Python Programming Simplified

An Absolute Beginners Guide

VIKAS THADA

2024

RSYN RESEARCH

India

RSYN RESEARCH

TEXTBOOK

RSYN RESEARCH LLP, Indore, India
Python Programming Simplified: An Absolute Beginners Guide
ISBN: 978-81-979716-7-9
Author: Vikas Thada

Second Edition, 2024

Digital download and online
books.rsyn.org
rsynresearch.com/books/

Published in India
by RSYN RESEARCH LLP, Indore, India
ww.rsyn.org

978-81-979716-7-9

Preface

Completing a book is an arduous task besides fun. It is like a long journey with sleepless nights. During this journey of the book which is completed now many persons have helped me directly or indirectly. First & foremost I would like to thank Almighty for giving the mental strength and potential for writing the book. I'm indebted to various authors whose books I've consulted to prepare this book, though a large portion of the text is from my own experience teaching this subject number of times. Special thanks go to google search engine for providing fast and accurate searching on various keywords. I thank all my friends ,colleague , HOD, Director and all higher authorities at Amity University MadhyaPradesh

I also thank my mother, my wife without their support writing this book was not possible. In the end I'd like to thank my sweet daughter 'Parishi' and my son 'Chinmay' whose one smile takes all my fatigue away and rejuvenate me after a tiresome work.

I also thank my lots of student whose conceptual queries and ideas helped me in strengthening the concepts and problems of data structures.

Organization of the book: *The book has been divided into 14 chapters. Each chapter has been written with clear understanding of the subject matter and with numerous examples to try practically. The selection of 14 chapters have been done so that it gives you knowledge of all essentials topics related to Python programming lectures.*

I hope to take out a second version of the book covering some advanced topics that have not become part of this book.

Though every program/script has been run before making it part of the book, but for any error if you find in the book, feel free to brought to my notice. Except any error, I welcome any suggestion if you feel to improve the book in the coming versions.

You can contact me at: vikasthada@rediffmail.com

Table of Contents

1. Starting With Python

1.1 PYTHON OVERVIEW **1**

1.2 FEATURES OF PYTHON **1**

1.3 INSTALLING ANACONDA **1**

1.4 OTHER OPTIONS **10**

1.5 WORKING WITH JUPYTER NOTEBOOK **10**

1.6 PYTHON SHELL **16**

1.7 WRITING FIRST PYTHON SCRIPT **22**

1.8 PYTHON CHARACTER SET **25**

1.9 PYTHON TOKENS **26**

1.9.1 KEYWORDS 27
1.9.2 IDENTIFIER 27
1.9.3 LITERALS 28
1.9.4 STRINGS 36
1.9.5 OPERATORS 36
1.9.6 SPECIAL SYMBOLS 36

1.10 VARIABLES **36**

1.11 DATA TYPES **37**

1.11.1 NUMBERS **38**

1.11.2 LIST 39

1.11.3 STRINGS **40**

1.11.4 SET 41
1.11.5 DICTIONARY 42

1.12 INDENTATION IN PYTHON **42**

1.13 PYTHON COMMENTS **43**

1.14 PYTHON STATEMENTS **44**

1.14.1 MULTILINE STATEMENTS 45

1.15 THE PRINT FUNCTION **46**

1.16 READING INPUT FROM USER **49**

1.17 TYPE CONVERSION **51**

1.18 SCRIPTING EXAMPLES **53**

1.19 WHY LEARN PYTHON? **55**

1.20 COMPANIES USING PYTHON **55**

1.21 POINTS TO PONDER **56**

2. Operators & Expressions

2.1 INTRODUCTION **57**

2.1.1 BINARY OPERATORS 57
2.1.2 UNARY OPERATORS 57

2.2 EXPRESSIONS 57

2.3 ARITHMETIC OPERATORS 58

2.3.1 THE STRING OPERATORS (+ AND *) 60
2.3.2 SCRIPTING EXAMPLES 61

2.4 RELATIONAL OPERATOR 64

2.5 LOGICAL OPERATORS 66

2.5.1 LOGICAL AND 66
2.5.2 LOGICAL OR 67
2.5.3 LOGICAL NOT (!) 68
2.5.4 SHORT CIRCUIT OPERATORS 68

2.6 ASSIGNMENT OPERATOR 69

2.7 BITWISE OPERATORS 71

2.7.1 BITWISE AND (&) 71
2.7.2 BITWISE OR (|) 73
2.7.3. BITWISE XOR (^) 74
2.7.4. 1'S COMPLEMENT (~) 76
2.7.5. LEFT SHIFT OPERATOR (<<) 77
2.7.6. RIGHT SHIFT OPERATOR (>>) 78

2.8 MEMBERSHIP OPERATOR 81

2.9 IDENTITY OPERATOR 82

2.10 PRECEDENCE OF OPERATORS 84

2.11 RVALUE AND LVALUE 87

2.12 THE MATH MODULE 88

2.13 POINTS TO PONDER 91

3. Decision Making

3.1 INTRODUCTION 93

3.2 THE IF STATEMENT 93
3.2.1 SHORT HAND IF 95

3.3 THE IF-ELSE STATEMENT 95

3.3.1 SHORT HAND IF-ELSE 99

3.4 NESTING OF IF-ELSE'S 100

3.5 ELSE-IF LADDER 103

3.6 PONDERABLE POINTS 110

4. Looping

4.1 INTRODUCTION 111

4.2 THE WHILE LOOP 111
4.2.1 NESTING OF WHILE LOOP 121

4.3 BREAK STATEMENT 122

4.4 THE CONTINUE STATEMENT **124**
4.5 THE *FOR* LOOP **125**
4.5.1 THE RANGE FUNCTION 125
4.6 NESTING OF FOR LOOP **131**
4.7 THE PASS STATEMENT **137**
4.8 PONDERABLE POINTS **138**

5. Functions

5.1 INTRODUCTION **139**
5.2 THE FUNCTION SYNTAX **139**
5.3 EXAMPLES OF FUNCTION **140**
5.4 ILLUSTRATIVE EXAMPLES **141**
5.5 PASSING PARAMETERS TO FUNCTIONS **143**
5.6 FUNCTION WITH PARAMETERS AND RETURN TYPE **149**
5.7 THE DEFAULT RETURN TYPE **153**
5.8 FUNCTION WITH DEFAULT ARGUMENTS **154**
5.9 CALL BY NAME **158**
5.10 RETURNING MORE THAN ONE VALUE **159**
5.11 GLOBAL VARIABLES IN FUNCTIONS **160**
5.12 PASSING FUNCTION AS ARGUMENT **164**
5.13 PASSING VARIABLE ARGUMENTS TO FUNCTIONS **166**
5.13.1 THE KEYWORD ARGUMENTS (**KWARGS) 168
5.14 RECURSION **170**
5.15 THE TOWER OF HANOI PUZZLE. **175**
5.15.1 SOLUTION AS RECURSIVE ALGORITHM 175
5.16 THE LAMBDA FUNCTION **177**
5.16.1 THE MAP FUNCTION 178
5.16.2 THE FILTER FUNCTION 180
5.16.3 THE REDUCE FUNCTION 180
5.17 PONDERABLE POINTS **181**

6. Strings

6.1 WHAT IS PYTHON STRING ? **183**
6.2 CREATING STRINGS **183**
6.3 ACCESSING STRING **184**
6.4 STRING SLICING **185**
6.5 STRING OPERATORS **186**
6.6 STRING TRAVERSAL **187**
6.7 STRING IS IMMUTABLE **190**

6.8 STRING COMPARISON **191**
6.9 METHODS OF STRING CLASS **192**
6.10 THE FORMAT METHOD **201**
6.10.1 POSITIONAL ARGUMENTS IN FORMAT FUNCTION 201
6.10.2 NUMBER FORMATTING 202
6.10.3 NUMBER PADDING 203
6.10.4 STRING FORMATTING 205
6.10.5 KEYWORD ARGUMENTS 207
6.11 PONDERABLE POINTS **207**

7. List

7.1 INTRODUCTION **208**
7.2 CREATION OF LIST 208
7.3 ACCESSING ELEMENTS FROM LIST **208**
7.4 LIST SLICING **208**
7.5 MODIFYING LIST **210**
7.5.1 UPDATING LIST 210
7.5.2 ADDING ELEMENTS TO LIST 210
7.5.3 REMOVING ELEMENTS FROM LIST 212
7.6 OPERATIONS ON LIST **213**
7.6.1 OPERATOR + WITH LIST 213
7.6.2 OPERATOR * WITH LIST 214
7.6.3 THE MEMBERSHIP OPERATOR ON LIST 214
7.7 LIST TRAVERSAL **214**
7.7.1 LIST TRAVERSAL USING WHILE LOOP 215
7.7.2 TRAVERSING LIST USING FOR LOOP 216
7.8 OTHER LIST METHODS **218**
7.8.1 THE COPY METHOD **218**
7.8.2 THE SORT METHOD 219
7.8.3 THE CLEAR AND COUNT METHOD 220
7.9 GENERAL METHODS APPLIED ON LIST **220**
7.9.1 THE MAX,MIN,SUM FUNCTIONS 220
7.9.2 THE ALL AND ANY FUNCTION 221
7.9.3 THE SORTED METHOD 221
7.9.4 THE ENUMERATE METHOD 221
7.10 LIST INPUT **222**
7.11 LIST AND FUNCTIONS **225**
7.12 LIST COMPREHENSION **227**
7.13 PONDERABLE POINTS **232**

8. Dictionary

8.1 INTRODUCTION **233**
8.2 CREATING DICTIONARIES **233**

8.2.1 RESTRICTION ON KEYS 234
8.3 ACCESSING ELEMENTS 235
8.4 ADDING AND MODIFYING ELEMENTS IN DICTIONARY 236
8.5 REMOVING ELEMENTS FROM DICTIONARY 236
8.6 TRAVERSING DICTIONARIES 238
8.6 METHODS OF DICTIONARY CLASS 239
8.8 MEMBERSHIP TESTING IN DICTIONARY 242
8.9 SCRIPTING EXAMPLES 242
8.10 DICTIONARY COMPREHENSION 244
8.11 PONDERABLE POINTS 245

9. Tuple

9.1 INTRODUCTION 246
9.2 CREATING TUPLES 246
9.3 ACCESSING TUPLE ELEMENTS 247
9.4 MODIFYING TUPLE ELEMENTS 248
9.5 DELETING TUPLE ELEMENTS 248
9.6 OPERATIONS ON TUPLE 249
9.6.1 TUPLE CONCATENATION 249
9.6.2 TUPLE REPETITION 249
9.6.3 TUPLE MEMBERSHIP 249
9.6.4 TUPLE COMPARISON 250
9.7 PONDERABLE POINTS 252

10. Modules in Python

10.1 INTRODUCTION 253
10.2 THE FIRST PYTHON MODULE 253
10.3 RELOADING THE MODULE 255
10.4 IMPORTING IN ANOTHER SCRIPT 256
10.5 UNDERSTANDING IMPORT 256
10.6 THE SEARCH PATH FOR MODULE 257
10.6.1THE PYTHONPATH VARIABLE 258
10.6.2 ADDING PATH TO SYS.PATH 258
10.7 THE DEFAULT MODULE 259
10.8 THE MAIN PROGRAM 260
10.9 PONDERABLE POINTS 261

11. Classes & Objects

11.1 INTRODUCTION 262

11.2 ADDING MEMBERS **263**
11.3 INITIALIZING OBJECT **264**
11.3.1 DEFAULT CONSTRUCTOR 265
11.4 PASSING AND RETURNING OBJECTS TO FUNCTIONS **267**
11.5 ARRAY OF OBJECTS **270**
11.6 STATIC MEMBERS IN CLASS **272**
11.7 STATIC METHODS **274**
11.8 CLASS METHOD **276**
11.9 PONDERABLE POINTS **278**

12. Inheritance

12.1 INTRODUCTION **279**
12. 2 TYPES OF INHERITANCE **279**
12.2.1 SINGLE LEVEL INHERITANCE 280
12.2.2 MULTILEVEL INHERITANCE 280
12.2.3 MULTIPLE INHERITANCE 281
12.2.4 HIERARCHICAL INHERITANCE 281
12.2.5 HYBRID INHERITANCE 282
12.3 SINGLE AND MULTILEVEL INHERITANCE IN PYTHON **283**
12.4 MULTILEVEL INHERITANCE **286**
12.5 MULTIPLE INHERITANCE **287**
12.6 HIERARCHICAL INHERITANCE **291**
12.7 METHOD OVERRIDING **293**
12.8 THE SUPER( ) METHOD **294**
12.9 CONSTRUCTOR (INITIALIZER) & INHERITANCE **295**
12.10 ABSTRACT BASE CLASS **299**
12.11 VISIBILITY MODIFIERS IN PYTHON **305**
12.12 FINAL CLASS **306**
12.13 THE DIAMOND PROBLEM **306**
12.14 COMPOSITION OR CONTAINERSHIP **310**
12.15 PONDERABLE POINTS **311**

13. Exception Handling

13.1 INTRODUCTION **312**
13.2 BASIS FOR EXCEPTION HANDLING **312**
13.3 EXCEPTION HIERARCHY **313**
13.4 SOME EXAMPLES OF EXCEPTIONS **314**
13.5 EXCEPTION HANDLING MECHANISM **315**
13.6 PYTHON STACK TRACE **316**

13.7 EXCEPTION HANDLING USING TRY AND EXCEPT 317

13.7.1 TRY-EXCEPT WITH MULTIPLE EXCEPTIONS 320
13.7.2 CATCHING EXCEPTIONS WITH EMPTY EXCEPT 321
13.7.3 CATCHING ALL EXCEPTIONS USING EXCEPTION CLASS 322
13.7.4 RAISING EXCEPTION 323
13.7.5 NESTING OF TRY EXCEPT BLOCK 324

13.8 THE FINALLY BLOCK 325

13.9 CREATING YOUR OWN EXCEPTIONS 327

13.10 PONDERABLE POINTS 329

14. File Handling

14.1 INTRODUCTION 330

14.2 FILE OPENING, READING AND CLOSING 330

14.3 FILE OPENING MODES 332

14.4 READING FROM FILE 333

14.4.1 THE READ FUNCTION 333
14.4.2 THE FUNCTION READLINE AND READLINES 334

14.5 WRITING TO FILE 335

14.5.1 READING AND WRITING 336
14.5.2 APPENDING DATA TO FILE 337

14.6 WORKING WITH MULTIPLE FILES 337

14.7 RANDOM ACCESS IN FILE 339

14.8 WORKING WITH NUMBERS 340

14.9 WORKING WITH BINARY MODE 342

14.10 FILES AND OBJECTS 344

14.10.1 PICKING OBJECTS 344
14.10.2 THE DILL MODULE 346

14.10 PONDERABLE POINTS 348

1. Starting with Python

1.1 Python Overview

Python is a general purpose and widely used high-level programming language created by Guido van Rossum in 1991.Python is an interpreted language that has features of procedural, object oriented and functional programming language. Python has a rich set of libraries, so it can be used for solving almost any type of programming problem related to fields like general purpose programming, developing websites, data science, robotics, game creation, security and cryptography, deep learning etc.

Contrary to its name Python is not scary language! Instead, the name was adopted from a British comedy series "Monty Python's Flying Circus".

Python interpreters are available for almost all types of operating systems like Windows, Linux, Mac. Some of the major companies that uses Python are Google, Microsoft, Yahoo, IBM, Dropbox, Mozila etc.

1.2 Features of Python

The features of Python are so many, and all these will be clearly visible as you spend your time with Python. Some of the important features are mentioned here:

- ✓ Python offers automatic memory management.
- ✓ Python is Written in C.
- ✓ Python is a dynamic interpreted language which has many strong features of various languages.
- ✓ It is an Object-oriented programming language.
- ✓ Python is open source.
- ✓ Python can be embedded into Hypertext Markup Language (HTML).
- ✓ Python has similar syntax to that of many programming languages such as C and Perl.
- ✓ Python is easy to learn because of simple English like syntax.
- ✓ It has features of Procedural, functional and object-oriented programming language.

1.3 Installing Anaconda

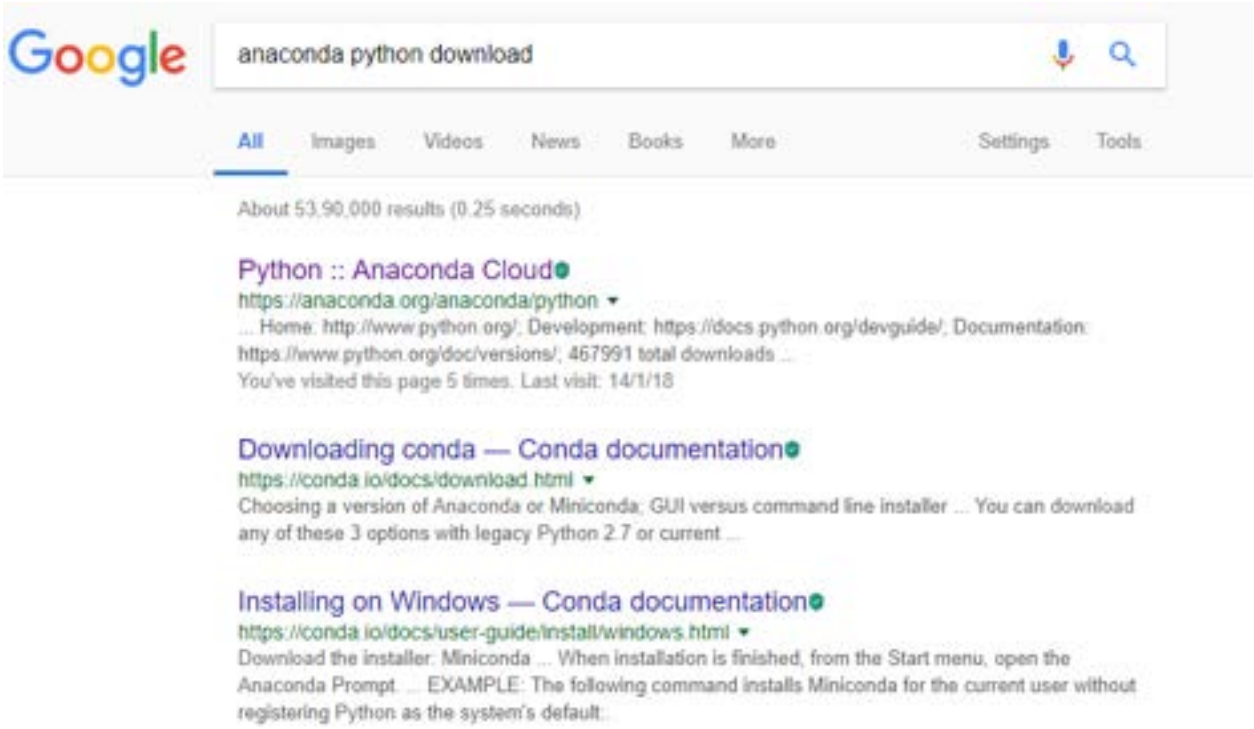
Python Anaconda Framework is one single solution for performing all your python related tasks. It bundles together number of other application programs / modules which are required for applications like graphics programming, machine learning, deep learning, game programming, robotics, internet of

things etc. The two main applications in which we will be interested and using in this book are: **Ipython and Jupyter Notebook**.

The main reason for using anaconda framework is that it is now become standard for python related tasks specially for data science, machine learning and deep learning. Second everything comes bundled with Anaconda. You do not need to install separately most of the modules. Even if you need it just need one or two commands with Internet connection.

In this section we see how we can get Anaconda from web and install on our windows system.

1. Type “anaconda python download” in Google search and right click “Open in new tab” the first result.

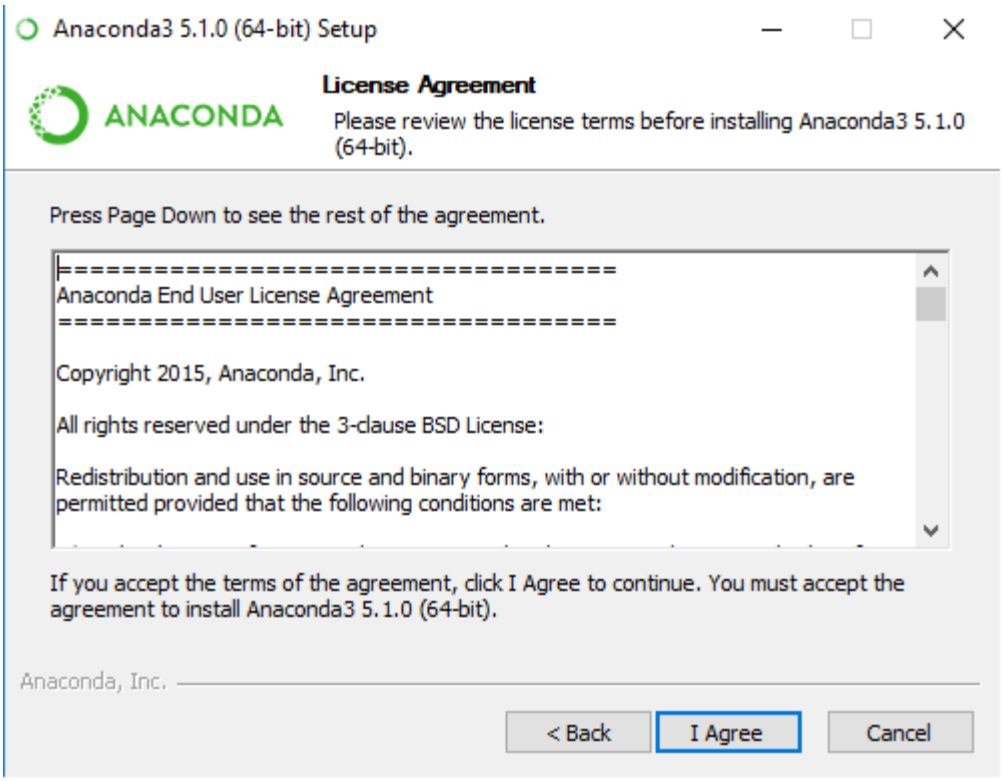


2. The page that is opened now is shown in the next figure. Now click on “Download Anaconda” shown in red rectangle.

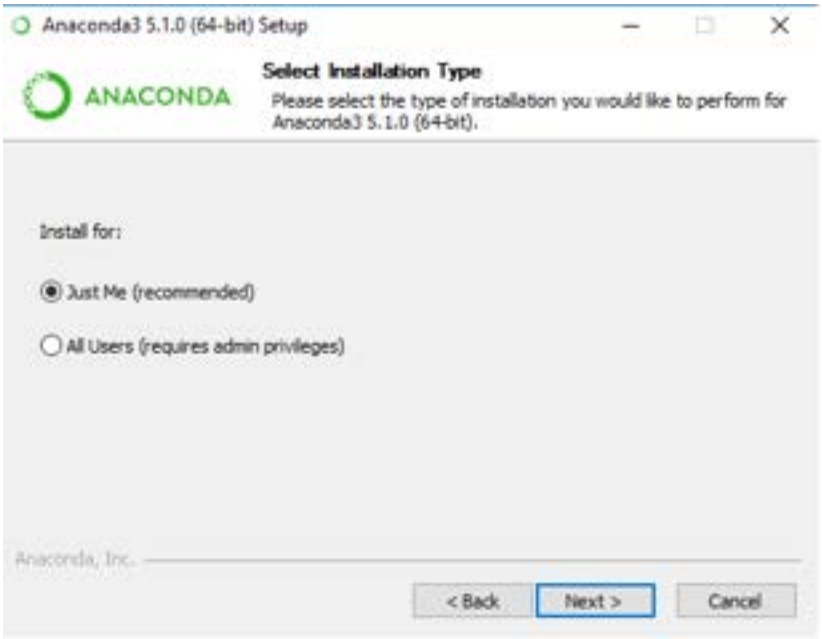
3. You'll have to scroll a bit down to see the next figure 3. Here you can download 64 bit or 32 bit Anaconda 5.2 Windows Installer but don't forget to choose Python 3.6 version. The Installer size is large (613 MB for 64) so it may take some time to download.

4. After downloaded you'll have file like: ***Anaconda3-5.1.0-Windows-x86_64.exe***. Just double click the file and you'll have the next figure 4

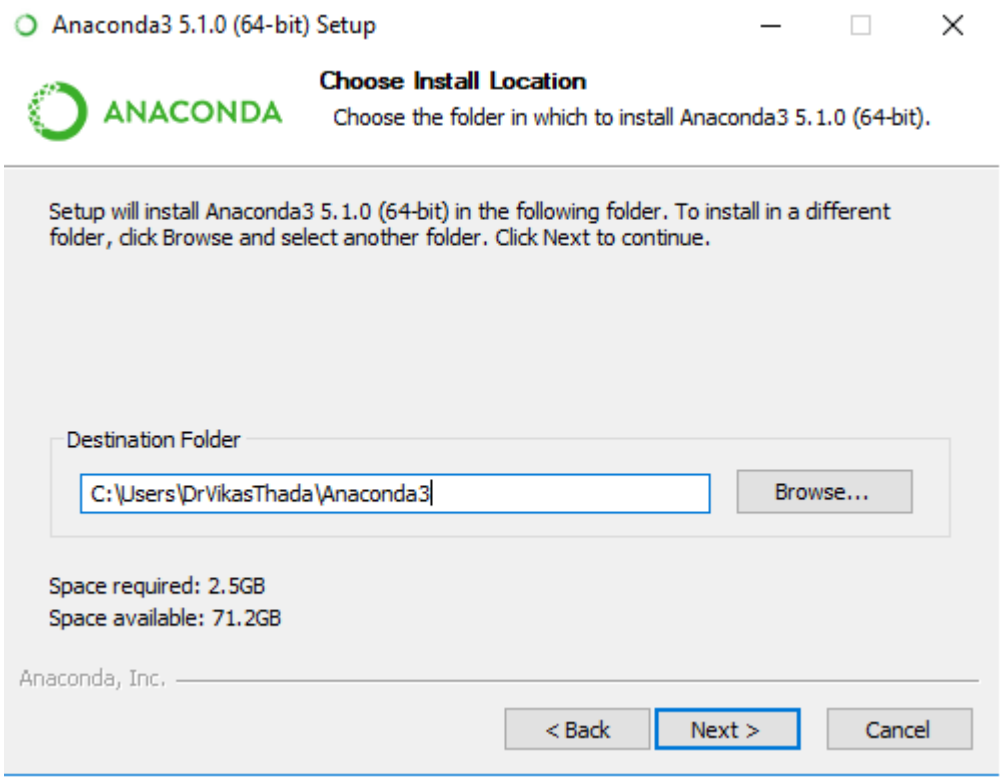
5. Click Next to continue the installation



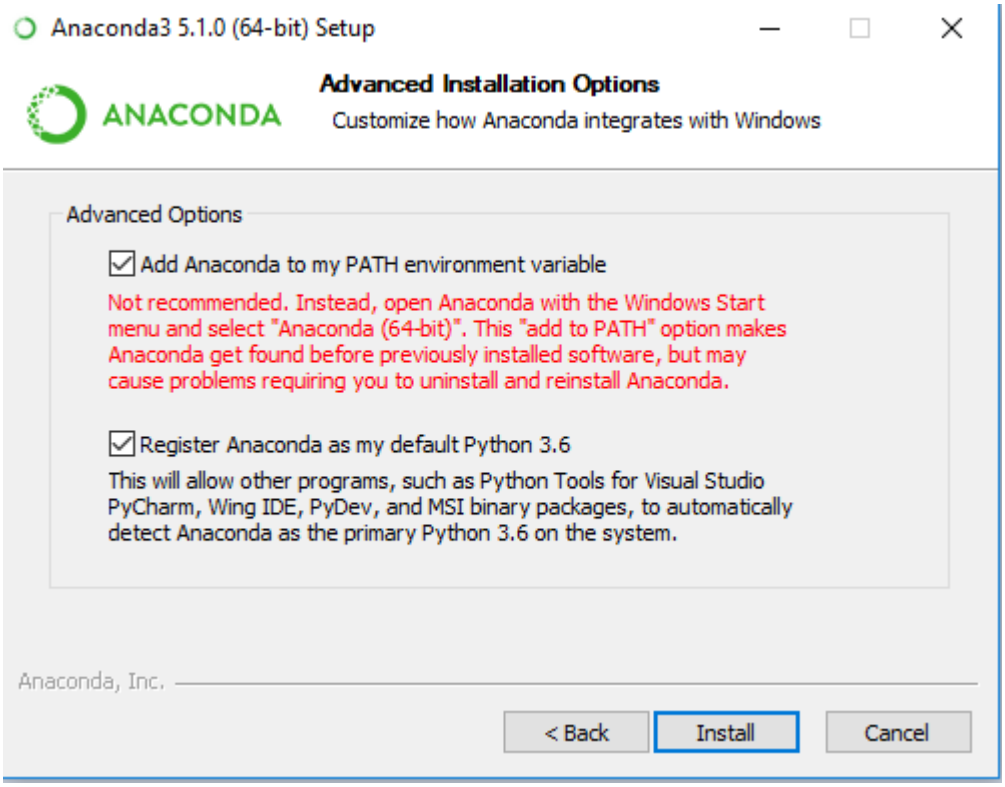
6. Select I Agree in the previous figure to get next figure. Here you can decide who is going to use installed Anaconda, either its you or everyone. I've selected 'Just Me' option.



