}".format(x,y))
     x=10, y=20
```

While using *format()* the arguments of *print()* must be numbered as 0, 1, 2, 3, etc. and they must be provided inside the *format()* in the same order.

- **Ex2:** The *format()* function can be used to specify the width of the variable (the number of spaces that the variable should occupy in the output) as well. Consider below given example which displays a number, its square and its cube.

```
for x in range(1,5):
     print("{0:1d} {1:3d} {2:4d}".format(x,x**2, x**3))
```

Output:

```
1   1    1
2   4    8
3   9   27
4  16   64
```

Here, 1d, 3d and 4d indicates 1-digit space, 2-digit space etc. on the output screen.

- **Ex3:** One can use % symbol to have required number of spaces for a variable. This will be useful in printing floating point numbers.

```
>>> x=19/3
>>> print(x)
6.333333333333333          #observe number of digits after dot
>>> print("%.3f"%(x))      #only 3 places after decimal point
6.333

>>> x=20/3
>>> y=13/7
>>> print("x= ",x, "y=",y)     #observe actual digits
     x=  6.666666666666667  y= 1.8571428571428572
>>> print("x=%0.4f, y=%0.2f"%(x,y))
     x=6.6667, y=1.86          #observe rounding off digits
```

To know more about possibilities with *format()*, read -
https://docs.python.org/3/tutorial/inputoutput.html

MODULE - 2

2.1 ITERATION

Iteration is a processing repeating some task. In a real time programming, we require a set of statements to be repeated certain number of times and/or till a condition is met. Every programming language provides certain constructs to achieve the repetition of tasks. In this section, we will discuss various such looping structures.

2.1.1 The *while* Statement

The *while* loop has the syntax as below -

```
while condition:
      statement_1
      statement_2
      …….
      statement_n

statements_after_while
```

Here, ***while*** is a keyword. The `condition` is evaluated first. Till its value remains true, the `statement_1` to `statement_n` will be executed. When the `condition` becomes false, the loop is terminated and statements after the loop will be executed. Consider an example -

```
n=1
while n<=5:
    print(n)    #observe indentation
    n=n+1

print("over")
```

The output of above code segment would be -

```
1
2
3
4
5
over
```

In the above example, a variable `n` is initialized to 1. Then the condition `n<=5` is being checked. As the condition is true, the block of code containing print statement (`print(n)`) and increment statement (`n=n+1`) are executed. After these two lines, condition is checked again. The procedure continues till condition becomes false, that is when n becomes 6. Now, the while-loop is terminated and next statement after the loop will be executed. Thus, in this example, the loop is ***iterated*** for 5 times.

Note that, a variable `n` is initialized before starting the loop and it is incremented inside the loop. Such a variable that changes its value for every iteration and controls the total execution of the loop is called as ***iteration variable*** or ***counter variable***. If the count variable is not updated properly within the loop, then the loop may not terminate and keeps executing infinitely.

2.1.2 Infinite Loops, *break* and *continue*

A loop may execute infinite number of times when the condition is never going to become false. For example,

```
n=1
while True:
     print(n)
     n=n+1
```

Here, the condition specified for the loop is the constant `True`, which will never get terminated. Sometimes, the condition is given such a way that it will never become false and hence by restricting the program control to go out of the loop. This situation may happen either due to wrong condition or due to not updating the counter variable.

In some situations, we deliberately want to come out of the loop even before the normal termination of the loop. For this purpose ***break*** statement is used. The following example depicts the usage of *break***.** Here, the values are taken from keyboard until a negative number is entered. Once the input is found to be negative, the loop terminates.

```
while True:
    x=int(input("Enter  a  number:"))
    if x>= 0:
        print("You  have  entered  ",x)
    else:
        print("You have entered a negative number!!")
        break       #terminates the loop
```

Sample output:

```
Enter a number:23
You have entered 23
Enter a number:12
You have entered 12
Enter a number:45
You have entered 45
Enter a number:0
You  have  entered  0
Enter a number:-2
You have entered a negative number!!
```

In the above example, we have used the constant `True` as condition for while-loop, which will never become false. So, there was a possibility of infinite loop. This has been avoided

by using *break* statement with a condition. The condition is kept inside the loop such a way that, if the user input is a negative number, the loop terminates. This indicates that, the loop may terminate with just one iteration (if user gives negative number for the very first time) or it may take thousands of iteration (if user keeps on giving only positive numbers as input). Hence, the number of iterations here is unpredictable. But, we are making sure that it will not be an infinite-loop, instead, the user has control on the loop.

Sometimes, programmer would like to move to next iteration by skipping few statements in the loop, based on some condition. For this purpose ***continue*** statement is used. For example, we would like to find the sum of 5 even numbers taken as input from the keyboard. The logic is -

- Read a number from the keyboard
- If that number is odd, without doing anything else, just move to next iteration for reading another number
- If the number is even, add it to *sum* and increment the accumulator variable.

When accumulator crosses 5, stop the program

The program for the above task can be written as -

```
sum=0
count=0
while True:
    x=int(input("Enter  a  number:"))
    if x%2 !=0:
        continue
    else:
       sum+=x
       count+=1

       if count==5:
           break

print("Sum= ", sum)
```

Sample Output:

```
Enter a number:13
Enter a number:12
Enter a number:4
Enter a number:5
Enter a number:-3
Enter a number:8
Enter a number:7
Enter a number:16
Enter a number:6
Sum=  46
```

2.1.3 Definite Loops using *for*

The *while* loop iterates till the condition is met and hence, the number of iterations are usually unknown prior to the loop. Hence, it is sometimes called as *indefinite loop*. When we know total number of times the set of statements to be executed, ***for*** loop will be used. This is called as a *definite loop***.** The for-loop iterates over a set of numbers, a set of words, lines in a file etc. The syntax of for-loop would be -

```
for var in list/sequence:
      statement_1
      statement_2
      ……..
      statement_n

statements_after_for
```

Here, `for` and `in` are keywords
`list/sequence` is a set of elements on which the loop is iterated. That is, the loop will be executed till there is an element in list/sequence
`statements` constitutes body of the loop

Ex: In the below given example, a ***list*** `names` containing three strings has been created. Then the counter variable x in the *for*-loop iterates over this *list*. The variable x takes the elements in `names` one by one and the body of the loop is executed.

```
names=["Ram", "Shyam", "Bheem"]
for x in names:
    print(x)
```

The output would be -

```
Ram
Shyam
Bheem
```

NOTE: In Python, list is an important data type. It can take a sequence of elements of different types. It can take values as a comma separated sequence enclosed within square brackets. Elements in the list can be extracted using index (just similar to extracting array elements in C/C++ language). Various operations like indexing, slicing, merging, addition and deletion of elements etc. can be applied on lists. The details discussion on Lists will be done in Module 3.

The *for* loop can be used to print (or extract) all the characters in a string as shown below -

```
for i in "Hello":
    print(i, end='\t')
```

Output:

```
H    e    l    l    o
```

When we have a fixed set of numbers to iterate in a *for* loop, we can use a function ***range()***. The function *range()* takes the following format -

```
range(start, end, steps)
```

The `start` and `end` indicates starting and ending values in the sequence, where `end` is excluded in the sequence (That is, sequence is up to `end-1`). The default value of `start` is 0. The argument `steps` indicates the increment/decrement in the values of sequence with the default value as 1. Hence, the argument `steps` is optional. Let us consider few examples on usage of *range()* function.

Ex1. Printing the values from 0 to 4 -

```
for i in range(5):
    print(i, end= '\t')
```

Output:

```
0    1    2    3    4
```

Here, 0 is the default starting value. The statement `range(5)` is same as `range(0,5)` and `range(0,5,1)`.

Ex2. Printing the values from 5 to 1 -

```
for i in range(5,0,-1):
    print(i, end= '\t')
```

Output:

```
5    4    3    2    1
```

The function `range(5,0,-1)` indicates that the sequence of values are 5 to 0(excluded) in steps of -1 (downwards).

Ex3. Printing only even numbers less than 10 -

```
for i in range(0,10,2):
    print(i, end= '\t')
```

Output:

```
0    2    4    6    8
```

2.1.4 Loop Patterns

The *while*-loop and *for*-loop are usually used to go through a list of items or the contents of a file and to check maximum or minimum data value. These loops are generally constructed by the following procedure -

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

The construction of these loop patterns are demonstrated in the following examples.

Counting and Summing Loops: One can use the *for* loop for counting number of items in the list as shown -

```
count = 0
for i in [4, -2, 41, 34, 25]:
    count = count + 1
print("Count:", count)
```

Here, the variable `count` is initialized before the loop. Though the counter variable `i` is not being used inside the body of the loop, it controls the number of iterations. The variable `count` is incremented in every iteration, and at the end of the loop the total number of elements in the list is stored in it.

One more loop similar to the above is finding the sum of elements in the list -

```
total = 0
for x in [4, -2, 41, 34, 25]:
    total = total + x
print("Total:", total)
```

Here, the variable `total` is called as ***accumulator*** because in every iteration, it accumulates the sum of elements. In each iteration, this variable contains *running total of values so far*.

NOTE: In practice, both of the counting and summing loops are not necessary, because there are built-in functions `len()` and `sum()` for the same tasks respectively.

Maximum and Minimum Loops: To find maximum element in the list, the following code can be used -

```
big = None
print('Before Loop:', big)
for x in [12, 0, 21,-3]:
    if big is None or x > big :
        big = x
    print('Iteration Variable:', x, 'Big:', big)

print('Biggest:', big)
```

Output:

```
Before Loop: None
Iteration Variable: 12    Big: 12
Iteration Variable: 0    Big: 12
Iteration Variable: 21   Big: 21
Iteration Variable: -3   Big: 21
Biggest: 21
```

Here, we initialize the variable `big` to `None`. It is a special constant indicating empty. Hence, we cannot use relational operator `==` while comparing it with `big`. Instead, the ***is*** operator must be used. In every iteration, the counter variable `x` is compared with previous value of `big`. If `x > big`, then `x` is assigned to `big`.

Similarly, one can have a loop for finding smallest of elements in the list as given below -

```
small = None
print('Before Loop:', small)
for x in [12, 0, 21,-3]:
    if small is None or x < small :
        small = x
    print('Iteration Variable:', x, 'Small:', small)

print('Smallest:', small)
```

Output:

```
Before Loop: None
Iteration Variable: 12 Small: 12
Iteration Variable: 0 Small: 0
Iteration Variable: 21 Small: 0
Iteration Variable: -3 Small: -3
Smallest: -3
```

NOTE: In Python, there are built-in functions `max()` and `min()` to compute maximum and minimum values among. Hence, the above two loops need not be written by the programmer explicitly. The inbuilt function `min()` has the following code in Python -

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

2.2 STRINGS

A string is a sequence of characters, enclosed either within a pair of single quotes or double quotes. Each character of a string corresponds to an index number, starting with zero as shown below -

S= "Hello World"

character	H	e	l	l	o		w	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

The characters of a string can be accessed using index enclosed within square brackets. For example,

```
>>> word1="Hello"
>>> word2='hi'
>>>  x=word1[1]      #2nd character of word1 is extracted
>>> print(x)
     e
>>>  y=word2[0]      #1st character of word1 is extracted
>>> print(y)
     h
```

Python supports negative indexing of string starting from the end of the string as shown below -

S= "Hello World"

character	H	e	l	l	o		w	o	r	l	D
Negative index	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

The characters can be extracted using negative index also. For example,

```
>>> var="Hello"
>>>  print(var[-1])
     o
>>>  print(var[-4])
     e
```

Whenever the string is too big to remember last positive index, one can use negative index to extract characters at the end of string.

2.2.1 Getting Length of a String using *len()*

The ***len()*** function can be used to get length of a string.

```
>>> var="Hello"
>>> ln=len(var)
>>> print(ln)
     5
```

The index for string varies from 0 to length-1. Trying to use the index value beyond this range generates error.

```
>>> var="Hello"
>>>  ln=len(var)
>>> ch=var[ln]
IndexError: string index out of range
```

2.2.2 Traversal through String with a Loop

Extracting every character of a string one at a time and then performing some action on that character is known as *traversal.* A string can be traversed either using *while* loop or using *for* loop in different ways. Few of such methods is shown here -

- **Using *for* loop:**

```
st="Hello"
for i in st:
    print(i, end='\t')
```

Output:

H e l l o

In the above example, the *for* loop is iterated from first to last character of the string `st`. That is, in every iteration, the counter variable `i` takes the values as H, e, l, l and o. The loop terminates when no character is left in `st`.

- **Using *while* loop:**

```
st="Hello"
i=0
while i<len(st):
    print(st[i],   end='\t')
    i+=1
```

Output:

```
H    e    l    l    o
```

In this example, the variable `i` is initialized to 0 and it is iterated till the length of the string. In every iteration, the value of `i` is incremented by 1 and the character in a string is extracted using `i` as index.

2.2.3 String Slices

A segment or a portion of a string is called as *slice*. Only a required number of characters can be extracted from a string using colon (:) symbol. The basic syntax for slicing a string would be -

```
st[i:j:k]
```

This will extract character from i^{th} character of st till $(j-1)^{th}$ character in steps of `k`. If first index `i` is not present, it means that slice should start from the beginning of the string. If the second index `j` is not mentioned, it indicates the slice should be till the end of the string. The third parameter `k`, also known as ***stride***, is used to indicate number of steps to be incremented after extracting first character. The default value of stride is 1.

Consider following examples along with their outputs to understand string slicing.

```
st="Hello World"       #refer this string for all examples
```

1. `print("st[:] is", st[:])  #output Hello World`
   As both index values are not given, it assumed to be a full string.

2. `print("st[0:5] is ", st[0:5])  #output is Hello`
   Starting from 0^{th} index to 4^{th} index (5 is exclusive), characters will be printed.

3. `print("st[0:5:1] is", st[0:5:1])  #output is Hello`
   This code also prints characters from 0th to 4th index in the steps of 1. Comparing this example with previous example, we can make out that when the stride value is 1, it is optional to mention.

4. `print("st[3:8] is ", st[3:8]) #output is lo Wo`
   Starting from 3rd index to 7th index (8 is exclusive), characters will be printed.

5. `print("st[7:] is ", st[7:])  #output is orld`
   Starting from 7th index to till the end of string, characters will be printed.

6. `print(st[::2])  #outputs HloWrd`
   This example uses stride value as 2. So, starting from first character, every alternative character (char+2) will be printed.

7. `print("st[4:4] is ", st[4:4]) #gives empty string`
   Here, `st[4:4]` indicates, slicing should start from 4th character and end with (4-1)=3rd character, which is not possible. Hence the output would be an empty string.

8. `print(st[3:8:2])  #output is l o`
   Starting from 3rd character, till 7th character, every alternative index is considered.

9. `print(st[1:8:3])  #output is eoo`
   Starting from index 1, till 7th index, every 3rd character is extracted here.

10. `print(st[-4:-1])  #output is orl`
    Refer the diagram of negative indexing given earlier. Excluding the -1st character, all characters at the indices -4, -3 and -2 will be displayed. Observe the role of stride with default value 1 here. That is, it is computed as -4+1 =-3, -3+1=-2 etc.

11. `print(st[-1:])  #output is d`
    Here, starting index is -1, ending index is not mentioned (means, it takes the index 10) and the stride is default value 1. So, we are trying to print characters from -1 (which is the last character of negative indexing) till 10th character (which is also the last character in positive indexing) in incremental order of 1. Hence, we will get only last character as output.

12. `print(st[:-1])  #output is Hello Worl`
    Here, starting index is default value 0 and ending is -1 (corresponds to last character in negative indexing). But, in slicing, as last index is excluded always, -1st character is omitted and considered only up to -2nd character.

13. `print(st[::])  #outputs Hello World`
    Here, two colons have used as if stride will be present. But, as we haven't mentioned stride its default value 1 is assumed. Hence this will be a full string.

14. `print(st[::-1])  #outputs dlroW olleH`

This example shows the power of slicing in Python. Just with proper slicing, we could able to ***reverse the string***. Here, the meaning is *a full string to be extracted in the order of -1*. Hence, the string is printed in the reverse order.

15. `print(st[::-2])  #output is drWolH`

Here, the string is printed in the reverse order in steps of -2. That is, every alternative character in the reverse order is printed. Compare this with example (6) given above.

By the above set of examples, one can understand the power of string slicing and of Python script. The slicing is a powerful tool of Python which makes many task simple pertaining to data types like strings, Lists, Tuple, Dictionary etc. (Other types will be discussed in later Modules)

2.2.4 Strings are Immutable

The objects of string class are immutable. That is, once the strings are created (or initialized), they cannot be modified. No character in the string can be edited/deleted/added. Instead, one can create a new string using an existing string by imposing any modification required.

Try to attempt following assignment -

```
>>> st= "Hello World"
>>> st[3]='t'
TypeError: 'str' object does not support item assignment
```

Here, we are trying to change the 4th character (index 3 means, 4th character as the first index is 0) to *t.* The error message clearly states that an assignment of new *item* (a string) is not possible on string object. So, to achieve our requirement, we can create a new string using slices of existing string as below -

```
>>> st= "Hello World"
>>> st1= st[:3]+ 't' + st[4:] >>>
print(st1)
     Helto World          #l is replaced by t in new string st1
```

2.2.5 Looping and Counting

Using loops on strings, we can count the frequency of occurrence of a character within another string. The following program demonstrates such a pattern on computation called as a ***counter***. Initially, we accept one string and one character (single letter). Our aim to find the total number of times the character has appeared in string. A variable *count* is initialized to zero, and incremented each time a character is found. The program is given below -

```
def  countChar(st,ch):
    count=0
    for i in st:
```

```
            if i==ch:
                count+=1
        return count

st=input("Enter a string:")
ch=input("Enter a character to be counted:")
c=countChar(st,ch)
print("{0} appeared {1} times in {2}".format(ch,c,st))
```

Sample Output:

Enter a string: hello how are you?
Enter a character to be counted: h
h appeared 2 times in hello how are you?

2.2.6 The *in* Operator

The ***in*** operator of Python is a Boolean operator which takes two string operands. It returns True, if the first operand appears in second operand, otherwise returns False. For example,

```
>>> 'el' in 'hello'      #el is found in hello
    True
>>> 'x' in 'hello'       #x is not found in hello
    False
```

2.2.7 String Comparison

Basic comparison operators like < (less than), > (greater than), == (equals) etc. can be applied on string objects. Such comparison results in a Boolean value True or False. Internally, such comparison happens using ASCII codes of respective characters. Consider following examples -

Ex1.
```
st= "hello"
if st== 'hello':
    print('same')
```

Output is `same`. As the value contained in `st` and `hello` both are same, the equality results in True.

Ex2.
```
st= "hello"
if st<= 'Hello':
    print('lesser')
else:
    print('greater')
```

Output is `greater`. The ASCII value of `h` is greater than ASCII value of `H`. Hence, `hello` is greater than `Hello`.

NOTE: A programmer must know ASCII values of some of the basic characters. Here are few -

A - Z	: 65 - 90
a - z	: 97 - 122
0 - 9	: 48 - 57
Space	: 32
Enter Key	: 13

2.2.8 String Methods

String is basically a ***class*** in Python. When we create a string in our program, an ***object*** of that class will be created. A class is a collection of member variables and member methods (or functions). When we create an object of a particular class, the object can use all the members (both variables and methods) of that class. Python provides a rich set of built-in classes for various purposes. Each class is enriched with a useful set of utility functions and variables that can be used by a Programmer. A programmer can create a class based on his/her requirement, which are known as user-defined classes.

The built-in set of members of any class can be accessed using the dot operator as shown-

```
objName.memberMethod(arguments)
```

The dot operator always binds the member name with the respective object name. This is very essential because, there is a chance that more than one class has members with same name. To avoid that conflict, almost all Object oriented languages have been designed with this common syntax of using dot operator. (Detailed discussion on classes and objects will be done in later Modules.)

Python provides a function (or method) ***dir*** to list all the variables and methods of a particular class object. Observe the following statements -

```
>>> s="hello"        #string object is created with the name s
>>> type(s)          #checking type of s
<class 'str'>        #s is object of type class str
>>> dir(s)           #display all methods and variables of object s

['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
```

```
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Students need not remember the above list !!

Note that, the above set of variables and methods are common for any object of string class that we create. Each built-in method has a predefined set of arguments and return type. To know the usage, working and behavior of any built-in method, one can use the command ***help***. For example, if we would like to know what is the purpose of `islower()` function (refer above list to check its existence!!), how it behaves etc, we can use the statement -

```
>>> help(str.islower)
Help on method_descriptor:

islower(...)
    S.islower() -> bool

    Return True if all cased characters in S are lowercase and there
is at least one cased character in S, False otherwise.
```

This is built-in help-service provided by Python. Observe the `className.memberName` format while using ***help***.

The methods are usually called using the object name. This is known as ***method invocation.*** We say that a method is invoked using an object.

Now, we will discuss some of the important methods of string class.

- **capitalize(s) :** This function takes one string argument ***s*** and returns a capitalized version of that string. That is, the first character of ***s*** is converted to upper case, and all other characters to lowercase. Observe the examples given below -

  Ex1.
  ```
  >>> s="hello"
  >>> s1=str.capitalize(s)
  >>> print(s1)
       Hello          #1st character is changed to uppercase
  ```

  Ex2.
  ```
  >>> s="hello World"
  >>>  s1=str.capitalize(s)
  >>> print(s1)
       Hello world
  ```

  Observe in Ex2 that the first character is converted to uppercase, and an in-between uppercase letter W of the original string is converted to lowercase.

- **s.upper():** This function returns a copy of a string s to uppercase. As strings are immutable, the original string s will remain same.

```
>>> st= "hello"
>>> st1=st.upper()
>>> print(st1)
     'HELLO'
>>> print( st)    #no change in original string
     'hello'
```

- **s.lower():** This method is used to convert a string s to lowercase. It returns a copy of original string after conversion, and original string is intact.

```
>>> st='HELLO'
>>> st1=st.lower()
>>> print(st1)
     hello
>>> print(st)        #no change in original string
     HELLO
```

- **s.find(s1) :** The `find()` function is used to search for a substring s1 in the string s. If found, the index position of first occurrence of s1 in s, is returned. If s1 is not found in s, then -1 is returned.

```
>>> st='hello'
>>> i=st.find('l')
>>> print(i)                    #output is 2
>>> i=st.find('lo')
>>> print(i)                    #output is 3
>>> print(st.find('x'))         #output is -1
```

The `find()` function can take one more form with two additional arguments viz. start and end positions for search.

```
>>> st="calender of Feb. cal of march" >>>
i= st.find('cal')
>>> print(i)                    #output is 0
```

Here, the substring `'cal'` is found in the very first position of `st`, hence the result is 0.

```
>>> i=st.find('cal',10,20)
>>> print(i)                    #output is 17
```

Here, the substring `cal` is searched in the string `st` between 10th and 20th position and hence the result is 17.

```
>>> i=st.find('cal',10,15)
>>> print(i)                    #ouput is -1
```

In this example, the substring `'cal'` has not appeared between 10th and 15th character of st. Hence, the result is -1.

- **s.strip():** Returns a copy of string *s* by removing leading and trailing white spaces.

```
>>> st="     hello world    "
>>>  st1  =  st.strip()
>>> print(st1)
     hello world
```

The *strip()* function can be used with an argument *chars*, so that specified *chars* are removed from beginning or ending of *s* as shown below -

```
>>> st="###Hello##"
>>> st1=st.strip('#')
>>> print(st1)            #all hash symbols are removed
     Hello
```

We can give more than one character for removal as shown below -

```
>>> st="Hello world"
>>>   st.strip("Hld")
     ello wor
```

- **S.startswith(prefix, start, end):** This function has 3 arguments of which *start* and *end* are option. This function returns True if S starts with the specified *prefix*, False otherwise.

```
>>> st="hello world"
>>> st.startswith("he")        #returns True
```

When *start* argument is provided, the search begins from that position and returns True or False based on search result.

```
>>> st="hello world"
>>> st.startswith("w",6)   #True because w is at 6th position
```

When both *start* and *end* arguments are given, search begins at *start* and ends at *end*.

```
>>> st="xyz abc pqr ab mn gh‟
>>> st.startswith("pqr ab mn",8,12)   #returns False
>>> st.startswith("pqr ab mn",8,18)     #returns True
```

The `startswith()` function requires case of the alphabet to match. So, when we are not sure about the case of the argument, we can convert it to either upper case or lowercase and then use `startswith()` function as below -

```
>>> st="Hello"
>>> st.startswith("he")            #returns False
>>> st.lower().startswith("he")    #returns True
```

- **S.count(s1, start, end):** The `count()` function takes three arguments - *string, starting position* and *ending position*. This function returns the number of non-overlapping occurrences of substring s1 in string S in the range of *start* and *end*.

```
>>> st="hello how are you? how about you?"
>>> st.count('h')          #output is 3
>>> st.count('how')        #output is 2
>>> st.count('how',3,10)   #output is 1 because of range given
```

There are many more built-in methods for string class. Students are advised to explore more for further study.

2.2.9 Parsing Strings

Sometimes, we may want to search for a substring matching certain criteria. For example, finding domain names from email-Ids in the list of messages is a useful task in some projects. Consider a string below and we are interested in extracting only the domain name.

"From chetanahegde@*ieee.org Wed Feb 21 09:14:16 2018*"

Now, our aim is to extract only *ieee.org*, which is the domain name. We can think of logic as-

- Identify the position of @, because all domain names in email IDs will be after the symbol @
- Identify a white space which appears after @ symbol, because that will be the end of domain name.
- Extract the substring between @ and white-space.

The concept of string slicing and *find()* function will be useful here.Consider the code given below -

```
st="From chetanahegde@ieee.org Wed Feb 21 09:14:16 2018"
atpos=st.find('@')                  #finds the position of @

print('Position of @ is', atpos)

spacePos=st.find('  ', atpos)       #position of white-space after @

print('Position of space after @ is', spacePos)
```

```
host=st[atpos+1:spacePos]      #slicing from @ till white-space
print(host)
```

Execute above program to get the output as *ieee.org*. One can apply this logic in a loop, when our string contains series of email IDs, and we may want to extract all those mail IDs.

2.2.10 Format Operator

The format operator, % allows us to construct strings, replacing parts of the strings with the data stored in variables. The first operand is the format string, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.

```
>>> sum=20
>>> '%d' %sum
     '20'            #string '20', but not integer 20
```

Note that, when applied on both integer operands, the % symbol acts as a modulus operator. When the first operand is a string, then it is a format operator. Consider few examples illustrating usage of format operator.

Ex1.
```
>>> "The sum value %d is originally integer"%sum
     'The sum value 20 is originally integer'
```

Ex2.
```
>>> '%d %f %s'%(3,0.5,'hello')
     '3 0.500000 hello'
```

Ex3.
```
>>> '%d %g %s'%(3,0.5,'hello')
     '3 0.5 hello'
```

Ex4.
```
>>> '%d'% 'hello'
TypeError: %d format: a number is required, not str
```

Ex5.
```
>>> '%d %d %d'%(2,5)
TypeError: not enough arguments for format string
```

MODULE 2 - FILES

2.3 FILES

File handling is an important requirement of any programming language, as it allows us to store the data permanently on the secondary storage and read the data from a permanent source. Here, we will discuss how to perform various operations on files using the programming language Python.

2.3.1 Persistence

The programs that we have considered till now are based on console I/O. That is, the input was taken from the keyboard and output was displayed onto the monitor. When the data to be read from the keyboard is very large, console input becomes a laborious job. Also, the output or result of the program has to be used for some other purpose later, it has to be stored permanently. Hence, reading/writing from/to files are very essential requirement of programming.

We know that the programs stored in the hard disk are brought into main memory to execute them. These programs generally communicate with CPU using conditional execution, iteration, functions etc. But, the content of main memory will be erased when we turn-off our computer. We have discussed these concepts in Module1 with the help of Figure 1.1. Here we will discuss about working with secondary memory or files. The files stored on the secondary memory are permanent and can be transferred to other machines using pen-drives/CD.

2.3.2 Opening Files

To perform any operation on a file, one must open a file. File opening involves communication with operating system. In Python, a file can be opened using a built-in function ***open()***. While opening a file, we must specify the name of the file to be opened. Also, we must inform the OS about the purpose of opening a file, which is termed as *file opening mode*. The syntax of ***open()*** function is as below -

```
fhand= open("filename", "mode")
```

Here, `filename` is name of the file to be opened. This string may be just a name of the file, or it may include pathname also. Pathname of the file is optional when the file is stored in current working directory

`mode` This string indicates the purpose of opening a file. It takes a pre-defined set of values as given in Table 2.1

`fhand` It is a reference to an object of ***file*** class, which acts as a handler or tool for all further operations on files.

When our Python program makes a request to open a specific file in a particular mode, then OS will try to serve the request. When a file gets opened successfully, then a file object is returned. This is known as *file handle* and is as shown in Figure 2.1. It will help to

perform various operations on a file through our program. If the file cannot be opened due to some reason, then error message (*traceback*) will be displayed.

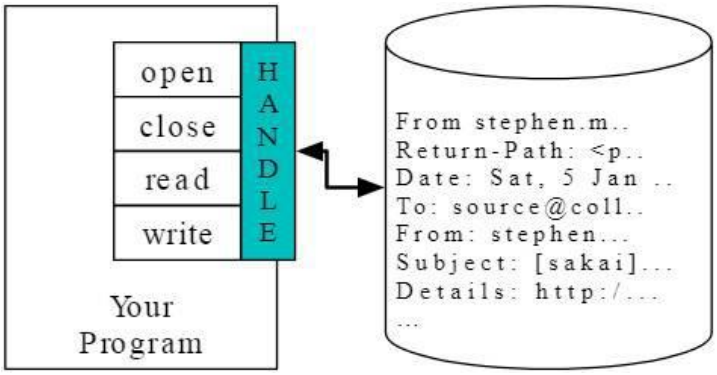


Figure 2.1 A File Handle

A file opening may cause an error due to some of the reasons as listed below -

- File may not exist in the specified path (when we try to read a file)
- File may exist, but we may not have a permission to read/write a file
- File might have got corrupted and may not be in an opening state

Since, there is no guarantee about getting a file handle from OS when we try to open a file, it is always better to write the code for file opening using *try-except* block. This will help us to manage error situation.

Mode	Meaning
r	Opens a file for reading purpose. If the specified file does not exist in the specified path, or if you don't have permission, error message will be displayed. This is the default mode of *open()* function in Python.
w	Opens a file for writing purpose. If the file does not exist, then a new file with the given name will be created and opened for writing. If the file already exists, then its content will be over-written.
a	Opens a file for appending the data. If the file exists, the new content will be appended at the end of existing content. If no such file exists, it will be created and new content will be written into it.
r+	Opens a file for reading and writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
rb	Opens a file for reading only in binary format
wb	Opens a file for writing only in binary format
ab	Opens a file for appending only in binary format

2.3.3 Text Files and Lines

A text file is a file containing a sequence of lines. It contains only the plain text without any images, tables etc. Different lines of a text file are separated by a newline character `\n`. In the text files, this newline character may be invisible, but helps in identifying every line in the file. There will be one more special entry at the end to indicate end of file (EOF).

NOTE: There is one more type of file called binary file, which contains the data in the form of bits. These files are capable of storing text, image, video, audio etc. All these data will be stored in the form of a group of bytes whose formatting will be known. The supporting program can interpret these files properly, whereas when opened using normal text editor, they look like messy, unreadable set of characters.

2.3.4 Reading Files

When we successfully open a file to read the data from it, the ***open()*** function returns the file handle (or an object reference to *file* object) which will be pointing to the first character in the file. A text file containing lines can be iterated using a for-loop starting from the beginning with the help of this file handle. Consider the following example of counting number of lines in a file.

NOTE: Before executing the below given program, create a text file (using Notepad or similar editor) *myfile.txt* in the current working directory (The directory where you are going store your Python program). Open this text file and add few random lines to it and then close. Now, open a Python script file, say *countLines.py* and save it in the same directory as that of your text file *myfile.txt*. Then, type the following code in Python script *countLines.py* and execute the program. (You can store text file and Python script file in different directories. But, if you do so, you have to mention complete path of text file in the *open()* function.)

Sample Text file *myfile.txt:*

hello how are you?
I am doing fine
what about you?

Python script file *countLines.py*

```
fhand=open('myfile.txt','r')
count =0
for line in fhand:
    count+=1
    print("Line Number ",count, ":", line)

print("Total lines=",count)
fhand.close()
```

Output:

```
Line Number 1 : hello how are you?
Line Number 2 : I am doing fine
```

```
Line Number 3 : what about you?
Total lines= 3
```

In the above program, initially, we will try to open the file `'myfile.txt`. As we have already created that file, the file handler will be returned and the object reference to this file will be stored in `fhand`. Then, in the for-loop, we are using `fhand` as if it is a sequence of lines. For each line in the file, we are counting it and printing the line. In fact, a line is identified internally with the help of new-line character present at the end of each line. Though we have not typed `\n` anywhere in the file `myfile.txt`, after each line, we would have pressed enter-key. This act will insert a `\n`, which is invisible when we view the file through notepad. Once all lines are over, `fhand` will reach end-of-file and hence terminates the loop. Note that, when end of file is reached (that is, no more characters are present in the file), then an attempt to read will return `None` or empty character `''` (two quotes without space in between).

Once the operations on a file is completed, it is a practice to close the file using a function *close()*. Closing of a file ensures that no unwanted operations are done on a file handler. Moreover, when a file was opened for writing or appending, closure of a file ensures that the last bit of data has been uploaded properly into a file and the end-of-file is maintained properly. If the file handler variable (in the above example, `fhand`) is used to assign some other file object (using *open()* function), then Python closes the previous file automatically.

If you run the above program and check the output, there will be a gap of two lines between each of the output lines. This is because, the new-line character `\n` is also a part of the variable `line` in the loop, and the *print()* function has default behavior of adding a line at the end (due to default setting of *end* parameter of *print()*). To avoid this double-line spacing, we can remove the new-line character attached at the end of variable `line` by using built-in string function *rstrip()* as below -

```
print("Line Number ",count, ":", line.rstrip())
```

It is obvious from the logic of above program that from a file, each line is read one at a time, processed and discarded. Hence, there will not be a shortage of main memory even though we are reading a very large file. But, when we are sure that the size of our file is quite small, then we can use ***read()*** function to read the file contents. This function will read entire file content as a single string. Then, required operations can be done on this string using built-in string functions. Consider the below given example -

```
fhand=open('myfile.txt')
s=fhand.read()
print("Total number of characters:",len(s))
print("String up to 20 characters:", s[:20])
```

After executing above program using previously created file *myfile.txt*, then the output would be -

```
Total number of characters:50
String up to 20 characters: hello how are you?
I
```

2.3.5 Writing Files

To write a data into a file, we need to use the mode ***w*** in *open()* function.

```
>>> fhand=open("mynewfile.txt","w")
>>> print(fhand)
<_io.TextIOWrapper name='mynewfile.txt' mode='w' encoding='cp1252'>
```

If the file specified already exists, then the old contents will be erased and it will be ready to write new data into it. If the file does not exists, then a new file with the given name will be created.

The ***write()*** method is used to write data into a file. This method returns number of characters successfully written into a file. For example,

```
>>> s="hello how are you?"
>>> fhand.write(s)
     18
```

Now, the file object keeps track of its position in a file. Hence, if we write one more line into the file, it will be added at the end of previous line. Here is a complete program to write few lines into a file -

```
fhand=open('f1.txt','w')
for i in range(5):
     line=input("Enter a line: ")
     fhand.write(line+"\n")

fhand.close()
```

The above program will ask the user to enter 5 lines in a loop. After every line has been entered, it will be written into a file. Note that, as ***write()*** method doesn't add a new-line character by its own, we need to write it explicitly at the end of every line. Once the loop gets over, the program terminates. Now, we need to check the file `f1.txt` on the disk (in the same directory where the above Python code is stored) to find our input lines that have been written into it.

2.3.6 Searching through a File

Most of the times, we would like to read a file to search for some specific data within it. This can be achieved by using some string methods while reading a file. For example, we may be interested in printing only the line which starts with a character ***h*.** Then we can use *startswith()* method.

```
fhand=open('myfile.txt')
for line in fhand:
    if line.startswith('h'):
        print(line)
fhand.close()
```

Assume the input file *myfile.txt* is containing the following lines -

```
hello how are you?
I am doing fine
how about you?
```

Now, if we run the above program, we will get the lines which starts with *h* -

```
hello how are you?
how about you?
```

2.3.7 Letting the User Choose the File Name

In a real time programming, it is always better to ask the user to enter a name of the file which he/she would like to open, instead of hard-coding the name of a file inside the program.

```
fname=input("Enter a file name:")
fhand=open(fname)

count =0
for line in fhand:
    count+=1
    print("Line Number ",count, ":", line)

print("Total lines=",count)
fhand.close()
```

In this program, the user input filename is received through variable `fname`, and the same has been used as an argument to `open()` method. Now, if the user input is *myfile.txt* (discussed before), then the result would be

```
Total lines=3
```

Everything goes well, if the user gives a proper file name as input. But, what if the input filename cannot be opened (Due to some reason like - file doesn't exists, file permission denied etc)? Obviously, Python throws an error. The programmer need to handle such runtime errors as discussed in the next section.

2.3.8 Using *try*, *except* to Open a File

It is always a good programming practice to write the commands related to file opening within a *try* block. Because, when a filename is a user input, it is prone to errors. Hence, one should handle it carefully. The following program illustrates this -

```
fname=input("Enter a file name:")
try:
    fhand=open(fname)
except:
```

```
    print("File cannot be opened")
    exit()

count =0
for line in fhand:
    count+=1
    print("Line Number ",count, ":", line)

print("Total lines=",count)
fhand.close()
```

In the above program, the command to open a file is kept within *try* block. If the specified file cannot be opened due to any reason, then an error message is displayed saying `File cannot be opened`, and the program is terminated. If the file could able to open successfully, then we will proceed further to perform required task using that file.

2.3.9 Debugging

While performing operations on files, we may need to extract required set of lines or words or characters. For that purpose, we may use string functions with appropriate delimiters that may exist between the words/lines of a file. But, usually, the invisible characters like white-space, tabs and new-line characters are confusing and it is hard to identify them properly. For example,

```
>>> s="1 2\t 3\n 4"
>>> print(s)
1 2   3
 4
```

Here, by looking at the output, it may be difficult to make out where there is a space, where is a tab etc. Python provides a utility function called as ***repr()*** to solve this problem. This method takes any object as an argument and returns a string representation of that object. For example, the *print()* in the above code snippet can be modified as -

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Note that, some of the systems use \n as new-line character, and few others may use \r (carriage return) as a new-line character. The ***repr()*** method helps in identifying that too.

MODULE - 3

3.1 LISTS

A list is an ordered sequence of values. It is a data structure in Python. The values inside the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc) and are called as *elements* or *items.* The elements of lists are enclosed within square brackets. For example,

```
ls1=[10,-4, 25, 13]
ls2=["Tiger", "Lion", "Cheetah"]
```

Here, `ls1` is a list containing four integers, and `ls2` is a list containing three strings. A list need not contain data of same type. We can have mixed type of elements in list. For example,

```
ls3=[3.5, 'Tiger', 10, [3,4]]
```

Here, `ls3` contains a float, a string, an integer and a list. This illustrates that a list can be nested as well.

An empty list can be created any of the following ways -

```
>>> ls =[]
>>> type(ls)
     <class 'list'>
```

or

```
>>> ls =list()
>>> type(ls)
     <class 'list'>
```

In fact, `list()` is the name of a method (special type of method called as constructor - which will be discussed in Module 4) of the class *list*. Hence, a new list can be created using this function by passing arguments to it as shown below -

```
>>> ls2=list([3,4,1])
>>> print(ls2)
     [3, 4, 1]
```

3.1.1 Lists are Mutable

The elements in the list can be accessed using a numeric index within square-brackets. It is similar to extracting characters in a string.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[1])
     hi
>>> print(ls[2])
     [2, 3]
```

Observe here that, the inner list is treated as a single element by outer list. If we would like to access the elements within inner list, we need to use double-indexing as shown below -

```
>>> print(ls[2][0])
     2
>>> print(ls[2][1])
     3
```

Note that, the indexing for inner-list again starts from 0. Thus, when we are using doubleindexing, the first index indicates position of inner list inside outer list, and the second index means the position particular value within inner list.

Unlike strings, lists are mutable. That is, using indexing, we can modify any value within list. In the following example, the 3rd element (i.e. index is 2) is being modified -

```
>>> ls=[34, 'hi', [2,3],-5]
>>> ls[2]='Hello'
>>> print(ls)
     [34, 'hi', 'Hello', -5]
```

The list can be thought of as a relationship between indices and elements. This relationship is called as a ***mapping***. That is, each index maps to one of the elements in a list.

The index for extracting list elements has following properties -

- Any integer expression can be an index.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[2*1])
     'Hello'
```

- Attempt to access a non-existing index will throw and IndexError.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[4])
IndexError: list index out of range ·
```

A negative indexing counts from backwards.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[-1])
     -5
>>> print(ls[-3])
     hi
```

The ***in*** operator applied on lists will results in a Boolean value.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> 34 in ls
     True
>>> -2 in ls
     False
```

3.1.2 Traversing a List

A list can be traversed using *for* loop. If we need to use each element in the list, we can use the *for* loop and *in* operator as below -

```
>>> ls=[34, 'hi', [2,3],-5]
>>> for item in ls:
        print(item)

     34
     hi
     Hello
     -5
```

List elements can be accessed with the combination of *range()* and *len()* functions as well -

```
ls=[1,2,3,4]
for i in range(len(ls)):
     ls[i]=ls[i]**2

print(ls)       #output is [1, 4, 9, 16]
```

Here, we wanted to do modification in the elements of list. Hence, referring indices is suitable than referring elements directly. The *len()* returns total number of elements in the list (here it is 4). Then *range()* function makes the loop to range from 0 to 3 (i.e. 4-1). Then, for every index, we are updating the list elements (replacing original value by its square).

3.1.3 List Operations

Python allows to use operators + and * on lists. The operator + uses two list objects and returns concatenation of those two lists. Whereas * operator take one list object and one integer value, say n, and returns a list by repeating itself for n times.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6,7]
>>> print(ls1+ls2)              #concatenation using +
     [1, 2, 3, 5, 6, 7]

>>>    ls1=[1,2,3]
>>> print(ls1*3)                #repetition using *
     [1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> [0]*4                       #repetition using *
     [0, 0, 0, 0]
```

3.1.4 List Slices

Similar to strings, the slicing can be applied on lists as well. Consider a list `t` given below, and a series of examples following based on this object.

```
t=['a','b','c','d','e']
```

- Extracting full list without using any index, but only a slicing operator -

```
>>> print(t[:])
     ['a', 'b', 'c', 'd', 'e']
```

- Extracting elements from 2nd position -

```
>>> print(t[1:])
     ['b', 'c', 'd', 'e']
```

- Extracting first three elements -

```
>>> print(t[:3])
     ['a', 'b', 'c']
```

- Selecting some middle elements -

```
>>> print(t[2:4])
     ['c', 'd']
```

- Using negative indexing -

```
>>> print(t[:-2])
     ['a', 'b', 'c']
```

- **Reversing a list** using negative value for stride -

```
>>> print(t[::-1])
     ['e', 'd', 'c', 'b', 'a']
```

- **Modifying (reassignment) only required set of values -**

```
>>> t[1:3]=['p','q']
>>> print(t)
     ['a', 'p', 'q', 'd', 'e']
```

Thus, slicing can make many tasks simple.

3.1.5 List Methods

There are several built-in methods in *list* class for various purposes. Here, we will discuss some of them.

- **append():** This method is used to add a new element at the end of a list.

```
>>> ls=[1,2,3]
>>> ls.append('hi')
>>> ls.append(10) >>>
print(ls)
     [1, 2, 3, 'hi', 10]
```

- **extend():** This method takes a list as an argument and all the elements in this list are added at the end of invoking list.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>>  ls2.extend(ls1)
>>> print(ls2)
     [5, 6, 1, 2, 3]
```

Now, in the above example, the list `ls1` is unaltered.

- **sort():** This method is used to sort the contents of the list. By default, the function will sort the items in ascending order.

```
>>> ls=[3,10,5, 16,-2]
>>> ls.sort()
>>> print(ls)
     [-2, 3, 5, 10, 16]
```

When we want a list to be sorted in descending order, we need to set the argument as shown -

```
>>>  ls.sort(reverse=True)
>>> print(ls)
[16, 10, 5, 3, -2]
```

- **reverse():** This method can be used to reverse the given list.

```
>>> ls=[4,3,1,6]
>>> ls.reverse()
>>> print(ls)
     [6, 1, 3, 4]
```

- **count():** This method is used to count number of occurrences of a particular value within list.

```
>>>  ls=[1,2,5,2,1,3,2,10]
>>> ls.count(2)
     3              #the item 2 has appeared 3 tiles in ls
```

- **clear():** This method removes all the elements in the list and makes the list empty.