7. After selecting your option and pressing next button you'll have choose destination folder where Anaconda will be installed. For selection of different folder you can browse the location by clicking Browse button. On my system I've used the path: C:\Anaconda3.



8. Finally the last screen before installation begin will appear after you press next button in previous figure. Here select the first option (tick mark) for setting the path and adding necessary anaconda files /folder to your system environment variable PATH.

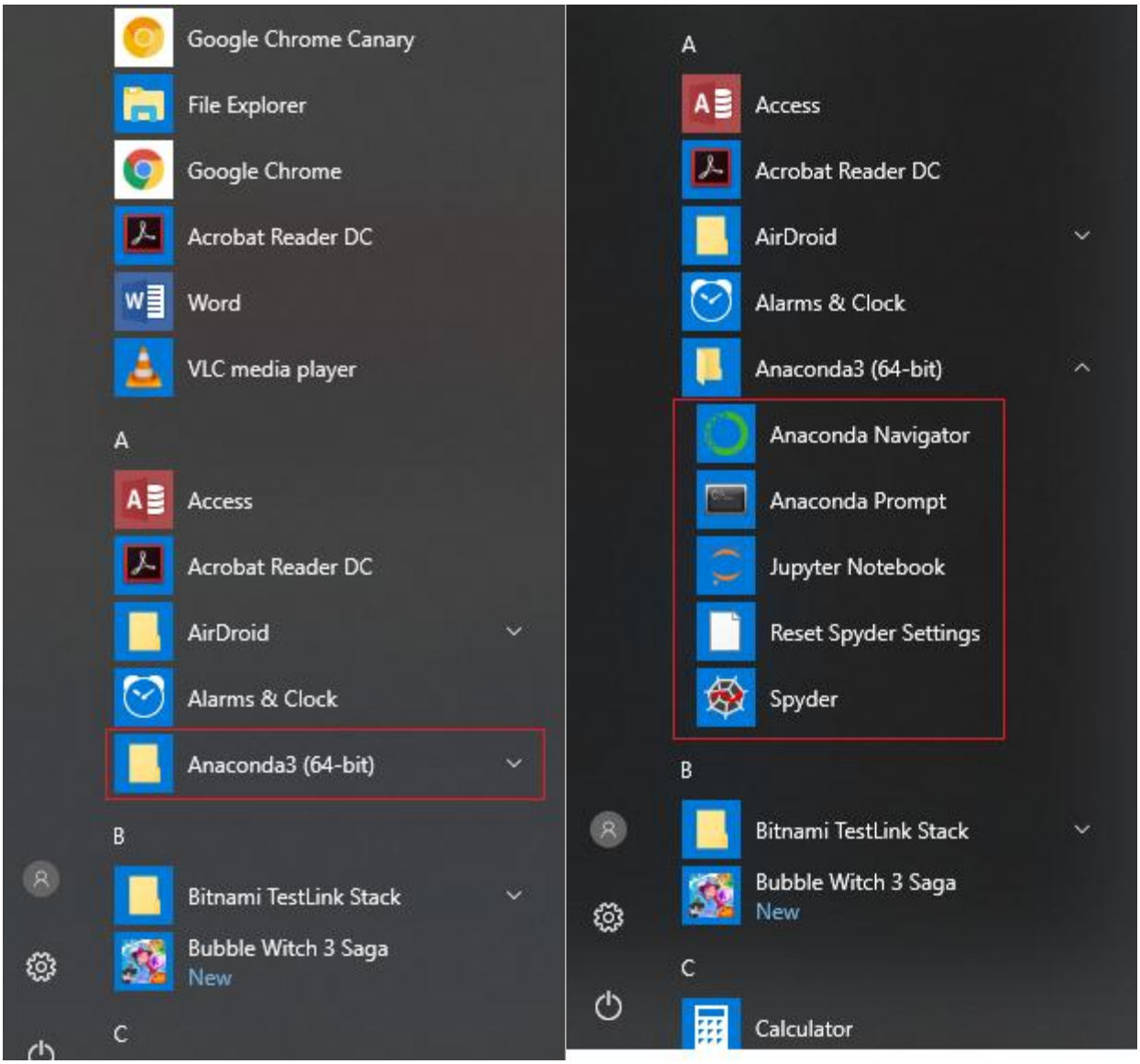


Finally click install and in few minutes it will be installed.

9. To verify the successful installation you can perform many checks :

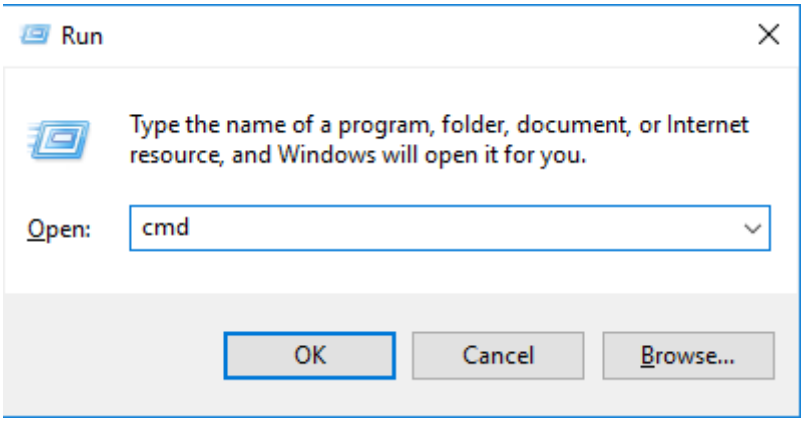
First Check

Just click start /windows icon button and see in Program files. You will find it :

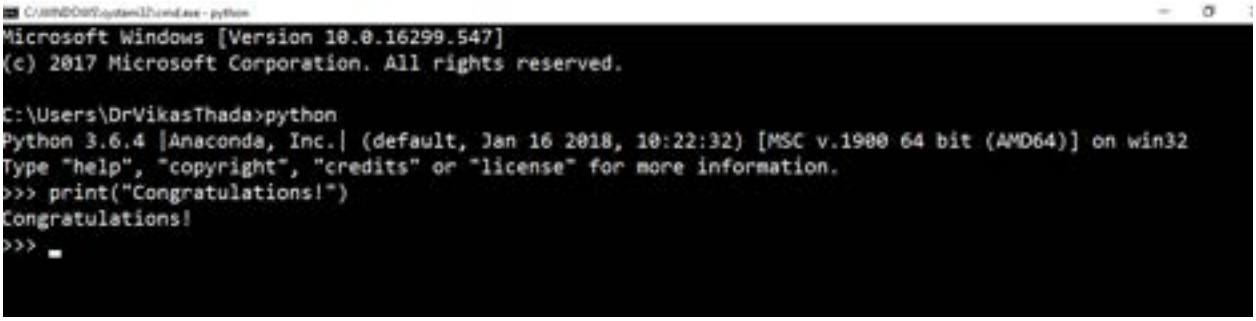


Second Check

Open command prompt by pressing “Win Logo+R” and type cmd:



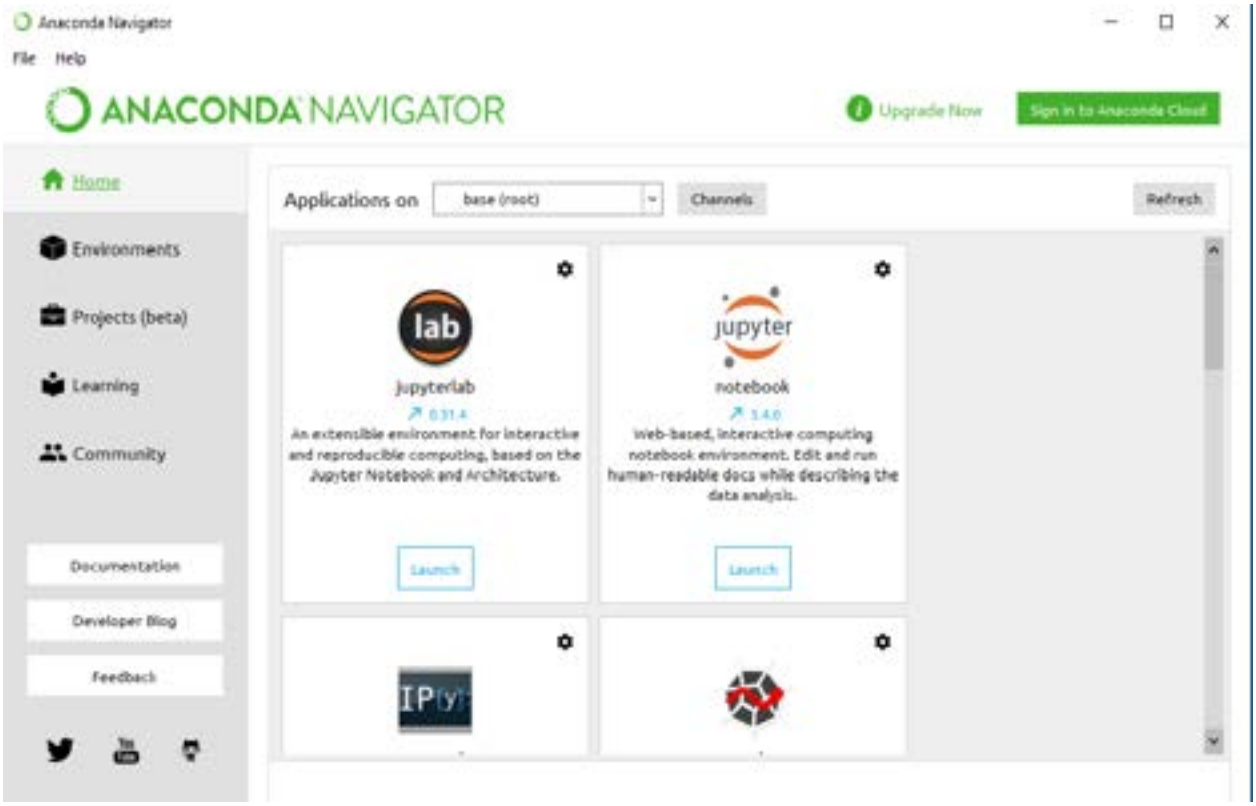
Click OK and you will be in command prompt. There just type **python.** Assuming the python executable is added to the environment variable PATH. You'll see following screen with python default prompt ">>>". This is known as primary prompt. Python also has secondary prompt "…" when you want your code to span multiple lines in shell. You'll see this later in this chapter. For the time being just make use of print function as shown in the figure below:



To come out from python shell just type: exit()

Third Check

Just type : anaconda-navigator at the command prompt. It will take some time to launch. It's a command based tool to launch number of other applications like Spyder, Jupyter Notebook etc. Some of them may be installed and other need to be installed.



1.4 Other Options

The other alternatives to Anaconda, if you simply want to work with python are installing any popular python IDE like PyCharm Community Edition , Wing Python IDE, Eclipse with Python plugin. Other than these two, you can also make use of Spyder IDE that comes with Anaconda. Simple editors (but not full fledged IDE) like visual studio code, Atom, Sublime editors, notepad++ etc can be used. For this book we have used both Jupyter notebook and PyCharm Community Edition.

1.5 Working with Jupyter Notebook

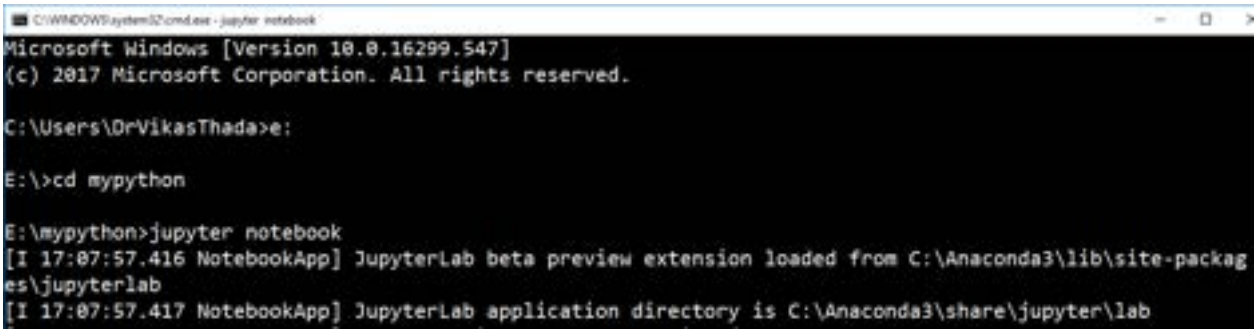
The Jupyter Notebook is a file with extension '.ipynb' which stands for interactive python notebook. It's a web application so to open it you need to have a web server which gets installed when you install jupyter notebook into system. It comes integrated with Anaconda Framework.

The notebook can be used for creating interactive code in python along with html, images, plots, latex, machine learning and deep learning tools etc. Much discussion of jupyter is beyond the scope of the book. For more details just google it or follow link: **http://jupyter.org/**

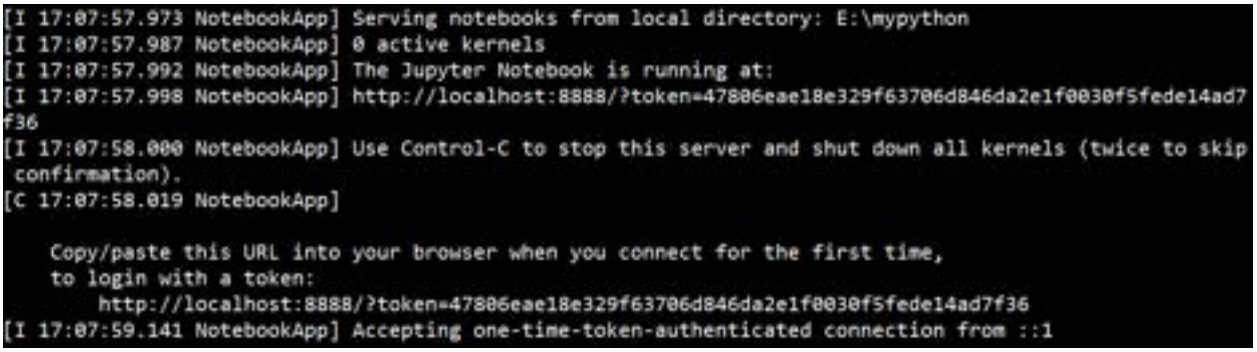
We'll just stick ourselves as how to use it anaconda and how to share code using it.

To run jupyter from command line just go to any directory where you are jupyter notebooks saved or you want to save the new ones. Here in the following figure we are in *mypthon* directory under *E:*. Now type at the command prompt: **jupyter notebook**

The above command starts the server.

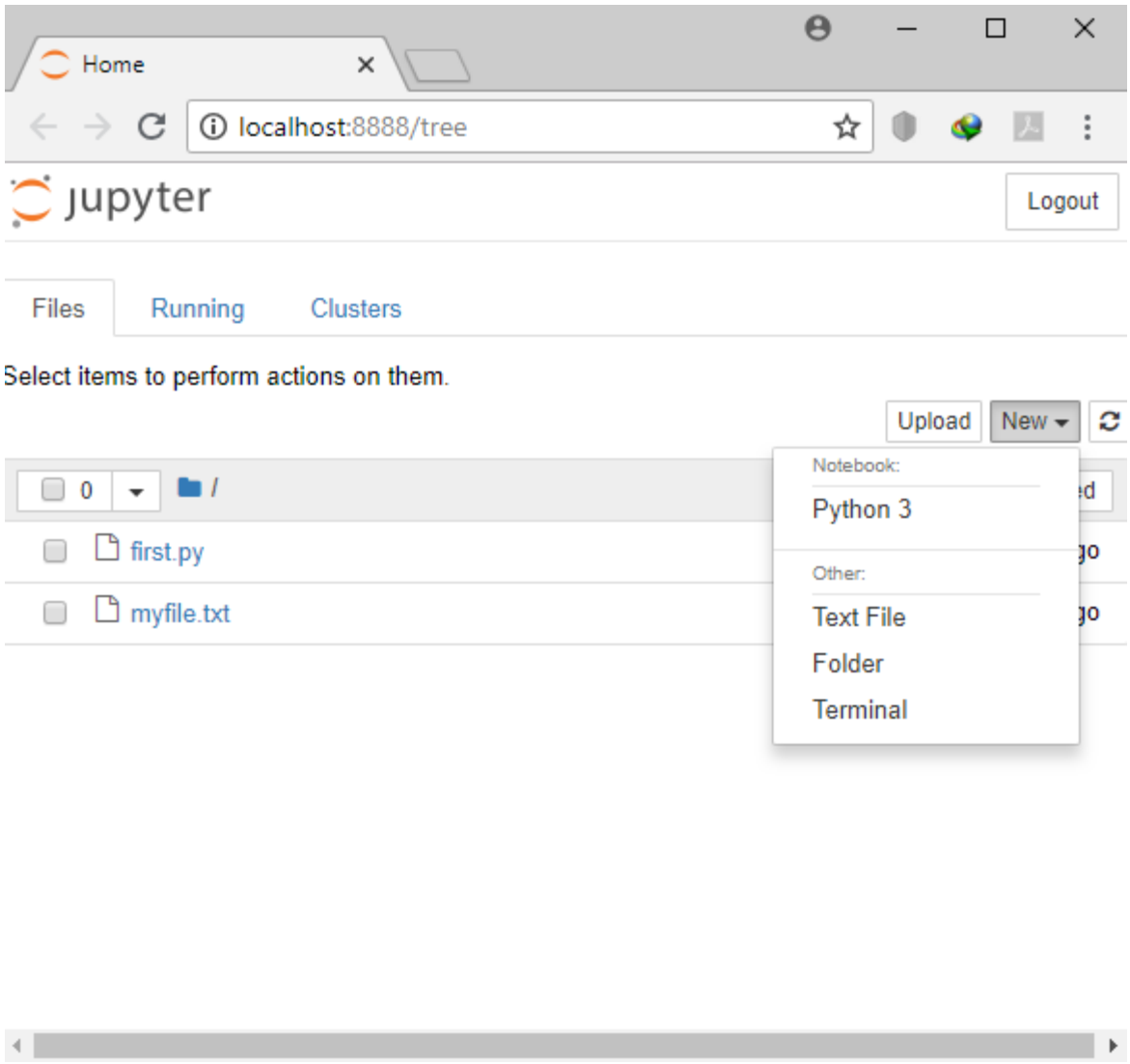


You'll see the contents like shown in the following figure:

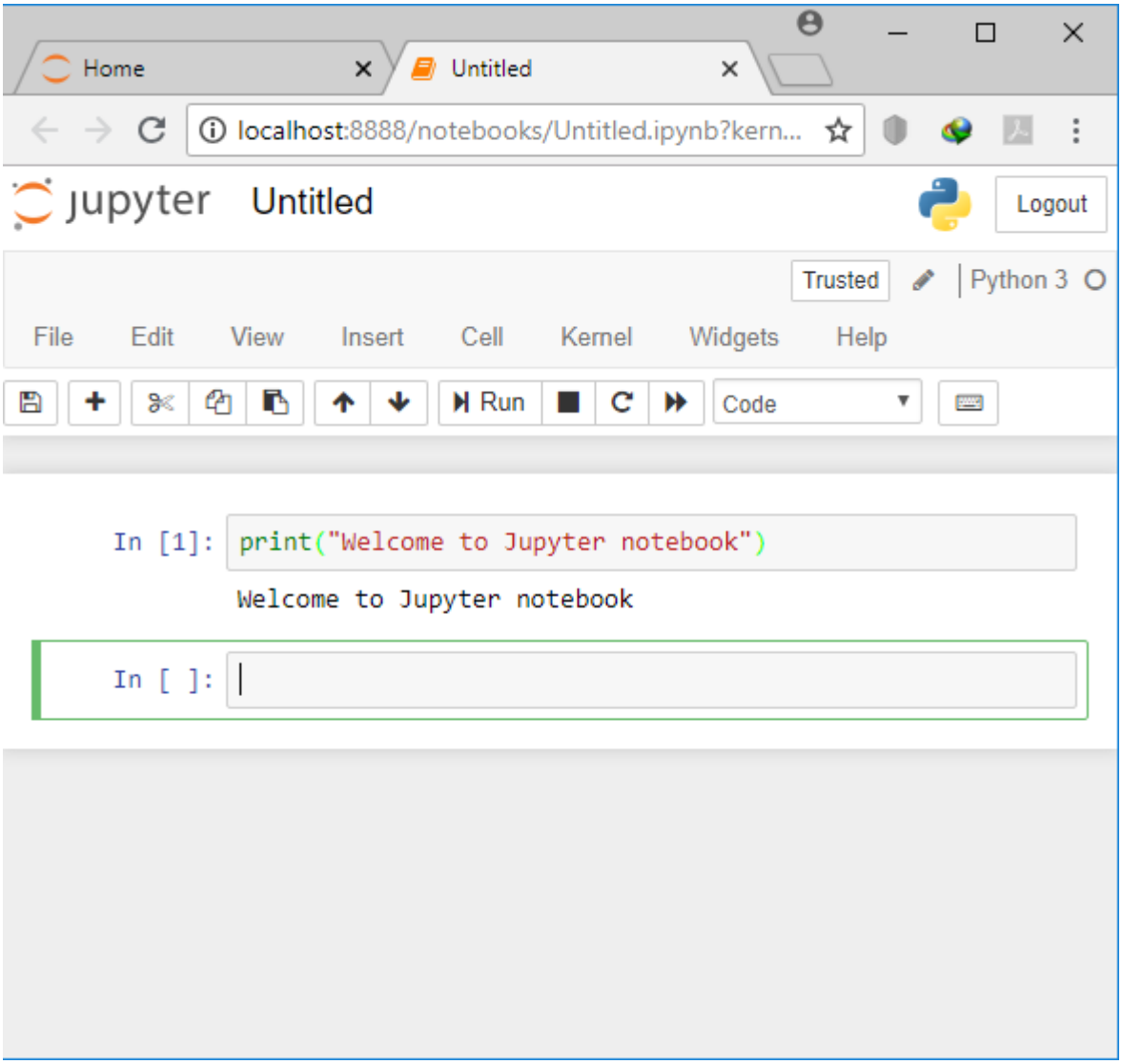


Immediately after this the application going to launch in your default web browser. If you want to run the web application in some another web browser just copy past the line that start with **http://localhost:/8888..... .** You will see the following figure then:

Here the figure shows the contents of the directory **"E:\mypthon".** To create a new jupyter notebook click on New and select Python 3.