```
>>> ls=[1,2,3]
>>> ls.clear()
>>> print(ls)
      []
```

- **insert():** Used to insert a value before a specified index of the list.

```
>>> ls=[3,5,10]
>>>  ls.insert(1,"hi")
>>> print(ls)
     [3, 'hi', 5, 10]
```

- **index():** This method is used to get the index position of a particular value in the list.

```
>>> ls=[4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
     1
```

Here, the number 2 is found at the index position 1. Note that, this function will give index of only the first occurrence of a specified value. The same function can be used with two more arguments *start* and *end* to specify a range within which the search should take place.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6] >>>
ls.index(2)
     2
>>> ls.index(2,3,7)
     6
```

If the value is not present in the list, it throws ValueError.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(53)
     ValueError: 53 is not in list
```

Few important points about List Methods:

1. There is a difference between *append()* and *extend()* methods. The former adds the argument as it is, whereas the latter enhances the existing list. To understand this, observe the following example -

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>>  ls2.append(ls1)
>>> print(ls2)
  [5, 6, [1, 2, 3]]
```

Here, the argument `ls1` for the *append()* function is treated as one item, and made as an inner list to `ls2`. On the other hand, if we replace *append()* by *extend()* then the result would be -

```
>>>  ls1=[1,2,3]
>>> ls2=[5,6]
>>>  ls2.extend(ls1)
>>> print(ls2)
  [5, 6, 1, 2, 3]
```

2. The ***sort()*** function can be applied only when the list contains elements of compatible types. But, if a list is a mix non-compatible types like integers and string, the comparison cannot be done. Hence, Python will throw TypeError. For example,

```
>>> ls=[34, 'hi', -5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Similarly, when a list contains integers and sub-list, it will be an error.

```
>>> ls=[34,[2,3],5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'list' and 'int'
```

Integers and floats are compatible and relational operations can be performed on them. Hence, we can sort a list containing such items.

```
>>> ls=[3, 4.5, 2]
>>> ls.sort()
>>> print(ls)
     [2, 3, 4.5]
```

3. The ***sort()*** function uses one important argument ***keys***. When a list is containing tuples, it will be useful. We will discuss tuples later in this Module.

4. Most of the list methods like *append()*, *extend()*, *sort()*, *reverse()* etc. modify the list object internally and return None.

```
>>> ls=[2,3]
>>>  ls1=ls.append(5)
>>> print(ls)
     [2,3,5]
>>>  print(ls1)
     None
```

3.1.6 Deleting Elements

Elements can be deleted from a list in different ways. Python provides few built-in methods for removing elements as given below -

- **pop():** This method deletes the last element in the list, by default.

```
>>> ls=[3,6,-2,8,10]
>>>  x=ls.pop()          #10 is removed from list and stored in x
>>> print(ls)
   [3, 6, -2, 8]
>>> print(x)
   10
```

When an element at a particular index position has to be deleted, then we can give that position as argument to *pop()* function.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)        #item at index 1 is popped
>>> print(t)
    ['a',  'c']
>>> print(x)
     b
```

- **remove():** When we don't know the index, but know the value to be removed, then this function can be used.

```
>>> ls=[5,8, -12,34,2]
>>> ls.remove(34)
>>> print(ls)
    [5, 8, -12, 2]
```

Note that, this function will remove only the first occurrence of the specified value, but not all occurrences.

```
>>> ls=[5,8, -12, 34, 2, 6, 34] >>>
ls.remove(34)
>>> print(ls)
   [5, 8, -12, 2, 6, 34]
```

Unlike *pop()* function, the *remove()* function will not return the value that has been deleted.

- **del:** This is an operator to be used when more than one item to be deleted at a time. Here also, we will not get the items deleted.

```
>>> ls=[3,6,-2,8,1]
>>> del ls[2]           #item at index 2 is deleted
>>> print(ls)
   [3, 6, 8, 1]

>>> ls=[3,6,-2,8,1]
>>> del ls[1:4]         #deleting all elements from index 1 to 3
>>> print(ls)
        [3, 1]
```

Deleting all odd indexed elements of a list -

```
>>> t=['a', 'b', 'c', 'd', 'e'] >>>
del t[1::2]
>>> print(t)
     ['a', 'c', 'e']
```

3.1.7 Lists and Functions

The utility functions like ***max(), min(), sum(), len()*** etc. can be used on lists. Hence most of the operations will be easy without the usage of loops.

```
>>> ls=[3,12,5,26, 32,1,4]
>>> max(ls)              # prints  32
>>> min(ls)              # prints  1
>>> sum(ls)              # prints  83
>>> len(ls)              # prints  7

>>>  avg=sum(ls)/len(ls)
>>> print(avg)
   11.857142857142858
```

When we need to read the data from the user and to compute sum and average of those numbers, we can write the code as below -

```
ls= list()
while (True):
   x= input('Enter a number: ') if
   x== 'done':
        break

   x=   float(x)
   ls.append(x)

average = sum(ls) / len(ls)
print('Average:', average)
```

In the above program, we initially create an empty list. Then, we are taking an infinite *while*loop. As every input from the keyboard will be in the form of a string, we need to convert x into float type and then append it to a list. When the keyboard input is a string ‘done’, then the loop is going to get terminated. After the loop, we will find the average of those numbers with the help of built-in functions *sum()* and *len()*.

3.1.8 Lists and Strings

Though both lists and strings are sequences, they are not same. In fact, a list of characters is not same as string. To convert a string into a list, we use a method ***list()*** as below -

```
>>> s="hello"
>>>  ls=list(s)
>>> print(ls)
     ['h', 'e', 'l', 'l', 'o']
```

The method ***list()*** breaks a string into individual letters and constructs a list. If we want a list of words from a sentence, we can use the following code -

```
>>> s="Hello how are you?"
>>> ls=s.split()
>>> print(ls)
     ['Hello', 'how', 'are', 'you?']
```

Note that, when no argument is provided, the ***split()*** function takes the delimiter as white space. If we need a specific delimiter for splitting the lines, we can use as shown in following example -

```
>>>   dt="20/03/2018"
>>>  ls=dt.split('/')
>>> print(ls)
     ['20', '03', '2018']
```

There is a method ***join()*** which behaves opposite to ***split()*** function. It takes a list of strings as argument, and joins all the strings into a single string based on the delimiter provided. For example -

```
>>> ls=["Hello", "how", "are", "you"]
>>> d=' '
>>> d.join(ls)
'Hello how are you'
```

Here, we have taken delimiter `d` as white space. Apart from space, anything can be taken as delimiter. When we don't need any delimiter, use empty string as delimiter.

3.1.9 Parsing Lines

In many situations, we would like to read a file and extract only the lines containing required pattern. This is known as ***parsing***. As an illustration, let us assume that there is a log file containing details of email communication between employees of an organization. For all received mails, the file contains lines as -

```
From stephen.marquard@uct.ac.za Fri Jan 5 09:14:16 2018
From georgek@uct.ac.za Sat Jan 6 06:12:51 2018
……
```

Apart from such lines, the log file also contains mail-contents, to-whom the mail has been sent etc. Now, if we are interested in extracting only the days of incoming mails, then we can go for parsing. That is, we are interested in knowing on which of the days, the mails have been received. The code would be -

```
fhand = open('logFile.txt')
for line in fhand:
     line = line.rstrip()
     if not line.startswith('From '):
          continue
     words = line.split()
     print(words[2])
```

Obviously, all received mails starts from the word `From`. Hence, we search for only such lines and then split them into words. Observe that, the first word in the line would be `From`, second word would be email-ID and the 3rd word would be day of a week. Hence, we will extract `words[2]` which is 3rd word.

3.1.10 Objects and Values

Whenever we assign two variables with same value, the question arises - whether both the variables are referring to same object, or to different objects. This is important aspect to know, because in Python everything is a class object. There is nothing like elementary data type.

Consider a situation -

```
a= "hi"
b= "hi"
```

Now, the question is whether both `a` and `b` refer to the ***same string***. There are two possible states -

In the first situation, `a` and `b` are two different objects, but containing same value. The modification in one object is nothing to do with the other. Whereas, in the second case, both `a` and `b` are referring to the same object. That is, a is an ***alias name*** for b and viceversa. In other words, these two are referring to same memory location.

To check whether two variables are referring to same object or not, we can use ***is*** operator.

```
>>> a= "hi"
>>> b= "hi"
>>> a is b          #result is True
>>> a==b            #result is True
```

When two variables are referring to same object, they are called as ***identical objects.*** When two variables are referring to different objects, but contain a same value, they are known as ***equivalent objects***. For example,

```
>>> s1=input("Enter a string:")    #assume you entered hello
>>> s2= input("Enter a string:") #assume you entered hello

>>> s1 is s2              #check s1 and s2 are identical
    False
>>> s1 == s2              #check s1 and s2 are equivalent
    True
```

Here **`s1`** and **`s2`** are equivalent, but not identical.

If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

String literals are ***interned*** by default. That is, when two string literals are created in the program with a same value, they are going to refer same object. But, string variables read from the key-board will not have this behavior, because their values are depending on the user's choice.

Lists are not interned. Hence, we can see following result -

```
>>> ls1=[1,2,3]
>>> ls2=[1,2,3]
>>> ls1 is ls2      #output is False
>>> ls1 == ls2      #output is True
```

3.1.11 Aliasing

When an object is assigned to other using assignment operator, both of them will refer to same object in the memory. The association of a variable with an object is called as ***reference.***

```
>>>  ls1=[1,2,3]
>>> ls2= ls1
>>> ls1 is ls2      #output is True
```

Now, ls2 is said to be ***reference*** of ls1. In other words, there are two references to the same object in the memory.

An object with more than one reference has more than one name, hence we say that object is ***aliased.*** If the aliased object is mutable, changes made in one alias will reflect the other.

```
>>> ls2[1]= 34
>>> print(ls1)      #output is [1, 34, 3]
```

Strings are safe in this regards, as they are immutable.

3.1.12 List Arguments

When a list is passed to a function as an argument, then function receives reference to this list. Hence, if the list is modified within a function, the caller will get the modified version. Consider an example -

```
def del_front(t):
     del t[0]

ls = ['a', 'b', 'c']
del_front(ls)
print(ls)      # output is ['b', 'c']
```

Here, the argument `ls` and the parameter `t` both are aliases to same object.

One should understand the operations that will modify the list and the operations that create a new list. For example, the ***append()*** function modifies the list, whereas the + operator creates a new list.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>>  print(t1)            #output is [1 2 3]
>>> print(t2)            #prints None

>>> t3 = t1 + [5]
>>> print(t3)            #output is [1 2 3 5]
>>> t2 is t3             #output is False
```

Here, after applying *append()* on t1 object, the t1 itself has been modified and t2 is not going to get anything. But, when + operator is applied, t1 remains same but t3 will get the updated result.

The programmer should understand such differences when he/she creates a function intending to modify a list. For example, the following function has no effect on the original list -

```
def test(t):
     t=t[1:]

ls=[1,2,3]
test(ls)
print(ls)      #prints [1, 2, 3]
```

One can write a return statement after slicing as below -

```
def test(t):
     return t[1:]

ls=[1,2,3]
ls1=test(ls)
print(ls1)     #prints [2, 3]
print(ls)      #prints [1, 2, 3]
```

In the above example also, the original list is not modified, because a return statement always creates a new object and is assigned to LHS variable at the position of function call.

3.2 DICTIONARIES

A dictionary is a collection of unordered set of ***key:value*** pairs, with the requirement that keys are unique in one dictionary. Unlike lists and strings where elements are accessed using index values (which are integers), the values in dictionary are accessed using keys. A key in dictionary can be any immutable type like strings, numbers and tuples. (The tuple can be made as a key for dictionary, only if that tuple consist of string/number/ sub-tuples). As lists are mutable - that is, can be modified using index assignments, slicing, or using methods like *append()*, *extend()* etc, they cannot be a key for dictionary.

One can think of a dictionary as a mapping between set of indices (which are actually keys) and a set of values. Each key maps to a value.

An empty dictionary can be created using two ways -

```
d= {}
```

OR

```
d=dict()
```

To add items to dictionary, we can use square brackets as -

```
>>> d={}
>>>   d["Mango"]="Fruit"
>>>  d["Banana"]="Fruit"
>>>  d["Cucumber"]="Veg"
>>> print(d)
{'Mango': 'Fruit', 'Banana': 'Fruit', 'Cucumber': 'Veg'}
```

To initialize a dictionary at the time of creation itself, one can use the code like -

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135}
>>> print(tel_dir)
     {'Tom': 3491, 'Jerry': 8135}

>>>  tel_dir['Donald']=4793
>>> print(tel_dir)
     {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
```

NOTE that the order of elements in dictionary is unpredictable. That is, in the above example, don't assume that `'Tom': 3491` is first item, `'Jerry': 8135` is second item etc. As dictionary members are not indexed over integers, the order of elements inside it may vary. However, using a *key,* we can extract its associated value as shown below -

```
>>> print(tel_dir['Jerry'])
     8135
```

Here, the key `'Jerry'` maps with the value `8135`, hence it doesn't matter where exactly it is inside the dictionary.

If a particular key is not there in the dictionary and if we try to access such key, then the *KeyError* is generated.

```
>>> print(tel_dir['Mickey'])
     KeyError: 'Mickey'
```

The ***len()*** function on dictionary object gives the number of key-value pairs in that object.

```
>>> print(tel_dir)
     {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793} >>>
len(tel_dir)
     3
```

The ***in*** operator can be used to check whether any ***key*** (not value) appears in the dictionary object.

```
>>> 'Mickey' in tel_dir        #output is False
>>> 'Jerry' in tel_dir         #output is True
>>> 3491 in tel_dir            #output is False
```

We observe from above example that the value 3491 is associated with the key `'Tom'` in `tel_dir`. But, the ***in*** operator returns False.

The dictionary object has a method ***values()*** which will ***return a list*** of all the values associated with keys within a dictionary. If we would like to check whether a particular value exist in a dictionary, we can make use of it as shown below -

```
>>> 3491 in tel_dir.values()   #output is True
```

The ***in*** operator behaves differently in case of lists and dictionaries as explained hereunder-

- When ***in*** operator is used to search a value in a list, then *linear search* algorithm is used internally. That is, each element in the list is checked one by one sequentially. This is considered to be expensive in the view of total time taken to process. Because, if there are 1000 items in the list, and if the element in the list which we are search for is in the last position (or if it does not exists), then before yielding result of search (True or False), we would have done 1000 comparisons. In other words, linear search requires *n* number of comparisons for the input size of *n* elements. Time complexity of the linear search algorithm is O(*n*).
- The keys in dictionaries of Python are basically ***hashable*** elements. The concept of ***hashing*** is applied to store (or maintain) the keys of dictionaries. Normally hashing techniques have the time complexity as *O(log n)* for basic operations like insertion, deletion and searching. Hence, the ***in*** operator applied on keys of dictionaries works better compared to that on lists. (Hashing technique is explained at the end of this Section, for curious readers)

3.2.1 Dictionary as a Set of Counters

Assume that we need to count the frequency of alphabets in a given string. There are different methods to do it -

- Create 26 variables to represent each alphabet. Traverse the given string and increment the corresponding counter when an alphabet is found.
- Create a list with 26 elements (all are zero in the beginning) representing alphabets. Traverse the given string and increment corresponding indexed position in the list when an alphabet is found.
- Create a dictionary with characters as keys and counters as values. When we find a character for the first time, we add the item to dictionary. Next time onwards, we increment the value of existing item.

Each of the above methods will perform same task, but the logic of implementation will be different. Here, we will see the implementation using dictionary.

```
s=input("Enter a string:")      #read a string
d=dict()                        #create empty dictionary

for ch in s:                    #traverse through string
    if ch not in d:             #if new character found
        d[ch]=1                 #initialize counter to 1
    else:                       #otherwise, increment counter
        d[ch]+=1

print(d)                        #display the dictionary
```

The sample output would be -

```
Enter a string: Hello World
{'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'W': 1, 'r': 1, 'd': 1}
```

It can be observed from the output that, a dictionary is created here with characters as keys and frequencies as values. **Note** that, here we have computed ***histogram*** of counters.

Dictionary in Python has a method called as ***get()***, which takes key and a default value as two arguments. If key is found in the dictionary, then the *get()* function returns corresponding value, otherwise it returns default value. For example,

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
>>> print(tel_dir.get('Jerry',0))
     8135
>>>  print(tel_dir.get('Donald',0))
     0
```

In the above example, when the *get()* function is taking `'Jerry'` as argument, it returned corresponding value, as `'Jerry'` is found in `tel_dir`. Whereas, when *get()* is used with `'Donald'` as key, the default value 0 (which is provided by us) is returned.

The function ***get()*** can be used effectively for calculating frequency of alphabets in a string. Here is the modified version of the program -

```
s=input("Enter  a  string:")
d=dict()

for ch in s:
   d[ch]=d.get(ch,0)+1

print(d)
```

In the above program, for every character `ch` in a given string, we will try to retrieve a value. When the `ch` is found in `d`, its value is retrieved, 1 is added to it, and restored. If `ch` is not found, 0 is taken as default and then 1 is added to it.

3.2.2 Looping and Dictionaries

When a *for*-loop is applied on dictionaries, it will iterate over the keys of dictionary. If we want to print key and values separately, we need to use the statements as shown -

```
tel_dir={'Tom':  3491,  'Jerry':8135,  'Mickey':1253}
for k in tel_dir:
     print(k, tel_dir[k])
```

Output would be -

```
Tom 3491
Jerry   8135
Mickey 1253
```

Note that, while accessing items from dictionary, the keys may not be in order. If we want to print the keys in alphabetical order, then we need to make a list of the keys, and then sort that list. We can do so using ***keys()*** method of dictionary and ***sort()*** method of lists. Consider the following code -

```
tel_dir={'Tom':  3491,  'Jerry':8135,  'Mickey':1253}
ls=list(tel_dir.keys())
print("The  list  of  keys:",ls)
ls.sort()
print("Dictionary  elements  in  alphabetical  order:")
for k in ls:
    print(k, tel_dir[k])
```

The output would be -

```
The list of keys: ['Tom', 'Jerry', 'Mickey']
Dictionary elements in alphabetical order:
Jerry 8135
Mickey  1253
Tom 3491
```

Note: The key-value pair from dictionary can be together accessed with the help of a method ***items()*** as shown -

```
>>> d={'Tom':3412, 'Jerry':6781, 'Mickey':1294}
>>> for k,v in d.items():
        print(k,v)
```

Output:

```
Tom 3412
Jerry   6781
Mickey 1294
```

The usage of comma-separated list `k,v` here is internally a tuple (another data structure in Python, which will be discussed later).

3.2.3 Dictionaries and Files

A dictionary can be used to count the frequency of words in a file. Consider a file *myfile.txt* consisting of following text -

hello, how are you?
I am doing fine.
How about you?

Now, we need to count the frequency of each of the word in this file. So, we need to take an outer loop for iterating over entire file, and an inner loop for traversing each line in a file. Then in every line, we count the occurrence of a word, as we did before for a character. The program is given as below -

```
fname=input("Enter file name:")
try:
    fhand=open(fname)
except:
    print("File cannot be opened")
    exit()

d=dict()

for line in fhand:
    for word in line.split():
        d[word]=d.get(word,0)+1

print(d)
```

The output of this program when the input file is *myfile.txt* would be -

```
Enter file name: myfile.txt
{'hello,': 1, 'how': 1, 'are': 1, 'you?': 2, 'I': 1, 'am': 1,
'doing': 1, 'fine.': 1, 'How': 1, 'about': 1}
```

Few points to be observed in the above output -

- The punctuation marks like comma, full point, question mark etc. are also considered as a part of word and stored in the dictionary. This means, when a particular word appears in a file with and without punctuation mark, then there will be multiple entries of that word.
- The word 'how' and 'How' are treated as separate words in the above example because of uppercase and lowercase letters.

While solving problems on text analysis, machine learning, data analysis etc. such kinds of treatment of words lead to unexpected results. So, we need to be careful in parsing the text and we should try to eliminate punctuation marks, ignoring the case etc. The procedure is discussed in the next section.

3.2.4 Advanced Text Parsing

As discussed in the previous section, during text parsing, our aim is to eliminate punctuation marks as a part of word. The *string* module of Python provides a list of all punctuation marks as shown -

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

The *str* class has a method *maketrans()* which returns a translation table usable for another method *translate()*. Consider the following syntax to understand it more clearly -

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

The above statement replaces the characters in `fromstr` with the character in the same position in `tostr` and delete all characters that are in `deletestr`. The `fromstr` and `tostr` can be empty strings and the `deletestr` parameter can be omitted.

Using these functions, we will re-write the program for finding frequency of words in a file.

```
import string

fname=input("Enter file name:")

try:
    fhand=open(fname)
except:
    print("File cannot be opened")
    exit()

d=dict()
```

```
for line in fhand:
     line=line.rstrip()
     line=line.translate(line.maketrans('','',string.punctuation))
     line=line.lower()

     for word in line.split():
          d[word]=d.get(word,0)+1

print(d)
```

Now, the output would be -

```
Enter file name:myfile.txt
{'hello': 1, 'how': 2, 'are': 1, 'you': 2, 'i': 1, 'am': 1,
'doing': 1, 'fine': 1, 'about': 1}
```

Comparing the output of this modified program with the previous one, we can make out that all the punctuation marks are not considered for parsing and also the case of the alphabets are ignored.

3.2.5 Debugging

When we are working with big datasets (like file containing thousands of pages), it is difficult to debug by printing and checking the data by hand. So, we can follow any of the following procedures for easy debugging of the large datasets -

- **Scale down the input**: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just first 10 lines or with the smallest example you can find. You can either edit the files themselves, or modify the program so it reads only the first n lines. If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you correct the errors.
- **Check summaries and types**: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers. A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.
- **Write self-checks**: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a ***sanity check*** because it detects results that are “completely illogical”. Another kind of check compares the results of two different computations to see if they are consistent. This is called a ***consistency check***.
- **Pretty print the output**: Formatting debugging output can make it easier to spot an error.

Hashing Technique (For curious minds - Only for understanding, not for Exams!!)

Hashing is a way of representing dictionaries (Not a Python data structure Dictionary!!). Dictionary is an abstract data type with a set of operations searching, insertion and deletion defined on its elements. The elements of dictionary can be numeric or characters or most of the times, records. Usually, a record consists of several fields; each may be of different data types. For example, student record may contain student id, name, gender, marks etc. Every record is usually identified by some ***key***. Hashing technique is very useful in database management, because it is considered to be very efficient searching technique.

Here we will consider the implementation of a dictionary of *n* records with keys k1, k2 …kn. Hashing is based on the idea of distributing keys among a one-dimensional array

H[0…m-1], called ***hash table***.

For each key, a value is computed using a predefined function called ***hash function***. This function assigns an integer, called ***hash address***, between 0 to m-1 to each key. Based on the hash address, the keys will be distributed in a hash table.

For example, if the keys k1, k2, …., kn are integers, then a hash function can be

h(K) = K mod m.

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be,

h(k) = k%10.

Then the hash addresses may be any value from 0 to 9. For each key, hash address will be computed as -

h(65) = 65 %10 = 5
h(78) = 78%10 = 8
h(22)= 22 % 10 =2
h(30)= 30 %10 =0
h(47) = 47 %10 = 7
h(89)=89 % 10 = 9

Now, each of these keys can be hashed into a hash table as -

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89

In general, a hash function should satisfy the following requirements:

- A hash function needs to distribute keys among the cells of hash table as evenly as possible.
- A hash function has to be easy to compute.

Hash Collisions: Let us have *n* keys and the hash table is of size *m* such that *m<n.* As each key will have an address with any value between 0 to m-1, it is obvious that more than one key will have same hash address. That is, two or more keys need to be hashed into the same cell of hash table. This situation is called as ***hash collision***.

In the worst case, all the keys may be hashed into same cell of hash table. But, we can avoid this by choosing proper size of hash table and hash function. Anyway, every hashing scheme must have a mechanism for resolving hash collision. There are two methods for hash collision resolution, viz.

- Open hashing
- closed hashing

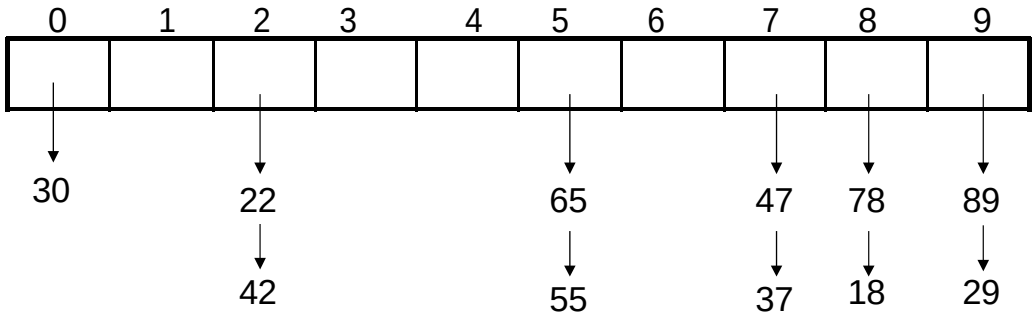
Open Hashing (or Separate Chaining): In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell. For example, consider the elements

65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.

If we take the hash function as h(k)= k%10, then the hash addresses will be -

h(65) = 65 %10 = 5	h(78) = 78%10 = 8
h(22)= 22 % 10 =2	h(30)= 30 %10 =0
h(47) = 47 %10 = 7	h(89)=89 % 10 = 9
h(55)=55%10 =5	h(42)=42%10 =2
h(18)=18%10 =8	h(29)=29%10=9
h(37)=37%10 =7	

The hash table would be -



Operations on Hashing:

- **Searching:** Now, if we want to search for the key element in a hash table, we need to find the hash address of that key using same hash function. Using the obtained hash address, we need to search the linked list by tracing it, till either the key is found or list gets exhausted.
- **Insertion**: Insertion of new element to hash table is also done in similar manner. Hash key is obtained for new element and is inserted at the end of the list for that particular cell.
- **Deletion**: Deletion of element is done by searching that element and then deleting it from a linked list.

Closed Hashing (or Open Addressing): In this technique, all keys are stored in the hash table itself without using linked lists. Different methods can be used to resolve hash collisions. The simplest technique is ***linear probing.***

This method suggests to check the next cell from where the collision occurs. If that cell is empty, the key is hashed there. Otherwise, we will continue checking for the empty cell in a circular manner. Thus, in this technique, the hash table size must be at least as large as the total number of keys. That is, if we have n elements to be hashed, then the size of hash table should be greater or equal to n.

Example: Consider the elements 65, 78, 18, 22, 30, 89, 37, 55, 42
Let us take the hash function as h(k)= k%10, then the hash addresses will be -

h(65) = 65 %10 = 5
h(78) = 78%10 = 8
h(18)=18%10 =8
h(22)= 22 % 10 =2
h(30)= 30 %10 =0
h(89)=89 % 10 = 9
h(37)=37%10 =7
h(55)=55%10 =5
h(42)=42%10 =2

Since there are 9 elements in the list, our hash table should at least be of size 9. Here we are taking the size as 10.

Now, hashing is done as below -

0	1	2	3	4	5	6	7	8	9
30	89	22	42		65	55	37	78	18

Drawbacks:

- Searching may become like a linear search and hence not efficient.

3.3 TUPLES

A tuple is a sequence of items, similar to lists. The values stored in the tuple can be of any type and they are indexed using integers. Unlike lists, tuples are immutable. That is, values within tuples cannot be modified/reassigned. Tuples are *comparable* and *hashable* objects. Hence, they can be made as keys in dictionaries.

A tuple can be created in Python as a comma separated list of items - may or may not be enclosed within parentheses.

```
>>> t='Mango', 'Banana', 'Apple'        #without parentheses
>>> print(t)
     ('Mango', 'Banana', 'Apple')

>>> t1=('Tom', 341, 'Jerry')            #with parentheses
>>> print(t1)
     ('Tom', 341, 'Jerry')
```

Observe that tuple values can be of mixed types.

If we would like to create a tuple with single value, then just a parenthesis will not suffice. For example,

```
>>> x=(3)            #trying to have a tuple with single item >>>
print(x)
     3               #observe, no parenthesis found
>>> type(x)
<class 'int'>        #not a tuple, it is integer!!
```

Thus, to have a tuple with single item, we must include a comma after the item. That is,

```
>>> t=3,             #or use the statement t=(3,)
>>> type(t)          #now this is a tuple
<class 'tuple'>
```

An empty tuple can be created either using a pair of parenthesis or using a function ***tuple()*** as below -

```
>>> t1=()
>>> type(t1)
     <class 'tuple'>

>>> t2=tuple()
>>> type(t2)
     <class 'tuple'>
```

If we provide an argument of type sequence (a list, a string or tuple) to the method ***tuple()***, then a tuple with the elements in a given sequence will be created -

Create tuple using string:

```
>>>  t=tuple('Hello')
>>> print(t)
     ('H', 'e', 'l', 'l', 'o')
```

Create tuple using list:

```
>>>  t=tuple([3,[12,5],'Hi'])
>>> print(t)
     (3, [12, 5], 'Hi')
```

Create tuple using another tuple:

```
>>> t=('Mango', 34, 'hi')
>>> t1=tuple(t)
>>> print(t1)
     ('Mango', 34, 'hi')
>>> t is t1
     True
```

Note that, in the above example, both t and t1 objects are referring to same memory location. That is, t1 is a reference to t.

Elements in the tuple can be extracted using square-brackets with the help of indices. Similarly, slicing also can be applied to extract required number of items from tuple.

```
>>> t=('Mango', 'Banana', 'Apple')
>>> print(t[1])
     Banana
>>> print(t[1:])
     ('Banana', 'Apple')
>>> print(t[-1])
     Apple
```

Modifying the value in a tuple generates error, because tuples are immutable -

```
>>> t[0]='Kiwi'
TypeError: 'tuple' object does not support item assignment
```

We wanted to replace ‘Mango’ by ‘Kiwi’, which did not work using assignment. But, a tuple can be replaced with another tuple involving required modifications -

```
>>> t=('Kiwi',)+t[1:]
>>> print(t)
     ('Kiwi', 'Banana', 'Apple')
```

3.3.1 Comparing Tuples

Tuples can be compared using operators like >, <, >=, == etc. The comparison happens lexicographically. For example, when we need to check equality among two tuple objects, the first item in first tuple is compared with first item in second tuple. If they are same, 2nd items are compared. The check continues till either a mismatch is found or items get over. Consider few examples -

```
>>>  (1,2,3)==(1,2,5)
     False
>>>  (3,4)==(3,4)
     True
```

The meaning of < and > in tuples is not exactly *less than* and *greater than*, instead, it means *comes before* and *comes after.* Hence in such cases, we will get results different from checking equality (==).

```
>>>  (1,2,3)<(1,2,5)
     True
>>>  (3,4)<(5,2)
     True
```

When we use relational operator on tuples containing non-comparable types, then TypeError will be thrown.

```
>>> (1,'hi')<('hello','world')
TypeError: '<' not supported between instances of 'int' and 'str'
```

The *sort()* function internally works on similar pattern - it sorts primarily by first element, in case of tie, it sorts on second element and so on. This pattern is known as ***DSU*** -

- **Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,
- **Sort** the list of tuples using the Python built-in *sort*(), and
- **Undecorate** by extracting the sorted elements of the sequence.

Consider a program of sorting words in a sentence from longest to shortest, which illustrates DSU property.

```
txt = 'Ram and Seeta went to forest with Lakshman' words
= txt.split()
t = list()
for word in words:
    t.append((len(word), word))

print('The list is:',t)
t.sort(reverse=True) res
= list()
```

```
    for length, word in t:
        res.append(word)
    print('The sorted list:',res)
```

The output would be -

```
The list is: [(3, 'Ram'), (3, 'and'), (5, 'Seeta'), (4, 'went'),
(2, 'to'), (6, 'forest'), (4, 'with'), (8, 'Lakshman')]

The sorted list: ['Lakshman', 'forest', 'Seeta', 'went', 'with',
'and', 'Ram', 'to']
```

In the above program, we have split the sentence into a list of words. Then, a tuple containing length of the word and the word itself are created and are appended to a list. Observe the output of this list - it is a list of tuples. Then we are sorting this list in descending order. Now for sorting, length of the word is considered, because it is a first element in the tuple. At the end, we extract length and word in the list, and create another list containing only the words and print it.

3.3.2 Tuple Assignment

Tuple has a unique feature of having it at LHS of assignment operator. This allows us to assign values to multiple variables at a time.

```
>>> x,y=10,20
>>> print(x)        #prints 10
>>> print(y)        #prints 20
```

When we have list of items, they can be extracted and stored into multiple variables as below -

```
>>> ls=["hello", "world"]
>>> x,y=ls
>>> print(x)        #prints hello
>>> print(y)        #prints world
```

This code internally means that -

```
x= ls[0]
y= ls[1]
```

The best known example of assignment of tuples is ***swapping two values*** as below -

```
>>> a=10
>>> b=20
>>> a, b = b, a
>>> print(a, b)     #prints 20 10
```

In the above example, the statement `a, b = b, a` is treated by Python as - LHS is a set of variables, and RHS is set of expressions. The expressions in RHS are evaluated and assigned to respective variables at LHS.

Giving more values than variables generates ValueError -

```
>>> a, b=10,20,5
ValueError: too many values to unpack (expected 2)
```

While doing assignment of multiple variables, the RHS can be any type of sequence like list, string or tuple. Following example extracts user name and domain from an email ID.

```
>>> email='chetanahegde@ieee.org'
>>> usrName, domain = email.split('@')
>>> print(usrName)                          #prints chetanahegde
>>> print(domain)                           #prints ieee.org
```

3.3.3 Dictionaries and Tuples

Dictionaries have a method called ***items()*** that returns a list of tuples, where each tuple is a key-value pair as shown below -

```
>>> d = {'a':10, 'b':1, 'c':22} >>>
t = list(d.items())
>>> print(t)
     [('b', 1), ('a', 10), ('c', 22)]
```

As dictionary may not display the contents in an order, we can use ***sort()*** on lists and then print in required order as below -

```
>>> d = {'a':10, 'b':1, 'c':22} >>>
t = list(d.items())
>>> print(t)
     [('b', 1), ('a', 10), ('c', 22)] >>>
t.sort()
>>> print(t)
     [('a', 10), ('b', 1), ('c', 22)]
```

3.3.4 Multiple Assignment with Dictionaries

We can combine the method ***items()***, tuple assignment and a for-loop to get a pattern for traversing dictionary:

```
d={'Tom': 1292, 'Jerry': 3501, 'Donald': 8913} for
key, val in list(d.items()):
     print(val,key)
```

The output would be -

```
1292 Tom
3501   Jerry
8913 Donald
```

This loop has two iteration variables because ***items()*** returns a list of tuples. And `key, val` is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary. For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary in hash order.

Once we get a key-value pair, we can create a list of tuples and sort them -

```
d={'Tom': 9291, 'Jerry': 3501, 'Donald': 8913}
ls=list()
for key, val in d.items():
    ls.append((val,key))      #observe inner parentheses

print("List of tuples:",ls)
ls.sort(reverse=True)
print("List of sorted tuples:",ls)
```

The output would be -

```
List of tuples: [(9291, 'Tom'), (3501, 'Jerry'), (8913, 'Donald')]
List of sorted tuples: [(9291, 'Tom'), (8913, 'Donald'), (3501,
'Jerry')]
```

In the above program, we are extracting `key, val` pair from the dictionary and appending it to the list `ls`. While appending, we are putting inner parentheses to make sure that each pair is treated as a tuple. Then, we are sorting the list in the descending order. The sorting would happen based on the telephone number (`val`), but not on name (`key`), as first element in tuple is telephone number (`val`).

3.3.5 The Most Common Words

We will apply the knowledge gained about strings, tuple, list and dictionary till here to solve a problem - write a program to find most commonly used words in a text file.

The logic of the program is -

- Open a file
- Take a loop to iterate through every line of a file.
- Remove all punctuation marks and convert alphabets into lower case (Reason explained in Section 3.2.4)
- Take a loop and iterate over every word in a line.
- If the word is not there in dictionary, treat that word as a key, and initialize its value as 1. If that word already there in dictionary, increment the value.
- Once all the lines in a file are iterated, you will have a dictionary containing distinct words and their frequency. Now, take a list and append each key-value (wordfrequency) pair into it.
- Sort the list in descending order and display only 10 (or any number of) elements from the list to get most frequent words.

```
import string
fhand  =  open('test.txt')
counts = dict()
for line in fhand:
    line  =  line.translate(str.maketrans('',  '',string.punctuation))
    line = line.lower()

    for  word  in  line.split():
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

lst = list()
for  key,  val  in  list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)
for  key,  val  in  lst[:10]:
    print(key, val)
```

Run the above program on any text file of your choice and observe the output.

3.3.6 Using Tuples as Keys in Dictionaries

As tuples and dictionaries are hashable, when we want a dictionary containing composite keys, we will use tuples. For Example, we may need to create a telephone directory where name of a person is Firstname-last name pair and value is the telephone number. Our job is to assign telephone numbers to these keys. Consider the program to do this task -

```
names=(('Tom','Cat'),('Jerry','Mouse'),  ('Donald',  'Duck'))
number=[3561, 4014, 9813]

telDir={}

for i in range(len(number)):
    telDir[names[i]]=number[i]

for fn, ln in telDir:
    print(fn, ln, telDir[fn,ln])
```

The output would be -

```
Tom Cat 3561
Jerry  Mouse  4014
Donald Duck 9813
```

3.3.7 Summary on Sequences: Strings, Lists and Tuples

Till now, we have discussed different types of sequences viz. strings, lists and tuples. In many situations these sequences can be used interchangeably. Still, due their difference in behavior and ability, we may need to understand pros and cons of each of them and then to decide which one to use in a program. Here are few key points -

1. Strings are more limited compared to other sequences like lists and Tuples. Because, the elements in strings must be characters only. Moreover, strings are immutable. Hence, if we need to modify the characters in a sequence, it is better to go for a list of characters than a string.
2. As lists are mutable, they are most common compared to tuples. But, in some situations as given below, tuples are preferable.
    a. When we have a return statement from a function, it is better to use tuples rather than lists.
    b. When a dictionary key must be a sequence of elements, then we must use immutable type like strings and tuples
    c. When a sequence of elements is being passed to a function as arguments, usage of tuples reduces unexpected behavior due to aliasing.
3. As tuples are immutable, the methods like ***sort()*** and ***reverse()*** cannot be applied on them. But, Python provides built-in functions ***sorted()*** and ***reversed()*** which will take a sequence as an argument and return a new sequence with modified results.

3.3.8 Debugging

Lists, Dictionaries and Tuples are basically data structures. In real-time programming, we may require compound data structures like lists of tuples, dictionaries containing tuples and lists etc. But, these compound data structures are prone to ***shape errors*** - that is, errors caused when a data structure has the wrong type, size, composition etc. For example, when your code is expecting a list containing single integer, but you are giving a plain integer, then there will be an error.

When debugging a program to fix the bugs, following are the few things a programmer can try -

- **Reading:** Examine your code, read it again and check that it says what you meant to say.
- **Running:** Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.
- **Ruminating:** Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
- **Retreating:** At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

3.4 REGULAR EXPRESSIONS

Searching for required patterns and extracting only the lines/words matching the pattern is a very common task in solving problems programmatically. We have done such tasks earlier using string slicing and string methods like *split()*, *find()* etc. As the task of searching and extracting is very common, Python provides a powerful library called ***regular expressions*** to handle these tasks elegantly. Though they have quite complicated syntax, they provide efficient way of searching the patterns.

The regular expressions are themselves little programs to search and parse strings. To use them in our program, the library/module ***re*** must be imported. There is a ***search()*** function in this module, which is used to find particular substring within a string. Consider the following example -

```
import re
fhand = open('myfile.txt')
for line in fhand:
    line = line.rstrip()
    if re.search('how', line):
        print(line)
```

By referring to file *myfile.txt* that has been discussed in previous Chapters, the output would be -

```
hello, how are you?
how about you?
```

In the above program, the *search()* function is used to search the lines containing a word *how*.

One can observe that the above program is not much different from a program that uses ***find()*** function of strings. But, regular expressions make use of special characters with specific meaning. In the following example, we make use of caret (^) symbol, which indicates beginning of the line.

```
import re
hand = open('myfile.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^how', line):
        print(line)
```

The output would be -

```
how about you?
```

Here, we have searched for a line which starts with a string *how.* Again, this program will not makes use of regular expression fully. Because, the above program would have been

written using a string function *startswith()*. Hence, in the next section, we will understand the true usage of regular expressions.

3.4.1 Character Matching in Regular Expressions

Python provides a list of meta-characters to match search strings. Table 3.1 shows the details of few important metacharacters. Some of the examples for quick and easy understanding of regular expressions are given in Table 3.2.

Table 3.1 List of Important Meta-Characters

Character	Meaning
^ (caret)	Matches beginning of the line
$	Matches end of the line
. (dot)	Matches any single character except newline. Using option *m*, then newline also can be matched
[…]	Matches any single character in brackets
[^…]	Matches any single character NOT in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
\d	Matches digits. Equivalent to [0-9].
\D	Matches non-digits.
\w	Matches word characters.
\W	Matches non-word characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches non-whitespace.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\b	Matches the empty string, but only at the start or end of a word.
\B	Matches the empty string, but not at the start or end of a word.
( )	When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using findall()

Table 3.2 Examples for Regular Expressions

Expression	Description
`[Pp]ython`	Match "Python" or "python"
`rub[ye]`	Match "ruby" or "rube"
`[aeiou]`	Match any one lowercase vowel
`[0-9]`	Match any digit; same as [0123456789]
`[a-z]`	Match any lowercase ASCII letter
`[A-Z]`	Match any uppercase ASCII letter
`[a-zA-Z0-9]`	Match any of uppercase, lowercase alphabets and digits
`[^aeiou]`	Match anything other than a lowercase vowel
`[^0-9]`	Match anything other than a digit

Most commonly used metacharacter is dot, which matches any character. Consider the following example, where the regular expression is for searching lines which starts with `I` and has any two characters (any character represented by two dots) and then has a character `m`.

```
import re
fhand  =  open('myfile.txt')
for line in fhand:
    line = line.rstrip()
    if re.search('^I..m', line):
          print(line)
```

The output would be -

```
I am doing fine.
```

Note that, the regular expression `^I..m` not only matches '`I am`', but it can match '`Isdm`', '`I*3m`' and so on. That is, between `I` and `m`, there can be any two characters.

In the previous program, we knew that there are exactly two characters between I and m. Hence, we could able to give two dots. But, when we don't know the exact number of characters between two characters (or strings), we can make use of dot and + symbols together. Consider the below given program -

```
import re
hand  =  open('myfile.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^h.+u', line):
        print(line)
```

The output would be -

```
hello,  how  are  you?
how about you?
```

Observe the regular expression `^h.+u` here. It indicates that, the string should be starting with h and ending with u and there may by any number of (dot and +) characters in-between.

Few examples:
To understand the behavior of few basic meta characters, we will see some examples. The file used for these examples is *mbox-short.txt* which can be downloaded from -
https://www.py4e.com/code3/mbox-short.txt

Use this as input and try following examples -

- **Pattern to extract lines starting with the word *From* (or *from*) and ending with *edu*:**

```
import re
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    pattern = '^[Ff]rom.*edu$'
    if re.search(pattern, line):
        print(line)
```

  Here the pattern given for regular expression indicates that the line should start with either *From* or *from*. Then there may be 0 or more characters, and later the line should end with *edu.*

- **Pattern to extract lines ending with any digit:**
  Replace the `pattern` by following string, rest of the program will remain the same.

```
pattern = '[0-9]$'
```

- **Using *Not* :**

```
pattern = '^[^a-z0-9]+'
```

  Here, the first ^ indicates we want something to match in the beginning of a line. Then, the ^ inside square-brackets indicate *do not match any single character within bracket*. Hence, the whole meaning would be - line must be started with anything other than a lower-case alphabets and digits. In other words, the line should not be started with lowercase alphabet and digits.

- **Start with upper case letters and end with digits:**

```
pattern = '^[A-Z].*[0-9]$'
```

  Here, the line should start with capital letters, followed by 0 or more characters, but must end with any digit.

3.4.2 Extracting Data using Regular Expressions

Python provides a method ***findall()*** to extract all of the substrings matching a regular expression. This function returns a list of all non-overlapping matches in the string. If there is no match found, the function returns an empty list. Consider an example of extracting anything that looks like an email address from any line.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting
    @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)
```

The output would be -

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Here, the pattern indicates at least one non-white space characters (`\S`) before @ and at least one non-white space after @. Hence, it will not match with `@2pm`, because of a whitespace before @.