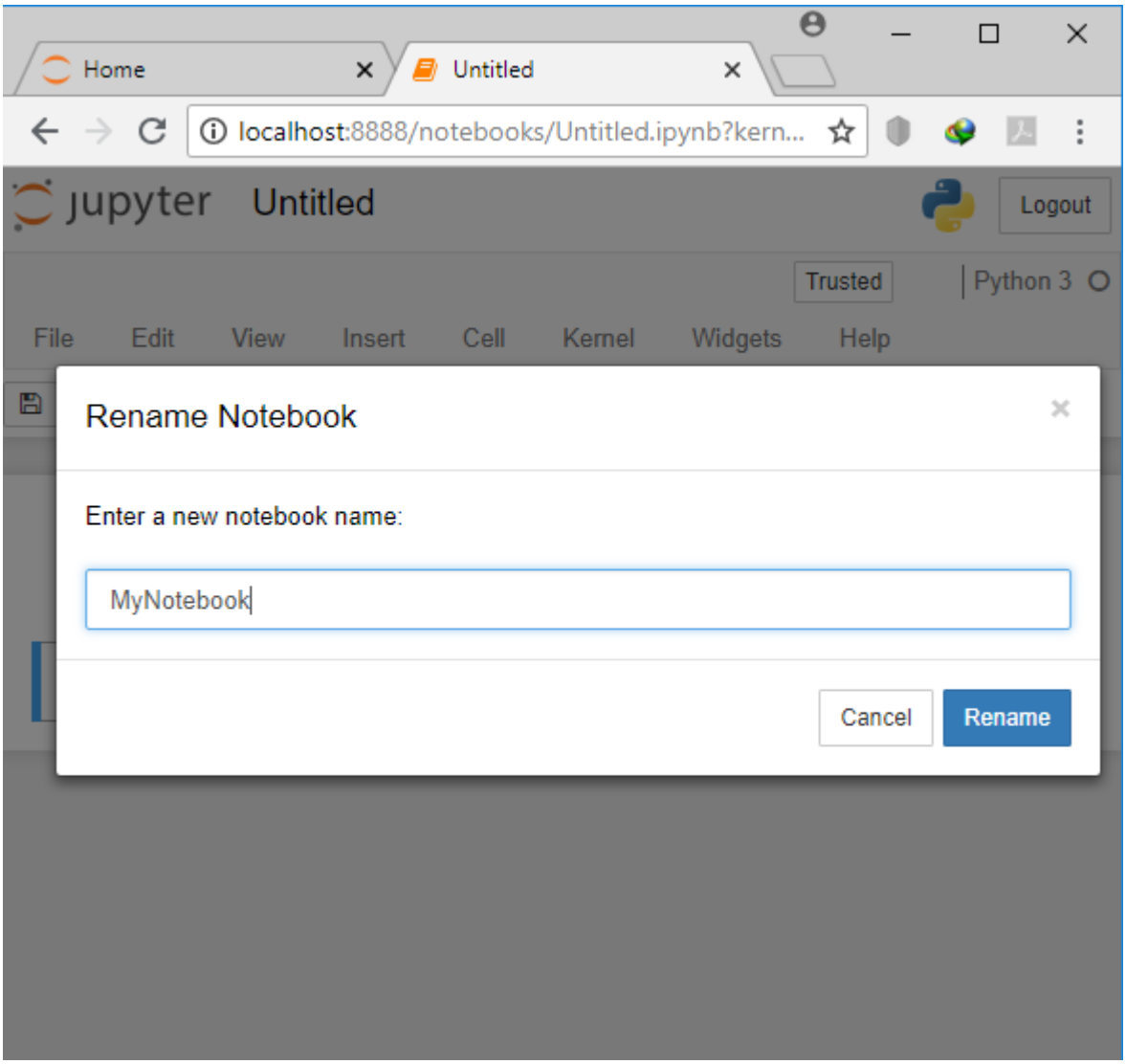


You will have a new jupyter notebook opened for you with the name **Untitled**. The jupyter notebook consists of number of cells which can be inserted or deleted. You have to type the code inside the cells and for execution press “Shift+Enter”. Try to create one now and type some code as shown in the figure given below:

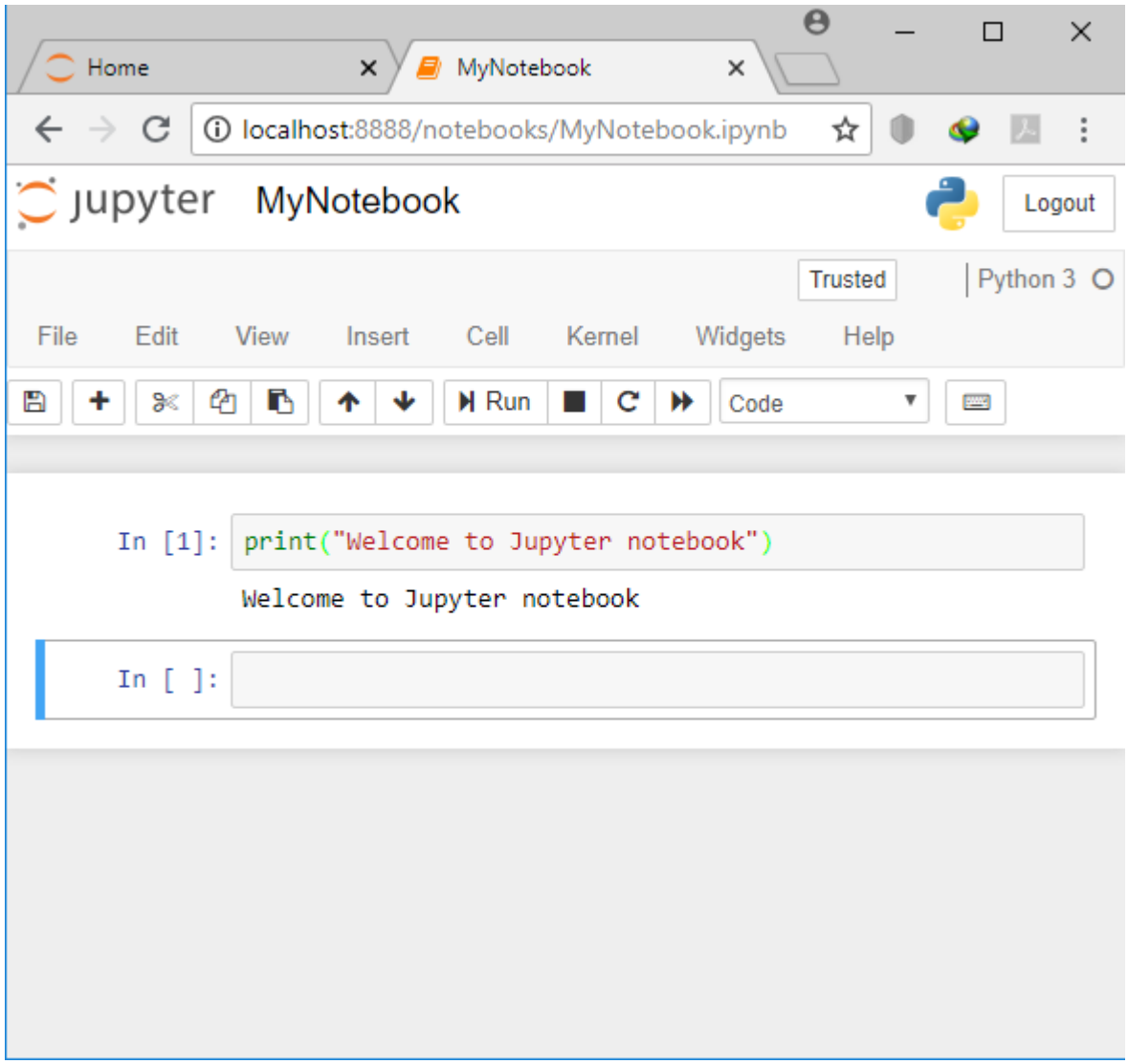


The advantage of Jupyter Notebook is that you can write any amount of code, including Python, HTML, Markdown, or any other language (Jupyter supports 40 languages to work with). The code remains in the notebook cells. You can modify them and rerun by pressing **shift+enter**.

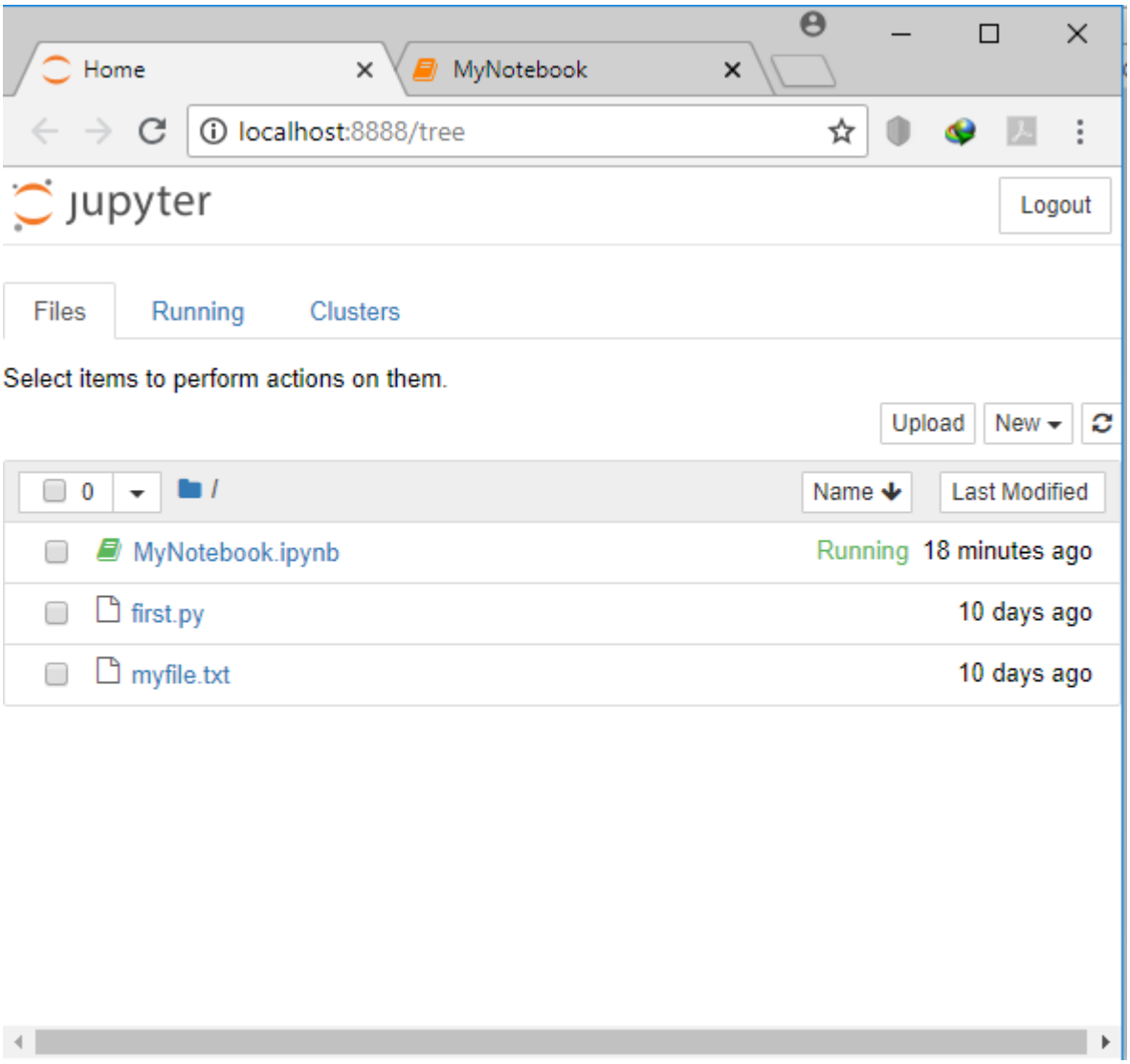
Once we have written some code inside our newly created notebook, we can save it by new name instead of default “Untitled”. Just click on File and select Rename.



Give a new name to your jupyter notebook and click Rename. Your jupyter notebook is now renamed.



Your juptyer notebook will be found in current directory as:

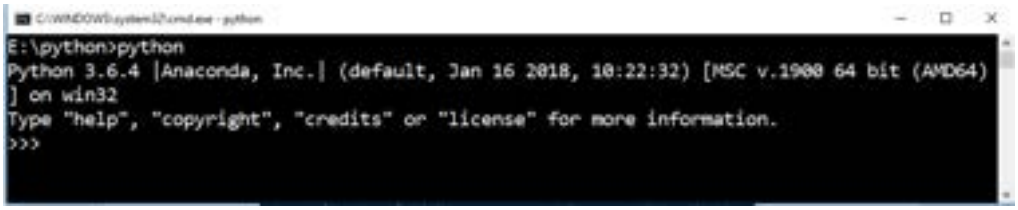


If you have any other notebook downloaded from web, the same can be uploaded to your notebook tree by just uploading it using Upload button next to New. Other way is to just copy it in current directory i.e. E:\mypython. It will be visible in the tree.

1.6 Python Shell

Python provides an interactive command line interface for running python statements, commands, functions etc. It's the python interpreter who interprets commands written in python thus the interpreter in command mode is python shell.

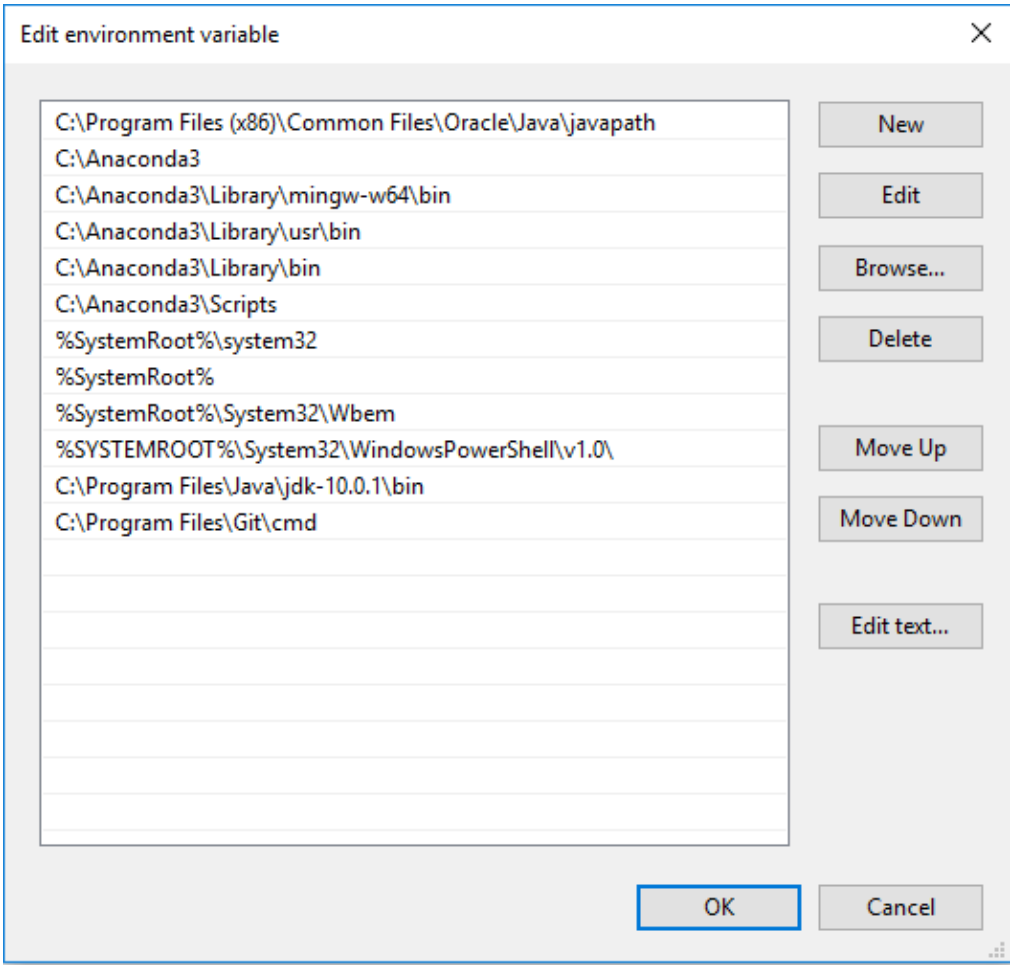
I assume you have followed the instructions above for setting up your python environment. Start python shell by typing: **python** at command prompt.



Now see the shell session with some of the commands typed in python shell:

```
>>> print("Hello Python")
Hello Python
>>> s='I love python'
>>> print(s)
I love python
>>> x=10;y=20
>>> print('sum of',x,'and',y,'is',x+y)
sum of 10 and 20 is 30
>>> sum=x+y
>>> print(sum)
30
>>> 2+3
5
>>> _
5
>>> sum
30
>>> _
30
>>> _ * 2
60
```

Congratulations on your first tryst with python shell. If in any case your python shell does not start, then it may be the path issue or some other issue. For path related issues just include python executable file in your path environment variable. This is shown in my laptop:

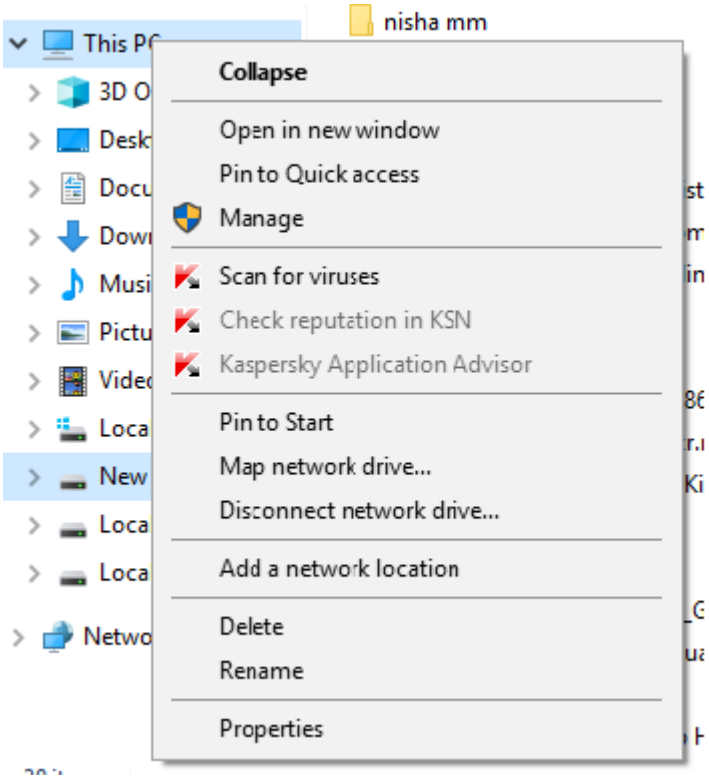


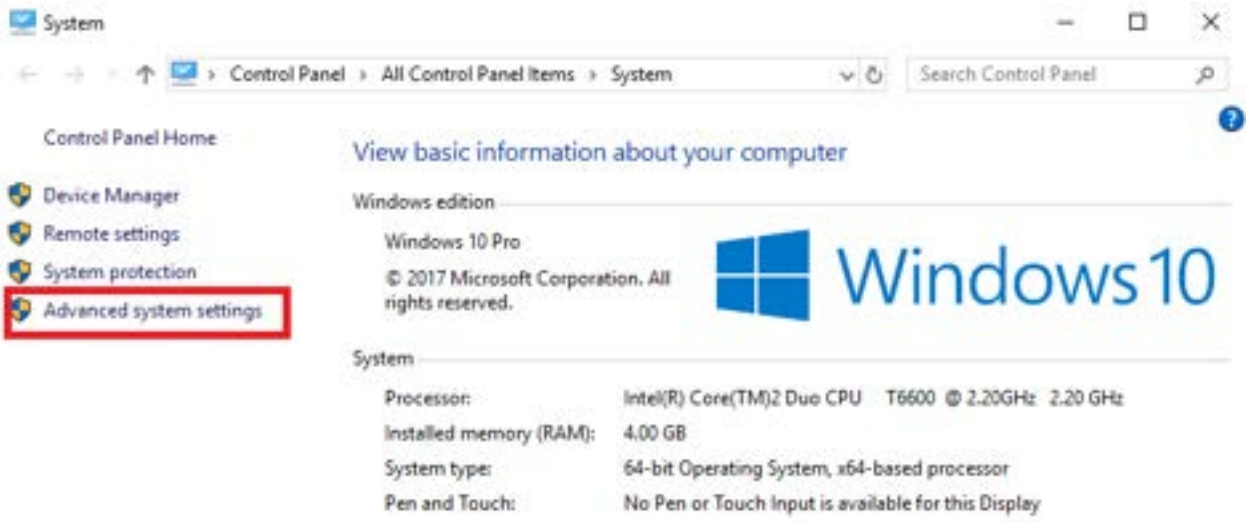
The file python.exe is present in **C:\Anaconda3** and ipython (we will cover it later) is in **C:\Anaconda\Scripts.**

Setting up Path

If you have not set the path earlier during anaconda installation, you can do it now by following the steps:

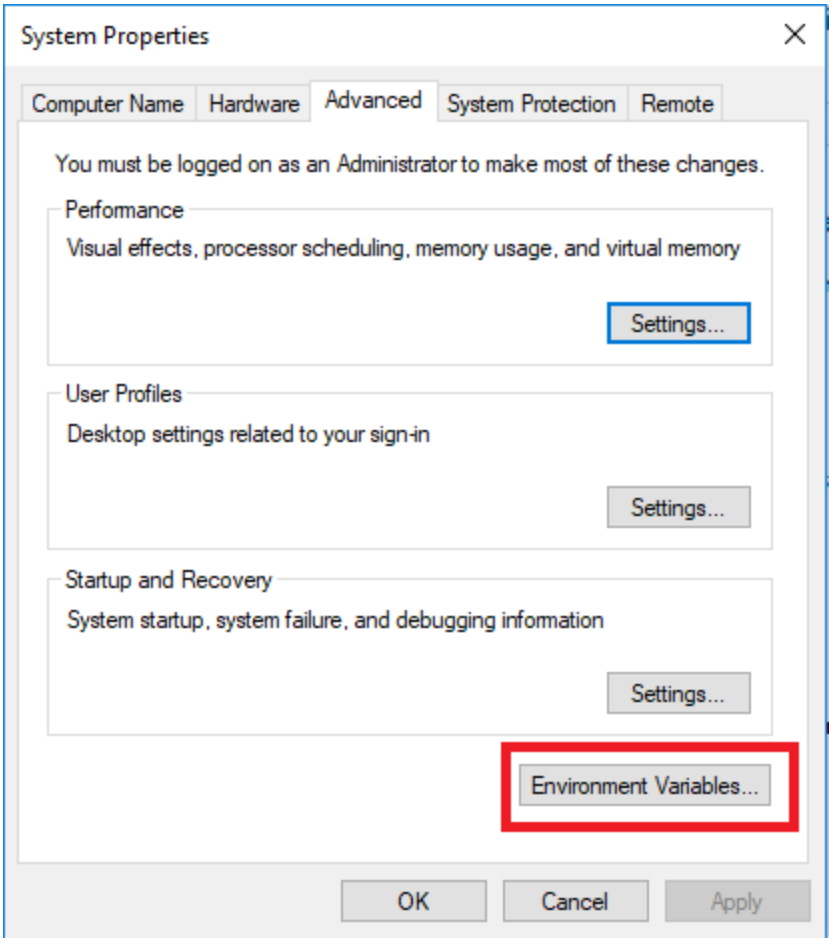
1. Right click of My Computer of *This PC* and select Properties:



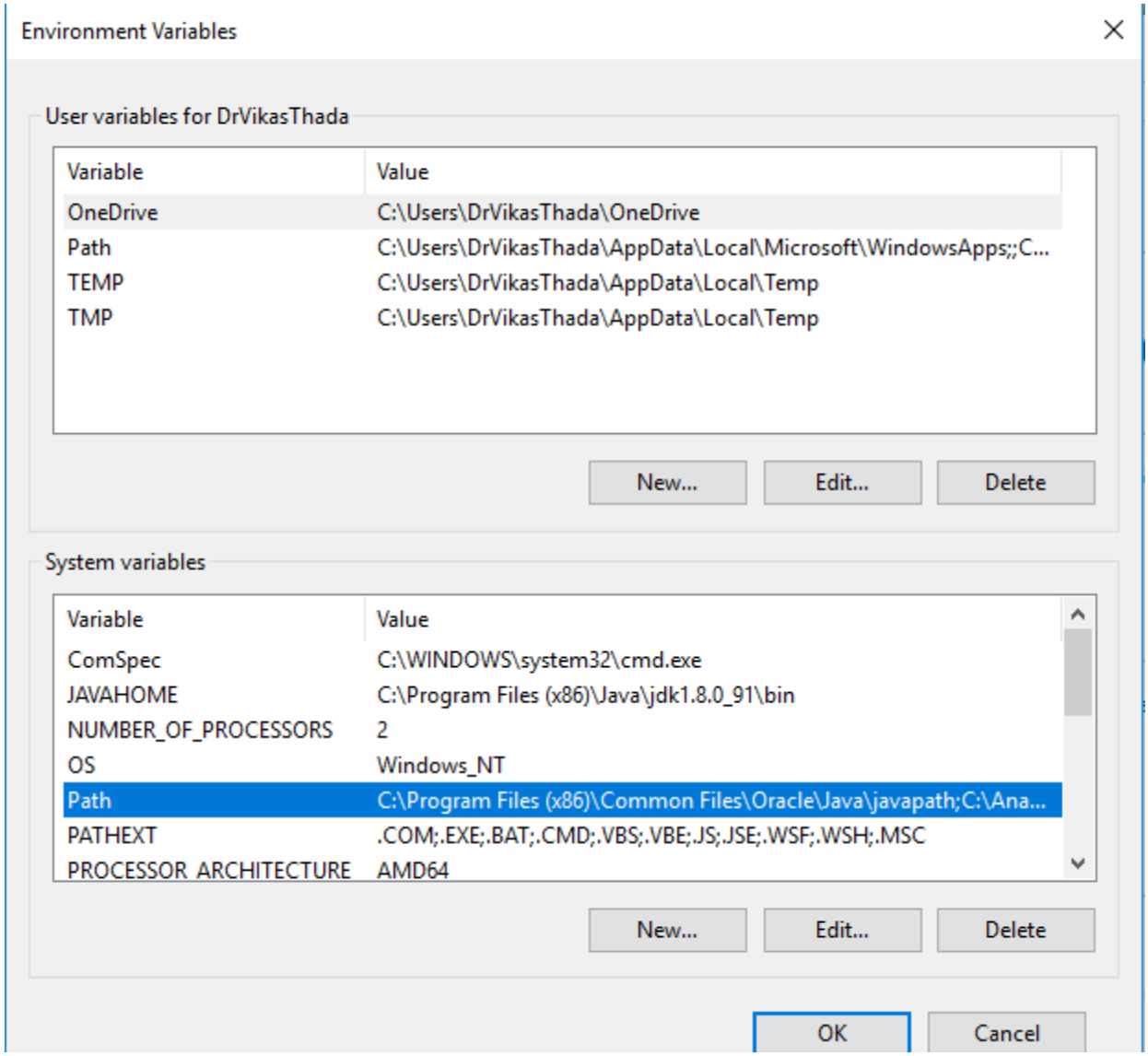


2. Then in the next window that open select “Advanced system settings”.

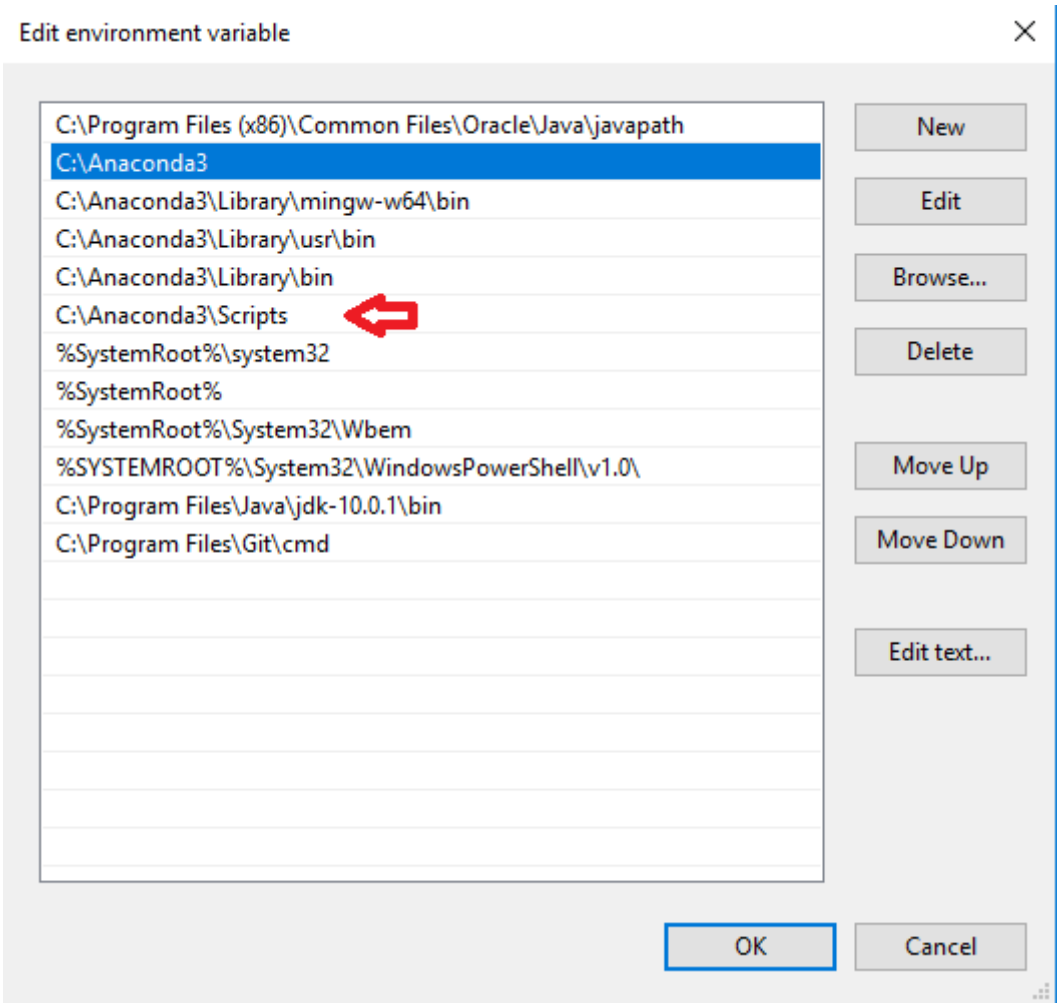
3. Next select “Environment Variables” as shown by red rectangle in the following figure.