Now, we can write a complete program to extract all email-ids from the file.

```
import re
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line) if
    len(x) > 0:
        print(x)
```

Here, the condition `len(x) > 0` is checked because, we want to print only the line which contain an email-ID. If any line do not find the match for a pattern given, the *findall()* function will return an empty list. The length of empty list will be zero, and hence we would like to print the lines only with length greater than 0.

The output of above program will be something as below -

```
['stephen.marquard@uct.ac.za']
['<postmaster@collab.sakaiproject.org>']
['<200801051412.m05ECIaH010327@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
……..
……..
```

Note that, apart from just email-ID's, the output contains additional characters (<, >, ; etc) attached to the extracted pattern. To remove all that, refine the pattern. That is, we want email-ID to be started with any alphabets or digits, and ending with only alphabets. Hence, the statement would be -

```
x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
```

3.4.3 Combining Searching and Extracting

Assume that we need to extract the data in a particular syntax. For example, we need to extract the lines containing following format -

```
X-DSPAM-Confidence:    0.8475
X-DSPAM-Probability: 0.0000
```

The line should start with `X-`, followed by 0 or more characters. Then, we need a colon and white-space. They are written as it is. Then there must be a number containing one or more digits with or without a decimal point. Note that, we want dot as a part of our pattern string, but not as meta character here. The pattern for regular expression would be -

```
^X-.*: [0-9.]+
```

The complete program is -

```
import re
hand  =  open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
          print(line)
```

The output lines will as below -

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6961
X-DSPAM-Probability: 0.0000
……
……
```

Assume that, we want only the numbers (representing confidence, probability etc) in the above output. We can use *split()* function on extracted string. But, it is better to refine regular expression. To do so, we need the help of parentheses.

When we add parentheses to a regular expression, they are ignored when matching the string. But when we are using ***findall()***, parentheses indicate that while we want the whole expression to match, we only are interested in extracting a portion of the substring that matches the regular expression.

```
import re
hand  =  open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X-\S*: ([0-9.]+)', line) if
    len(x) > 0:
        print(x)
```

Because of the parentheses enclosing the pattern above, it will match the pattern starting with X- and extracts only digit portion. Now, the output would be -

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
.....
.....
```

Another example of similar form: The file *mbox-short.txt* contains lines like -

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

We may be interested in extracting only the revision numbers mentioned at the end of these lines. Then, we can write the statement -

```
x = re.findall('^Details:.*rev=([0-9.]+)', line)
```

The regex here indicates that the line must start with `Details:`, and has something with `rev=` and then digits. As we want only those digits, we will put parenthesis for that portion of expression. Note that, the expression `[0-9]` is greedy, because, it can display very large number. It keeps grabbing digits until it finds any other character than the digit. The output of above regular expression is a set of revision numbers as given below -

```
['39772']
['39771']
['39770']
['39769']
.....
.....
```

Consider another example - we may be interested in knowing time of a day of each email. The file *mbox-short.txt* has lines like -

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Here, we would like to extract only the hour 09. That is, we would like only two digits representing hour. Hence, we need to modify our expression as -

```
x = re.findall('^From .* ([0-9][0-9]):', line)
```

Here, `[0-9][0-9]` indicates that a digit should appear only two times. The alternative way of writing this would be -

```
x = re.findall('^From .* ([0-9]{2}):', line)
```

The number 2 within flower-brackets indicates that the preceding match should appear exactly two times. Hence `[0-9]{2}` indicates there can be exactly two digits. Now, the output would be -

```
['09']
['18']
['16']
['15']
……
……
```

3.4.4 Escape Character

As we have discussed till now, the character like dot, plus, question mark, asterisk, dollar etc. are meta characters in regular expressions. Sometimes, we need these characters themselves as a part of matching string. Then, we need to escape them using a backslash. For example,

```
import re
x = 'We just received $10.00 for cookies.' y =
re.findall('\$[0-9.]+',x)
```

Output:

```
['$10.00']
```

Here, we want to extract only the price $10.00. As, $ symbol is a metacharacter, we need to use \ before it. So that, now $ is treated as a part of matching string, but not as metacharacter.

3.4.5 Bonus Section for Unix/Linux Users

Support for searching files using regular expressions was built into the Unix OS. There is a command-line program built into Unix called ***grep*** (Generalized Regular Expression Parser) that behaves similar to ***search()*** function.

```
$ grep '^From:'  mbox-short.txt
```

Output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Note that, *grep* command does not support the non-blank character `\S`, hence we need to use `[^ ]` indicating not a white-space.

MODULE - 4

4.1 CLASSES AND OBJECTS

Python is an object-oriented programming language, and *class* is a basis for any object oriented programming language. Class is a user-defined data type which binds data and functions together into single entity. Class is just a prototype (or a logical entity/blue print) which will not consume any memory. An object is an instance of a class and it has physical existence. One can create any number of objects for a class. A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).

(Overview of general OOP concepts is given at the end of this module as an extra topic. Those who are new to OOP concepts, it is suggested to have a glance and then continue reading).

4.1.1 Programmer-defined Types

A class in Python can be created using a keyword `class`. Here, we are creating an empty class without any members by just using the keyword `pass` within it.

```
class Point:
    pass

print(Point)
```

The output would be -

```
<class '__main__.Point'>
```

The term `__main__` indicates that the class `Point` is in the main scope of the current module. In other words, this class is at the top level while executing the program.

Now, a user-defined data type `Point` got created, and this can be used to create any number of objects of this class. Observe the following statements -

```
p=Point()
```

Now, a reference (for easy understanding, treat reference as a pointer) to `Point` object is created and is returned. This returned reference is assigned to the object `p`. The process of creating a new object is called as ***instantiation*** and the object is ***instance*** of a class. When we print an object, Python tells which class it belongs to and where it is stored in the memory.

```
print(p)
```

The output would be -

```
<__main__.Point object at 0x003C1BF0>
```

The output displays the address (in hexadecimal format) of the object in the memory. It is now clear that, the object occupies the physical space, whereas the class does not.

4.1.2 Attributes

An object can contain named elements known as ***attributes***. One can assign values to these attributes using dot operator. For example, keeping coordinate points in mind, we can assign two attributes `x` and `y` for the object `p` of a class `Point` as below -

```
p.x =10.0
p.y =20.0
```

A state diagram that shows an object and its attributes is called as ***object diagram.*** For the object `p`, the object diagram is shown in Figure 4.1.

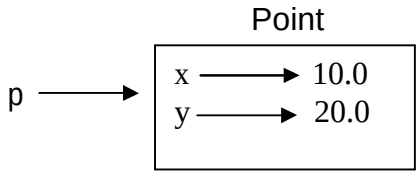


Figure 4.1 Object Diagram

The diagram indicates that a variable (i.e. object) `p` refers to a `Point` object, which contains two attributes. Each attributes refers to a floating point number.

One can access attributes of an object as shown -

```
>>> print(p.x)
     10.0
>>> print(p.y)
     20.0
```

Here, `p.x` means *"Go to the object p refers to and get the value of x".* Attributes of an object can be assigned to other variables -

```
>>> x= p.x
>>> print(x)
     10.0
```

Here, the variable `x` is nothing to do with attribute `x`. There will not be any name conflict between normal program variable and attributes of an object.

A complete program: Write a class Point representing a point on coordinate system. Implement following functions -

- A function `read_point()` to receive x and y attributes of a Point object as user input.

- A function `distance()` which takes two objects of Point class as arguments and computes the Euclidean distance between them.
- A function `print_point()` to display one point in the form of ordered-pair.

```
import math

class Point:
    """ This is a class Point representing a
    coordinate point
    """

def read_point(p):
    p.x=float(input("x  coordinate:"))
    p.y=float(input("y coordinate:"))

def print_point(p):
    print("(%g,%g)"%(p.x, p.y))

def distance(p1,p2):
    d=math.sqrt((p1.x-p2.x)**2+(p1.y-p2.y)**2)
    return d

p1=Point()                      #create  first  object
print("Enter First point:")
read_point(p1)                  #read x and y for p1

p2=Point()                      #create  second  object
print("Enter Second point:")
read_point(p2)                  #read x and y for p2

dist=distance(p1,p2)            #compute  distance
print("First point is:")
print_point(p1)                 #print p1
print("Second point is:")
print_point(p2)                 #print p2

print("Distance is: %g" %(distance(p1,p2)))   #print d
```

The sample output of above program would be -

```
Enter First point:
x coordinate:10
y coordinate:20
Enter Second point: x
coordinate:3
y coordinate:5
First point is: (10,20)
```

```
Second point is:(3,5)
Distance is: 16.5529
```

Let us discuss the working of above program thoroughly -

- The class `Point` contains a string enclosed within 3 double-quotes. This is known as ***docstring***. Usually, a string literal is written within 3 consecutive double-quotes inside a class, module or function definition. It is an important part of documentation and is to help someone to understand the purpose of the said class/module/function. The docstring becomes a value for the special attribute viz. `__doc__` available for any class (and objects of that class). To get the value of docstring associated with a class, one can use the statements like -

```
>>> print(Point.__doc__)
 This is a class Point representing a coordinate point

>>> print(p1.__doc__)
 This is a class Point representing a coordinate point
```

  Note that, you need to type two underscores, then the word `doc` and again two underscores.

  In the above program, there is no need of docstring and we would have just used `pass` to indicate an empty class. But, it is better to understand the professional way of writing user-defined types and hence, introduced docstring.

- The function `read_point()` take one argument of type `Point` object. When we use the statements like,

```
read_point(p1)
```

  the parameter `p` of this function will act as an alias for the argument `p1`. Hence, the modification done to the alias `p` reflects the original argument `p1`. With the help of this function, we are instructing Python that the object p1 has two attributes x and y.

- The function `print_point()` also takes one argument and with the help of format-strings, we are printing the attributes x and y of the Point object as an ordered-pair (x,y).

- As we know, the Euclidean distance between two points (x1,y1) and (x2,y2) is

$$\sqrt{(x1-x2)^2+(y1-y2)^2}$$

  In this program, we have Point objects as (p1.x, p1.y) and (p2.x, p2.y). Apply the formula on these points by passing objects p1 and p2 as parameters to the function `distance()`. And then return the result.

Thus, the above program gives an idea of defining a class, instantiating objects, creating attributes, defining functions that takes objects as arguments and finally, calling (or invoking) such functions whenever and wherever necessary.

NOTE: User-defined classes in Python have two types of attributes viz. ***class attributes*** and ***instance attributes***. Class attributes are defined inside the class (usually, immediately after class header). They are common to all the objects of that class. That is, they are shared by all the objects created from that class. But, instance attributes defined for individual objects. They are available only for that instance (or object). Attributes of one instance are not available for another instance of the same class. For example, consider the class Point as discussed earlier -

```
class Point:
     pass

p1= Point()                #first object of the class
p1.x=10.0                  #attributes for p1
p1.y=20.0
print(p1.x, p1.y)          #prints 10.0 20.0

p2=  Point()               #second object of the class
print(p2.x)                #displays error as below

AttributeError: 'Point' object has no attribute 'x'
```

This clearly indicates that the attributes x and y created are available only for the object p1, but not for p2. Thus, x and y are instance attributes but not class attributes.

We will discuss class attributes late in-detail. But, for the understanding purpose, observe the following example -

```
class Point:
     x=2
     y=3

p1=Point()                 #first object of the class
print(p1.x, p1.y)          # prints 2 3

p2=Point()                 #second object of the class
print(p2.x, p2.y)          # prints 2 3
```

Here, the attributes `x` and `y` are defined inside the definition of the class `Point` itself. Hence, they are available to all the objects of that class.

4.1.3 Rectangles

It is possible to make an object of one class as an attribute to other class. To illustrate this, consider an example of creating a class called as Rectangle. A rectangle can be created using any of the following data -

- By knowing width and height of a rectangle and one corner point (ideally, a bottom-left corner) in a coordinate system
- By knowing two opposite corner points

Let us consider the first technique and implement the task: Write a class Rectangle containing numeric attributes width and height. This class should contain another attribute *corner* which is an instance of another class Point. Implement following functions -

- A function to print corner point as an ordered-pair
- A function *find_center()* to compute center point of the rectangle
- A function *resize()* to modify the size of rectangle

The program is as given below -

```
class Point:
    """ This is a class Point
        representing  coordinate  point
    """

class Rectangle:
    """ This is a class Rectangle.
    Attributes: width, height and Corner Point
    """

def find_center(rect):
    p=Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p

def resize(rect, w, h):
    rect.width +=w
    rect.height +=h

def print_point(p):
    print("(%g,%g)"%(p.x, p.y))


box=Rectangle()          #create Rectangle object
box.corner=Point()       #define an attribute corner for box
box.width=100            #set attribute width to box
box.height=200           #set attribute height to box
box.corner.x=0           #corner itself has two attributes x and y
box.corner.y=0           #initialize x and y to 0
```

```
print("Original Rectangle is:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of rectangle is:")
print_point(center)

resize(box,50,70)
print("Rectangle after resize:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of resized rectangle is:")
print_point(center)
```

A sample output would be:

```
Original Rectangle is: width=100, height=200
The center of rectangle is: (50,100)
Rectangle after resize: width=150, height=270
The center of resized rectangle is: (75,135)
```

The working of above program is explained in detail here -

- Two classes `Point` and `Rectangle` have been created with suitable docstrings. As of now, they do not contain any class-level attributes.
- The following statement instantiates an object of `Rectangle` class.

  ```
  box=Rectangle()
  ```

  The statement

  ```
  box.corner=Point()
  ```

  indicates that `corner` is an attribute for the object `box` and this attribute itself is an object of the class `Point`. The following statements indicate that the object `box` has two more attributes -

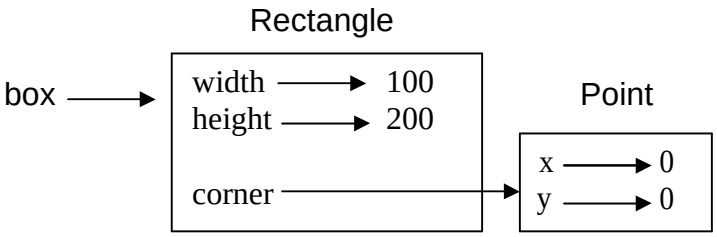
  ```
  box.width=100        #give any numeric value
  box.height=200       #give any numeric value
  ```

  In this program, we are treating the corner point as the origin in coordinate system and hence the following assignments -

  ```
  box.corner.x=0
  box.corner.y=0
  ```

  (Note that, instead of origin, any other location in the coordinate system can be given as corner point.)

  Based on all above statements, an object diagram can be drawn as -



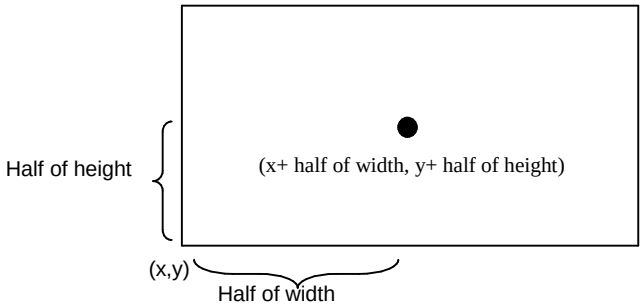
The expression `box.corner.x` means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."

- The function `find_center()` takes an object `rect` as an argument. So, when a call is made using the statement -

```
center=find_center(box)
```

the object `rect` acts as an alias for the argument `box`.

A local object `p` of type `Point` has been created inside this function. The attributes of p are x and y, which takes the values as the coordinates of center point of rectangle. Center of a rectangle can be computed with the help of following diagram.



The function `find_center()` returns the computed center point. Note that, the return value of a function here is an instance of some class. That is, one can have an ***instance as return values*** from a function.

- The function `resize()` takes three arguments: `rect` - an instance of Rectangle class and two numeric variables `w` and `h`. The values `w` and `h` are added to existing attributes `width` and `height`. This clearly shows that ***objects are mutable***. State of an object can be changed by modifying any of its attributes. When this function is called with a statement -

```
resize(box,50,70)
```

the `rect` acts as an alias for `box`. Hence, width and height modified within the function will reflect the original object `box`.

Thus, the above program illustrates the concepts: ***object of one class is made as attribute for object of another class, returning objects from functions*** and ***objects are mutable.***

4.1.4 Copying

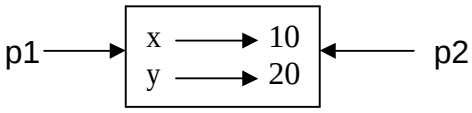
An object will be aliased whenever there an object is assigned to another object of same class. This may happen in following situations -

- Direct object assignment (like p2=p1)
- When an object is passed as an argument to a function
- When an object is returned from a function

The last two cases have been understood from the two programs in previous sections. Let us understand the concept of aliasing more in detail using the following program -

```
>>> class Point:
        pass

>>> p1=Point()
>>> p1.x=10
>>> p1.y=20
>>> p2=p1
>>> print(p1)
     <__main__.Point object at 0x01581BF0>
>>> print(p2)
     <__main__.Point object at 0x01581BF0>
```

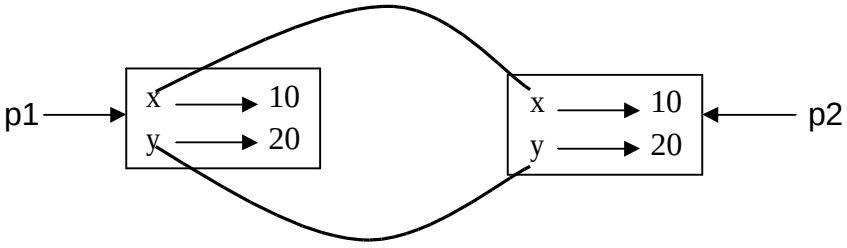
Observe that both p1 and p2 objects have same physical memory. It is clear now that the object p2 is an alias for p1. So, we can draw the object diagram as below -



Hence, if we check for equality and identity of these two objects, we will get following result.

```
>>> p1 is p2
     True
>>>  p1==p2
     True
```

But, the aliasing is not good always. For example, we may need to create a new object using an existing object such that - the new object should have a different physical memory, but it must have same attribute (and their values) as that of existing object. Diagrammatically, we need something as below -



In short, ***we need a copy of an object, but not an alias.*** To do this, Python provides a module called ***copy*** and a method called ***copy()***. Consider the below given program to understand the concept.

```
>>> class Point:
        pass

>>> p1=Point()
>>> p1.x=10
>>> p1.y=20

>>> import copy                  #import module copy
>>> p3=copy.copy(p1)            #use the method copy()
>>> print(p1)
     <__main__.Point object at 0x01581BF0>
>>> print(p3)
     <__main__.Point object at 0x02344A50>
>>> print(p3.x,p3.y)
     10 20
```

Observe that the physical address of the objects `p1` and `p3` are now different. But, values of attributes `x` and `y` are same. Now, use the following statements -

```
>>> p1 is p3
     False
>>>  p1  ==  p3
     False
```

Here, the `is` operator gives the result as False for the obvious reason of `p1` and `p3` are being two different entities on the memory. But, why `==` operator is generating False as the result, though the contents of two objects are same? The reason is - `p1` and `p3` are the objects of user-defined type. And, Python cannot understand the meaning of equality on the new data type. The default behavior of equality (`==`) is identity (`is` operator) itself. Hence, Python applies this default behavior on `p1 == p3` and results in `False`.

(**NOTE:** If we need to define the meaning of equality (==) operator explicitly on user-defined data types (i.e. on class objects), then we need to override the method `__eq__()` inside the class. This will be discussed later in detail.)

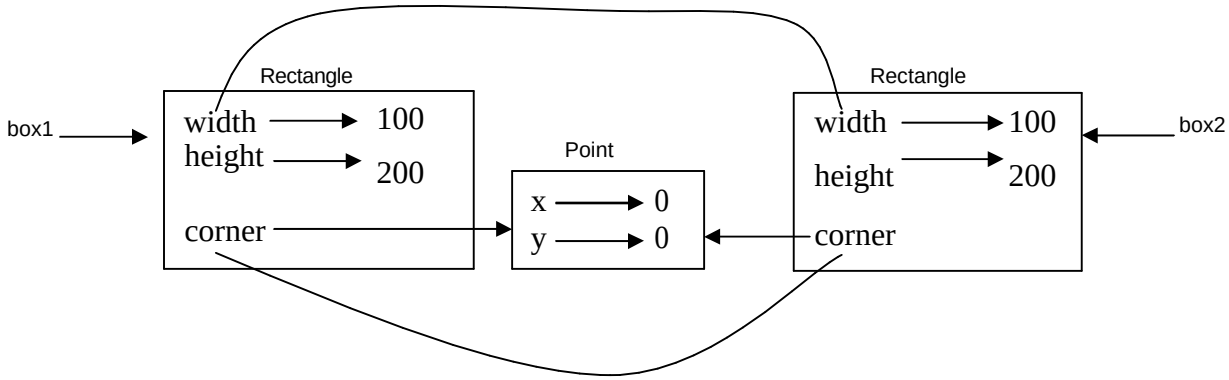
The ***copy()*** method of ***copy*** module duplicates the object. The content (i.e. attributes) of one object is copied into another object as we have discussed till now. But, when an object itself is an attribute inside another object, the duplication will result in a strange manner. To understand this concept, try to copy Rectangle object (created in previous section) as given below -

```
import   copy
class Point:
    """ This is a class Point
        representing  coordinate  point
    """

class Rectangle:
    """ This is a class Rectangle.
    Attributes: width, height and Corner Point
    """

box1=Rectangle()
box1.corner=Point()
box1.width=100
box1.height=200
box1.corner.x=0
box1.corner.y=0

box2=copy.copy(box1)
print(box1 is box2)                          #prints False
print(box1.corner is box2.corner)            #prints True
```

Now, the question is - why **box1.corner** and **box2.corner** are same objects, when **box1** and **box2** are different? Whenever the statement

```
box2=copy.copy(box1)
```

is executed, the contents of all the attributes of `box1` object are copied into the respective attributes of `box2` object. That is, `box1.width` is copied into `box2.width`, `box1.height` is copied into `box2.height`. Similarly, `box1.corner` is copied into `box2.corner`. Now, recollect the fact that corner is not exactly the object itself, but it is a reference to the object of type Point (Read the discussion done for Figure 4.1 at the beginning of this Chapter). Hence, the value of reference (that is, the physical address) stored in `box1.corner` is copied into `box2.corner`. Thus, the physical object to which `box1.corner` and `box2.corner` are pointing is only one. This type of copying the objects is known as ***shallow copy.*** To understand this behavior, observe the following diagram -



Now, the attributes `width` and `height` for two objects `box1` and `box2` are independent. Whereas, the attribute `corner` is shared by both the objects. Thus, any modification done to `box1.corner` will reflect `box2.corner` as well. Obviously, we don't want this to happen, whenever we create duplicate objects. That is, we want two independent physical objects. Python provides a method ***deepcopy()*** for doing this task. This method copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on.

```
box3=copy.deepcopy(box1)
print(box1 is box3)                          #prints False
print(box1.corner is box3.corner)            #prints False
```

Thus, the objects `box1` and `box3` are now completely independent.

4.1.5 Debugging

While dealing with classes and objects, we may encounter different types of errors. For example, if we try to access an attribute which is not there for the object, we will get ***AttributeError***. For example -

```
>>> p= Point()
>>> p.x = 10
>>> p.y = 20
>>> print(p.z)
    AttributeError: 'Point' object has no attribute 'z'
```

To avoid such error, it is better to enclose such codes within try/except as given below -

```
try:
     z = p.x
except  AttributeError:
     z = 0
```

When we are not sure, which type of object it is, then we can use ***type()*** as -

```
>>> type(box1)
     <class '__main__.Rectangle'>
```

Another method ***isinstance()*** helps to check whether an object is an instance of a particular class -

```
>>>  isinstance(box1,Rectangle)
     True
```

When we are not sure whether an object has a particular attribute or not, use a function ***hasattr()*** -

```
>>> hasattr(box1,  'width')
     True
```

Observe the string notation for second argument of the function ***hasattr()***. Though the attribute `width` is basically numeric, while giving it as an argument to function ***hasattr()***, it must be enclosed within quotes.

4.2 CLASSES AND FUNCTIONS

Though Python is object oriented programming languages, it is possible to use it as functional programming. There are two types of functions viz. ***pure functions*** and ***modifiers***. A pure function takes objects as arguments and does some work without modifying any of the original argument. On the other hand, as the name suggests, modifier function modifies the original argument.

In practical applications, the development of a program will follow a technique called as ***prototype*** and ***patch***. That is, solution to a complex problem starts with simple prototype and incrementally dealing with the complications.

4.2.1 Pure Functions

To understand the concept of pure functions, let us consider an example of creating a class called Time. An object of class Time contains hour, minutes and seconds as attributes. Write a function to print time in HH:MM:SS format and another function to add two time objects. Note that, adding two time objects should yield proper result and hence we need to check whether number of seconds exceeds 60, minutes exceeds 60 etc, and take appropriate action.

```
class Time:
    """Represents the time of a day
    Attributes: hour, minute, second """

def printTime(t):
    print("%.2d:%.2d:%.2d"%(t.hour,t.minute,t.second))

def add_time(t1,t2):
    sum=Time()
    sum.hour = t1.hour + t2.hour
    sum.minute  =  t1.minute  +  t2.minute
    sum.second = t1.second + t2.second
```

```
        if sum.second >= 60:
            sum.second -= 60
            sum.minute += 1 if
        sum.minute >= 60:
            sum.minute  -=  60
            sum.hour += 1

        return sum

    t1=Time()
    t1.hour=10
    t1.minute=34
    t1.second=25
    print("Time1   is:")
    printTime(t1)

    t2=Time()
    t2.hour=2
    t2.minute=12
    t2.second=41
    print("Time2  is  :")
    printTime(t2)

    t3=add_time(t1,t2)
    print("After adding two time objects:")
    printTime(t3)
```

The output of this program would be -

Time1 is: 10:34:25
Time2 is : 02:12:41
After adding two time objects: 12:47:06

Here, the function `add_time()` takes two arguments of type `Time`, and returns a `Time` object, whereas, it is not modifying contents of its arguments t1 and t2. Such functions are called as *pure functions.*

4.2.2 Modifiers

Sometimes, it is necessary to modify the underlying argument so as to reflect the caller. That is, arguments have to be modified inside a function and these modifications should be available to the caller. The functions that perform such modifications are known as ***modifier function.*** Assume that, we need to add few seconds to a time object, and get a new time. Then, we can write a function as below -

```
def increment(t, seconds):
    t.second += seconds

    while t.second >= 60:
        t.second -= 60
        t.minute += 1

    while t.minute >= 60:
        t.minute -= 60
        t.hour += 1
```

In this function, we will initially add the argument `seconds` to `t.second`. Now, there is a chance that `t.second` is exceeding 60. So, we will increment minute counter till `t.second` becomes lesser than 60. Similarly, till the `t.minute` becomes lesser than 60, we will decrement minute counter. Note that, the modification is done on the argument `t` itself. Thus, the above function is a ***modifier.***

4.2.3 Prototyping v/s Planning

Whenever we do not know the complete problem statement, we may write the program initially, and then keep of modifying it as and when requirement (problem definition) changes. This methodology is known as ***prototype and patch.*** That is, first design the prototype based on the information available and then perform patch-work as and when extra information is gathered. But, this type of incremental development may end-up in unnecessary code, with many special cases and it may be unreliable too.

An alternative is ***designed development***, in which high-level insight into the problem can make the programming much easier. For example, if we consider the problem of adding two time objects, adding seconds to time object etc. as a problem involving numbers with base 60 (as every hour is 60 minutes and every minute is 60 seconds), then our code can be improved. Such improved versions are discussed later in this chapter.

4.2.4 Debugging

In the program written in Section 4.2.1, we have treated time objects as valid values. But, what if the attributes (second, minute, hour) of time object are given as wrong values like negative number, or hours with value more than 24, minutes/seconds with more than 60 etc? So, it is better to write error-conditions in such situations to verify the input. We can write a function similar to as given below -

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False

    if time.minute >= 60 or time.second >= 60:
        return False

    return True
```

Now, at the beginning of `add_time()` function, we can put a condition as -

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')

    #remaining statements of add_time() functions
```

Python provides another debugging statement ***assert***. When this keyword is used, Python evaluates the statement following it. If the statement is True, further statements will be evaluated sequentially. But, if the statement is False, then ***AssertionError*** exception is raised. The usage of *assert* is shown here -

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    #remaining statements of add_time() functions
```

The *assert* statement clearly distinguishes the normal conditional statements as a part of the logic of the program and the code that checks for errors.

4.3 CLASSES AND METHODS

The classes that have been considered till now were just empty classes without having any definition. But, in a true object oriented programming, a class contains class-level attributes, instance-level attributes, methods etc. There will be a tight relationship between the object of the class and the function that operate on those objects. Hence, the object oriented nature of Python classes will be discussed here.

4.3.1 Object-Oriented Features

As an object oriented programming language, Python possess following characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways objects in the real world interact.

To establish relationship between the object of the class and a function, we must define a function as a member of the class. A function which is associated with a particular class is known as a ***method***. Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

Now onwards, we will discuss about classes and methods.

4.3.2 The __init__() Method

(A method __init__() has to be written with two underscores before and after the word *init*)

Python provides a special method called as `__init__()` which is similar to constructor method in other programming languages like C++/Java. The term *init* indicates initialization. As the name suggests, this method is invoked automatically when the object of a class is created. Consider the example given here -

```
import math

class Point:
    def __init__(self,a,b):
        self.x=a
        self.y=b

    def dist(self,p2):
        d=math.sqrt((self.x-p2.x)**2  +  (self.y-p2.y)**2)
        return d

    def __str__(self):
        return "(%d,%d)"%(self.x, self.y)

p1=Point(10,20)           #__init__() is called automatically
p2=Point(4,5)             #__init__() is called automatically

print("P1  is:",p1)       #__str__() is called automatically
print("P2 is:",p2)        #__str__() is called automatically

d=p1.dist(p2)             #explicit call for dist()

print("The distance is:",d)
```

The sample output is -

```
P1  is:  (10,20)
P2 is: (4,5)
Distance is: 16.15549442140351
```

Let us understand the working of this program and the concepts involved:

- Keep in mind that every method of any class must have the first argument as ***self***. The argument ***self*** is a reference to the current object. That is, it is reference to the object which invoked the method. (Those who know C++, can relate ***self*** with ***this*** pointer). The object which invokes a method is also known as ***subject***.

- The method `__init__()` inside the class is an initialization method, which will be invoked automatically when the object gets created. When the statement like -

```
p1=Point(10,20)
```

is used, the `__init__()` method will be called automatically. The internal meaning of the above line is -

```
p1.__init__(10,20)
```

Here, p1 is the object which is invoking a method. Hence, reference to this object is created and passed to `__init__()` as *`self`*. The values 10 and 20 are passed to formal parameters a and b of `__init__()` method. Now, inside `__init__()` method, we have statements

```
self.x=10
self.y=20
```

This indicates, `x` and `y` are instance attributes. The value of `x` for the object `p1` is 10 and, the value of `y` for the object `p1` is 20.

When we create another object `p2`, it will have its own set of `x` and `y`. That is, memory locations of instance attributes are different for every object.

Thus, state of the object can be understood by instance attributes.

- The method `dist()` is an ordinary member method of the class `Point`. As mentioned earlier, its first argument must be self. Thus, when we make a call as -

```
d=p1.dist(p2)
```

a reference to the object `p1` is passed as `self` to `dist()` method and `p2` is passed explicitly as a second argument. Now, inside the `dist()` method, we are calculating distance between two point (Euclidian distance formula is used) objects. Note that, in this method, we cannot use the name `p1`, instead we will use `self` which is a reference (alias) to `p1`.

- The next method inside the class is ***__str__(). It is a special method used for string representation of user-defined object.*** Usually, `print()` is used for printing basic types in Python. But, user-defined types (class objects) have their own meaning and a way of representation. To display such types, we can write functions or methods like `print_point()` as we did in Section 4.1.2. But, more polymorphic way is to use `__str__()` so that, when we write just `print()` in the main part of the program, the `__str__()` method will be invoked automatically. Thus, when we use the statement like -

```
print("P1 is:",p1)
```

the ordinary print() method will print the portion “P1 is:” and the remaining portion is taken care by `__str__()` method. In fact, `__str__()` method will return the string

format what we have given inside it, and that string will be printed by `print()` method.

4.3.3 Operator Overloading

Ability of an existing operator to work on user-defined data type (class) is known as operator overloading. It is a polymorphic nature of any object oriented programming. Basic operators like +, -, * etc. can be overloaded. To overload an operator, one needs to write a method within user-defined class. Python provides a special set of methods which have to be used for overloading required operator. The method should consist of the code what the programmer is willing to do with the operator. Following table shows gives a list of operators and their respective Python methods for overloading.

Operator	Special Function in Python	Operator	Special Function in Python
+	__add__()	<=	__le__()
-	__sub__()	>=	__ge__()
*	__mul__()	==	__eq__()
/	__truediv__()	!=	__ne__()
%	__mod__()	in	__contains__()
<	__lt__()	len	__len__()
>	__gt__()	str	__str__()

Let us consider an example of Point class considered earlier. Using operator overloading, we can try to add two point objects. Consider the program given below -

```
class Point:
    def __init__(self,a=0,b=0):
        self.x=a
        self.y=b

    def __add__(self, p2):
        p3=Point()
        p3.x=self.x+p2.x
        p3.y=self.y+p2.y
        return p3

    def __str__(self):
        return "(%d,%d)"%(self.x, self.y)
```

```
p1=Point(10,20)
p2=Point(4,5)
print("P1  is:",p1)
print("P2 is:",p2)
p4=p1+p2                    #call for __add__() method
print("Sum is:",p4)
```

The output would be -

```
P1  is:  (10,20)
P2 is: (4,5)
Sum is: (14,25)
```

In the above program, when the statement `p4 = p1+p2` is used, it invokes a special method `__add__()` written inside the class. Because, internal meaning of this statement is-

```
p4 = p1.__add__(p4)
```

Here, `p1` is the object invoking the method. Hence, `self` inside `__add__()` is the reference (alias) of `p1`. And, `p4` is passed as argument explicitly.

In the definition of `__add__()`, we are creating an object `p3` with the statement -

```
p3=Point()
```

The object `p3` is created without initialization. Whenever we need to create an object with and without initialization in the same program, we must set arguments of `__init__()` for some default values. Hence, in the above program arguments `a` and `b` of `__init__()` are made as default arguments with values as zero. Thus, `x` and `y` attributes of `p3` will be now zero. In the `__add__()` method, we are adding respective attributes of `self` and `p2` and storing in `p3.x` and `p3.y`. Then the object `p3` is returned. This returned object is received as `p4` and is printed.

NOTE that, in a program containing operator overloading, the overloaded operator behaves in a normal way when basic types are given. That is, in the above program, if we use the statements

```
m= 3+4
print(m)
```

it will be usual addition and gives the result as 7. But, when user-defined types are used as operands, then the overloaded method is invoked.

Let us consider a more complicated program involving overloading. Consider a problem of creating a class called Time, adding two Time objects, adding a number to Time object etc. that we had considered in previous section. Here is a complete program with more of OOP concepts.

```
class Time:
    def __init__(self, h=0,m=0,s=0):
        self.hour=h
        self.min=m
        self.sec=s

    def time_to_int(self):
        minute=self.hour*60+self.min
        seconds=minute*60+self.sec
        return seconds

    def int_to_time(self, seconds):
        t=Time()
        minutes,   t.sec=divmod(seconds,60)
        t.hour,    t.min=divmod(minutes,60)
        return t

    def __str__(self):
        return "%.2d:%.2d:%.2d"%(self.hour,self.min,self.sec)

    def __eq__(self,t):
        return self.hour==t.hour and self.min==t.min and self.sec==t.sec

    def __add__(self,t):
        if isinstance(t, Time):
            return  self.addTime(t)
        else:
            return self.increment(t)

    def addTime(self, t):
        seconds=self.time_to_int()+t.time_to_int()
        return self.int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return self.int_to_time(seconds)

    def __radd__(self,t):
        return self.__add__(t)

T1=Time(3,40)
T2=Time(5,45)
print("T1  is:",T1)
print("T2 is:",T2)
print("Whether T1 is same as T2?",T1==T2)      #call for __eq__()

T3=T1+T2                        #call for __add__()
```

```
print("T1+T2 is:",T3)

T4=T1+75                        #call for __add__()
print("T1+75=",T4)

T5=130+T1                       #call for __radd__()
print("130+T1=",T5)

T6=sum([T1,T2,T3,T4])
print("Using sum([T1,T2,T3,T4]):",T6)
```

The output would be -

```
T1  is:  03:40:00
T2 is: 05:45:00
Whether T1 is same as T2? False
T1+T2 is: 09:25:00
T1+75=   03:41:15
130+T1= 03:42:10
Using sum([T1,T2,T3,T4]): 22:31:15
```

Working of above program is explained hereunder -

- The class Time has `__init__()` method for initialization of instance attributes `hour`, `min` and `sec`. The default values of all these are being zero.
- The method `time_to_int()` is used convert a Time object (hours, min and sec) into single integer representing time in number of seconds.
- The method `int_to_time()` is written to convert the argument `seconds` into time object in the form of hours, min and sec. The built-in method ***divmod()*** gives the quotient as well as remainder after dividing first argument by second argument given to it.
- Special method `__eq__()` is for overloading equality (==) operator. We can say one Time object is equal to the other Time object if underlying hours, minutes and seconds are equal respectively. Thus, we are comparing these instance attributes individually and returning either True of False.
- When we try to perform addition, there are 3 cases -
  - Adding two time objects like `T3=T1+T2`.
  - Adding integer to Time object like `T4=T1+75`
  - Adding Time object to an integer like `T5=130+T1`

  Each of these cases requires different logic. When first two cases are considered, the first argument will be `T1` and hence `self` will be created and passed to `__add__()` method. Inside this method, we will check the type of second argument using `isinstance()` method. If the second argument is Time object, then we call `addTime()` method. In this method, we will first convert both Time objects to integer (seconds) and then the resulting sum into Time object again. So, we make use `time_to_int()` and `int_to_time()` here. When the 2nd argument is an integer,

it is obvious that it is number of seconds. Hence, we need to call `increment()` method.

Thus, based on the type of argument received in a method, we take appropriate action. This is known as ***type-based dispatch.***

In the 3rd case like `T5=130+T1`, Python tries to convert first argument 130 into `self`, which is not possible. Hence, there will be an error. This indicates that for Python, `T1+5` is not same as `5+T1` (Commutative law doesn't hold good!!). To avoid the possible error, we need to implement ***right-side addition*** method `__radd__()`. Inside this method, we can call overloaded method `__add__()`.

- The beauty of Python lies in surprising the programmer with more facilities!! As we have implemented `__add__()` method (that is, overloading of + operator), the builtin `sum()` will is capable of adding multiple objects given in a sequence. This is due to ***Polymorphism*** in Python. Consider a list containing Time objects, and then call `sum()` on that list as -

```
T6=sum([T1,T2,T3,T4])
```

The `sum()` internally calls `__add__()` method multiple times and hence gives the appropriate result. Note down the square-brackets used to combine Time objects as a list and then passing it to `sum()`.

Thus, the program given here depicts many features of OOP concepts.

4.3.4 Debugging

We have seen earlier that ***hasattr()*** method can be used to check whether an object has particular attribute. There is one more way of doing it using a method ***vars()***. This method maps attribute names and their values as a dictionary. For example, for the Point class defined earlier, use the statements -

```
>>> p = Point(3, 4)
>>> vars(p)                  #output is {'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
     for attr in vars(obj):
          print(attr, getattr(obj, attr))
```

Here, `print_attributes()` traverses the dictionary and prints each attribute name and its corresponding value. The built-in function `getattr()` takes an object and an attribute name (as a string) and returns the attribute's value.

GENERAL OOP CONCEPTS

At the earlier age of computers, the programming was done using assembly language. Even though, the assembly language can be used to produce highly efficient programs, it is not easy to learn or to use effectively. Moreover, debugging assembly code is quite difficult. At the later stage, the programming languages like BASIC, COBOL and FORTRAN came into existence. But, these languages are non-structured and consisting of a mass of tangled jumps and conditional branches that make a program virtually impossible to understand.

To overcome these problems, the structured or procedural programming methodology was developed. Here, the actual problem is divided into small independent tasks/modules. Then the programs are written for each of these tasks and they are grouped together to get the final solution for the given problem. Thus, the solution design technique for this method is known as *top-down approach*. C is the one successful language that adopted structured programming style. However, even with the structured programming methods, once a project/program reaches a certain size, its complexity exceeds what a programmer can manage.

The new approach - object oriented programming was developed to overcome the problems with structured approach. In this methodology, the actual data and the operations to be performed on that are grouped as a single entity called object. The objects necessary to get the solution of a problem are identified initially. The interactions between various objects are then identified to achieve the solution of original problem. Thus, it is also known as *bottom-up approach.* Object oriented concepts inhabits many advantages like reusability of the code, security etc.

In structured programming approach, the programs are written around *what is happening* rather than *who is being affected.* That is, structured programming focuses more on the process or operation and not on the data to be processed. This is known as *process oriented model* and this can be thought of as *code acting on data.* For example, a program written in C is defined by its functions, any of which may operate on any type of data used by the program.

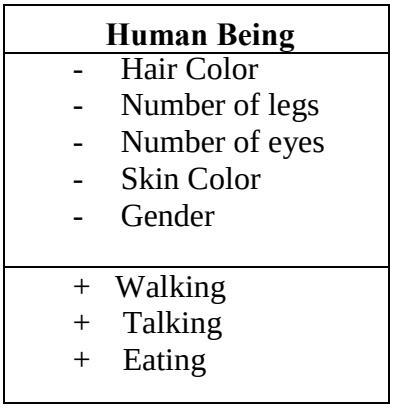
But, the real world problems are not organized into data and functions independent of each other and are tightly closed. So, it is better to write a program around '*who is being affected'.* This kind of data-centered programming methodology is known as *object oriented programming* (*OOP*) and this can be thought of as *data controlling access to code.* Here, the behavior and/or characteristics of the data/objects are used to determine the function to be written for applying them. Thus, the basic idea behind OOP language is to combine both data and the function that operates on data into a single unit. Such a unit is called as an *object*. A function that operates on data is known as a *member function* and it is the only means of accessing an object's data.

Elements of OOP: The object oriented programming supports some of the basic concepts as building blocks. Every OOPs language normally supports and developed around these features. They are discussed hereunder.

Class: A class is a user defined data type which binds data and functions together into a single entity.

Class is a building block of any object oriented language. As it is discussed earlier, object oriented programming treats data and the code acting on that data as a connected component. That is, data and code are not treated separately as procedure oriented languages do. Thus, OOPs suggests to wrap up the data and functions together into a single entity. Normally, a class represents the prototype of a real world entity. Hence, a class, by its own, is not having any physical existence. It can be treated as a user-defined data type.

Since a class is a prototype or blueprint of a real world entity, it consists of number of properties (known as data members) and behavior (known as member functions). To illustrate this, consider an example of a class representing a human being shown in the following Figure -



Class diagram for Human Being

Few of the properties of human can be *number of legs, number of eyes, gender, number of hands, hair color, skin color etc.* And the functionality or behavior of a human may be *walking, talking, eating, thinking* etc.

Object : An object is an instance of a class.
A class is just a prototype, representing a new data type. To use this new data type, we need to create a variable of this type. Such a variable is known as an object. Thus, an object is a physical entity, which consumes memory. In OOPs, the object represents a real world data which defines the properties and behavior of any real world entity. Every object has its unique existence and it is different from the other objects of a same class.

Let us refer to the class of human being discussed in previous section. Assume that there are two persons: Ramu and Radha. Now, properties of Ramu and Radha may be having different values as shown in the following Table.

Properties of Objects

Property/Attribute	Objects	
	Ramu	Radha
Skin color	Wheatish	Fair
Hair	gray	black
Number of legs	2	2
Number of eyes	2	2

Also, the walking and talking style of both of them may be different. Hence there are two different objects of the same class.

Thus, two or more objects of the same class will differ in the values of their properties and way they behave. But, they share common set of types of properties and behavior.

Encapsulation: The process of binding data and code together into a single entity is called encapsulation.

It is the mechanism that binds code and the data it manipulates together and keeps both safe from misuse and unauthorized manipulation. In any OOP language, the basis of encapsulation is the *class.* Class defines the structure and behavior (i.e. data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class. So, object is also referred to as an *instance of a class* and it is thought just as a variable of user-defined data type. Thus, class is a logical construct and an object is a physical entity. The data defined by the class are known as *member variables* and the functions written to operate on that data are known as *member functions or methods.* The member variables and functions can be *private* or *public.* If a particular member is private, then only the other members of that class can access that member. That is, a private member of the class can't be accessed from outside the class. But, if any member is public, then it can be accessed from anywhere in the program. Thus, through encapsulation, an OOP technique provides high security for user's data and for the entire system.

Data Abstraction: Hiding the implementation details from the end user.

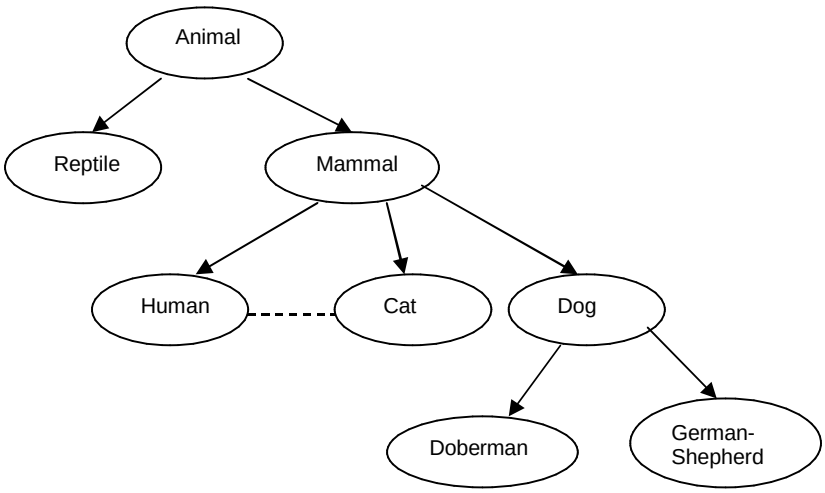
Many a times, abstraction and encapsulation are used interchangeably. But, actually, they are not same. In OOPs, the end user of the program need not know how the actual function works, or what is being done inside a function to make it work. The user must know only the abstract meaning of the task, and he can freely call a function to do that task. The internal details of function may not be known to the user. Designing the program in such a way is called as abstraction.

To understand the concept of abstraction, consider a scenario: When you are using Microsoft Word, if you click on Save icon, a dialogue box appears which allows you to save the document in a physical location of your choice. Similarly, when you click Open icon, another dialogue box appears to select a file to be opened from the hard disk. You, as a user will not be knowing how the internal code would have written so as to open a dialogue box when an icon is being clicked. As a user, those details are not necessary for you. Thus, such implementation details are hidden from the end user. This is an abstraction.

Being an OOPs programmer, one should design a class (with data members and member functions) such a way that, the internal code details are hidden from the end user. OOPs provide a facility of having member functions to achieve this technique and the external world (normally, a main() function) needs to call the member function using an object to achieve a particular task.

Inheritance: Making use of existing code.

It is a process by which one object can acquire the properties of another object. It supports the concept of hierarchical (top-down) classification. For example, consider a large set of animals having their own behaviors. In that, mammals are of one kind having all the properties of animals with some additional behaviors that are unique to them. Again, we can divide mammals class into various mammals like dogs, cats, human etc. Again among the dogs, differentiation is there like Doberman, German-shepherd, Labrador etc. Thus, if we consider a German-shepherd, it is having all the qualities of a dog along with its own special features. Moreover, it exhibits all the properties of a mammal, and in turn of an animal. Hence it is inheriting the properties of animals, then of mammals and then of dogs along with its own specialties. We can depict it as shown in the Figure given below.



Example of Inheritance

Normally, inheritance of this type is also known as “is-a” relationship. Because, we can easily say “Doberman ***is a*** dog”, “Dog ***is a*** mammal” etc. Hence, inheritance is termed as ***Generalization to Specialization*** if we consider from top-to-bottom level. On the other hands, it can be treated as ***Specialization to Generalization*** if it is bottom-to-top level.

This indicates, in inheritance, the topmost base class will be more generalized with only properties which are common to all of its derived classes (various levels) and the bottom-most class is most specialized version of the class which is ready to use in a real-world.

If we apply this concept for programming, it can be easily understood that a code written is reusable. Thus, in this mechanism, it is possible for one object to be a specific instance of a more general case. Using inheritance, an object need only define those qualities that make it unique object within its class. It can inherit its general attributes from its parent.

Polymorphism: ***This can be thought of as one interface, multiple methods.***
It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Using this mechanism, function overloading and operator overloading can be done.

Consider an example of performing stack operation on three different types of data viz. integer, floating-point and characters. In a non-object oriented programming, we have to write functions with different name for push and pop operations for all these types of data even though the logic is same for all the data types. But in OOP languages, we can use the same function names with the data types of the parameters being different. This is an example for function overloading.

We know that the '+' operator is used for adding two numbers. Conceptually, the concatenation of two strings is also an addition. But, in non-object oriented programming language, we cannot use '+' operator to concatenate two strings. This is possible in object oriented programming language by overloading the '+' operator for string operands.

Polymorphism is also meant for *economy of expression*. That is, the way you express things is more economical when you use polymorphism. For example, if you have a function to add two matrices, you can use just a + symbol as:

```
m3 = m1 + m2;
```

here, m1, m2 and m3 are objects of matrix class and + is an overloaded operator. In the same program, if you have a function to concatenate two strings, you can write an overloaded function for + operator to do so -

```
s3= s1+ s2;     where s1, s2 and s3 are strings.
```

Moreover, for adding two numbers, the same + operator is used with its default behavior. Thus, one single operator + is being used for multiple purposes without disturbing its abstract meaning - addition. But, type of data it is adding is different. Hence, the way you have expressed your statements is more economical rather than having multiple functions like -

```
addMat(m1, m2);
concat(s1, s2);   etc.
```

MODULE - 5

5.1 NETWORKED PROGRAMS

In this era of internet, it is a requirement in many situations to retrieve the data from web and to process it. In this section, we will discuss basics of network protocols and Python libraries available to extract data from web.

5.1.1 HyperText Transfer Protocol (HTTP)

HTTP (HyperText Transfer Protocol) is the media through which we can retrieve web-based data. The **HTTP** is an application protocol for distributed and hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP is the protocol to exchange or transfer hypertext.

Consider a situation:

- you try to read a socket, but the program on the other end of the socket has not sent any data, then you need to wait.
- If the programs on both ends of the socket simply wait for some data without sending anything, they will wait for a very long time.

So an important part of programs that communicate over the Internet is to have some sort of protocol. A protocol is a set of precise rules that determine

- Who will send request for what purpose
- What action to be taken
- What response to be given

To send request and to receive response, HTTP uses GET and POST methods.

NOTE: To test all the programs in this section, you must be connected to internet.

5.1.2 The World's Simplest Web Browser

The built-in module ***socket*** of Python facilitates the programmer to make network connections and to retrieve data over those sockets in a Python program. ***Socket*** is bidirectional data path to a remote system. **A *socket is much like a file, except that a single socket provides a two-way connection* between two programs.** You can both read from and write to the same socket. If you write something to a socket, it is sent to the application at the other end of the socket. If you read from the socket, you are given the data which the other application has sent.

Consider a simple program to retrieve the data from a web page. To understand the program given below, one should know the meaning of terminologies used there.

- **AF_INET** is an address family (IP) that is used to designate the type of addresses that your socket can communicate with. When you create a socket, you have to

specify its address family, and then you can use only addresses of that type with the socket.

- **SOCK_STREAM** is a constant indicating the type of socket (TCP). It works as a file stream and is most reliable over the network.
- **Port** is a logical end-point. Port 80 is one of the most commonly used **port** numbers in the Transmission Control Protocol (TCP) suite.
- The command to retrieve the data must use CRLF(Carriage Return Line Feed) line endings, and it must end in \r\n\r\n (line break in protocol specification).
- ***encode()*** method applied on strings will return bytes-representation of the string. Instead of *encode()* method, one can attach a character *b* at the beginning of the string for the same effect.
- ***decode()*** method returns a string decoded from the given bytes.

A socket connection between the user program and the webpage is shown in Figure 5.1.

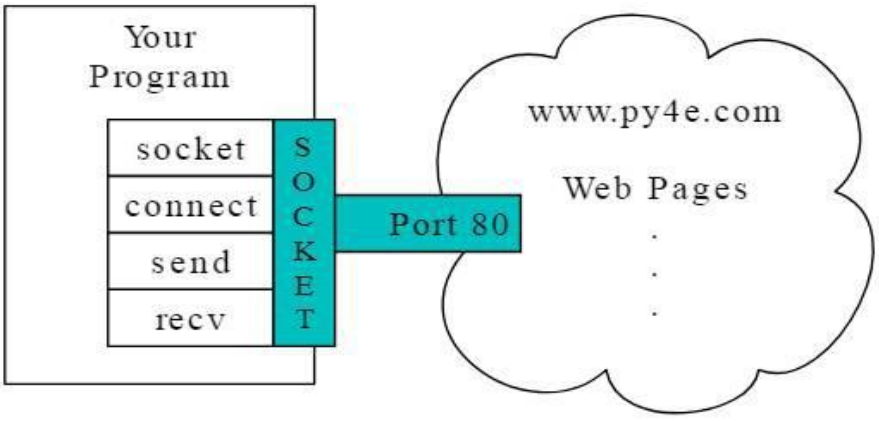


Figure 5.1 A Socket Connection

Now, observe the following program -

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd='GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if (len(data) < 1):
        break

    print(data.decode(),end='')

mysock.close()
```

When we run above program, we will get some information related to web-server of the website which we are trying to scrape. Then, we will get the data written in that web-page. In this program, we are extracting 512 bytes of data at a time. (One can use one's convenient number here). The extracted data is decoded and printed. When the length of data becomes less than one (that is, no more data left out on the web page), the loop is terminated.

5.1.3 Retrieving an Image over HTTP

In the previous section, we retrieved the text data from the webpage. Similar logic can be used to extract images on the webpage using HTTP. In the following program, we extract the image data in the chunks of 5120 bytes at a time, store that data in a string, trim off the headers and then store the image file on the disk.

```
import socket
import time

HOST = 'data.pr4e.org'          #host name
PORT = 80                       #port number

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))

mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')

count = 0
picture = b""                   #empty string in binary format

while True:
    data = mysock.recv(5120)  #retrieve 5120 bytes at a time
    if (len(data) < 1):
        break

    time.sleep(0.25)     #programmer can see data retrieval easily
    count = count + len(data)
    print(len(data), count)    #display cumulative data retrieved
    picture = picture + data

mysock.close()

pos = picture.find(b"\r\n\r\n") #find end of the header (2 CRLF)
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]

fhand = open("stuff.jpg", "wb")    #image is stored as stuff.jpg
```

```
fhand.write(picture)
fhand.close()
```

When we run the above program, the amount of data (in bytes) retrieved from the internet is displayed in a cumulative format. At the end, the image file 'stuff.jpg' will be stored in the current working directory. (One has to verify it by looking at current working directory of the program).

5.1.4 Retrieving Web Pages with urllib

Python provides simpler way of webpage retrieval using the library ***urllib.*** Here, webpage is treated like a file. ***urllib*** handles all of the HTTP protocol and header details. Following is the code equivalent to the program given in Section 5.1.2.

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

for line in fhand:
    print(line.decode().strip())
```

Once the web page has been opened with `urllib.urlopen`, we can treat it like a file and read through it using a for-loop. When the program runs, we only see the output of the contents of the file. The headers are still sent, but the `urllib` code consumes the headers and only returns the data to us.

Following is the program to retrieve the data from the file romeo.txt which is residing at www.data.pr4e.org, and then to count number of words in it.

```
import urllib.request
fhand  =  urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
counts = dict()

for line in fhand:
     words  =  line.decode().split()
     for word in words:
        counts[word] = counts.get(word, 0) + 1

print(counts)
```

5.1.5 Reading Binary Files using urllib

Sometimes you want to retrieve a non-text (or binary) file such as an image or video file. The data in these files is generally not useful to print out, but you can easily make a copy of a URL to a local file on your hard disk using ***urllib***. In Section 5.1.3, we have seen how to retrieve image file from the web using sockets. Now, here is an equivalent program using ***urllib***.

```
import urllib.request
img=urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()
```

Once we execute the above program, we can see a file `cover3.jpg` in the current working directory in our computer.

The program reads all of the data in at once across the network and stores it in the variable ***img*** in the main memory of your computer, then opens the file `cover.jpg` and writes the data out to your disk. This will work if the size of the file is less than the size of the memory (RAM) of your computer. However, if this is a large audio or video file, this program may crash or at least run extremely slowly when your computer runs out of memory. In order to avoid memory overflow, we retrieve the data in blocks (or buffers) and then write each block to your disk before retrieving the next block. This way the program can read any size file without using up all of the memory you have in your computer.

Following is another version of above program, where data is read in chunks and then stored onto the disk.

```
import urllib.request

img=urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0

while True:
    info = img.read(100000)
    if len(info) < 1:
        break
    size = size + len(info)
    fhand.write(info)

print(size, 'characters copied.')
fhand.close()
```

Once we run the above program, an image file `cover3.jpg` will be stored on to the current working directory.

5.1.6 Parsing HTML and Scraping the Web

One of the common uses of the urllib capability in Python is to *scrape the web.* Web scraping is when we write a program that pretends to be a web browser and retrieves pages, then examines the data in those pages looking for patterns. Example: a search engine such as Google will look at the source of one web page and extract the links to other pages and retrieve those pages, extracting links, and so on. Using this technique,

Google *spiders its way through nearly all of the* pages on the web. Google also uses the frequency of links from pages it finds to a particular page as one measure of how "important" a page is and how high the page should appear in its search results.

5.1.7 Parsing HTML using Regular Expressions

Sometimes, we may need to parse the data on the web which matches a particular pattern. For this purpose, we can use regular expressions. Now, we will consider a program that extracts all the hyperlinks given in a particular webpage. To understand the Python program for this purpose, one has to know the pattern of an HTML file. Here is a simple HTML file -

```
<h1>The  First  Page</h1>
<p>
     If you like, you can switch to the
       <a   href="http://www.dr-chuck.com/page2.htm">
     Second Page</a>.
</p>
```

Here,
`<h1>` and `</h1>` are the beginning and end of header tags
`<p>` and `</p>` are the beginning and end of paragraph tags
`<a>` and `</a>` are the beginning and end of anchor tag which is used for giving links
`href` is the attribute for anchor tag which takes the value as the link for another page.

The above information clearly indicates that if we want to extract all the hyperlinks in a webpage, we need a regular expression which matches the `href` attribute. Thus, we can create a regular expression as -

```
href="http://.+?"
```

Here, the question mark in `.+?` indicate that the match should find smallest possible matching string.

Now, consider a Python program that uses the above regular expression to extract all hyperlinks from the webpage given as input.

```
import urllib.request
import re

url = input('Enter - ')  #give URL of any website

html = urllib.request.urlopen(url).read()
links = re.findall(b'href="(http://.*?)"', html)

for link in links:
     print(link.decode())
```

When we run this program, it prompts for user input. We need to give a valid URL of any website. Then all the hyperlinks on that website will be displayed.

5.1.8 Parsing HTML using BeautifulSoup

There are a number of Python libraries which can help you parse HTML and extract data from the pages. Each of the libraries has its strengths and weaknesses and you can pick one based on your needs. ***BeautifulSoup*** *library* is one of the simplest libraries available for parsing. To use this, *download and install the BeautifulSoup code* from:

http://www.crummy.com/software/

Consider the following program which uses `urllib` to read the page and uses BeautifulSoup to extract `href` attribute from the anchor tag.

```
import urllib.request
from bs4 import BeautifulSoup
import ssl                          #Secure Socket Layer

ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url,context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')
tags = soup('a')

for tag in tags:
    print(tag.get('href', None))
```

A sample output would be -

```
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
```

The above program prompts for a web address, then opens the web page, reads the data and passes the data to the BeautifulSoup parser, and then retrieves all of the anchor tags and prints out the href attribute for each tag.

The BeautifulSoup can be used to extract various parts of each tag as shown below -

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

ctx = ssl.create_default_context()
```

```
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')

html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")
tags = soup('a')
for tag in tags:
     print('TAG:', tag)
     print('URL:', tag.get('href', None))
     print('Contents:', tag.contents[0])
     print('Attrs:', tag.attrs)
```

The sample output would be -

```
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Contents:
Second Page
Attrs: {'href': 'http://www.dr-chuck.com/page2.htm'}
```

5.2 USING WEB SERVICES

There are two common formats that are used while exchanging data across the web. One is HTML and the other is XML (eXtensible Markup Language). In the previous section we have seen how to retrieve the data from a web-page which is in the form of HTML. Now, we will discuss the retrieval of data from web-page designed using XML.

XML is best suited for exchanging document-style data. When programs just want to exchange dictionaries, lists, or other internal information with each other, they use JavaScript Object Notation or JSON (refer www.json.org). We will look at both formats.

5.2.1 eXtensible Markup Language (XML)

XML looks very similar to HTML, but XML is more structured than HTML. Here is a sample of an XML document:

```
<person>
     <name>Chuck</name>
     <phone type="intl">
          +1 734 303 4456
     </phone>
     <email hide="yes"/>
</person>
```

Often it is helpful to think of an XML document as a tree structure where there is a top tag `person` and other tags such as `phone` are drawn as children of their parent nodes. Figure 5.2 is the tree structure for above given XML code.

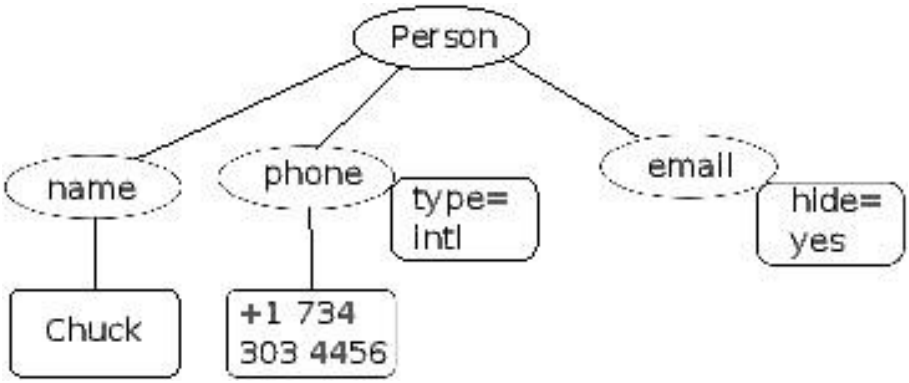


Figure 5.2 Tree Representation of XML

5.2.2 Parsing XML

Python provides library `xml.etree.ElementTree` to parse the data from XML files. One has to provide XML code as a string to built-in method `fromstring()` of `ElementTree` class. `ElementTree` acts as a parser and provides a set of relevant methods to extract the data. Hence, the programmer need not know the rules and the format of XML document syntax. The `fromstring()` method will convert XML code into a tree-structure of XML nodes. When the XML is in a tree format, Python provides several methods to extract data from XML. Consider the following program.

```
import xml.etree.ElementTree as ET

#XML code embedded in a string format
data = '''
<person>
    <name>Chuck</name>
    <phone type="intl">
        +1 734 303 4456
    </phone>
    <email  hide="yes"/>
</person>'''

tree = ET.fromstring(data)
print('Attribute for tag email:', tree.find('email').get('hide'))
print('Attribute for tag phone:', tree.find('phone').get('type'))
```

The output would be -

```
Name: Chuck
Attribute for the tag email: yes
Attribute for the tag phone: intl
```

In the above example, `fromstring()` is used to convert XML code into a tree. The `find()` method searches XML tree and retrieves a node that matches the specified tag. The `get()` method retrieves the value associated with the specified attribute of that tag. Each node can have some text, some attributes (like hide), and some "child" nodes. Each node can be the parent for a tree of nodes.

5.2.3 Looping Through Nodes

Most of the times, XML documents are hierarchical and contain multiple nodes. To process all the nodes, we need to loop through all those nodes. Consider following example as an illustration.

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
    <users>
        <user x="2">
            <id>001</id>
            <name>Chuck</name>
        </user>
        <user x="7">
            <id>009</id>
            <name>Brent</name>
        </user>
    </users>
</stuff>'''

stuff = ET.fromstring(input)
lst  =  stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
     print('Name', item.find('name').text)
     print('Id', item.find('id').text)
     print('Attribute', item.get("x"))
```

The output would be -

```
User count: 2
Name Chuck
Id 001
Attribute  2
Name   Brent
Id 009
Attribute 7
```

The `findall()` method retrieves a Python list of subtrees that represent the `user` structures in the XML tree. Then we can write a for-loop that extracts each of the `user` nodes, and prints the `name` and `id`, which are text elements as well as the attribute `x` from the `user` node.

5.2.4 JavaScript Object Notation (JSON)

The JSON format was inspired by the object and array format used in the JavaScript language. But since Python was invented before JavaScript, Python's syntax for dictionaries and lists influenced the syntax of JSON. So the format of JSON is a combination of Python lists and dictionaries. Following is the JSON encoding that is roughly equivalent to the XML code (the string `data`) given in the program of Section 5.2.2.

```
{
      "name" : "Chuck",
      "phone": {"type" : "intl", "number" : "+1 734 303 4456"},
      "email": {"hide" : "yes"}
}
```

Observe the differences between XML code and JSON code:

- In XML, we can add attributes like "intl" to the "phone" tag. In JSON, we simply have key-value pairs.
- XML uses tag "person", which is replaced by a set of outer curly braces in JSON.

In general, JSON structures are simpler than XML because JSON has fewer capabilities than XML. But JSON has the advantage that it maps *directly* to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python's dictionaries and lists, JSON is a very natural format to have two compatible programs exchange data. JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

5.2.5 Parsing JSON

Python provides a module `json` to parse the data in JSON pages. Consider the following program which uses JSON equivalent of XML string written in Section 5.2.3. Note that, the JSON string has to embed a list of dictionaries.