4. In the next windows select Path under System Variables and click Edit:



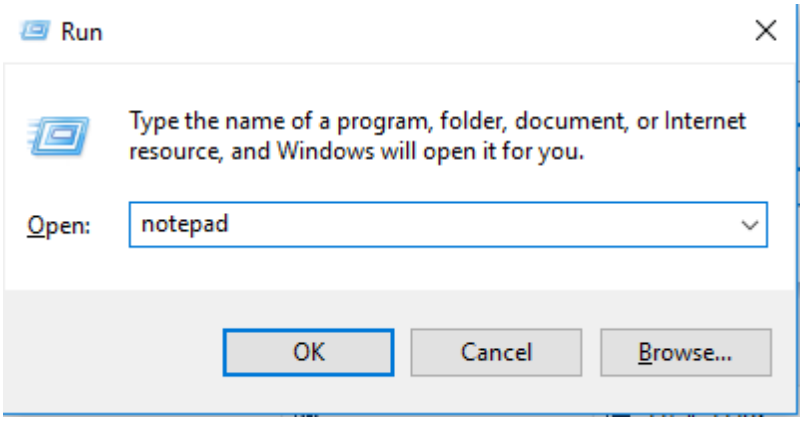
5. Add Path for Python executables as highlighted in blue and indicated by red both by selecting New and browsing for the files.



1.7 Writing First Python Script

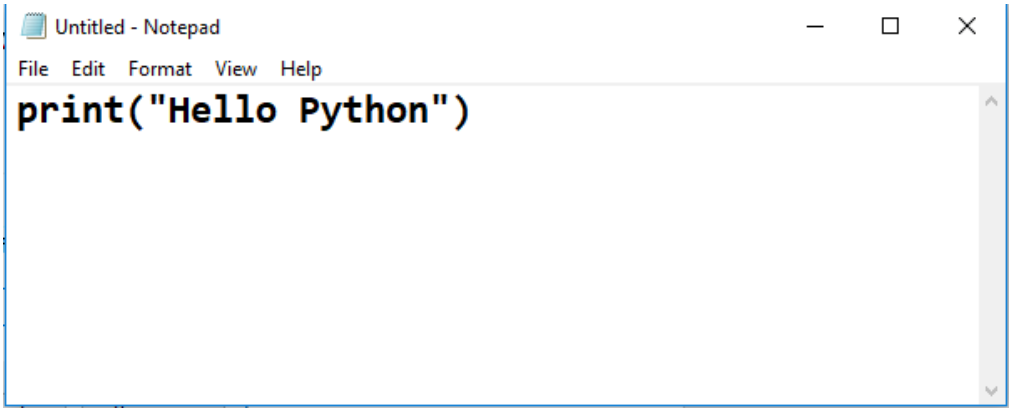
After having our first meeting with python shell, lets write our first script in python. For that you can use any editor. You can try notepad, notepad++, sublime, PyCharm or WinIDE or Spyder editor. As this is our first script you can just be happy with notepad.

Just create one folder by the name “mypython” in any of your favorite drive other than C: drive. Now open notepad by typing “Win (logo) +R” and typing “notepad” in it without quotes.

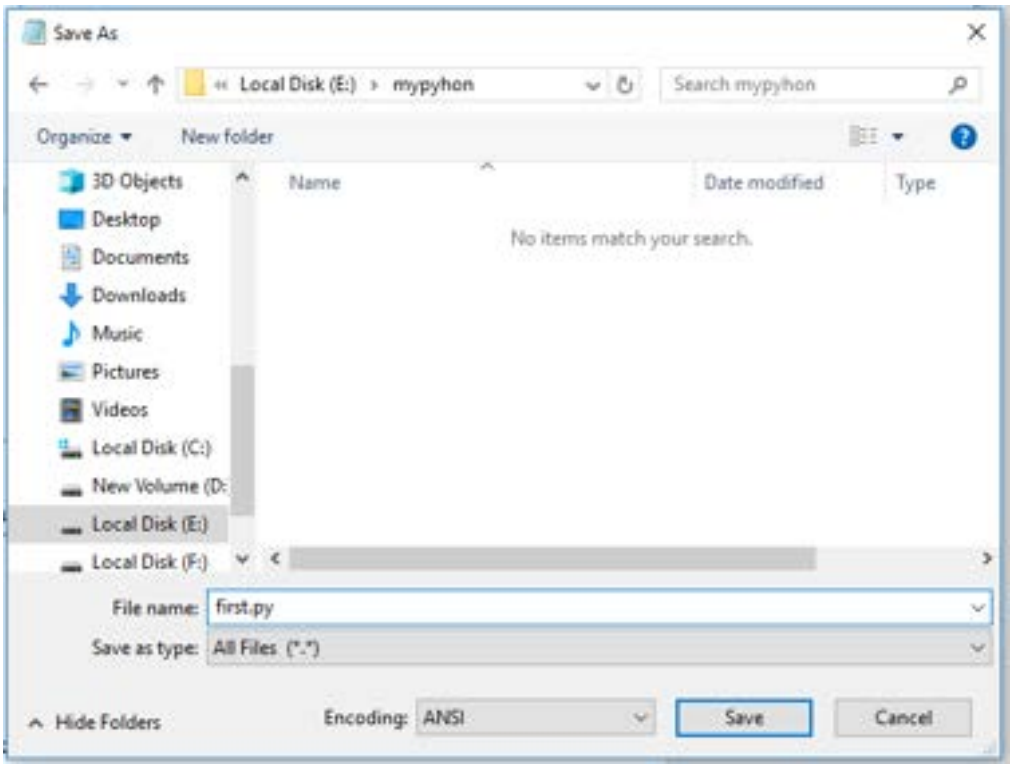


This will open the notepad window. Now type the following in it:

```
print("Hello Python")
```

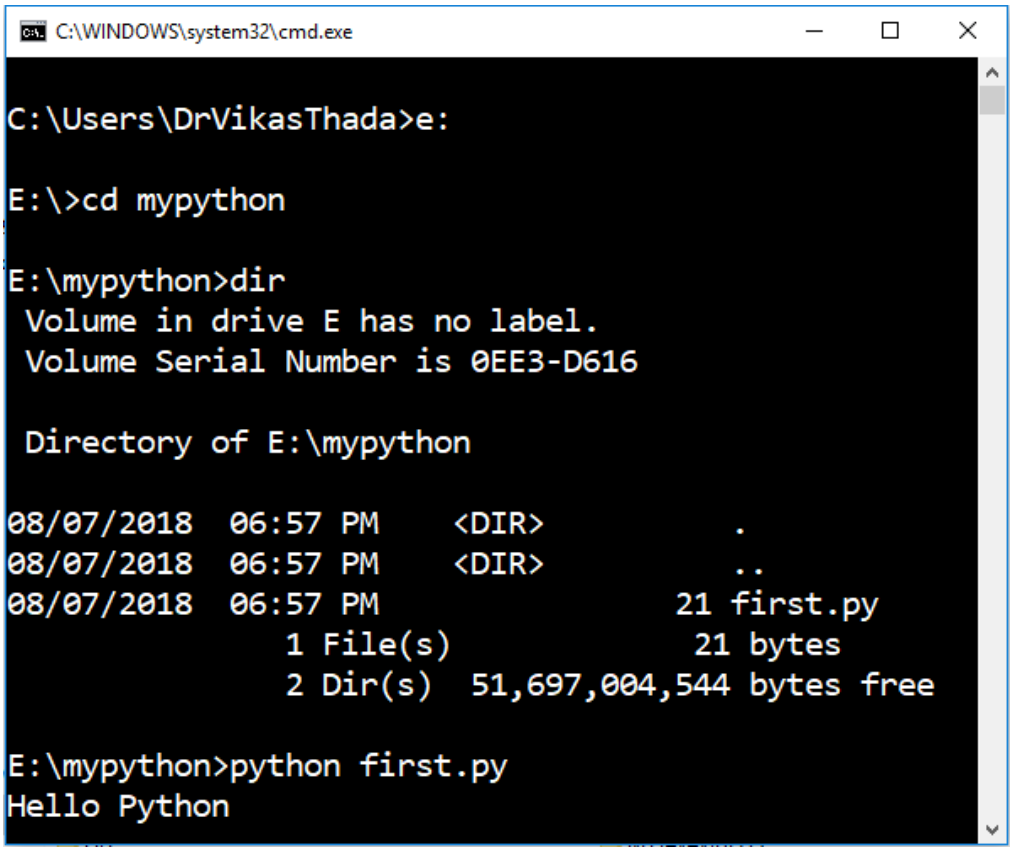


Now save it by selecting "save as" from File menu in your folder : "mypython" by the name "first.py" . You can choose any other name , but extension must be ".py".



One important point to remember is that In “Save as type” box you have to change from text files to All Files (*.*) else your file be saved as: first.py.txt.

Once you have done this and assuming path to python executable is set as discussed above, come to DOS prompt and go to your folder/dir. See the figure below:



If you have followed instructions as shown in the figure above then viola!!. Congratulations on running your first python script.

1.8 Python Character Set

A Python program is a collection of number of instructions written in a meaningful order. Further instructions are made up of keywords, variables, functions, objects etc which uses the Python character set defined by Python. It is a collection of various characters, digits and symbols which can be used in a Python program. It comprises followings:

Table 1.1: Python Character Set

S.N	Elements of Python character Set
1.	Upper Case letters: A to Z
2.	Lower Case letters: a to z
3.	Digits: 0 to 9
4.	Symbols (See below)

Symbol	Name	Symbol	Name
~	tilde	>	greater than
<	less than	&	ampersand
|	or/pipe	#	hash
>=	greater than equal	<=	less than equal
==	equal	=	assignment
!=	not equal	^	caret
{	left brace	}	right brace
(	left parenthesis	)	right parenthesis
[	left square bracket	]	right square bracket
/	forward slash	\	backward slash
:	colon	;	semicolon
+	plus	-	minus
*	multiply	/	division
%	mod	,	comma
'	single quote	"	double quote
>>	right shift	<<	left shift
.	period	_	underscore

1.9 Python Tokens

Smallest individual unit in a Python program is called Python Token. Python defines six types of tokens.

1. **Keywords**
2. **Identifiers**
3. **Literals**

4. Strings

5. Operators

6. Special Symbols

1.9.1 Keywords

Keywords are those words whose meaning has already been known to the python interpreter. That is meaning of each keyword is fixed. You can simply use the keyword for its intended meaning. You cannot change the meaning of Keywords. Also you cannot use keywords as names for variables, function, arrays etc. Keywords are required as they help us to create scripts, define structure and syntax of the python scripts.

There are **33** keywords in Python 3.6. Following figure lists all keywords available in Python.

Table 1.2: Python Keywords

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

As clear from the above table except the first three keywords: False, True, None, all other keywords are written in small case letters.

1.9.2 Identifier

Identifiers are names given to various program elements like variables, array, functions, class etc. To identify any programming element like variable, class, function, array, module etc. identifiers are required.

Rules for writing identifiers

(a) First letter must be an alphabet or underscore _.

(b) From second character onwards any combination of digits, alphabets or underscore is allowed.

(c) Only digits, alphabets, underscore are allowed. No other symbol is allowed.

(d) Keywords cannot be used as identifiers.

(e) Identifier can be of any length.

Examples of valid and invalid identifiers (On the basis of above rules)

Valid identifier	Order_no, name, _err, _123,xyz,radius, a23, int_rate
Invalid Identifiers	a. order-no (hypen not allowed) b. 12name (cannot start with digit) c. e$rr (here $ is for space which is not allowed) d. def (cannot use as it is a keyword) e. x+123 (+ cannot be used) f. 123 (just numbers not allowed)

1.9.3 Literals

Literals in Python refer to fixed values that do not change during the execution of a program. They are also known as nameless constants. There are various types of literals in Python. They are classified into the following categories as given below.

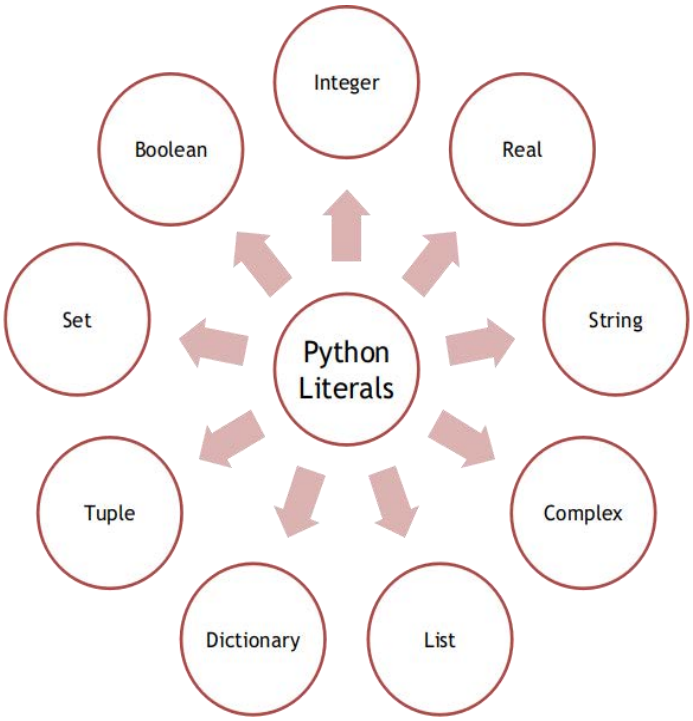


Figure 1.1: Various python literals

(a) Integer Literals

(b) Real Literals

(c) String Literals

(d) Complex Literals

(e) List Literals

(f) Dictionary Literals

(g) Set Literals

(h) Tuple Literals

(i) Boolean Literals

We discuss each literal one by one.

Integer Literals

They are of three types

(a) Decimal Literals: They are sequence of digits from 0 to 9 without fractional part. It may be negative, positive or zero.

Example: 12,455, -5644,0.

(b) Octal Literals: They have sequence of numbers from 0 to 7 and first digit must be 0o. The alphabet O may be upper case or lower case.

Example: 0O34,0O,0o564,0o123

(c) Hex Literals: They have sequence of digits from 0 to 9 and A to F(represents 10 to 15).They start with 0x or 0X.

Example: 0x34, 0xab3, 0X3E**.**

(d) Binary Literals: They have sequence of digits from 0 and 1. They start with 0b or 0B.

Example: 0b01, 0B111, 0b10101**.**

The shell tour of all the above literals in shown in the following figure:

```
>>> x=10; print(x)
10
>>> x=0O45; print(x) # octal number, output in decimal
37
>>> x=0x34; print(x) # hex number, output in decimal
52
>>> x=0b1010; print(x)
10
>>> x=1_23_45_678; print(x) # long number separated by _
12345678
>>> x=0b1011_1110; print(x) # binary number
190
```

From the above small python shell session, you have understood the concepts of integer literals. You also have noticed that all binary, octal and hex literals on printing gives output in decimal. This can be changed with the use of built in functions: bin, oct and hex. See them in action in following session:

```
>>> x=0b11
>>> y=0b101
>>> x+y
8
>>> bin(x+y)
'0b1000'
>>> x=0o23
>>> y=0o12
>>> x+y
29
>>> oct(x+y)
'0o35'
>>> x=0x12
>>> y=0xa1
>>> x+y
179
>>> hex(x+y)
'0xb3'
```

Real Literals

They are the numbers with fractional part. They are also known as floating point literals.

Example: 34.56, 0.67, 1.23**.**

Real constants can also be represented in exponential or scientific notation which consists of two parts. For example the number **212.345** can be represented as **2.12345e+2** where **e+2** mean **10 to the power 2**. Here the portion before the **e** that is **2.12345** is known as **mantissa** and **+2** is the exponent. Exponent is always an integer number which can be written either in lower or upper case.

```
>>> x=3.4e+2
>>> x
340.0
>>> x=3.4e-2
>>> x
0.034
```

String or String Literals

They are sequence of characters, digits or any symbol enclosed in double quotes or single quotes.

Example: "hello", ‘23 twenty three’, ‘&^ABC’, “2.456”

```
>>> x='python'
>>> print(x)
python
>>> x="python"
>>> print(x)
python
>>> x="'python'"
>>> print(x)
'python'
>>> x='"python"'
>>> print(x)
"python"
```

Single Character Literals

Python does not any have concept of character literals but a string with just one character enclosed in either single or double quotes work as character literals. Python uses Unicode character set and supports ASCII characters sets too.

```
>>> x="A"
>>> ord(x) # Ascii value of A
65
>>> x='a'
>>> ord(x)
97
>>> x=65
>>> chr(x)# character corresponding to Ascii value 65
'A'
>>> x=191
>>> chr(x)
'¿'
>>> x="AB"
>>> ord(x)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: ord() expected a character, but string of length 2 found
```

Boolean Literals

Boolean literals are simply **True** and **False**.

```
>>> True
True
>>> False
False
>>> x=True
>>> print(x)
True
>>> type(x)
<class 'bool'>
```

Backslash Literals

Python defines several backslash character constants which are used for special purpose. They are called so because each backslash constant starts with backslash (\ ). They are represented with 2 characters whose general form is **\char** but treated as a single character. They are also called **escape sequences.** You will see their usage in number of programs later in this chapter and all other chapters of the book. Given below is the list of backslash character literals: -

S.N	BCC	MEANING	ASCII VALUE
1.	\b	backspace	08
2.	\f	formfeed	12
3.	\n	newline	10
4.	\r	carriage return	13
5.	\"	double quotes	34
6.	\'	single quotes	39
7.	\a	alert	07
8.	\t	horizontal tab	09
9.	\v	vertical tab	11

Table 1.3:

10.	**\0**	**null**	**00**

Escape Sequences

```
>>> print("\a")
>>> print("hello\tpython")
hello   python
>>> print("hello\npython")
hello
python
>>> print("hello\rpython")
python
>>> print("hello\r\npython")
hello
python
>>> print("hello\bs")
hells
>>> print("hello\b ")
hell
>>> print("\"hello\"")
"hello"
>>> print("\'hello\'")
'hello'
>>> print("a\\b")
a\b
```

```
>>> print("It\'s fun")
It's fun
```

Complex Literals

Complex literals are having two parts: real and imaginary. The both the parts are written with a "+" as separator. Any part can be integer or float type. Examples: 2+3j,1.2+5j, 3j,

Shell session is given below:

```
>>> x=2+3j
>>> print(x)
(2+3j)
>>> x=1.2+4.5j
>>> print(x)
(1.2+4.5j)
>>> print(x.real,x.imag)
(1.2,4.5)
>>> x=0.2j
>>> print(x)
0.2j
>>> type(x)
   <class 'complex'>
```

List Literals

List literals are any combination of other literal types enclosed within square brackets. They will be discussed in detail in coming chapters,

Examples: [1,2,3,4],['A',22,True],["hello",23,34.45].

Shell session is given below:

```
>>> L=[1,2,3,4]
>>> print(L)
[1, 2, 3, 4]
>>> L=[True,"hello",34.45]
>>> print(L)
[True, 'hello', 34.45]
>>> L=[True,2+3j,[3,4],45.56]
>>> print(L)
```

```
[True, (2+3j), [3, 4], 45.56]
>>> type(L)
<class 'list'>
```

Dictionary Literals

Dictionary literals are created within curly braces and with key-value pairs. More on this in coming chapters. Examples: {‘age’:23,’name’:’harsh’},{‘jan’:31,’march’:31} etc.

See shell session below:

```
>>> x={'age':12,'name':'harsh'}
>>> x
{'age': 12, 'name': 'harsh'}
>>> x.keys()
dict_keys(['age', 'name'])
>>> x.values()
dict_values([12, 'harsh'])
>>> x.items()
dict_items([('age', 12), ('name', 'harsh')])
```

Set Literals

Set literals are simply set with elements enclosed within braces. They can be created with set() or by putting elements in curly braces. Set store only unique elements even if duplicates are present.Examples: {2,3,4,5,2},{“A”,”BC”,”D”}, set([3,4,5,6,2]}.

See shell session in practice:

```
>>> {"juhi",'purvi','purvi','anil'}
{'purvi', 'juhi', 'anil'}
>>> set(["juhi",'purvi','purvi','anil'])
{'purvi', 'juhi', 'anil'}
>>> {4,5,6,3,2,3,4,5}
{2, 3, 4, 5, 6}
>>> set({4,5,6,3,2,3,4,5})
{2, 3, 4, 5, 6}
>>> set([4,5,6,3,2,3,4,5])
{2, 3, 4, 5, 6}
```

Tuple Literals

Tuple literals are simply elements enclosed within parentheses. Examples: (2,3,4,5,2),(“one”, “two”), (3,)

See shell session in practice:

```
>>> x=(2,3,4,5)
>>> print(x)
(2, 3, 4, 5)
>>> x=(3,)
>>> type(x)
<class 'tuple'>
>>> x=('hello',34,56.78)
>>> print(x)
('hello', 34, 56.78)
```

List, sets, tuple, dictionary all belong to class of collections and the treatment of all will be done in coming chapters.

1.9.4 Strings

See string literals.

1.9.5 Operators

They are discussed in chapter 2.

1.9.6 Special symbols

They are also known as separator and they are square brackets **[ ]** , braces **{ }** , parentheses **( )** etc. They **[ ]** used in array and known as subscript operator, the symbol **( )** is known as function symbol.

1.10 Variables

A variable is a named location in memory that is used to hold a value that can be modified in the program by the instruction. Python is a dynamic language so declaration of variables stating their types and name is not required. You can declare and initialize the variable right at the time of use. The type of variable is determined by the type of contents it is storing. Every variable has a class which can easily be seen using the type function. The id function lets you see the address of the variable in memory.

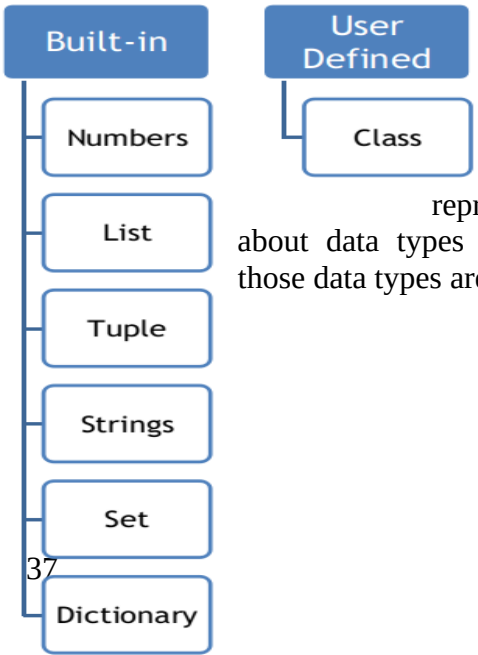
```
>>> x=10
>>> type(x)
<class 'int'>
```

```
>>> x=23.4
>>> type(x)
<class 'float'>
>>> x='python'
>>> type(x)
<class 'str'>
>>> x=True
>>> type(x)
<class 'bool'>
>>> x={3,4}
>>> type(x)
<class 'set'>
>>> x=[1,2,3]
>>> type(x)
<class 'list'>
>>> id(x)
1958436684552
>>> x=23
>>> id(x)
1781690528
```

As you can see from the above shell session variables are declared and initialized at the point of use and type of variable changes dynamically as initialized contents changes say from integer to string, or string to boolean. This can be easily understood using the **type** function. One more point regarding variables is the use of **id** function which tells the memory address of the variable. Further note that as you assign the same variable having different contents the **id** after new assignment also changes. This indicates that old variable cease to exist and new variable with the same name with new content is created.

1.11 Data Types

Python defines several data types which can be used under different programming situations like an **int** data type can be used to represent whole numbers as age of a person, roll number etc or **float** data type can be used to represent salary of a person, interest rate etc. The main thing to note about data types in python that they all are classes and variables/instances of those data types are objects.



The basic Python data types are as shown below:

1. **Built-in Types**
   (a) Numbers

(b) List

(c) Tuple

(d) Strings

(e) Set

(f) Dictionary

2. **User defined data types**

(a) class

1.11.1 Numbers

Python number types are int, float and complex. As we have seen in the Literals Section above they all are classes. Integers can be decimal, octal, hexadecimal and binary. Real values are just values with a decimal point and complex numbers are in pair of real and imaginary with a + operator joining them and suffix of "j" after imaginary part. An example through shell is illustrated:

```
>>> x=12
>>> type(x)
<class 'int'>
>>> y=3.45
>>> type(y)
<class 'float'>
>>>
x=342349954385394583953453405934059345845938459384530958349534503845930
5
>>> print(x)
342349954385394583953453405934059345845938459384530958349534503845930
5
>>> type(x)
<class 'int'>
>>> a=3+4.4j
>>> type(a)
<class 'complex'>
>>> print('type of',a,' is ',type(a))
```

```
type of (3+4.4j) is  <class 'complex'>
```

There is also a function: **isinstance** that can be used to check if an instance is of specific class or any of it subclass. The function is a part of default imported module that we will discuss later. The syntax of this can easily be seen using the help function as:

```
>>> help(isinstance)
Help on built-in function isinstance in module builtins:
isinstance(obj, class_or_tuple, /)
Return whether an object is an instance of a class or of a subclass
thereof.A tuple, as in ``isinstance(x, (A, B, ...))``, may be given as
the target to check against. This is equivalent to ``isinstance(x, A)
or isinstance(x, B) or ...`` etc.
```

Lets see how do we use it using some examples:

```
>>> x=10
>>> isinstance(x,int) # check x is of type int
True
>>> x=10.45
>>> isinstance(x,float)
True
>>> isinstance(x,int)
False
>>> x=10+4j
>>> isinstance(x,int)
False
>>> isinstance(x,complex)
True
>>> isinstance([3,4],list)
True
>>> x=10
>>> isinstance(x,(int,complex))
True
>>> x=10+5j
>>> isinstance(x,(int,complex))
True
```

As can be seen in the last two examples, instead of one single class type a tuple of class types can also be given. So if any of the class type matches then the result returned is True else False.

1.11.2 List

List is an ordered sequence of items of any type. Even another list can be a member of List. All list elements are enclosed within square brackets. Detailed discussion is in Chapter X. See some examples:

```
>>> L=[1,2,3,5]
>>> print(L)
[1, 2, 3, 5]
>>> len(L)
4
>>> L=['hello','this','is','example']
>>> print(L)
['hello', 'this', 'is', 'example']
>>> L=['hello',34.56,True,[4,5]]
>>> print(L)
['hello', 34.56, True, [4, 5]]
>>> type(L)
<class 'list'>
```

1.11.3 Strings

Strings are just sequence of characters enclosed within double or single quotes. Individual character of strings can be accessed using subscript operator [ ]. The first character is at 0 index and last at -1. Strings are immutable collections. See shell in action:

```
>>> s="hello";
>>> print(s)
hello
>>> s="hello"+" "+"python"
>>> print(s)
hello python
>>> "hi"*3
'hihihi'
>>> s[0]
'h'
>>> s[-1]
'n'
```

```
>>> s="'hello'"
>>> print(s)
'hello'
>>> type(s)
<class 'str'>
>>> s[0]='c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

String can also span multiple lines in shell. For continuation on next line "\" can be used or multiline string using triple double quotes or triple single quotes can be used. See shell in action:

```
>>> s='this \
... is \
... an example'
>>> print(s)
this is an example
>>> s
'this is an example'
>>> s='''hello
... this is
... an example
... '''
>>> s
'hello\nthis is\nan example\n'
>>> print(s)
hello
this is
an example
>>> s="""hello
... python"""
>>> s
'hello\npython'
>>> print(s)
hello
```

```
python
```

1.10.4 Set

Set is an unordered collection of various elements of any type. As we have in literal section, the elements are separated by comma and for opening and closing the set ,braces are used. The special point about set is that set stores only unique items. See shell in action.

```
>>> s={1,3,2,3,2,3}
>>> s
{1, 2, 3}
>>> print(s)
{1, 2, 3}
>>> type(s)
<class 'set'>
>>> s=set('aabrakadabra')
>>> print(s)
{'a', 'k', 'd', 'r', 'b'}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

1.10.5 Dictionary

Dictionary is an unordered collection where each item is a **key: value** pair. All the key value pair are put into curly braces. Each key is separated by value using colon. See shell in action:

```
>>> d={1:"one",2:'two'}
>>> d
{1: 'one', 2: 'two'}
>>> d[1]
'one'
>>> d[2]
'two'
>>> d[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
KeyError: 3
>>> d[3]='three'
>>> d
{1: 'one', 2: 'two', 3: 'three'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> d.values()
dict_values(['one', 'two', 'three'])
>>> d.items()
dict_items([(1, 'one'), (2, 'two'), (3, 'three')])
```

1.12 Indentation in Python

Python is not free form language like C/C++/Java. In a free form language there is no restriction on writing code in any column or any line. Blocks can easily be created using opening braces '{' and closed using closing braces '}'.

In Python indentation plays an important role. All the statements under one block must follow same indentation. Even a single space not matching indentation can create problem. The standard says 4 space characters for indentation for creating any block :**for ,if, else , any loop, function, class** etc. A block in python is created by colon(:). It will be clearer as you see some python code in coming chapters. For now, see some examples:

Table 1.4 : Some Indentation Examples

Indentation Example	Explanation
`print("First line")` ` print("second line")`	Second **print** statement is not indented. Must be aligned with first **prin**t statement. Gives error.
`if True:` `    print("its true")` `    print("Indented")` `else:` `    print("False")`	**if** and **else** are indented, **if** block is created because of **:** after **True** . The **print** statement within **if** block is having 4 space indentation. Tab can also be used for indentation. Similarly, **else** block is created and having one **prin**t statement indented by 4 space.
`if a>b:` `    if a>c:` `        print(a," is Max")` `    else:` `        print (c," is Max)` `else:`	An example of nested **if-else**. The **print** within inner **if** of outer **if** is indented with respect to inner **if**. If-else within outer **if** is indented with respected to outer **if**.

if b>c: **print(b," is Max")** **else:** **print (c," is Max)**	
i=1; **while i<=10:** **print("i=",i)** **i=i+1** **print("outside loop")**	First, second and last line are indented. The **while** loop creates a block and all statements within **while** loop is indented.

1.13 Python Comments

Comments are simple text which makes it easy to understand the source code. You as a coder in any programming language wish to provide some code to be commented so that after some time (in a month or two) it can give you idea about as what you did earlier. Further some portions of code can also be commented as not to be interpreted by python. It is a good programming practice to always document your code using comment.

In Python comments can easily be created by '#' before the start of the line like:

```
# Example of print function
print("Hello Python")
```

Multiline comments are not supported by python and you must use # in every line to make it a comment like:

```
# Example of print function
# use for display purpose
print ("Hello Python")
```

One way of achieving multiline comment is to use triple double quotes or triple single quotes. They serve special purpose while creating functions and will be illustrated when we study functions. They are used for creating multiline string as we have seen earlier in "Strings".

```
"""
Function print
Is used for
Printing on the screen"""
```

1.14 Python Statements

Python is very rich in types of statements. Python contains many types of statements that can be used in a variety of programming situations. All statements are part of the Python program which executes when the program

executes. You don't make use of all the statements in every program, but you use them as par the demand of the problem you are going to solve using Python.

A python statement is any instruction that can be executed by python interpreter. For example, the **print** function is a statement. When executed by python interpreter it displays output onto the screen. As another example: **x=10** is an assignment statement, on execution is sets the value of x as 10.

The various types of statements supported by Python are discussed below:

1. **Selection/Conditional/Decision Making Statement**- By default the program flows sequentially. These statements decide the flow of statements based on evaluation of results of conditions. Types of statements in this category are: **simple if, if-else , else if ladder.**

2. **Iteration/Looping Statement**– Looping statements used to run a block of statements repeatedly for a finite number of times. Thus, they form a loop. Examples of these types of statements are: **for** and **while** statements. Python also has **range** function that gives range of values. It is like loop generating values within range. Both first and second category comes under the name "control statements".
3. **Jump Statements** – These statements are used to make the flow of your statements from one section of program to other. Statements **break**, **continue** and **return** come under this category of statements.
4. **Expression Statement** - Any valid expression makes an expression statement consists of any operator, variable, literals which we will discuss in the next chapter. Assignment statement discussed above in some python code comes under Expression Statement.
5. **Block Statement** – Block is a group of statements which are bind together by using colon (**:**) as discussed in indentation section.
6. **Input Statements** – All statements which are responsible for taking input from keyboard or file are input statements. The **input** function discussed later is an example of input statement.
7. **Output Statements**- All statements which are responsible for displaying something onto screen or to the file are output statements. The function **print** is an example of output statements.
8. **Empty /null statement**: An empty statement does nothing but sometimes require in programming situations. One example is creating delay using loop or creating an empty class. The **pass** keyword act as an empty statement in python.

1.14.1 Multiline Statements

Statement terminator in python is new line character. But when you want to continue your statement over multiple line python provides a line continuation symbol '\'. It can be used as:

```
>>> x=10*3+5 \
... -3/4 \
... +5
>>> x
39.25
>>> x= (10*3+5
... -3/4
... +5)
```

```
>>> x
 39.25
```

In the first example line continuation character was used but in the second example the whole expression was split onto multiple lines but within parenthesis. This eliminated the need of line continuation character. *In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }.* **But remember this does not work dealing with strings.** See below:

```
>>> x=("this is
  File "<stdin>", line 1
    x=("this is
               ^
SyntaxError: EOL while scanning string literal
```

For multiline strings you can use triple double/single quotes as discussed earlier.

See one more example of multiline statements to wrap up this section:

```
>>> x=("string",
... "is",
... "fun"
... )
>>> x
('string', 'is', 'fun')
>>> x=['string','is','fun']
>>> x=['string',
... 'is','fun']
>>> x
['string', 'is', 'fun']
>>> x={'string',
... 'is','fun'
... }
>>> x
{'is', 'string', 'fun'}
```

1.15 The print function

In this section we are going to see number of examples of print functions. Some examples using scripts and some within python shell. Let's start with first example within shell.

Example 1 within shell

```
>>> print("one");print("two");print("three")
one
two
three
>>> print("one\ttwo\tthree")
one     two     three
>>> print("This is a long \
... example which is \
... spanning multiple \
... lines")
This is a long example which is spanning multiple lines
```

The code above shown in shell is easy to understand. Each **print** statement leaves a line by default after printing. *Multiple commands can be put on same line by separating them with semicolon.* The tab character has been used for printing contents on the same line. For printing long lines and respecting the width of the screen "\" can be used as a continuation character.

To see the various parameters for the **print** function we can simply use **help** function as:

```
>>> help(print)
Help on built-in function print in module builtins:
print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:a file-like object(stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

Keyword or named arguments means we can supply the values to these arguments with their name. The default value for **file** argument is **sys.stdout** ie. by default the output will appear on monitor or screen. The **sep** argument is separator and default value is space and **end** argument's default value is newline character. Let's understand them in shell:

```
>>> print("first");print("second")
first
second
>>> print("first",end="\t");print("second")
```

```
first   second
>>> print(10,12,13)
10 12 13
>>> print(10,12,13,sep=";")
10;12;13
>>> print(10,12,13,sep="\n")
10
12
13
```

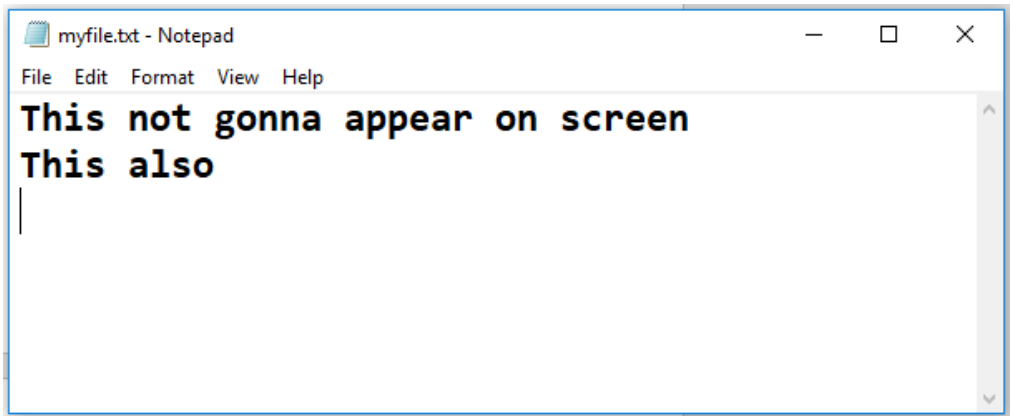
The concepts can easily be build upon just by seeing the output. Now lets understand the concept of file argument. See the next shell session :

```
>>>print("This not gonna appear on screen",file=open("myfile.txt","a"))
>>> print("This also",file=open("myfile.txt","a"))
>>> print("But this is")
But this is
```

The file argument opens a file by the name **"myfile.txt"** in append mode in the current working directory. Assuming we are in our directory **"mypython"**. This file will be saved in **mypython** directory. Append mode means if file does not exist it will be created and matter will be appended at the end.

As you can see from the shell that output of first two print statement does not appear on screen instead it goes into file **"myfile.txt"**. In the last print statement as we have not used the file keyword argument it appears on the screen.

See the output of the file **"myfile.txt"** in notepad:



Or you can just open the file and print its content (File handling will be covered in chapter XXX).

```
>>> for line in open("myfile.txt"):
...     print(line)
...
This not gonna appear on screen
This also
```

Don't forget to give 4 spaces after for loop in next line and press enter in last line.

For printing the values of variables of different types, the print function has different formats. Lets explore them in shell:

```
>>> name='Pari';age=23;
>>> print("Hello ",name," You are ",age, "years old")
Hello Pari  You are  23 years old
```

The first format for printing the variables is to simply concatenate them using comma and we have done for **name** and **age**.

Lets see the second way of doing the same :

```
>>> print("Hello {0} you are {1} years old". format(name,age))
Hello Pari you are 23 years old
>>> print("Hello {} you are {} years old". format(name,age))
Hello Pari you are 23 years old
```

In the above we make use of format method of string class. The variables to be printed are arguments to format and assigned positions starting from 0. Thus {0} in print function means displaying the value of name and {1} means displaying the value of age. When positional number is not mentioned variables are printed in order from left to right. See one more example:

```
>>> name='Pari';age=23
>>> print("Hello {} you are {} years old".format(name,age))
Hello Pari you are 23 years old
>>> print("Hello {} you are {} years old".format(age,name))
Hello 23 you are Pari years old
>>> print("Hello {1} you are {0} years old".format(age,name))
Hello Pari you are 23 years old
```

The other print option for variables is using format specifiers.

```
>>> print("Hello %s you are %d years old"%(name,age))
Hello Pari you are 23 years old
>>> print("Your salary is %f"%54000.0)
Your salary is 54000.000000
```

Here **%d, %s and %f** is known as format specifier and used for printing integer, string and float type of values.

In next way you can use named or keyword arguments in **print** function as:

```
>>>print("Hello {name} you are {age} years old".format(name='Pari',age=21))
Hello Pari you are 21 years old
```

Finally you can use string concatenation using + symbol. But here non string variables require to be converted into string using function 'str'.

```
>>> print("Hello "+name+" you are "+str(age)+" years old")
Hello Pari you are 23 years old
```

1.16 Reading input from user

To read any type of input from user you can make use of **input** function. The function returns a string read from standard input device i.e. keyboard. The input to the function is a string prompting user for entering input. See the signature of the function using help.

```
>>> help(input)
Help on built-in function input in module builtins:
input(prompt=None, /)
    Read a string from standard input.  The trailing newline is stripped.
    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.
    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise
EOFError.
    On *nix systems, readline is used if available.
```

The method says that after giving input we press Enter for returning and that newline character does not become part of the input. Even the **prompt** is None. It means we can leave the input function empty. See some examples in shell.

```
>>> name=input('Enter your name:')
Enter your name:Kuntal
>>> print('Hello ',name)
Hello  Kuntal
>>> name=input()
Jay
>>> print('Hello ',name)
Hello  Jay
```

In the first example of **input** the **prompt** is assigned the string ‘Enter your name’. In the second no string is passed and we just entered the text ‘Jay’..

One important point to note in **input** that it always return string ie.even in input your supply any integer, float, Boolean value it will take that as string. To prove my point lets see shell in action.

```
>>> x=input("Enter integer")
Enter integer10
>>> x+10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> x+str(10)
'1010'
>>> type(x)
<class 'str'>
```

As can be seen from the above shell session return type of **input** function is string. The first error is as we are trying to add integer with string. As **x** is string we convert 10 to string using **str** function. But output is again string ‘1010’ and finally the type of **x** returned is string.

1.17 Type Conversion

To convert the read item from **input** function to the type we want or in general for type conversion, python provides number of conversion functions. Lets start with **int** function.

```
>>> x='10'
>>> y=int(x)
>>> y+10
20
>>> x=input('Enter x')
Enter x10
>>> x=int(x)
>>> x=x+20
>>> x
30
>>> z=x+y
>>> z
40
```

In the above shell session you can easily understand that how int function has been used for converting a string to integer. See some more examples in next shell session:

```
>>> int(23.45)
23
>>> int(True)
1
>>> int(False)
0
>>> int(2+3j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to int
```

.

*The **int** function can also be used for converting a number from any other base to decimal base 10. See shell session:*

```
>>> int('123',8) # convert octal '123' to integer
83
>>> int('0b1010',2)
10
>>> int('0o123',8)
83
>>> int('1010',2)
10
>>> int('0x123',16)
291
>>> int('123',16)
291
```

The other function and their use are shown in the table given below:

Table 1.5: Examples of Type Conversions

>>> **float('23.34')** **23.34**	Converts string to float, try other types
>>> **str(23)** **'23'**	Converts integer to string, try other types as arguments.

>>> **complex(2)** **(2+0j)** >>> **complex(2,3)** **(2+3j)**	Creates complex number with one or two arguments. Try with zero argument.
>>> **list('python')** **['p', 'y', 't', 'h', 'o', 'n']**	Converts a string into list.
>>> **tuple('python')** **('p', 'y', 't', 'h', 'o', 'n')**	Converts a string into tuple.
>>> **set('python')** **{'o', 'h', 't', 'n', 'y', 'p'}**	Converts a string into unordered set with duplicates removed.
>>> **dict([[1,2],[3,4]])** **{1: 2, 3: 4}**	Creates a dictionary from a list of lists.
>>> **dict([(3,26),(4,44)])** **{3: 26, 4: 44}**	Creates a dictionary from list of tuples.

1.18 Scripting Examples

```
# Script 1.1 to read name from user and display
name=input('Enter your name\n')
print('Hello ',name)
OUTPUT:
Enter your name
Juhi
Hello  Juhi
```

The script is self-explanatory.

```
# Script 1.2 to read name, age and salary from user and display.
name=input('Enter your name\n')
age=int(input('Enter your age\n'))
salary=float(input('Enter your salary: '))
print('Hello ',name,' your age is ',age,' and salary is ',salary)
OUTPUT:
Enter your name
Navin
```

```
Enter your age
27
Enter your salary: 35000
Hello  Navin  your age is  27  and salary is  35000.0
```

Here we have taken three different input from user: **name** is of string type, **age** of integer type and **salary** of float type. The same are displayed back. *As we are not performing any operation on age and salary variables its ok to not type convert them into integer and float respectively but it is advisable to convert them into their respective types to avoid any unforeseen error or any ambiguous operation.*

Script 1.3 Reading multiple variables with single input function

```
name,age=input('Enter your name and  age\n').split()
print('Hello ',name)
print('After 10 years your age will be:',int(age)+10)
OUTPUT:
Enter your name and age
jatin 12
Hello jatin
After 10 years your age will be: 22
```

As we know **input** method returns a string, the string class has a method **split** which splits the input string on space(default separator). The input must be supplied as separated by space or tab. The splitted tokens are assigned to name and age.

Script 1.4 Number Conversion from decimal to other number system

```
num=int(input('Enter an integer:'))
print('Original number: ',num)
print('Octal equivalent: ',oct(num))
print('Hex equivalent: ',hex(num))
print('Binary equivalent: ',bin(num))
OUTPUT:
Enter an integer:125
Original number:  125
Octal equivalent:  0o175
Hex equivalent:  0x7d
Binary equivalent:  0b1111101
```

The script is easy to understand. We take a decimal integer from user and convert the same into octal, hex and binary using built in functions.

```
# Script 1.5 To swap two integer numbers
a,b=input('Enter two numbers:').split()
a=int(a)
b=int(b)
print('Before Swapping')
print('a=',a,'\tb=',b)
a,b=b,a
print('After Swapping')
print('a=',a,'\tb=',b)
OUTPUT:
Enter two numbers:12 34
Before Swapping
a= 12      b= 34
After Swapping
a= 34      b= 12
```

In other programming languages swapping operation is achieved as:

```
t=a;
a=b;
b=t;
```

where t is some temporary variable. But in python just one single line of code does this task: **a,b=b,a** without needing any temporary variable.

1.19 Why Learn Python?

1. A language that can be easily and quickly learn. If you are a first-time programmer within no time you'll learn python.
2. Developing web applications is quite easy as compared to other web development framework and languages.
3. The language allows you to *code quickly*, building complex applications with minimal lines of code as compare to C/C++/Java etc. That boost developer productivity.
4. Programs can be easily ported across different platforms.
5. Its takes minimal time and amount of code to go from Idea to implementation.
6. Python is powering scientific, machine learning, data science applications.
7. Python is in demand because of artificial intelligence, data science, internet of things, machine learning and deep learning.

8. Python code is easily readable even if its written by someone else. It's as simple as plain English.
9. Plenty of support libraries are available for numerical computing, game development, machine learning, deep learning, robotics, cryptography, networking and many more.
10. Easy integration with other languages either through software or hardware.
11. An expertise in python can fetch you a lucrative job !

1.20 Companies Using Python

Sr.No	Company	Uses
1.	Google	Server side official language, web search systems, Machine learning, deep learning
2.	Facebook	Services in infrastructure management, deep learning, machine learning
3.	YouTube	Major portion written in Python
4.	Instagram	Django framework for web development
5.	Raspberry Pi	Uses python for programming
6.	Industrial Light & Magic	In the production of animated movies
7.	Dropbox	Client and server software written in python
8.	Maya	API are in python
9.	NSA	Cryptography and intelligence analysis
10.	Netflix	Software infrastructure
11.	Microsoft	Machine learning and deep learning
12.	iRobot	Development of robotic devices
13.	Intel	Hardware testing
14.	Cisco	Hardware testing
15.	IBM	Hardware testing, machine learning and deep learning

1.21 Points to Ponder

1. Python is a general purpose and widely used high-level programming language created by **Guido van Rossum in 1991**

2. The name Python was adopted from a British comedy series "Monty Python's Flying Circus".

3. Python shell is an interactive command line interface for running python statements, commands, functions etc.
4. The default python shell prompt is **'>>>'** and known as primary shell prompt.
5. The secondary shell prompt is **'…'.**
6. The last executed command is stored in special variable known as **_** .
7. Python is a scripting language and python files has extension '.py'.
8. Smallest individual unit in a Python program is called Python Token. They are keywords, identifiers, operators, literals, strings, special symbols.
9. Python 3.6 has 33 keywords.

10. .Literals in Python refer to fixed values that do not change during the execution of a program. They are also known as nameless constants.
11. Python is a dynamic language so declaration of variables stating their types and name is not required.
12. Every variable has a class which can easily be seen using the type function and address which can be seen using id function.
13. Python has built-in data types as: Numbers, List, Tuple, Strings, Set, Dictionary
14. Python is not freeform language. Indentation is required for the creation of blocks.
15. Python supports single line comment using # symbol and multiline comment using triple double/single quotes.
16. A python statement is any instruction that can be executed by python interpreter.
17. For continuation of statements on multiple lines '\' can be used. line continuation is implied inside parentheses ( ), brackets [ ] and braces { }.
18. The syntax of the print function is :

    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

19.To read any type of input from user you can make use of input function. The function returns a string read from standard input device i.e. keyboard

2. Operators & Expressions

2.1 Introduction

For performing different kind of operations, various types of operators are required. An operator denotes an operation to be performed on some data that generates some value. For example, plus operator (+) on **2** and **3** generates **5.** Here **2** and **3** are called operands.

Python is very rich in built-in operators. The whole list of operators provided by Python is given in the table below:

Table 2.1: Operators in Python

S.N	Operator	Symbol Representation
1.	**Arithmetic**	**+ , - , / , * , %**
2.	**Logical**	**and, or, not**
3.	**Relational /Comparison**	**>, < , >= , <= , == , !=**
4.	**Assignment**	**=**
5.	**Bitwise**	**& , | , ^ , ! , >> , <<**
6.	**Membership**	in, not in
7.	**Identity**	**is, not is**

2.1.1 Binary Operators

All the operators which require two operands to operate on are known as binary operators. For example as shown in the above table all the arithmetic, relational, logical (except **not** operator) assignment, bitwise (except **~** operator) operators etc. are binary operators.

Examples: **2+4, 34>45, x=23, 2 and 5.**

2.1.2 Unary Operators

Unary operators are those operators which require only one operand to operate on are known as unary operators. For example as shown in the above table **not logical operator**, **~**(1's complement operator) etc. are unary operators. Python also provides unary plus and unary minus i.e. **+20** and **-34**. Here **+** and **–** are known as unary plus and unary minus operators.

2.2 Expressions

Operator together with operands constitutes an expression or a valid combination of constants, variables, and operators forms an expression. An expression is usually recognized by type of operator

used within the expression. Depending upon that you may have integer expression, floating point expression, relational expression etc. You may also have different types of operators in an expression which is a mixed mode expression. See the table given below for few examples.

Table 2.2: Types of Expressions

S.N.	Expression	Type of Expression
1.	**2+3*4/ 6-7**	**Integer Arithmetic**
2.	**2.3 * 4.5 /7.0**	**Real Arithmetic**
3.	**a>b!=c**	**Relational**
4.	**x and 10 or y**	**Logical**
5.	**2>3+x and y**	**Mixed (relational, arithmetic and logical)**

2.3 Arithmetic Operators

As given in the **Table 2.3** there are mainly 7 arithmetic operators in Python. They are **+, -, *, /,** // , **%** and ** which are used for addition, subtraction, multiplication, float division , integer division, remainder and power respectively. See the table given below:

Table 2.3 Arithmetic Operators

S.N	Operator	Example	Meaning
1.	**+**	x+y or +x	Perform simple addition; perform unary plus also.
2.	**-**	x-y or -x	Perform simple addition; perform unary plus also
3.	**/**	x/y	Divide x by y and return float value
4.	**//**	x // y	Divide x by y and return integer value
5.	*****	x * y	Perform simple multiplication
6.	**%**	x % y	Returns the remainder when x is divided by y. Works for both integer and float.
7.	******	x**y	Finds x to the power y. Multiplies x by itself y times.

Let's try all the above operators in python shell

```
>> x,y=23,5
>>> x+y
28
>>> x-y
```

```
18
>>> x*y
115
>>> x/y
4.6
>>> x//y
4
>>> x%y
3
>>> 2**y
32
>>>3.4%2
1.4
```

In the first line **x** and **y** are initialized to **23** and **5** respectively. This type of assignment is unique to python. We will explore this later in this chapter. The simple **/** symbol is **float** division and **//** is for integer division. The remainder operator works both with integer and floating-point numbers. Power operator ** takes two operands: base and exponent and returns base to the exponent: 2**3 gives **8**.

Let's have one more session to understand **/, //** and **%** operators in detail but this time we also provide explanation along with expression in tabular form.

>>> **-10//3** **-4** >>> **10//-3** **-4**	-10/3 gives -3.3333333333333335 and -10//3 is floor of -10/3 i.e smallest integer value of the result -10/3. As -4 is next smallest integer after -3 so is the answer.
>>> **3//10** **0** >>> **-3//10** **-1** >>> **3//-10** **-1**	-3/10 gives -0.3 and -3//10 is floor of -3/10 i.e smallest integer value of the result -3/10. As -1 is next smallest integer after -0.3 so is the answer.
>>> **10%3** **1** >>> **-10%3**	# Remainder Formula A%B=A-(A//B)*B -10%3= -10-(-10//3)*3

2 **>>> 10%-3** **-2**	= -10 –(-4)*3 = -10 +12 = 2 10%-3= 10-(10//-3)*-3 = 10 –(-4)*-3 = 10 -12 = -2

In the last example to remember what the output should be, perform normal remainder operation without -ve sign. Make the sign of the dividend as sign of the remainder and add remainder to the divisor.

For example, to tell quickly what the answer should be of 23%-5, just performing normal remainder operation (without sign) and remainder is 3. As 23 is +ve, remainder 3 remains 3. Now to get actual answer just add -5 to 3 and -2 is your answer. Similarly -23%5 remainder will be 3 and as sign of 23 is -ve , remainder 3 changes to -3 and adding this to 5 gives answer as 2.

Another example with some easy method: 16%-7 , just add some multiple of -7 which is more than 16 (in magnitude) which is 21 so adding -7*3=-21 to 16 gives us -5. Similarly, for -16%7 add 21 to -16 that gives us answer as 5.

2.3.1 The string operators (+ and *)

The arithmetic operator + and * has special use with strings. The + operator is used for string concatenation and * for creating copies of the strings. To use * operator with string just multiply string with an integer number. See small shell session:

```
>>> s1='hello'
>>> s2='world'
>>> s3=s1+' '+s2
>>> s3
'hello world'
>>> s3='hello'+' '+'world'
>>> s3
'hello world'
>>> 'cool'*3
'coolcoolcool'
>>> print('cool\n'*3)
cool
cool
cool
```

```
>>> print('*'*20);print('I LOVE PYTHON');print('*'*20)
********************
I LOVE PYTHON
********************

>>> x='@'
>>> x*10
'@@@@@@@@@@'
```

The above examples are quite easy to understand.

2.3.2 Scripting Examples

Let's write some python scripts which illustrate use of these operators.