```
import json

data  =  '''
[
    { "id" : "001",
      "x" : "2",
      "name"  :  "Chuck"
    } ,
    { "id" : "009",
      "x" : "7",
```

```
        "name" : "Chuck"
    }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])
```

The output would be -

```
User count: 2
Name Chuck
Id 001
Attribute  2
Name   Chuck
Id 009
Attribute 7
```

Here, the string `data` contains a list of users, where each user is a key-value pair. The method `loads()` in the `json` module converts the string into a list of dictionaries. Now onwards, we don't need anything from `json`, because the parsed data is available in Python native structures. Using a for-loop, we can iterate through the list of dictionaries and extract every element (in the form of key-value pair) as if it is a dictionary object. That is, we use index operator (a pair of square brackets) to extract value for a particular key.

NOTE: Current IT industry trend is to use JSON for web services rather than XML. Because, JSON is simpler than XML and it directly maps to native data structures we already have in the programming languages. This makes parsing and data extraction simpler compared to XML. But XML is more self descriptive than JSON and so there are some applications where XML retains an advantage. For example, most word processors store documents internally using XML rather than JSON.

5.2.6 Application Programming Interface (API)

Till now, we have discussed how to exchange data between applications using HTTP, XML and JSON. The next step is to understand API. Application Programming Interface defines and documents the contracts between the applications. When we use an API, generally one program makes a set of services available for use by other applications and publishes the APIs (i.e., the "rules") that must be followed to access the services provided by the program.

When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a ***Service-Oriented Architecture***(SOA). A SOA approach is one where our overall application makes use of

the services of other applications. A non-SOA approach is where the application is a single stand-alone application which contains all of the code necessary to implement the application.

Consider an example of SOA: Through a single website, we can book flight tickets and hotels. The data related to hotels is not stored in the airline servers. Instead, airline servers contact the services on hotel servers and retrieve the data from there and present it to the user. When the user agrees to make a hotel reservation using the airline site, the airline site uses another web service on the hotel systems to actually make the reservation. Similarly, to reach airport, we may book a cab through a cab rental service. And when it comes time to charge your credit card for the whole transaction, still other computers become involved in the process. This process is depicted in Figure 5.3.

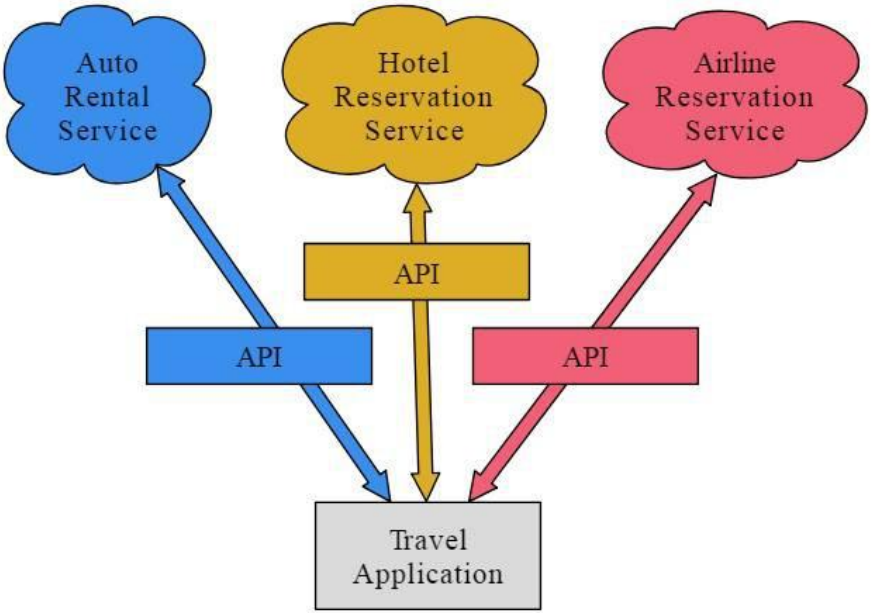


Figure 5.3 Server Oriented Architecture

SOA has following major advantages:

- we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit)
- the owners of the data can set the rules about the use of their data.

With these advantages, an SOA system must be carefully designed to have good performance and meet the user's needs. When an application makes a set of services in its API available over the web, then it is called as ***web services.***

5.2.7 Google Geocoding Web Service

Google has a very good web service which allows anybody to use their large database of geographic information. We can submit a geographic search string like "Rajarajeshwari Nagar" to their geocoding API. Then Google returns the location details of the string submitted.

The following program asks the user to provide the name of a location to be searched for. Then, it will call Google geocoding API and extracts the information from the returned JSON.

```
import urllib.request, urllib.parse, urllib.error
import json

serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'

address = input('Enter location: ')
if len(address) < 1:
    exit()

url = serviceurl + urllib.parse.urlencode({'address': address})
print('Retrieving', url)
uh = urllib.request.urlopen(url)
data = uh.read().decode()
print('Retrieved', len(data), 'characters')

try:
    js = json.loads(data)
except:
    js = None

if not js or 'status' not in js or js['status'] != 'OK':
    print('==== Failure To Retrieve ====')
    print(data)


print(json.dumps(js, indent=4))
lat = js["results"][0]["geometry"]["location"]["lat"]
lng = js["results"][0]["geometry"]["location"]["lng"]
print('lat', lat, 'lng', lng)
location = js['results'][0]['formatted_address']
print(location)
```

(Students are advised to run the above program and check the output, which will contain several lines of Google geographical data).

The above program retrieves the search string and then encodes it. This encoded string along with Google API link is treated as a URL to fetch the data from the internet. The data retrieved from the internet will be now passed to JSON to put it in JSON object format. If the input string (which must be an existing geographical location like Channasandra, Malleshwaram etc!!) cannot be located by Google API either due to bad internet or due to unknown location, we just display the message as 'Failure to Retrieve'. If Google successfully identifies the location, then we will dump that data in JSON object. Then, using

indexing on JSON (as JSON will be in the form of dictionary), we can retrieve the location address, longitude, latitude etc.

5.2.8 Security and API Usage

Public APIs can be used by anyone without any problem. But, if the API is set up by some private vendor, then one must have ***API key*** to use that API. If API key is available, then it can be included as a part of POST method or as a parameter on the URL while calling API.

Sometimes, vendor wants more security and expects the user to provide cryptographically signed messages using shared keys and secrets. The most common protocol used in the internet for signing requests is ***OAuth.***

As the Twitter API became increasingly valuable, Twitter went from an open and public API to an API that required the use of OAuth signatures on each API request. But, there are still a number of convenient and free OAuth libraries so you can avoid writing an OAuth implementation from scratch by reading the specification. These libraries are of varying complexity and have varying degrees of richness. The OAuth web site has information about various OAuth libraries.

5.3 USING DATABASES AND SQL

A structured set of data stored in a permanent storage is called as ***database***. Most of the databases are organized like a dictionary - that is, they map keys to values. Unlike dictionaries, databases can store huge set of data as they reside on permanent storage like hard disk of the computer.

There are many database management softwares like Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite etc. They are designed to insert and retrieve data very fast, however big the dataset is. Database software builds *indexes* as data is added to the database so as to provider quicker access to particular entry.

In this course of study, SQLite is used because it is already built into Python. SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a non-standard variant of the SQL query language. SQLite is designed to be *embedded* into other applications to provide database support within the application. For example, the Firefox browser also uses the SQLite database internally. SQLite is well suited to some of the data manipulation problems in Informatics such as the Twitter spidering application etc.

5.3.1 Database Concepts

For the first look, database seems to be a spreadsheet consisting of multiple sheets. The primary data structures in a database are ***tables, rows*** and ***columns.*** In a relational database terminology, tables, rows and columns are referred as ***relation, tuple*** and ***attribute*** respectively. Typical structure of a database table is as shown below. Each table may consist of n number of attributes and m number of tuples (or records). Every tuple gives the information about one individual. Every cell(i, j) in the table indicates value of j^{th} attribute for i^{th} tuple.

	Attribute1	Attribute2	……	Attribute_n
Tuple1	V11	V12	……	V1n
Tuple2	V21	V22	……	V2n
……..	……	…….	……	…….
……	……	…….	……..	…….
Tuple_m	Vm1	Vm2	……..	Vmn

Consider the problem of storing details of students in a database table. The format may look like -

	RollNo	Name	DoB	Marks
Student1	1	Ram	22/10/2001	82.5
Student2	2	Shyam	20/12/2000	81.3
……..	……	…….	……	…….
……	……	…….	……..	…….
Student_m	……..	…….	……	……

Thus, table columns indicate the type of information to be stored, and table rows gives record pertaining to every student. We can create one more table say *addressTable* consisting of attributes like DoorNo, StreetName, Locality, City, PinCode. To relate this table with a respective student stored in *studentTable,* we need to store RollNo also in *addressTable* (Note that, RollNo will be unique for every student, and hence there won't be any confusion). Thus, there is a relationship between two tables in a single database. There are softwares that can maintain proper relationships between multiple tables in a single database and are known as Relational Database Management Systems (RDBMS). The detailed discussion on RDBMS is out of the scope of this study.

5.3.2 Structured Query Language (SQL) Summary

To perform operations on databases, one should use structured query language. SQL is a standard language for storing, manipulating and retrieving data in databases. Irrespective of RDBMS software (like Oracle, MySQL, MS Access, SQLite etc) being used, the syntax of SQL remains the same. The usage of SQL commands may vary from one RDBMS to the other and there may be little syntactical difference. Also, when we are using some programming language like Python as a front-end to perform database applications, the way we embed SQL commands inside the program source-code is as per the syntax of respective programming language. Still, the underlying SQL commands remain the same. Hence, it is essential to understand basic commands of SQL.

There are some *clauses* like FROM, WHERE, ORDER BY, INNER JOIN etc. that are used with SQL commands, which we will study in a due course. The following table gives few of the SQL commands.

Command	Meaning
CREATE DATABASE	creates a new database
ALTER DATABASE	modifies a database
CREATE TABLE	creates a new table
ALTER TABLE	modifies a table
DROP TABLE	deletes a table
SELECT	extracts data from a database
INSERT INTO	inserts new data into a database
UPDATE	updates data in a database
DELETE	deletes data from a database

As mentioned earlier, every RDBMS has its own way of storing the data in tables. Each of RDBMS uses its own set of data types for the attribute values to be used. SQLite uses the data types as mentioned in the following table -

Data Type	Description
NULL	The value is a NULL value.
INTEGER	The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
REAL	The value is a floating point value, stored as an 8-byte floating point number
TEXT	The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE)
BLOB	The value is a blob (Binary Large Object) of data, stored exactly as it was input

Note that, SQL commands are case-insensitive. But, it is a common practice to write commands and clauses in uppercase alphabets just to differentiate them from table name and attribute names.

Now, let us see some of the examples to understand the usage of SQL statements -

- `CREATE TABLE Tracks (title TEXT, plays INTEGER)`

This command creates a table called as `Tracks` with the attributes `title` and `plays` where `title` can store data of type `TEXT` and `plays` can store data of type `INTEGER`.

- `INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)`
  This command inserts one record into the table `Tracks` where values for the attributes `title` and `plays` are 'My Way' and 15 respectively.

- `SELECT * FROM Tracks`
  Retrieves all the records from the table `Tracks`

- `SELECT * FROM Tracks WHERE title = 'My Way'`
  Retrieves the records from the table `Tracks` having the value of attribute `title` as 'My Way'

- `SELECT title, plays FROM Tracks ORDER BY title`
  The values of attributes `title` and `plays` are retrieved from the table `Tracks` with the records ordered in ascending order of `title`.

- `UPDATE Tracks SET plays = 16 WHERE title = 'My Way'`
  Whenever we would like to modify the value of any particular attribute in the table, we can use UPDATE command. Here, the value of attribute `plays` is assigned to a new value for the record having value of `title` as 'My Way'.

- `DELETE FROM Tracks WHERE title = 'My Way'`
  A particular record can be deleted from the table using DELETE command. Here, the record with value of attribute `title` as 'My Way' is deleted from the table `Tracks`.

5.3.3 Database Browser for SQLite

Many of the operations on SQLite database files can be easily done with the help of software called *Database Browser for SQLite* which is freely available from:

http://sqlitebrowser.org/

Using this browser, one can easily create tables, insert data, edit data, or run simple SQL queries on the data in the database. This database browser is similar to a text editor when working with text files. When you want to do one or very few operations on a text file, you can just open it in a text editor and make the changes you want. When you have many changes that you need to do to a text file, often you will write a simple Python program. You will find the same pattern when working with databases. You will do simple operations in the database manager and more complex operations will be most conveniently done in Python.

5.3.4 Creating a Database Table

When we try to create a database table, we must specify the names of table columns and the type of data to be stored in those columns. When the database software knows the type of data in each column, it can choose the most efficient way to store and look up the data based on the type of data. Here is the simple code to create a database file and a table named Tracks with two columns in the database:

Ex1.

```
import sqlite3
conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
conn.close()
```

The `connect()` method of `sqlite3` makes a "connection" to the database stored in the file `music.sqlite3` in the current directory. If the file does not exist, it will be created. Sometimes, the database is stored on a different database server from the server on which we are running our program. But, all the examples that we consider here will be local file in the current working directory of Python code.

A `cursor()` is like a file handle that we can use to perform operations on the data stored in the database. Calling `cursor()` is very similar conceptually to calling `open()` when dealing with text files. Hence, once we get a cursor, we can execute the commands on the contents of database using `execute()` method.

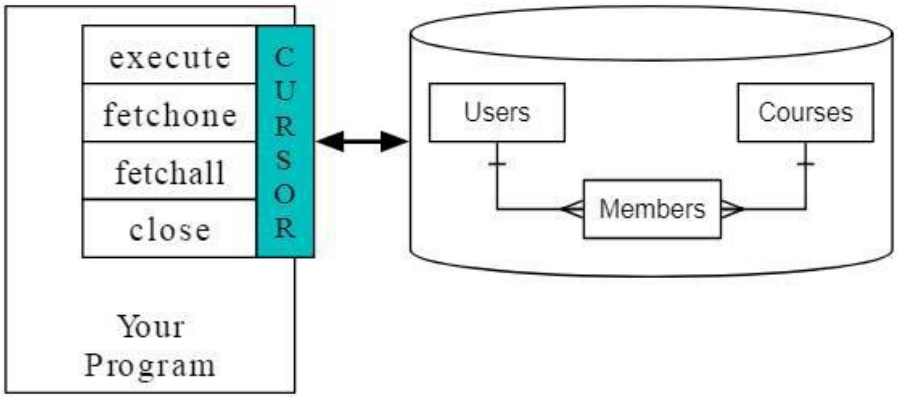


Figure 5.4 A Database Cursor

In the above program, we are trying to remove the database table `Tracks`, if at all it existed in the current working directory. The DROP TABLE command deletes the table along with all its columns and rows. This procedure will help to avoid a possible error of trying to create a table with same name. Then, we are creating a table with name `Tracks` which has two columns viz. `title`, which can take `TEXT` type data and `plays`, which can

take `INTEGER` type data. Once our job with the database is over, we need to close the connection using `close()` method.

In the previous example, we have just created a table, but not inserted any records into it. So, consider below given program, which will create a table and then inserts two rows and finally delete records based on some condition.

Ex2.

```
import sqlite3

conn  =  sqlite3.connect('music.sqlite')
cur = conn.cursor()
cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

cur.execute("INSERT INTO Tracks (title, plays) VALUES
                                    ('Thunderstruck', 20)")
cur.execute("INSERT INTO Tracks (title, plays) VALUES (?, ?)",
                                           ('My Way', 15))
conn.commit()

print('Tracks:')
cur.execute('SELECT  title,  plays  FROM  Tracks')
for row in cur:
     print(row)

cur.execute('DELETE  FROM  Tracks  WHERE  plays  <  100')
cur.close()
```

In the above program, we are inserting first record with the SQL command -

```
"INSERT INTO Tracks (title, plays) VALUES('Thunderstruck', 20)"
```

Note that, execute() requires SQL command to be in string format. But, if the value to be store in the table is also a string (TEXT type), then there may be a conflict of string representation using quotes. Hence, in this example, the entire SQL is mentioned within double-quotes and the value to be inserted in single quotes. If we would like to use either single quote or double quote everywhere, then we need to use escape-sequences like `\'` or `\"`.

While inserting second row in a table, SQL statement is used with a little different syntax -

```
"INSERT INTO Tracks (title, plays) VALUES (?, ?)",('My Way', 15)
```

Here, the question mark acts as a place-holder for particular value. This type of syntax is useful when we would like to pass user-input values into database table.

After inserting two rows, we must use `commit()` method to store the inserted records permanently on the database table. If this method is not applied, then the insertion (or any other statement execution) will be temporary and will affect only the current run of the program.

Later, we use SELECT command to retrieve the data from the table and then use for-loop to display all records. When data is retrieved from database using SELECT command, the cursor object gets those data as a list of records. Hence, we can use for-loop on the cursor object. Finally, we have used a DELETE command to delete all the records WHERE `plays` is less than 100.

Let us consider few more examples -

Ex3.

```
import sqlite3
from sqlite3 import Error

def create_connection():
    """ create a database connection to a database that resides
        in the memory
    """
    try:
        conn = sqlite3.connect(':memory:')
        print("SQLite  Version:",sqlite3.version)
    except Error as e:
        print(e)
    finally:
        conn.close()

create_connection()
```

Few points about above program:

- Whenever we try to establish a connection with database, there is a possibility of error due to non-existing database, authentication issues etc. So, it is always better to put the code for connection inside try-except block.
- While developing real time projects, we may need to create database connection and close it every now-and-then. Instead of writing the code for it repeatedly, it is better to write a separate function for establishing connection and call that function whenever and wherever required.
- If we give the term `:memory:` as an argument to `connect()` method, then the further operations (like table creation, insertion into tables etc) will be on memory (RAM) of the computer, but not on the hard disk.

Ex4. Write a program to create a Student database with a table consisting of student name and age. Read n records from the user and insert them into database. Write queries to display all records and to display the students whose age is 20.

```
import sqlite3
conn=sqlite3.connect('StudentDB.db')
c=conn.cursor()
c.execute('CREATE TABLE tblStudent(name text, age Integer)')

n=int(input("Enter number of records:"))
for i in range(n):
    nm=input("Enter Name:")
    ag=int(input("Enter age:"))
    c.execute("INSERT INTO tblStudent VALUES(?,?)",(nm,ag))

conn.commit()
c.execute("select * from tblStudent ")
print(c.fetchall())

c.execute("select * from tblStudent where age=20")
print(c.fetchall())

conn.close()
```

In the above program we take a for-loop to get user-input for student's name and age. These data are inserted into the table. Observe the question mark acting as a placeholder for user-input variables. Later we use a method fetchall() that is used to display all the records form the table in the form of a list of tuples. Here, each tuple is one record from the table.

5.3.5 Three Kinds of Keys

Sometimes, we need to build a data model by putting our data into multiple linked tables and linking the rows of those tables using some ***keys***. There are three types of keys used in database model:

- A ***logical key*** is a key that the "real world" might use to look up a row. It defines the relationship between primary keys and foreign keys. Most of the times, a UNIQUE constraint is added to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.
- A ***primary key*** is usually a number that is assigned automatically by the database. It generally has no meaning outside the program and is only used to link rows from different tables together. When we want to look up a row in a table, usually searching for the row using the primary key is the fastest way to find the row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly.

- A ***foreign key*** is usually a number that points to the primary key of an associated row in a different table.

Consider a table consisting of student details like RollNo, name, age, semester and address as shown below -

RollNo	Name	Age	Sem	Address
1	Ram	29	6	Bangalore
2	Shyam	21	8	Mysore
3	Vanita	19	4	Sirsi
4	Kriti	20	6	Tumkur

In this table, RollNo can be considered as a primary key because it is unique for every student in that table. Consider another table that is used for storing marks of students in all the three tests as below -

RollNo	Sem	M1	M2	M3
1	6	34	45	42.5
2	6	42.3	44	25
3	4	38	44	41.5
4	6	39.4	43	40
2	8	37	42	41

To save the memory, this table can have just RollNo and marks in all the tests. There is no need to store the information like name, age etc of the students as these information can be retrieved from first table. Now, RollNo is treated as a foreign key in the second table.

5.3.6 Basic Data Modeling

The relational database management system (RDBMS) has the power of linking multiple tables. The act of deciding how to break up your application data into multiple tables and establishing the relationships between the tables is called ***data modeling***. The design document that shows the tables and their relationships is called a ***data model***. Data modeling is a relatively sophisticated skill. The data modeling is based on the concept of ***database normalization*** which has certain set of rules. In a raw-sense, we can mention one of the basic rules as never put the same string data in the database more than once. If we need the data more than once, we create a numeric ***key*** (primary key) for the data and reference the actual data using this key. This is because string requires more space on the disk compared to integer, and data retrieval (by comparing) using strings is difficult compared to that with integer.

Consider the example of Student database discussed in Section 5.3.5. We can create a table using following SQL command -

```
CREATE TABLE tblStudent
    (RollNo INTEGER PRIMARY KEY, Name TEXT, age INTEGER, sem INTEGER, address TEXT)
```

Here, RollNo is a primary key and by default it will be unique in one table. Now, another take can be created as -

```
CREATE TABLE tblMarks
        (RollNo INTEGER, sem INTEGER, m1 REAL, m2 REAL, m3 REAL, UNIQUE(RollNo,sem))
```

Now, in the tblMarks consisting of marks of 3 tests of all the students, RollNo and sem are together unique. Because, in one semester, only one student can be there having a particular RollNo. Whereas in another semester, same RollNo may be there.

Such types of relationships are established between various tables in RDBMS and that will help better management of time and space.

5.3.7 Using JOIN to Retrieve Data

When we follow the rules of database normalization and have data separated into multiple tables, linked together using primary and foreign keys, we need to be able to build a SELECT that reassembles the data across the tables. SQL uses the JOIN clause to reconnect these tables. In the JOIN clause you specify the fields that are used to reconnect the rows between the tables.

Consider the following program which creates two tables tblStudent and tblMarks as discussed in the previous section. Few records are inserted into both the tables. Then we extract the marks of students who are studying in 6th semester.

```
import sqlite3

conn=sqlite3.connect('StudentDB.db')
c=conn.cursor()

c.execute('CREATE TABLE tblStudent
     (RollNo INTEGER PRIMARY KEY, Name TEXT, age INTEGER, sem INTEGER,
     address TEXT)')

c.execute('CREATE TABLE tblMarks
     (RollNo INTEGER, sem INTEGER, m1 REAL, m2 REAL, m3 REAL,
     UNIQUE(RollNo,sem))')

c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (1,'Ram',20,6,'Bangalore'))
c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (2,'Shyam',21,8,'Mysore'))
c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (3,'Vanita',19,4,'Sirsi'))
c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (4,'Kriti',20,6,'Tumkur'))
```

```
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(1,6,34,45,42.5))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(2,6,42.3,44,25))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(3,4,38,44,41.5))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(4,6,39.4,43,40))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(2,8,37,42,41))

conn.commit()

query="SELECT tblStudent.RollNo, tblStudent.Name, tblMarks.sem,
     tblMarks.m1, tblMarks.m2, tblMarks.m3
     FROM tblStudent JOIN tblMarks ON
     tblStudent.sem = tblMarks.sem AND
     tblStudent.RollNo = tblMarks.RollNo
     WHERE tblStudent.sem=6"

c.execute(query)
for row in c:
    print(row)

conn.close()
```

The output would be -

```
     (1, 'Ram', 6, 34.0, 45.0, 42.5)
     (4, 'Kriti', 6, 39.4, 43.0, 40.0)
```

The query joins two tables and extracts the records where RollNo and sem matches in both the tables, and sem must be 6.