```
# Script 2.1 Arithmetic operations on two numbers
n1=float(input('Enter first float number\n'))
n2=float(input('Enter second float number\n'))
sum=n1+n2
sub=n1-n2
mul=n1*n2
fdiv=n1/n2
idiv=n1//n2
print('sum=',sum)
print('sub=',sub)
print('mul=',mul)
print('integer div=',idiv)
print('float division=',fdiv)
```

OUTPUT:

```
Enter first float number
34
Enter second float number
5
sum= 39.0
sub= 29.0
mul= 170.0
integer div= 6.0
```

```
float division= 6.8
```

The program is easy to understand as all concepts have been cleared in the preceding section. Remember that division by zero is not allowed, if you do so run time error occurs:

```
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Instead of using separate variables for storing results of various operations, we can directly print them in **print** function as:

```
print('sum=',n1+n2)
print('sub=',n1-n2)
print('mul=',n1*n2)
print('integer div=',n1//n2)
print('float division=',n1/n2)
```

```
# Script 2.2: Finding quotient and remainder, given two
# number as input.
n1=float(input('Enter first float number\n'))
n2=float(input('Enter second float number\n'))
q=n1//n2
r=n1%n2
print('N1=',n1,'\tN2=',n2)
print("Quotient=",q,'\tRemainder=',r)
```

OUTPUT:

```
Enter first float number
23
Enter second float number
4
N1= 23.0     N2= 4.0
Quotient= 5.0      Remainder= 3.0
```

Two numbers have taken from user and script simply finds quotient and remainder when n1 is divided by n2.

```
# Script 2.3 to find area of a circle
r=float(input("Enter the radius\n"))
```

```
area=3.14*r**2
print('Radius=',r)
print('Area of Circle %10.6f'%area)
```

OUTPUT:

```
Enter the radius
2
Radius= 2.0
Area of Circle  12.560000
```

The script ask use to enter the radius. The area of circle is PI*r*r where value of PI=3.14. To display the answer in a width of 10 with 6 point after decimal, one width for decimal point and 3 before decimal point we have used format **%10.6f**, **f** for floating point numbers.

```
# Script 2.4 to find simple interest
p=float(input("Enter the principal \n"))
r=float(input("Enter the rate \n"))
t=float(input("Enter the years \n"))
si=p*r*t/100
print('Principal=',p)
print('Interest=',r)
print('Time=',t)
print('Simple Interest=',si)
```

OUTPUT:

```
Enter the principal
2000
Enter the rate
5
Enter the years
2
Principal= 2000.0
Interest= 5.0
Time= 2.0
Simple Interest= 200.0
```

The script has calculated the simple interest for given principal, rate of interest and time in years.

```
# Script 2.5 to find area of a triangle
base=float(input("Enter the base\n"))
height=float(input("Enter the height\n"))
```

```
area=0.5*base*height
print('base=',base)
print('height=',height)
print('Area of triangle is %6.2f'%area)
```

OUTPUT:

```
Enter the base
15
Enter the height
8
base= 15.0
height= 8.0
Area of triangle is 60.00
```

Area of a triangle is ½ * base * height. We have taken base and height from user and stored in variables base and height. Area is calculated using the formula and stored in area. The same is displayed using **print** statement.

2.4 Relational Operator

Table 2.4 Relational Operators

S.N	Operator	Example	Meaning / used for
1.	**>**	x > y	Returns True when value of x is greater than value of y else return False.
2.	**<**	x < y	Returns True when value of x is less than value of y else return False
3.	**>=**	x >= y	Returns True when value of x is greater than equal to value of y else return False
4.	**<=**	x <= y	Returns True when value of x is less than equal to value of y else return False
5.	**==**	x == y	Returns True when value of x is equal to value of y else return False
6.	**!=**	x != y	Returns True when value of x is not equal to the value of y else return False

All relational operators yield boolean values i.e. True or False. A true value is represented by True and has numerical value 1. A false is represented by False and has numerical value 0. In expression any non-zero value is termed as true value. See short shell session

```
>>> 2>3
```

```
False
>>> 10>5
True
>>> 3!=4
True
>>> 4>=5
False
>>> 5<=6
True
>>> 4==4
True
>>> True+True
2
>>> False+True
1
>>> x=True+False
>>> x
1
```

As you can see from the above shell session that output of all relational operators is either True or False. Further True and False can be used is any expression and their numerical value is used in the expression.

Let's see one simple example script that make use of all operators. We will see other examples in the coming chapters when we introduce several new concepts of python.

```
# Script 2.6 To demonstrate relational operators
a,b=6,14
print(a,'>',b,'is',a>b)
print(a,'>=',b,'is',a>=b)
print(a,'<',b,'is',a<b)
print(a,'<=',b,'is',a<=b)
print(a,'==',b,'is',a==b)
print(a,'!=',b,'is',a!=b)
```

OUTPUT:

```
6 > 14 is False
6 >= 14 is False
6 < 14 is True
```

```
6 <= 14 is True
6 == 14 is False
6 != 14 is True
```

2.5 Logical Operators

Logical operators are used to check logical relation between two expressions. Depending upon the truth or falsehood of the expression they are assigned value **True** and **False.** The expressions may be variables, constants, functions etc. See the table given below:

Table 2.5: Logical Operators

S.N	Symbol	Example	Meaning
1.	**and**	**x and y**	Returns true when both operands are true else return False
2.	**Or**	**x or y**	Returns true when either or both operands are true else return False
3.	**Not**	**not x**	Returns complement of the operand (true to false or false to true)

The **and** and **or** are binary operators. For **and** to return true value both of its operand must yield true value. For **or** to yield true value at least one of the operands yield true value. The **not** operator is a unary operator. It negates its operand i.e. if operand is true it converts it into false and vice versa.

2.5.1 Logical and

The operator works with two operands which may be any expression, variable, constant or function. It checks both of its operand returns true value or not. If they it returns True value. If either of its operand is false, a False value is returned.

Table 2.5 **Truth Table of Logical and**

Operand 1	Operand 2	Returned Value
False	**False**	**False**
False	**True**	**False**
True	**False**	**False**
True	**True**	**True**

Script	Explanation
`a=10;b=20;`	***As discussed above*** the **and** is called **AND** operator. On both side of this

`res=(a>=10 and b==20)` `print("returned value in res=",res)` `OUTPUT:` `returned value in res=True`	operator condition is specified. If both conditions are true the returned value is True else False value is returned. In the above program both the conditions in the expression are true so **res** contains ***True*** as result
`a=0;b=2;` `res=(a!=0 and b<=2)` `print("returned value in res=",res)` `OUTPUT:` `returned value in res=False`	Simply changed the values of **a** and **b** and first operand is false so False is returned.

2.5.2 Logical or

The operator works with two operands which may be any expression, variable, constant or function. It checks any of its operand returns true value or not. If any operator is True it returns True value (i.e. a decimal 1). If both of its operand is false, a False value is returned (i.e. a decimal 0).

Table 2.6 Truth Table Of OR

Operand 1	Operand 2	Returned Value
False	False	False
False	True	True
True	False	True
True	True	True

Script	Explanation
`a=100;b=120;` `res=(a>=100 or b<0)` `print("returned value in res=",res)` `OUTPUT:` `returned value in res=True`	***As discussed above*** the or is called OR operator. On both side of this operator condition is specified. If either of the condition is true the returned value is True. False value is returned when both operands are false. In the script first condition is true but second is false so **res** contains ***True*** as result
`a=0;b=2;` `res=(a!=0 or b<=2)` `print("returned value in res=",res)` `OUTPUT:`	Simply changed the values of a and b and first operand is false but second is true so True is returned.

returned value in res=True	

2.5.3 Logical NOT (!)

The operator converts a true value into false and vice versa. Again, the operand may be any expression, constant, variable or function.

Table 2.7 Truth Table of NOT

Operand	Returned Value
False	**True**
True	**False**

See a small shell session to understand **not** operator

```
>>> x=12
>>> not x
False
>>> y=not x==12
>>> y
False
>>> a=not(x>10 and y==0)
>>> a
False
```

2.5.4 Short circuit operators

Before we delve into the discussion of what short circuiting is all about let's understand one simple concept. Writing logical expression with any operand without relational operator defaults to! = operator. It means that writing: **x and y>20** interprets to **x! =0 and y>20.** As another example writing **x and y** means **x! =0 and y! =0**. In those expressions instead of True or False the numerical values are returned.

In case of logical operator **and** and **or ,** if the left operand yields false value, the right operand is not evaluated by a compiler in a logical expression using **and**. If the left operand yields true value, the right operand is not evaluated by the compiler in a logical expression with the operator **or.** The operators **and** and **or** have left to right associativity, hence the left operand is evaluated first and based on the output, the right operand may or may not be evaluated. As an example, consider the following expression:

(10>=15) and (5!=4)

The left operand of **and** i.e **(10>=15)** is false so right operand i.e (**5!=4)** is not evaluated.

If it were **or** in the above expression in place of **and** , second operand would have been checked.

Let's see some examples:

Sr.No.	Shell Expression	Explanation
1.	>>> x, y=10,20 >>> x and y 20	As **x!=0** second operand is checked and as **y!=0** value of y is returned
2.	>>> x, y=10,0 >>> x and y 0	As **x!=0** second operand is checked and as **y!=0** value of y is returned
3.	>>> x, y=0,20 >>> x and y 0	As **x!=0** is false second operand is not checked because of short circuiting value of **x** is returned
4.	**>>> x,y=10,20** **>>> x or y** **10**	As **x!=0** is true and logical operator is **or** so second condition is not evaluated and answer is 10, value of x.
5.	**>>> x,y=0,20** **>>> x or y** **20**	As **x!=0** is false second operand is evaluated and 20 is returned.
6.	**>>> 2==(2 and 3)** **False**	**2 and 3** returns 3 and **2==3** is False
7.	**>>> 2==(2 or 3)** **True**	2 or 3 returns 2 and 2==2 is True
8.	**>>> 'x'==('x' and 'y')** **False**	**'x' and 'y'** returns 'y' which is not equal to 'x' so False is returned
9.	**>>> 'x'==('x' or 'y')** **False**	'x' or 'y' returns 'x' which is equal to 'x' so True is returned

2.5 Assignment Operator

The = operator is called assignment operator. We have seen several instances of this operator in many of the earlier programs. The new thing about assignment in python is that it supports parallel assignment. See number of examples as illustrated through python shell:

Shell Expressions	Explanation
`>>> a,b=10,20`	Assigning 10 to a and 20 to b
`>>> a,b=(10,20)`	Another way of previous one
`>>> a,b,c="CAT"`	"CAT" is unpacked , a='C', b='A',c='T'
`>>>a,b,c,d="CAT"`	Error not enough values to unpack
`>>>a,b='CAT'`	Error not enough values to unpack
`>>>x,y,z=10,20,30` `>>>x,y,z=z,z+y,y+x`	Second line is equal to three parallel assignment: x=z; y=z+y;z=y+x ; and x,y,z stores values 30,50,30. Remember new assignments are not immediately visible unless all assignments are executed.
`>>>x,y,z=[1,2,3]`	List elements are unpacked and assigned to x, y and z respectively
`>>>str="python"` `>>>a,b,c=str[0],str[1],str[2:]`	a='p',b='y' and c='thon'

One more use of this operator is the shortening of following types of expressions:

```
x=x+1,      y=y*(x-4),      a=a/10 ,   t=t%10
```

In all the above expressions the variable on both side of = operator is same, so we can change the above expression in shorter form as follows:

```
x=x+1       =>    x+=1
y=y*(x-4)   =>    y*=x-4
a=a/10      =>    a/=10
t=t%10      =>    t%=10
p=p-3       =>    p-=3
```

This form **op=** where operator may be any operator is called **compound operator** or **shorthand assignment operator.**

2.6 Bitwise operators

They are called so because they operate on bits. They can be used for the manipulation of bits. All these

operators are extensively used when interfacing with the hardware and for settings of bits in registers of the device. Python provides total 6 types of bitwise operators. They are as follows:

Table 2.8 Bitwise Operators

S.N	Operator	Example	Meaning / used for
1.	**&**	**x & y**	Perform **and** operation on bits of x and y
2.	**|**	**x | y**	Perform or operation on bits of x and y
3.	^	**x ^ y**	Perform **xor** operation on bits of x and y
4.	**~**	**~x**	Perform negation on bits on x
5.	>>	**x>>n**	Shift bits of **x** by **n** positions towards right
6.	<<	**x<<n**	Shift bits of **x** by **n** positions towards left

For all the following scripts/shell session we consider only first **8** bits of the number for explanation purpose. So, range of possible numbers is **-127 to 128** As an example decimal 10 can be written in **8** bits as **00001010.**

2.6.1 Bitwise AND (&)

It takes two bits as operand and returns the value **1** if both are **1**. If either of them is **0**, the result is **0**.

Table 2.9 Truth Table Of Bitwise AND

First Bit	Second Bit	Result
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

Let's take a simple example**.** Binary values of **a=2 is 0010** and **b=3 is 0011.**

Bitwise AND of these two values is performed as follows:

0 0 1 0

0 0 1 1

----

0 0 1 0 (output will be 2 in decimal)

----

if both bit are **1** output bit will be **1** using **&** operator else **0**.

See execution in python shell.

```
>>> a,b=2,3
>>> a&b
2
>>> bin(a)
'0b10'
>>> bin(b)
'0b11'
>>> bin(a&b)
'0b10'
```

2.6.1.1 Masking using Bitwise AND

The **AND** operator is used for **masking** purpose. For example we have a binary number 1010***1101*** which is 173 in decimal integer. We want to preserve the right most 4 bits (shown in italics) and make the remaining bits to zero. For this purpose we choose a binary number **00001111**. **We choose one's (1) for those bits which we want to preserve and zero for which we do not want to preserve.** Then we perform Bitwise **AND** operation of these two binary numbers. That is

1 0 1 0 *1 1 0 1* (Original Number)

0 0 0 0 *1 1 1 1* (Mask)

----

0 0 0 0 1 1 0 1

This is known as masking. The number together with we perform Bitwise **AND** is known as Mask. As another example say you want to preserve bit number **0,1,4,6** (0^{th} bit is rightmost). So mask will be **01010101.**

1 0 1 0 1 1 0 1 (Original Number)

0 1 0 1 0 1 0 1 (Mask)

----

0 0 0 0 0 1 0 1

See the shell execution given below:

```
>>> a=173
>>> b=15
```

```
>>> c=a&b
>>> c
13
>>> bin(a)
'0b10101101'
>>> bin(b)
'0b1111'
>>> bin(c)
'0b1101'
```

Use of format function

You might have noticed that output of **bin** function is limited in number of bits. It displays number in binary depending upon actual number of bits needed. For example, if we want in the previous masking example, to display 15 as: '0b00001111' is not possible. The **bin** function does not display preceding zeros. To get the binary pattern in the desired format we can make use of **format** function. See one example as:

```
>>> a=15
>>> format(15,'#010b')
'0b00001111'
>>> format(15,'#08b')
'0b001111'
```

In the **format** function first argument is the number whose binary we want. Second argument is interpreted as: The `#` makes the format include the `0b` prefix, and the `010` size formats the output to fit in 10 characters width, with `0` padding; 2 characters for the `0b` prefix, the other 8 for the binary digits. We have presented two examples. See in the second example the preceding 0s are just two.

You can try the format function for displaying the binary number in any desired pattern you want.

2. 6.2 Bitwise OR (|)

It takes two bits as operand and returns the value **1** if at least one bit is **1**. If both are **0** only then result is **0** else it is **1**.

Table 2.10 Truth Table Of Bitwise OR

First Bit	Second Bit	Result
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

Let's take a simple example Binary values of **a=12 is 1100** and **b=7 is 0111.**

OR of these two values is performed as follows:

1 1 0 0

0 1 1 1

----

1 1 1 1 (output in c will be 15 in decimal) .

If any of the bit is **1** output will be **1** using **OR** operator.

Take a simple example in shell :

```
>>> a,b=12,7
>>> c=a|b
>>> format(a,'#010b')
'0b00001100'
>>> format(b,'#010b')
'0b00000111'
>>> format(c,'#010b')
'0b00001111'
```

2.6.3. Bitwise XOR (^)

This operator takes at least two bits (may be more than two). If number of 1**'s** are odd then result is **1** else result is **0.**

Table 2.12 Truth Table Of Bitwise XOR

First Bit	Second Bit	Result
0	0	0
0	1	1
1	0	1
1	1	0

Let's take a simple example. Binary values of **a=5 is 0101** and **b=6 is 0110.**

XOR of these two values is performed as follows:

0 1 0 1

0 1 1 0

----

0 0 1 1 (output will be 3 in decimal)

If odd number of **1***'s* are there output will be one otherwise output will be **0** using **XOR** operator

See the execution in shell script.

```
>>> a,b=5,6
>>> c=a^b
>>> format(a,'#06b')
'0b0101'
>>> format(b,'#06b')
'0b0110'
>>> format(c,'#06b')
'0b0011'
```

Though python provides swapping of two numbers without using third number as: a,b,=b,a; but the bitwise operator ^ can also be used for achieving the same. See a python script:

```
# Script 2.7 to swap two numbers using XOR (^) operator
a=int(input("Enter first number"))
b=int(input("Enter second number"))
print('Before swapping\n')
print('a=',a,'\tb=b',b)
# logic Starts
a=a^b
b=a^b
a=a^b
# logic ends
print('After swapping\n')
print('a=',a,'\tb=',b)
```

OUTPUT:

```
Enter first number10
Enter second number20
Before swapping
a= 10   b=b 20
After swapping
a= 20   b= 10
```

The code is quite easy to understand. The reader is encouraged to try the 3 line logic code for swapping using dry run.

2.6.4. 1's Complement (~)

The symbol (**~**) denotes one's complement. It is a unary operator and complements the bits in its operand i.e. **1** is converted to **0** and **0** is converted to **1.**

Let's see an example in shell and understand:

```
>>> b=12
>>> c=~b
>>> format(b,'#010b')
'0b00001100'
>>> format(c,'#010b')
'-0b0001101'
>>> c
-13
```

Binary value of **b=12 is 00001100** in 8 bit representation.

In one's complement we invert the bit values i.e **0** is inverted into **1** and vice-versa. So the binary output will be:

c=11110011

But internally the computer represents the negative number in 2's complement so this number in c we convert to 2's complement form. For this we leave the first 1 from right side and complement all other bits:

00001101

This value is 13 and because of leftmost bit was 1 before performing 2's complement, the number is -13 which is what ~ operators returns.

Lets take one more example to understand:

binary value of 6 → 00000110

1's complement of 6 →11111001 (sign bit is left most ,0:+ve,1:-ve)

2's complement of 6 → 00000111

As the sign bit was 1 in 1's complement, the answer will be negative i.e -7.

2.6.5. Left Shift Operator (<<)

The operator is used to shift the bits of its operand. It is written as **x<<num;** which means shifting the bits of **x** towards left by **num** number of times. A new zero is entered in the Least Significant Bit (LSB) position. See execution in shell followed by explanation.

```
>>> a=2
>>> b=a<<1
>>> b
4
>>> format(a,'#06b')
'0b0010'
>>> format(b,'#06b')
'0b0100'
```

a<<1 means shifting the contents of **a** towards left by **1** position. If we represent the number **2** in binary as:

b3	**b2**	**b1**	**b0**
0	**0**	**1**	**0**

where **b0** is the first bit from right which is called **Least Significant Bit(LSB)** and **b3** is called **Most significant Bit(MSB)**.Shifting left by **1 bit** position results in **b3** losing its value and taking from **b2,b2** getting from **b1** and **b1** from **b0**, a new zero is inserted at **b0**.So the resultant bit pattern will be:

b3	**b2**	**b1**	**b0**
0	**1**	**0**	**0**

Which is **4** in decimal .

Now instead of writing **a<<1** if we write **a<<2** it means shifting the contents of **a** twice towards left.

Original value in **a=0010**

b3	**b2**	**b1**	**b0**
0	**0**	**1**	**0**

Shifting once

b3	**b2**	**b1**	**b0**
0	**1**	**0**	**0**

(4 in decimal)

Shifting the value obtained in first step

b3	**b2**	**b1**	**b0**
1	**0**	**0**	**0**

Which is *8* in decimal and this is the final output

We have given just two examples of +ve numbers. The operators << equally work well with negative numbers also.

```
>>> a=-10
>>> b=a<<2
>>> b
-40
>>> format(a,'#010b')
'-0b0001010'
>>> format(b,'#010b')
'-0b0101000'
```

NOTE: shifting the contents of a value val by num times means multiplying the value val by 2^{num} times. For example, 3<<5 means 3 * 2^5 and answer will be 3*32=96.

2.6.6. Right Shift Operator (>>)

The operator is used to shift the bits of its operand. It is written as **x>>num;** which means shifting the bits of **x** towards right by **num** number of times. A new binary bit zero/one is entered in the Most Significant Bit (MSB) position dependent upon sign of the number. If the number is +ve a binary zero bit is entered else binary one bit is entered.

```
>>> a=8
>>> b=a>>1
>>> b
4
>>> format(a,'#08b')
```

```
'0b001000'
>>> format(b,'#08b')
'0b000100'
```

a>>1 means shifting the contents of **a** towards right by **1** bit position. If we represent the number **8** in binary as:

b3	**b2**	**b1**	**b0**
1	**0**	**0**	**0**

Shifting right by **1 bit** position results in **b0** losing its value and taking from **b1, b1** getting from **b2** and **b2** from **b1**, a new zero is inserted in **b3 as number 8 is +ve**. So the resultant bit pattern will be:

b3	**b2**	**b1**	**b0**
0	**1**	**0**	**0**

In another example we shift the contents of a towards right by 2 bit positions. a>>2 means shifting the contents of *a* twice towards right.

Original value in **a=1000**

b3	**b2**	**b1**	**b0**
1	**0**	**0**	**0**

Shifting once

b3	**b2**	**b1**	**b0**
0	**1**	**0**	**0**

(4 in decimal)

Shifting the value obtained in first step

b3	**b2**	**b1**	**b0**
0	**0**	**1**	**0**

Which is *2* in decimal and is the final output.

```
>>> a=8
>>> b=a>>2
>>> b
2
>>> format(a,'#08b')
```

```
'0b001000'
>>> format(b,'#010b')
'0b00000010'
```

Let's see now one example of negative number.

```
>>> a=-10
>>> b=a>>1
>>> b
-5
>>> format(a,'#010b')
'-0b0001010'
>>> format(b,'#010b')
'-0b0000101'
```

For negative numbers the sign bit is 1 and a 1 is inserted from right during shifting. Now before wrapping up this section let's see what result we get for on odd number:

```
>>> a=9
>>> b=a>>1
>>> b
4
>>> format(a,'#08b')
'0b001001'
>>> format(b,'#010b')
'0b00000100'
>>> a=-9
>>> b=a>>1
>>> b
-5
>>> format(a,'#08b')
'-0b01001'
>>> format(b,'#08b')
'-0b00101'
```

For odd numbers floor of the output is taken i.e. smallest integer less than the result of a/2. For 4.5 it becomes 4 and for -4.5 it becomes -5.

2.7 Membership Operator

Python supports two membership operators: **in** and **not in**. They can be used to check whether an element is a member of any of the collection/sequence: list, string, dictionary, set etc.

Table 2.12 : Membership Operators

Operator	Syntax	Remarks	Example
in	element **in** sequence	Return True/False if element is in/not in sequence	'a' in 'hola' returns True
not in	element **not in** sequence	Return True/False if element is not in/in sequence	'a' not in 'hello' returns True

Let's see some examples in python shell:

```
>>> 'v' in 'victory'
True
>>> x="Hello"
>>> 'H' in x
True
>>> 'h' in x
False
>>> L= [1,2,3,4]
>>> x in L
False
>>> 2 in L
True
>>> d={1:'Pari',2:'chinu'}
>>> 1 in d
True
>>> 'Pari' in d
False
>>> 3 not in d
True
>>> x=set('python')
>>> x
{'o', 'h', 'y', 'n', 't', 'p'}
>>> 'c' in x
False
```

```
>>> 'c' not in x
True
```

Most of the examples in shell above are easy to understand. Just one explanation: for dictionary items membership operators **in** and **not in** compares with keys and not with values. In the example above as 1 is a key in dictionary **d** the operator **in** returns **True** but 'Pari' is a value so despite its presence in dictionary **d** the membership operator **in** returns **False**.

2.8 Identity Operator

The identity operator **is** and **not is** are used to check similarity of the two operands. By similarity means pointing to the same objects in the memory. This contrasts with == comparison operators which checks for values of its operands instead of memory locations.

Table 2.13 : Identity Operators

Operator	Syntax	Remarks	Example
is	opr1 **is** opr2	Return True/False if opr1 is same as opr2 (same object)	x=10;y=10; **x is y** returns True
is not	opr1 **is not** opr2	Return True/False if opr1 is not same as opr2 (same object)	x=10;y=12; **x is not y** returns True

Let's see some examples:

```
>>> x=10
>>> id(x)
1360162560
>>> id(10)
1360162560
>>> x is 10
True
```

In the above the location of x and 10 is same as clearly seen by the address returned by id function. That's why the expression: **x is 10** returns **True**. The same is true for strings also i.e.

```
>>> x='hell';y='hell'
>>> x is y
True
```

Try with y=10 along with the above and find out their ids.

```
>>> y=2.5
>>> y is 2.5
False
>>> y==2.5
```

```
True
>>> id(2.5)
2842696487608
>>> id(y)
2842696487560
```

The previous discussion does not apply to floating point numbers

```
>>> L=[1,2,3]
>>> [1,2,3] is L
False
>>> L1=[1,2,3]
>>> L is L1
False
>>> L2=L1
>>> L1 is L2
True
>>> y is not 2.5
True
```

In list also id of [1,2,3] is different though content of L is same. Even though content of L and L1 are same they do not point to same memory location.

Let's see some more examples to conclude this section.

```
>>> x=10
>>> x is int
False
>>> type(x) is int
True
>>> x=23.45
>>> type(x) is float
True
>>> L=[1,2,3]
>>> type(L) is list
True
>>> d={1:'aa'}
>>> type(d)
<class 'dict'>
>>> type(d) is dict
```

```
True
```

Important point to understand from above shell session is the use of **type** function to check type of any given variable. It is sometime useful to find out the type of the variable dynamically as python is dynamic type language. The use of **is** operator can be quite handy in those situations.

2.9 Precedence of Operators

Table 2.13: Precedence of Operators

Operator	Description
`()`	Parentheses (grouping)
f`(args...)`	Function call
x`[index:index]`	Slicing
x`[index]`	Subscription
x.attribute	Attribute reference
`**`	Exponentiation
`~`*x*	Bitwise not
`+`*x*`, -`*x*	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not` *x*	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression

Precedence tells in an expression which operation should be performed first depending upon priority of operators. Associativity means when two or more operators have same priority then from which side (left or right) we operate For example if we write **a * b / c** then in this expression ***** and **/** has got same priority so whether we perform **a * b** first or **b / c** first or in other way we should consider **b** as part of sub-expression **b / c** or part of sub-expression **a * b.** In python except assignment operator (right to left) all other operators have associativity from left to right.

As we know associativity of ***** and **/** is from left to right so sub-expression **a * b** will be performed first then the result of **a * b** will be divided by **c.**

See some of the examples scripts with explanations in table given below:

Sr.No	Script	Explanation
1.	`a=2;b=3;c=4` `d=a*4//b+c-10%3` `print('d=',d)` `OUTPUT:` `d=5`	As priority of say **opr1 (* , / , %)** is higher than say **opr2 (+ ,-).,**so expressions involving **opr1** are evaluated first . Again at the same level the associativity of **opr1** is from left to right, so evaluation will proceed as: *d= 2 * 4 // 3 + 4 – 10 % 3* *= 8 // 3 + 4 – 10%3* *= 2 + 4 – 10%3* *= 2 + 4 -1* *= 6-1* *= 5 .*
2.	`a=12;b=3;c=4` `d=(a-b)/c` `print('d=',d)` `OUTPUT:` `D=2.25`	We want to evaluate difference of **a** and **b** divided by **c.** For that we must write **(a - b) / c.** As precedence of **/** is more than – if we write **a – b / c** then initially **b/c** will be evaluated that we don't want. So whenever we want to evaluate an expression irrespective of the priority of the operator we write the expression within parenthesis.
3.	`a=4;b=4;c=2` `d=a * 2 and b > c - (not c)` `print('d=',d)` `OUTPUT:` `d=True`	First **not c** is evaluated so **c** becomes False(0). The rest of the expression is evaluated as: *= (4 * 2) and 4>3 – 0* *= 8 and 4 >(3 – 0)* *= 8 and (4>3)* *= True and True*

		= True In relational operator expression will be either true or false**. 1** denotes True and **0** denotes false. In logical operator operands are compared with **0.** If they are nonzero then condition is interpreted as true. In the above case as **4>3** is true so **1** is assumed at this place. In the next step **8** is nonzero as well as **1** is nonzero. As both conditions of logical and are true, the whole expression is true and a **True** is assigned to **d**.
4.	`a=2;b=4;c=3` `x=a**2>10+b%c*3` `print("x=",x)` `OUTPUT:` `x=False`	The expression is evaluated as: x=2**2>10+4%3*3 x=4>10+4%9 x=4>10+4 x=4>14 x=False
5.	`a=10;b=1;c=5` `x=a>>1-b+c%3<<2*c` `print("x=",x)` `OUTPUT:` `x=2048`	The expression is evaluated as: x=10>>1-1+5%3<<2*5 x=10>>1-1+2<<10 x=10>>0+2<<10 x=10>>2<<10 x=2<<10 x=2048

In the precedence table not all the operators we have discussed yet. The same will be explored in the coming chapters. But the table illustrating precedence of operators with the examples above must have given you good confidence about solving expression involving mix of operators.

2.10 Rvalue and Lvalue

The Python interpreter classifies all data types into two categories: One is **Lvalue** and second is **Rvalue.**

Lvalue is an expression that refers to an object that can be examined as well as altered. The **Lvalue** denotes an object that is the address of the data object. **Lvalue** has got its name because of values appearing on the left-hand side of assignment These values are something which can store values thus all objects which are mutable / modifiable are **Lvalue**. Examples are list, sets and dictionaries.

The second category **Rvalue** permits examination but not alteration. The **Rvalue** is the value residing in the address.That is, the python assigns the read only storage to objects under this category. All non-modifiable objects are **Rvalue** in python. They are also known as immutable. The examples of this category are: Constants/literals, Function names, int, float, complex, string, tuple

See some small examples:

```
>>> x=10
>>> y=20
>>> 20=x
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

In the above case values 10 and 20 are r-value. The variables **x** and **y** are storing the values 10 and 20 so they are l-value but last assignment gives error as you cannot assign x to an r-value 20.

Consider the following statements:

```
x=5
y=x
```

In the first assignment statement, x is an **Lvalue**. Hence , **x** is treated as a name for a particular memory location and the value 5 is stored in that memory location, which is **Rvalue**. In the second assignment statement, x is not an **Lvalue** and hence the value stored in the memory location is referred to by **x**.

But note that x is an integer and are not mutable. Let's check this in python session:

```
>>> x=10
>>> id(x)
1358917376
>>> x=x+1
>>> id(x)
1358917408
```

After writing **x=x+1** a new **x** is created as integer does not support modification.

As another example:

```
>>> x='example'
>>> x[0]='C'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Here in the above case 'C' is R-value and **x[0]** is also a R-value as it cannot be modified. To assign 'C' to the left of '=' operator we need Lvalue which x[0] is not.

List are mutable, and this can be easily seen using an example below:

```
>>> L=[1,2,3,4]
>>> id(L)
1903700100040
>>> x=L
>>> x[0]=90
>>> L
[90, 2, 3, 4]
>>> id(x)
1903700100040
```

In the above a simple List containing 4 integers is created. When you write x=L you are creating a reference to L and changing any element through x reflected in original L. Thus by writing x[0]=90 you changes first element of list L to 90 !. Here x[0] is an Lvalue that can be modified and 90 is Rvalue. Even **id** functions give you same object location.

2.11 The math module

A module is a python file that comprises functions and classes. All the functions we have seen so far belong to the default module: **builtins**. The detailed discussion of modules will be done in a separate chapter later. Here we discuss various mathematical functions of **math** module. To use a module, you need to import the module using **import** keyword as:

```
import math
```

Once imported you can have access to all the functions of this module as: ***math.funcname***

Here *funcname* is any function name. To see all the attributes and functions of math module just type ***dir(math)*** after importing math module. See the shell session below:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> len(dir(math))
54
```

The **dir** function returns a list of all attributes and functions in the math module. As you can starting from **acos** and excluding e and pi there are around 47 functions which can be used in various mathematical situations. For detail on every function you can make use of help function or refer python documentation.

```
>>> help(math.exp)
```

```
Help on built-in function exp in module math:
exp(...)
    exp(x)
    Return e raised to the power of x.
```

Here we discuss some of the commonly used math functions with examples

Table 2.14: Examples of math functions

Sr.No	Function	Description	Example
1.	ceil(x)	Return the ceiling of x as an Integral.This is the smallest integer >= x	>>>math.ceil(3.4) 4 >>>math.ceil(-3.4) -3
2.	floor(x)	Return the floor of x as an Integral.This is the largest integer <= x.	>>> math.floor(3.4) 3 >>> math.floor(-3.4) -4
3.	fabs(x)	Return the absolute value of the float x.	>>> math.fabs(-34) 34.0 >>> math.fabs(34) 34.0
4.	factorial(x)	Return factorial of input value Raise error if x is negative or non-integral	>>> math.factorial(5) 120 >>> math.factorial(-1) Traceback (most recent call last): File "<stdin>", line 1, in <module> ValueError: factorial() not defined for negative values
5.	trunc(x)	Truncates x to the nearest Integral toward 0	>>> math.trunc(3.45) 3 >>> math.trunc(-3.45) -3
6.	exp(x)	Return e raised to the power of	>>> math.exp(2)

		x.	7.38905609893065 >>> math.exp(-2) 0.135335283236612 7
7.	pow(x,y)	Return x**y (x to the power of y)	>>> math.pow(2,3) 8.0 >>> math.pow(2,-3) 0.125
8.	sqrt(x)	Return the square root of x.	>>> math.sqrt(5) 2.23606797749979 >>> math.sqrt(-5) Traceback (most recent call last): File "<stdin>", line 1, in <module> ValueError: math domain error
9.	gcd(a,b)	greatest common divisor of x and y	>>> math.gcd(99,78) 3 >>> math.gcd(97,11) 1
10.	radians(x)	Convert angle x from degrees to radians	>>> math.radians(30) 0.523598775598298 8
11.	sin(x)	Return the sine of x (measured in radians)	>>> x=math.radians(30) >>> math.sin(x) 0.499999999999999 94
12.	log2(x)	Return the base 2 logarithm of x	>>> math.log2(8) 3.0 >>> math.log2(32) 5.0
13.	log10(x)	Return the base 10	>>> math.log10(10)

		logarithm of x	1.0 >>> math.log10(100) 2.0
14.	log(x[, base])	Return the logarithm of x to the given base.If the base not specified, returns the natural logarithm (base e) of x.	>>> math.log(10) 2.302585092994046 >>> math.log(10,10) 1.0 >>> math.log(8,2) 3.0
15.	pi	Math constant , the ratio 22/7	>>> math.pi 3.141592653589793
16.	e	Math constant	>>> math.e 2.718281828459045

Like sin(x) we have functions for cos(x), tan(x).

2.12 Points to Ponder

1. An operator is a symbol used to manipulate the data. The data items that the operators act upon are called operands. In **a+b**, **a** and **b** are operands and **+** is a operator.
2. A valid combination of constants, variables and operators constitutes an expression.
3. When several operators appear in one expression, evaluation takes place according to certain predefined rules which specify the order of evaluation and are called precedence rules.
4. Python supports seven types of operators: Arithmetic, logical, relational, bitwise, assignment, membership, identity.
5. Arithmetic operators // is used for integer division and / for float division.
6. The operators % is known as remainder operator and works with both float and integers
7. Relational operator are also known as comparison operators. They return either True or False value. A True is equivalent to numeric 1 and False is equivalent to 0.
8. Python supports parallel assignment using comma separated values. For example : a, b ,c=10,20,30 initializes a to 10, b to 20 and c to 30 at once.
9. Relational expressions use numeric data and relational operators whereas logical expressions use logical values and logical operators. A logical expression may contain relational expression, but reverse is not true.
10. To divide an integer by $\mathbf{2^n}$ a right shift by **n** bit positions is applied. To multiply an integer by $\mathbf{2^n}$ a left shift by **n** positions is applied. As an example to divide **16(10000) by 4** which is $\mathbf{2^2}$ we right shift 16 (10000) by 2 bits and get the answer as **00100** which is **4** in decimal.

11. In case of logical operator **and** and **or ,** if the left operand yields false value, the right operand is not evaluated by a compiler in a logical expression using **and**. If the left operand yields true value, the right operand is not evaluated by the compiler in a logical expression with the operator **or.** The operators **and** and **or** have left to right associativity, hence the left operand is evaluated first and based on the output, the right operand may or may not be evaluated. As an example, consider the following expression:

(10>=15) and (5!=4)

The left operand of **and** i.e **(10>=15)** is false so right operand i.e (**5!=4)** is not evaluated.

If it were **or** in the above expression in place of **and** , second operand would have been checked.

12. Membership operator in and not in checks membership of an element within a collection.

13. The identity operator is and not is are used to check similarity of the two operands. By similarity means pointing to the same objects in the memory.

14. Precedence tells in an expression which operation should be performed first depending upon priority of operators. Associativity means when two or more operators have same priority then from which side (left or right) we operate.

3. Decision Making

3.1 Introduction

Decision making statements are needed to alter the sequence of the statements in the program depending upon certain circumstances. In the absence of decision-making statements, a program executes in the serial fashion statement by statement basis. We have seen examples in the programs given in earlier chapters. In this chapter we are going to write statements which control the flow of execution based on some decision. Decision can be made based on success or failure of some logical condition. They allow us to control the flow of our program. These conditions can be placed in the program using decision making statements. Python language supports the following decision making / control statements:

(a) The if statement.

(b) The if-else statement.

(c) The if-else-if ladder statement.

All these decision-making statements checks the given condition and then executes its sub block if the condition happens to be true. On falsity of condition the block is skipped. A block is a set of statements created with colon (**:**) and proper indention. All control statement uses a combination of relational and logical operators to form conditions as par the requirement of the programmer.

3.2 The if statement

The general syntax of **if** statement is as:

if (condition):

statement(s)

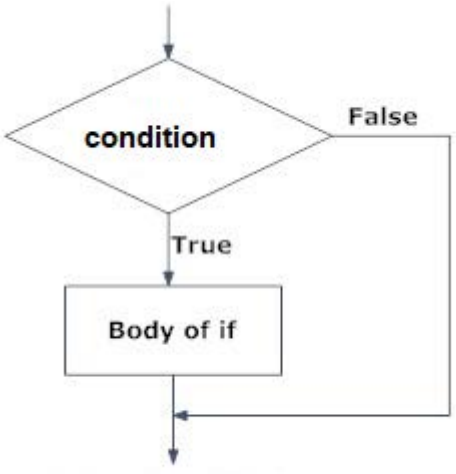


Figure 3.1: Flowchart of if statement

The **if** statement is used to execute / skip a block of statements based on truth or falsity of a condition. The condition to be checked is put inside the parenthesis (not necessary sometimes) which is preceded by keyword **if.** All the statements which are indented by the python rules (4 spaces or tab) constitute the body of the **if** block. The first statement not following the indention of **if** ends the **if** block.

In case of one or two statements constituting body of the **if** block, they can be written just after the colon **:** .

We present number of small scripts to illustrate **if** statement.

Sr.No	Script	Explanation
1.	`x=10` `if x>0:` `  print("x is greater than 0")` **OUTPUT:** `x is greater than 0`	**if** is a conditional execution statement. It's a keyword. The condition to be checked is written after **if**. For better readability parenthesis can also be used for enclosing condition. If the condition is true the first statement after the **if** gets executed, else it is skipped. Here the condition is true so the output.
2.	`x=1` `if x==0:` `  print("x is equal to 0")` **OUTPUT:** `Blank screen`	The **if** condition is false and there is only one statement just after **if** which will be skipped due to false condition, so the output.
3.	`x=51` `if x<40:` `  print("x less than 40")` `print("x greater/equal to forty")` **OUTPUT:** `x greater/equal to forty`	Only the first statement after if is in the body of if due to indention. The next **print** statement always executes regardless of truth or falsity of if condition.
4.	`x=int(input("Enter the value of x\n"))` `if x>=100:` `  print("x greater or equal to 100")` `  print("you think high")` `print("x less than 100")`	The statement after the **if** block executes regardless of truth or falsity of **if** condition. Just two statements constitute the body of the if block and last statement is not part of the if block's body.

	OUTPUT: **(FIRST RUN)** Enter the value of x 100 x greater or equal to 100 you think high x less than 100 **(SECOND RUN)** Enter the value of x 12 x less than 100	
5.	x=12 if x: print(x," is not zero") OUTPUT: 12 is not zero	**if x** is interpreted as if x!=0 or if x!=False or if x==True or if x==1. All convey same meaning.
6.	x=0 if not x: print(x," is zero") OUTPUT: 0 is zero	**if not x** is interpreted as if x!=1 or if x!=True or if x==False or if x==0. All convey same meaning.

3.2.1 Short hand if

Python also provide a short hand version of **if** when one or two statements only are body of the **if** block. See some examples:

```
Example 1:
x=10
if x>0: print (x,"is positive")

Example 2:
x=10
if x==10: print ('I executes'); print ('Me too')
```

The second example demonstrates that multiple statement can also be written separated by semicolon in the body of short hand **if**.

3.3 The if-else statement

In all the above programs we didn't write the other side of **if** condition ie we didn't take the action

when the condition fails. The **if-else** construct allows us to do this.

Its general syntax is

if(condition):
    statement(s)
else:
    statement(s)

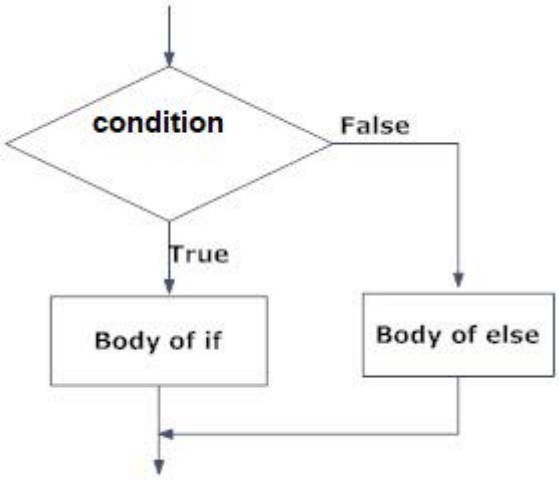


Figure 3.2: Flow chart of if-else

If the condition within **if** is True all the statements within the block(body of **if**) following **if** are executed else they are skipped and **else** part (body of else) get executed.

We present number of scripts to illustrate **if-else** statement.

```
#  Script 3.1 To check number is even or odd
x=int(input('Enter an integer\n'))
if x%2==0:
    print (x,' is even')
else:
    print(x,' is odd')
```

OUTPUT:

```
Enter an integer
34
34  is even
```

The priority of **%** is higher then **==** , so **x%2** is compared to ***0*** . If this is True then number is even else

number is not even. The **else** part executes only when **if** part is false and vice-versa.

```
# Script 3.2 To check whether a number is +ve or -ve
x=int(input('Enter any number\n'))
if x>0:
    print (x,' is positive')
else:
    print(x,' is negative')
```

OUTPUT:

```
Enter any number
54
54  is positive
```

If the number is greater than zero the number is positive and if the number is less than zero then number is negative.

The above code does not handle the case when instead of a +ve or -ve number the user supplies a 0 as input. If the user supplies zero (0) as input, **if** condition is false and **else** part get executed which prints **"number is –ve".** But in reality number is zero not negative so we modify the above script and give you the next script which handles this. .

```
# Script 3.3 To check whether a number is +ve , -ve or zero
x=int(input('Enter any number\n'))
if x==0:
    print(x,' is zero')
if x>0:
    print (x,' is positive')
else:
    print(x,' is negative')
```

OUTPUT:

```
Enter any number
0
0  is zero
0  is negative
```

There is one small change in the script from the previous one. We have added a simple **if** condition

which check whether number is zero or not.

The above program works fine but is less efficient. If the input happens to be zero then we need not check the later **if** condition. Out of the two : simple **if** and one if-**else** part we want only one part should get executed. The solution is simple exit from script once the first **if** condition for checking zero is satisfied. This can be done as:

```
if x==0:
    print(x,' is zero')
    import sys
    sys.exit()
```

To terminate the script at any point in the code we can make use of **exit** function present in **sys** module. Now whenever user enters 0 in the above script this **if** block executes and script terminates. If input is not zero then **if-else** part executes.

The code can also be written efficiently using **else-if ladder** that we will see later in this chapter.

```
# Script 3.4 Maximum of two numbers
x=int(input('Enter first number\n'))
y=int(input('Enter second number\n'))
if x>y:
    print(x,' is greater than ',y)
else:
    print(y,' is greater than ',x)
```

OUTPUT:

```
Enter first number
10
Enter second number
20
20  is greater than  10
```

The code is easy to understand but once again what if two number entered by user happens to be equal. The code does not handle this case. Can you rewrite the code to handle this?

```
# Script 3.5 To calculate gross salary of a person. Given #basic salary(bs) as input. If bs is >5000 da=55% of bs and #hra=15% of bs else da=45% of bs and hra=10% of bs
bs=float(input("Enter your basic salary\n"))
if bs<=5000:
  da=bs*0.45
  hra=bs*0.10
```

```
else:
  da=bs*0.55
  hra=bs*0.15
gs=bs+da+hra
print("Basic salary is %f"%bs)
print("HRA is %f"%hra)
print("DA is %f"%da)
print("Gross salary is %f"%gs)
```

OUTPUT:

```
Enter your basic salary
6000
Basic salary is 6000.000000
HRA is 900.000000
DA is 3300.000000
Gross salary is 10200.000000
```

We input the basic salary in **bs**. Through **if** condition we check basic salary **bs** against **5000.** Depending upon whether **if** condition is true or false **hra** and **da** are calculated as per the condition specified in the problem.

3.3.1 Short hand if-else

Like short hand **if** , Python provides short hand **if-else** statement. It can be used whenever body of the **if** and **else** contains one statement. The syntax is:

```
statement if condition else statement
```

Note the statement to be executed is written first for if but later for else.

See one simple example:

```
x=10
print('Hello') if x else print('Bye')
```

OUTPUT:

```
Hello
```

Here the value of **x** is 10 so **if** condition turns out to be true and **print** statement before **if** gets executed. Let's rewrite our even-odd script using short hand **if-else**

```
# script 3.6 Even odd using short hand if-else
x=int(input('Enter an integer\n'))
print(x,' is even') if x%2==0 else print(x,' is odd')
```

OUTPUT:

```
Enter an integer
```

```
10
10  is even
```

The code is easy to understand. Note the brevity and simplicity of code within just two lines.

3.4 Nesting of if-else's

Nesting of **if-else** means one **if-else** or simple **if** as the statement part of another **if-else** or simple **if** statement. There may be various syntaxes of nesting of if-else. We present few of them.

1. **if(condition):**
   if (condition):
   statement(s)
   else:
   statement(s)
   else:
   statement(s)

In the above case there is a nested if-else inside outer if.

2. **if(condition):**
   statement(s)
   else:
   if(condition):
   statement(s)
   else:
   statement(s)

Here the **else** part has nested **if-else.**

3. **if (condition):**
   if(condition):
   statement(s)
   else:
   statement(s)
   else:
   if(condition):

statement(s)

else:

statement(s)

Here both the outer if and outer else has nested if-else

We give number of examples of nesting of **if-else's** which are based on the above syntax .

```
#Script 3.7 To check whether a year is leap or not using nested if-else
year=int(input("Enter any year\n"))
if(year %100 ==0):
     if(year %400 ==0):
          print("The given year is leap year")
     else:
          print("The given year is not a leap year")
else:
     if(year %4 ==0):
          print("The given year is leap year")
     else:
          print("The given year is not a leap year")
```

OUTPUT:

```
(first run)
Enter any year
2000
The given year is  a leap year
(second run)
Enter any year
2005
The given year is not a leap year
```

A year is leap year if it is completely divisible by 100 *and* 400 or not divisible by 100 but divisible by 4. Initially if the **year %100** is zero, the inner **if** checks if the **year %400** is zero. If this is so the year is leap else year is not leap. If the outer **if** fails its corresponding **else** part executes in which we check **year *% 4==0*** , if this is true the year is leap else year is not leap.

```
# Script 3.8 To check whether a year is leap or not using logical
# operators and if-else
year=int(input("Enter any year\n"))
c1=year%100==0 and year%400==0
```

```
c2=year%100!=0 and year%4==0
if(c1 or c2):
        print("The given year is leap year ")
else:
        print("The given year is not a leap year")
```

The use of **and** and **or|** operator have reduced the length of the program as well as has made it more readable and efficient. **c1** and **c2** are temporary variables introduced. If any of them is **True** the **if** condition is true and year is leap year else year is not leap year.

```
# Script 3.9 Maximum of three numbers
a,b,c=input('Enter three integers separated by space\n').split()
a=int(a);b=int(b);c=int(c);
if(a==b and a==c):
    print("All three are equal")
else:
    if a>b:
        if a>c:
            max=a
        else:
            max=c
    else:
        if b>c:
            max=b
        else:
            max=c
    print(max," is greater ")
```

OUTPUT:

```
(First Run)
Enter three integers separated by space
10 20 5
20 is greater
(Second Run)
Enter three integers separated by space
10 10 10
All three are equal
```

If all the numbers are not equal we check **if a>b** , if this is true it means **a** is greater than **b**, we then check **a>c** if this is so then **a** is the greatest else **c** is greatest. If **a>b** is false initially it means **b** is greater than a , we then check whether **b>c ,** if this is so then **b** is greatest else **c** is greatest.

```
# Script 3.10 To check number is +ve,-ve or zero using nested if-else
num=int(input('Enter any number\n'))
if(num==0):
    print(num," is zero ")
else:
    if num>0:
        print(num,' is positive')
    else:
        print(num,' is negative')
```

OUTPUT:

```
Enter any number
-90
-90 is negative
```

There are three cases to consider: input number may be +ve,-ve or zero. The first possibility is handled by the outer **if** and rest two are handled by **if-else** nested within outer **else** part.

3.5 else-if ladder

The general syntax of **else-if** ladder is

```
if(condition):
  statement(s)
elif(condition):
  statement(s)
else(condition):
  statement(s)
```

If the first **if** condition is satisfied, then all its related statements are executed and all other **elif** *'s (*elif is short for else-if, one or more elif may be present*)* are skipped. The control reaches to first **elif** only if the first **if** fails. Same for second, third and other **elif** *'s* depending upon what your program required. **That is out of this *else-if* ladder only one *if* condition will be satisfied.**

We present number of programs to illustrate **else-if** ladder construct.

```
# Script 3.11 Maximum of three numbers using else-if ladder
a,b,c=input("Enter the three numbers separated by space\n").split()
a=int(a);b=int(b);c=int(c)
if(a==b and a==c):
```

```
    print("All three are equal")
elif (a>b and a>c):
    print("maximum is ",a)
elif(b>a and b>c):
    print("maximum is ",b)
else:
    print("maximum is" ,c)
```

OUTPUT:

```
(First Run)
Enter the three numbers separated by space
10 20 30
maximum is 30
(Second Run)
Enter the three numbers separated by space
40 40 40
All three are equal
```

In the script if all three inputs are equal , first **if** condition is satisfied and after executing its body code exits **.** If first **if** fails it is checked whether **a** is greater or **b** is greater by the subsequent two **elif** conditions using logical **and** operator else **c** is found to be the maximum. The logical operator **and** have been used for finding maximum of three numbers.

```
# Script 3.12 Arrange three numbers in Ascending order
a,b,c=input("Enter the three numbers separated by space\n").split()
a=int(a);b=int(b);c=int(c)
# logic to find max begins
if (a>b and a>c):
    max=a
elif(b>a and b>c):
    max=b
else:
 max=c
# logic to find max ends
# logic to find min begins
if (a<b and a<c):
    min=a
elif(b<a and b<c):
    min=b
```

```
else:
    min=c
# logic to find min begins
mid=a+b+c-(max+min)
print('Three numbers in ascending order')
print(min,",",mid,",",max)
```

OUTPUT:

```
Enter the three numbers separated by space
10 5 8
Three numbers in ascending order
5 , 8 , 10
```

In the variable **max** we have stored the maximum among three and in the variable **min** we have stored the minimum among three. The **mid** is calculated by subtracting **(min+max*)*** from the sum of **a, b** and **c** i.e. **(a+b+c).**

```
# Script 3.13 To determine the status of entered character
ch=input("Enter a character\n")
avalue=ord(ch)
if(avalue>=65 and avalue<=90):
    print("U entered uppercase letter")
elif (avalue>=97 and avalue<=122):
    print("U entered lowercase letter")
elif(avalue>=48 and avalue<=57):
    print("U entered a digit")
else:
 print("U entered a special symbol")
```

OUTPUT:

```
Enter a character
8
U entered a digit
```

The ASCII values for lowercase alphabets is from **97 to 122** (inclusive both) and for uppercase it is from **65 to 90** (inclusive both). The value is obtained by applying **ord** function to entered character. It is checked whether the entered character is within these two ranges using **else-if ladder.** Similarly, ASCII values for digits are from **48 to 57** (inclusive both) so character entered is also checked with this range. If all three conditions fail then the character entered must be a special symbol.

As arithmetic and relational operation can easily be carried out on characters too the above code can be written as:

```
ch=input("Enter a character\n")
if(ch>='A' and ch<='Z'):
    print("U entered uppercase letter")
elif (ch>='a' and ch<='z'):
    print("U entered lowercase letter")
elif(ch>='0' and ch<='9'):
    print("U entered a digit")
else:
    print("U entered a special symbol")
```

The code is same as previous one but instead of ASCII values of characters we have written characters within single quotes.

```
# Script 3.14 Case conversion
ch=input("Enter a character\n")
if(ch>='A' and ch<='Z'):
    print("lower case is :",chr(ord(ch)+32))
elif (ch>='a' and ch<='z'):
    print("upper case is :",chr(ord(ch)-32))
else:
    print("not an alphabet")
```

OUTPUT:

```
Enter a character
P
lower case is : p
```

Difference between ASCII values of uppercase and lowercase alphabet is 32 ie 'A'+32='a' or 'a'-32='A'. But python does not allow us to perform addition of string and an integer. The trick is to first convert character into integer and perform addition. The result then can be converted back to character using **chr** function.

Take an example to understand

ch='a'

ord(ch) gives you **97**; **ord(ch)-32** gives you **65** and **chr(65)** gives you **'A'.**

```
# Script 3.15 To determine grade of student based on its percentage
m1,m2,m3=input("Enter the marks in three subjects(max 100)\n").split()
m1=float(m1);m2=float(m2);m3=float(m3);
```

```
if((m1<1 or m1>100) or (m2<1 or m2>100) or (m3<1 or m3>100)):
     print("Marks must be within rang 1 to 100 ")
     import sys;sys.exit()
per=(m1+m2+m3)/3
if(per>=90):
     print("Grade is A\n")
elif(per>=80 and per<90):
     print("Grade is B\n")
elif(per>=70 and per<80):
     print("Grade is C\n")
elif(per>=60 and per<70):
     print("Grade is D\n")
elif(per>=50 and per<60):
     print("Grade is E\n")
else:
     print("Fail\n")
```

OUTPUT:

```
(First Run)
Enter the marks in three subjects(max 100)
94 95.5 93.5
Grade is A
(Second Run)
Enter the marks in three subjects(max 100)
101 78 -20
Marks must be within rang 1 to 100
```

Initial **if** condition ensures that user enters marks within range **1 to 100**.If they are not then we display a message and exit the script. Based on percentage, we display the grade of the student. For calculation of percentage we have reduced **(a+b+c)/100*300** to just **(a+b+c)/3**. If **per>=90** we display **grade is A** else for **per** in between **80 to 90** the **grade is B** and so on.

```
# Script 3.21 Determine root of quadratic equation
from math import sqrt
from sys import exit
a,b,c=input("Enter value of a, b and c\n").split()
a=float(a);b=float(b);c=float(c)
dis=b*b-4*a*c
if dis<0:
```

```
        print("Roots are imaginary\n")
        exit()
    elif dis==0:
        print("Roots are equal\n")
        r1=r2=-b/2*a
    else:
        print("Roots are unequal\n")
        r1=(-b+sqrt(dis))/(2.0 *a)
        r2=(-b-sqrt(dis))/(2.0 *a)
    print("Root 1= %f"%r1)
    print("Root 2= %f"%r2)
```

OUTPUT:

```
(First Run)
Enter value of a, b and c
1 7 12
Roots are unequal
Root 1= -3.000000
Root 2= -4.000000
(Second Run)
Enter value of a, b and c
1 2 1
Roots are equal

Root 1= -1.000000
Root 2= -1.000000
```

The quadratic equation in mathematics is given as:

$$AX^2 + BX + C = 0$$

Where **A, B** and **C** are constants. The solution of the equation comprises of two roots as power of **X** is **2.**

$$X1 = (-B + \sqrt{B*B - 4*A*C}) / (2 * A)$$

$$X2 = (-B - \sqrt{B*B - 4*A*C}) / (2 * A)$$

The expression **B*B- 4*A*C** is known as **discriminant** (say **dis**)and on the basis of its value the roots are determined as **equal (dis==0), imaginary (dis<0) or unequal (dis>0)** as shown in the program.

In the program we take as input the three constant's value using **A, B** and **C** and calculated the value of **dis**. The sqrt function has been imported from math module and exit function from sys module in the

first two lines of the script.

```
    #  Script 3.22 calculate electricity bill according to the given
    #  condition:
    #       For       first       50       units       Rs.       0.60/unit
#       For       next       100       units       Rs.       0.85/unit
#       For       next       100       units       Rs.       1.30/unit
#       For       unit       above       250       Rs.       1.60/unit
# An additional surcharge of 25% is added to the bill.
    unit=int(input("Enter number of units used "))
    # variables to hold charges per unit and surcharge rates
    c50=0.60
    c150=0.85
    c250=1.30
    cover250=1.60
    srate=0.25
    # Calculation of charges on units
    if unit <= 50:
        amt = unit * c50
    elif unit <= 150:
        amt = 50*c50 + (unit-50) * c150
    elif unit <= 250:
        amt = 50*c50 + 100*c150+(unit-150) * c250
    else:
        amt = 50*c50 + 100*c150+100*c250+(unit-250) * cover250

    surcharge = amt * srate
    total  = amt + surcharge
    print("Electricity Bill = Rs %6.2f"%total)
    OUTPUT:
    Enter number of units used 230
    Electricity Bill = Rs 273.75
```

The input to the script is the number of units consumed and stored in variable unit. The variables c50, c150 ,c250 and cover250 denotes the per unit rates for units<=50, units<=150 , units<=250 and units>250 respectively. The variable srate is for storing surcharge rate.

If unit consumed is say 90 units then for first 50 units charges will be 50*c50 and for remaining units (90-50) charges will be 40*c150. We show you one example for units consumed =230 then calculation is done as:

1. First 50 units: 50*c50 = 50 *0.60=30
2. Next 100 units 100*c150=100*0.85=85
3. Next 80 units (230-150) 80*c250=80*1.30=104

Total amount without surcharge will be: 30+85+104=219

Final amount with surcharge = 219+219*0.25 => 219+54.75=273.75

3.6 Ponderable Points

1. The flow of execution may be transferred from one part of a program to another part based on the output of the conditional test carried out. It is known as conditional execution.
2. **if, if-else, if-else-if** are known as selective control structures.
3. Any block in python can be created using : and proper indentation (tab or 4 space)
4. Between two codes **if (x = = 0)** and **if (0 = = x)** the later one is preferred. By mistake if **(0== x)** is written it will produce error, whereas the first one will accept.
5. Short hand **if** syntax: *if condition statement*
6. Multiple statements can be written in short hand **if** in single line separated by semicolon .
7. Short hand *if-else* syntax: *statement if condition else statement*
8. The else-if ladder can be used to check multiple conditions and out of that only one condition will be true.
9. There is no switch-case construct present in python as found in C/C++/Java .

4. LOOPING

4.1 Introduction

Looping is a process in which set of statements are executed repeatedly for a finite or infinite number of times. Python provides two ways to perform looping by providing two different types of loop. Looping can be called synonymously iteration or repetition. Loops are the most important part of almost all the programming language such as C, C++, java, C#, R, Rust etc.

In our practical life we see lots of examples where some repetitive tasks has to be performed like finding average marks of students of a class, finding maximum salary of group of employees, counting numbers etc.

A loop is a block of statements which are executed again and again till a specific condition is satisfied. Python provides two loops to perform repetitive actions.

1. while

2 .for

To work with any type of loop three things must be performed:

(a) Loop control variable and its initialization

(b) Condition for controlling the loop.

(c) Increment / decrement of control variable.

Let's work first with the **while** loop.

4.2 The while loop

The syntax of the **while** loop is simple.

```
while(condition):
  statement(s)
```

The statements that follows in the indented block of **while** loop is called body of the **while** loop. All the statements within the body are repeated till the condition specified in the parenthesis in **while** is **True**. As soon as condition becomes false the body is skipped and control is transferred to the next statement outside the loop which is not in indention of **while** loop.

Let's write few programs to understand the **while** loop better.

```
# Script 4.1 demo of while loop (printing number 1 to 5)
t=1
while t<=5:
    print("t=",t)
```

```
    t=t+1
print("Out of loop")
```

OUTPUT:

```
t= 1
t= 2
t= 3
t= 4
t= 5
Out of loop
```

In the **while** loop we have stated the condition **t<=5. t** is called loop control variable. Initially value of **t is 1**.This value of **t** is compared with **5** which is True so control reaches into the **while** loop body and **print** statement within loop executes which prints the value of **t**. Then **t** is incremented by **1** ie becomes **2.** Control reaches back to the condition of the **while** loop which is true (2<=5). This process continues. When value of **t** becomes **6**, which causes condition in the **while** loop to become false and control comes out of loop. The statement outside the **while** loop executes.

If you want to display just numbers in the same line then the above script can be modified as:

```
t=1
while t<=5:
    print(t,end=' ')
    t=t+1
print("\nOut of loop")
```

Here we have used named parameter **end** in **print** function to set space. Because of this '\n' is required in next print statement outside **while** loop to print on next line.

```
# Script 4.2 printing number 5 to 1 using while loop
t=5
while t>=1:
    print("t=",t)
    t=t-1
print("Out of loop")
```

OUTPUT:

```
t= 5
t= 4
t= 3
t= 2
t= 1
```

```
Out of loop
```

The script is simple. Here we have printed the value from **5 to 1** by decrementing the control variable **t by 1** in each successive iteration till **t>=1** condition is satisfied. .

```
# Script 4.3 Printing all even numbers between 1 and 30
t=1
print("Even numbers between 1 and 30")
while t<=30:
    if t%2==0:
        print(t,end=' ')
    t=t+1
```

OUTPUT:

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

The program simply checks remainder when **t** is divided by **2**. If it is **0** i.e. it is even number, and the value of **t** is printed. This continues till **t** is less than equal to **30**.

The other way is to start with t=2 and increment the control variable t by 2 in every loop iteration. This is shown below:

```
t=2
print("Even numbers between 1 and 30")
while t<=30:
    print(t,end=' ')
    t=t+2
```

On the similar ground you can try printing all odd numbers between 1 and 30 or between any range.

```
# Script 4.9 printing numbers between 1 and 100 which are
#completely divisible by 3 and 5
t=1
while t<=100:
    if(t%3==0 and t%5==0):
        print(t,end=' ')
    t=t+1
```

OUTPUT:

```
1530    45    60    90
```

If remainder divided by **3** is zero then number is completely divisible by **3** . Same is true for **5** so we have combined the two condition by ***and*** **(logical AND)** operator so if remainder in both the case is zero then **and** condition will hold true and that number will be printed.

```
# Script 4.11 generation of table of any given number
t=1
n=int(input("Enter any +ve number\n"))
while t<=10:
     value=n*t
     print("%2d x %2d = %3d"%(n,t,value))
     t=t+1
```

OUTPUT:

```
Enter any +ve number
5
 5 x  1 =   5
 5 x  2 =  10
 5 x  3 =  15
 5 x  4 =  20
 5 x  5 =  25
 5 x  6 =  30
 5 x  7 =  35
 5 x  8 =  40
 5 x  9 =  45
 5 x 10 =  50
```

Any **+ve** number is taken as input. The number **n** is multiplied by **1** through **10** inside the loop and the loop counter**,** number and value is displayed by formatting as to produce the desired output. The **%2d** means allocating width for 2 digits and **%3d** for 3 digits. This we have used for better alignment of digits in output.

```
# Script 4.12 maximum of n elements
t=1
n=int(input("Enter how many numbers\n"))
m=int(input("Enter the number\n"))
max=m
while t<=n-1:
     m=int(input("Enter the number\n"))
     if max<m:
```

```
            max=m;
      t=t+1
print("maximum element is ",max)
```

OUTPUT:

```
Enter how many numbers
5
Enter the number
34
Enter the number
12
Enter the number
56
Enter the number
78
Enter the number
4
maximum element is  78
```

Initially numbers of elements are taken in **n.** The first number is taken outside the loop and assumed to be maximum; this number is stored in **max.** Then remaining numbers are taken inside the loop. On each iteration the number is compared with the **max,** if the **max** is less then number taken then number will be the maximum one. This is checked through **if** statement. In the end when control comes out from **while** loop **max** is displayed.

```
# Script 4.13 to reverse the number
rev=0
orig=int(input("Enter the number\n"))
while orig:
    r=orig%10
    rev=rev*10+r
    orig=orig//10
print("Reverse Number=",rev)
```

OUTPUT:

```
Enter the number
3456
Reverse Number= 6543
```

The logic to reverse a number is quite simple. Let's understands it step by step inside the loop:

S 1	**orig=3456**	**r=3456%10=6**	**rev=0*10+6=6**	**orig=3456//10=345**
S 2	**orig=345**	**r=345%10=5**	**rev=6*10+5=65**	**orig=345//10=34**
S 3	**orig=34**	**r=34%10=4**	**rev=65*10+4=654**	**orig=34//10=3**
S 4	**orig=3**	**r=3%10=3**	**rev=654*10+3=6543**	**orig=3//10=0**

As **orig** is **0** so condition inside the **while** loop is false and control comes out of loop. The reverse number in variable **rev** which is printed.

```
# Script 4.14 To check number is palindrome or not
rev=0
orig=int(input("Enter the number\n"))
save=orig
while orig:
    r=orig%10
    rev=rev*10+r
    orig=orig//10
if save==rev:
    print(save," is palindrome")
else:
    print(save," is not palindrome")
```

OUTPUT:

```
(First Run)
Enter the number
7723277
7723277  is palindrome
(Second Run)
Enter the number
12123
12123  is not palindrome
```

A number is called palindrome if on reversing it is equal to the original number for e.g. 121 3223,656 etc. To check whether an entered number is palindrome or not simply reverse the number and compare with the original number but as we have seen in the program to reverse a number, the original number becomes zero when controls comes out from the loop. So we save the original number in a variable before starting processing, in the above program it is in the **save** variable.

```
# Script 4.16 To find number of digits in a given number
count=0
num=int(input("Enter the number\n"))
while num:
  num=num//10
  count=count+1
print("Number of digits=",count)
```

OUTPUT:

```
Enter the number
34545
Number of digits= 5
```

We keep on dividing the number by **10** and storing the result back in the original number. On each iteration i.e. after divide we increment the counter **count** which counts the number of digits in the number. See example below:

Initial	**num=3456**	**count=0**
S1	**num=345**	**count=1**
S2	**num=34**	**count=2**
S3	**num=3**	**count=3**
S4	**num=0**	**count=4**

In **step 4** when **num** becomes condition in the **while** loop becomes false and control comes out from loop and prints the result. .

```
# Script 4.17 To check whether a number is Armstrong or not
newnum=0
count=0
num=int(input("Enter the number\n"))
save=num;
# counting number of digits in num
while num:
    num=num//10;
    count=count+1
# storing back saved number in num
num=save;
```

```
# main logic
while num:
    r=num%10
    newnum=newnum+r**count
    num=num//10
if newnum==save:
    print("Number ",save," is Armstrong")
else:
    print("Number ",save," is not Armstrong")
```

OUTPUT:

```
(First Run)
Enter the number
1634
Number 1634  is Armstrong
(Second Run)
Enter the number
275
Number  275  is not Armstrong
```

A number is called Armstrong if sum of count number of power of each digit is equal to the original number For e.g. to check **153** is Armstrong number or not we see that number of digits are **3** then $\mathbf{1^3 + 5^3 + 3^3 => 1+125+27=>153}$ which is equal to the original number so number **153** is Armstrong. Let's see a four digit number **1634**. Number of digits is 4 **so** $\mathbf{1^4 + 6^4 + 3^4 + 4^4 => 1+1296+81+256=>1634.}$

So script proceeds as follows. First find out number of digits, before doing this save the number in the **save** variable. Now number of digits are stored in the **count** variable. **num** is **0** now so copy the value from **save** to **num.**

Now steps of second loop are as follows:

Initially num=153 newnum=0 count=3

S1 r=153%10= newnum=0+33=0+27=27 num=153//10=15**

S2 r=15%10=5 newnum=27+53=27+125=152 num=15//10=1**

S3 r=1%10=1 newnum=152+13=152+1=153 num=1//10=0**

As **newnum** contains **153** which is compared with the original number stored in **save** We get output **153** *is* Armstrong number.

```
# Script 4.18 To find sum of digits of given number
s=0
num=int(input('Enter the number\n'))
while num:
    r=num%10
    s+=r
    num=num//10
print("Sum of digits of given number is ",s)
```

OUTPUT:

```
Enter the number
34567
Sum of digits of given number is  25
```

Finding sum of digits of a given number is quite simple. We extract each digit from right and sum it till number does not become zero. For better understanding we follow the steps as :

Initial num=2345

S1	**2345!=0 (true)**	**r=2345%10=5**	**s=0+5=5**	**num=2345//10=234**
S2	**234!=0 (true)**	**r=234%10=4**	**s=5+4=9**	**num=234//10=23**
S3	**23!=0 (true)**	**r=23%10=3**	**s=9+3=12**	**num=23//10=2**
S4	**2!=0 (true)**	**r=2%10=2**	**s=12+2=14**	**num=2//10=0**
S5	**0!=0(false)**			

In the fifth step condition in the **while** loop becomes false and the result is printed through **s**.

```
# Script 4.19 Mini Area calculator using while and else if ladder
from sys import exit
choice=5
while(choice >=1 and choice <=5):
  print("Welcome to Area Zone")
  print("1.Area of Triangle")
  print("2.Area of Circle")
  print("3.Area of Rectangle")
  print("4.Area of Square")
```

```
  print("5.Exit")
  choice=int(input("Enter your choice( 1 to 5)\n"))
  if choice==1:
    b,h=input("Enter the base and height of triangle\n").split()
    b=float(b);h=float(h)
    area=0.5*b*h
    print("Area of triangle is ",area)
    input("Press any key to continue...")
  elif choice==2:
    r=float(input("Enter the radius of circle\n"))
    import math
    area=math.pi*r*r
    print("Area of circle is ",area)
    input("Press any key to continue...")
  elif choice==3:
      l,b=input("Enter the length and bredth rectangle\n").split()
      l=float(l);b=float(b)
      area=l*b
      print("Area of rectangle is ",area)
      input("Press any key to continue...")
  elif choice==4:
    s=float(input("Enter the side of square\n"))
    area=s*s
    print("Area of square ",area)
    input("Press any key to continue...")
  elif choice==5:
        print("Bye Bye\n")
        exit()
  else:
      print("Better u know numbers")
      exit()
```

OUTPUT:

```
Welcome to Area Zone
1.Area of Triangle
2.Area of Circle
```

```
3.Area of Rectangle
4.Area of Square
5.Exit
Enter your choice( 1 to 5)
1
Enter the base and height of triangle
20 7
Area of triangle is  70.0
Press any key to continue...
```

In the program the whole **else if ladder** is put into the **while** loop. In each different **if test** we find out areas of triangle, circle, rectangle and square. After fulfilling one choice for the user the menu again appears because of **while** loop. On entering **5** in the **choice** the program terminates.

4.2.1 Nesting of while loop

Nesting means one inside another. One **while** loop becomes body of the another **while** loop and for one iteration of outer **while** loop inner **while** loop runs. The syntax is :

```
while condition:
   statement(s)
   while condition:
         statement(s)
```

Let's understand it using a small example:

```
x=1
while(x<=3):
    y=1
    while(y<=4):
        print(x*y,end=' ')
        y=y+1
    x=x+1
    print('')
```

Initially the value of **x** is 1 and in the outer **while** loop condition **x<=3** is satisfied. The first line in the body of outer **while** is **y=1** . The condition in the inner **while** loop is satisfied. The **print** statement prints **1*1=1** and **y** is incremented. The inner **while** loop condition is checked and **1*2=2** is printed. This continues till condition in the inner **while** loop is satisfied. When condition in the inner **while** loop becomes false, **x** is increment by 1 and a new line is inserted because of **print** statement. The outer loop continues with new value of **x=2** and inner **while** loop runs for **y=1** to 4 with the value of **x=2**. The process continues, and we get the following output:

```
1 2 3 4
2 4 6 8
3 6 9 12
```

Let's take one more example where we find sum of digits of a given number up to single digit.

```
# Script 4.20 To find sum of digits of a given number upto single digit
count=sum=0
num=int(input("Enter the number\n"))
while(num>9):
    sum=0
    while num!=0:
        r=num%10
        sum=sum+r
        num=num//10;
    if(sum>9):
        num=sum
print("Sum of digits up to single digit is ",sum)
```

OUTPUT:

```
Enter the number
786
Sum of digits up to single digit is 3
```

For example, number is **4275** then sum of digits is **4+2+7+5 = 18**. As **18** is more than **9** we repeat the process and get the result **1+8 i.e. 9**. This time answer is in single digit so we stop the process. In the program for finding sum of digits we have used nesting of **while** loop.For sum of digits up to single digit we have used outer while loop. When **sum>9 , sum** *i*s assigned to **num** and for this **num** , sum of digits are determined using inner **while** loop.

4.3 Break Statement

The ***break*** statement is used to come out early from a loop without waiting for the condition to become false. When the **break** statement is encountered in the **while** loop or any of the loop which we will see later, **the control immediately transfers to first statement out of the loop i.e. loop is exited prematurely. If there is nesting of loops the *break* will exit only from the current loop containing it.**

Let's write some programs which make use of **break** statement

```
# Script 4.21 demo of break statement
x=1
while x<=5:
    if x==3:
        break;
    print("Inside the loop x=",x)
    x=x+1
print("Outside the loop x=",x)
```

OUTPUT:

```
Inside the loop x= 1
Inside the loop x= 2
Outside the loop x= 3
```

When **x** is **3 if** condition becomes true, the body of the **if** statement is single **break** statement so all the statements in the loop following the **break** are skipped and control is transferred to the first statement after the loop which is **print** which prints **Outside the loop x=3**.

```
# Script 4.22  Square of numbers
while True:
    num=int(input('Enter any number(-99 to quit)\n'))
    if num==-99:
        break
    print('Square of number is ',num*num)
print("Have a nice day !")
```

OUTPUT:

```
Enter any number(-99 to quit)
5
Square of number is  25
Enter any number(-99 to quit)
3
Square of number is  9
Enter any number(-99 to quit)
-99
Have a nice day !
```

The statement **while (True)** is an infinite loop as it interprets to **while (True!=False)** which is always

true so our loop runs for infinite number of times. To **break** out from loop we ask user to enter number **–99** which is checked through **if** condition and if it is true then we come out from loop through **break** else prints square of the number entered.

```
# Script 4.23 To check number is prime or not
    flag=True
    c=2
    num=int(input("Enter the number\n"))
    #Loop starts
    while c<=num//2:
      if(num%c==0):
        flag=False
        break
      c=c+1
    # Loop ends
    if flag:
      print("Number is prime")
    else:
      print("Number is not prime\n")
    OUTPUT:
    Enter the number
    11
    Number is prime
```

A number is prime if it is completely divisible by 1 and itself for example 1,3,5,7,11,13,17,19,23 etc. To check whether a number is prime or not we start from a counter **c=2** (every number divides by 1) and continues till **c<=num//2 since no number is completely divisible by a number which is more than half of that number**. For example **12** is not divisible by **7,8,9,10,11** which are more than **6.** So we check if the number is divisible by any number **<=num//2** then it cannot be prime we set **flag=False** and come out from loop. The **flag** was initialized to **True** in the beginning so **if num%c==0** is true control sets **flag=False** which means number is not prime else **flag** remains **True** which means control never transferred to ***if*** block i.e. number is prime. So outside the loop we check this value of **flag** and prints accordingly.

4.4 The continue statement

The *continue* statement causes the remainder of the statements following the *continue* to be skipped and continue with the next iteration of loop. So we can use *continue* statement to bypass curtain number of statements in the loop on the basis of some condition given by *if* generally. The syntax of *continue* statement is simply

continue

Lets write a program to illustrate **continue** statement.

```
# Script 4.24 demo of continue statement
t=0
while t<=10:
      t=t+1
      if t%2:
            continue
      print(t,end=' ')
OUTPUT:
2 4    6    8    10
```

when **t** is an odd number *continue* in the body of the **if** condition cause loop to continue with the next iteration of loop skipping **print** statement. If number is even the number is simply printed as **continue** itself is skipped.

In the coming part of this program as well as in other chapters you will see lots of examples of **continue** statement. The above program was just to give you an idea how **continue** statement works in loop.

4.5 The *for* loop

This is second loop which we are going to examine in this section. The **for** loop is most frequently used by programmers just because of its simplicity. The **for** loop in the python is different than other programming languages like C/C++/Java. In python the **for** loop is used for iterating over a sequence. It can be a range of numbers, list, tuple, dictionary etc. The syntax of **for** loop is :

```
for element in sequence:
    statement(s)
```

Here the sequence can be anything as discussed just now: list, tuple, dictionary, set, array, range etc. and element is any element that belongs to sequence. The body of the **for** loop continues if there are elements in the sequence. After element takes the last value in the sequence the **for** loop exits.

Before we see any example of for loop let's see what the range function in python it as most commonly uses with for loop.

4.5.1 The range function

The **range** function in python generates sequence of numbers within **range**. The general syntax is : **range(start, stop, stepsize)**

Numbers are produced in sequences starting from start, with an increment of stepsize for next element in sequence up to stop (excluding). The default value for start and stepsize is 0 and 1 respectively. See the first example in shell

```
>>> range(5)
```

```
range(0, 5)
>>> print(range(5))
range(0, 5)
```

As can be seen from the output in the shell that instead of getting sequence of numbers you are getting the **range(0,5)** as output which is actually a **range** object. To get the sequence you need to convert this range into list by applying list constructor as:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

This is an important concept and you must make note of it. Further not all elements are generated statically. The generation of next element of the sequence the range function simply remembers the *start, stop and stepsize* and generates the next element on the fly. Important point to note that *stop* is not included in generation of numbers.

See many other examples in shell:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> list(range(2,11,2))
[2, 4, 6, 8, 10]
>>> list(range(10,101,10))
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(1,50,5))
[1, 6, 11, 16, 21, 26, 31, 36, 41, 46]
>>> list(range(5,50,5))
[5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> list(range(5,51,5))
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

The examples in the shell are quite easy to understand. Just try them at your own and play with them to have a feeling of understanding and confidence together.

The list generated through range and list constructed can also be stored in some other variable and using len function the number of elements can also be found out.

```
>>> x=list(range(5))
>>> x
```

```
[0, 1, 2, 3, 4]
>>> len(x)
5
```

As you have now understood the concept of range function, let's now use it with for loop

Sr.No	Script	Explanation
1.	**for i in range(5):** **print(i,end=' ')** **OUTPUT:** **0 1 2 3 4**	The range(5) gives a list of elements from 0 to 4. In every iteration of for loop I takes on the values 0 to 4 and in the body the same value is displayed. Note list constructor is note required.
2.	**for i in range(5,0,-1):** **print(i,end=' ')** **OUTPUT:** **5 4 3 2 1**	The loop runs from 5 to 1(inclusive)as 0 is the stop element and step size is -1.
3.	**for i in range(1,20,2):** **print(i,end=' ')** **OUTPUT:** **1 3 5 7 9 11 13 15 17 19**	Displays all odd numbers between 1 and 20.
4.	**for i in range(2,21,2):** **print(i,end=' ')** **OUTPUT:** **2 4 6 8 10 12 14 16 18 20**	Displays all odd numbers between 1 and 20.

The table above has demonstrated some small examples of for loop with the range function. Many more you can try yourself. Now we write some scripts where some processing on the elements on the sequence is performed instead of just displaying them. Let's see some example scripts.

```
#Script 4.25 To print and find sum of series 1+2+3+4+5.....
sum=0
n=int(input("Enter the number of terms\n"))
for i in range(1,n+1):
    print(i,"+",end=' ')
    sum=sum+i
print("\nSum of series is ",sum)
```

OUTPUT:

```
Enter the number of terms
7
1 + 2 + 3 + 4 + 5 + 6 + 7 +
Sum of series is  28
```

The number of terms is taken in the variable **n**. We run the loop from **1 to n with increment of 1.** The sum is stored in the variable **sum** initialized to **0** in the beginning.

```
# Script 4.26 To print and find sum of series 1-2+3-4+5-…
sum=0
k=1
n=int(input("Enter the number of terms\n"))
for i in range(1,n+1):
    if i%2==0:
        print("-",i,end=' ')
    else:
        print("+",i,end=' ')
    sum=sum+(k*i)
    k=k*-1;
print("\nSum of series is ",sum)
```

OUTPUT:

```
Enter the number of terms
8
+ 1 - 2 + 3 - 4 + 5 - 6 + 7 - 8
Sum of series is  -4
```

In the series to be generated odd numbers are +ve and even numbers are **–ve.** We take one variable **k=1. k** is multiplied by **-1** inside the loop in each iteration.

Initially **sum=0 + (1 * 1)** gives **sum=1** then **k** becomes **k=1*-1= -1** which is used in the second iteration of the loop. In the second iteration **sum** becomes **sum=1+(2 * -1) =>sum=1 – 2 = -1** and value of **k** changes to **1** again **(-1 * -1 = 1)** and this continues for **n** times.

```
# Script 4.27 to print and find sum of series
#1^2+2^2+3^2+4^2+.....(^ stands for power)
sum=0
n=int(input("Enter the number of terms\n"))
for i in range(1,n+1):
       print("%d^%d+ "%(i,2),end=' ')
       sum=sum+i**2
```

```
print("\nSum of series is ",sum)
```

OUTPUT:

```
Enter the number of terms
5
1^2+  2^2+  3^2+  4^2+  5^2+
Sum of series is  55
```

We take number of terms in the *n* and run loop from **1 to n**. On each iteration we found **2 to the power i** using power operator ** and add it to **sum.**

```
# Script 4.28 To print and find sum of series 1 + x
# +x^2+x^3+.........
sum=1
n=int(input("Enter the number of terms\n"))
x=int(input("Enter the value of x\n"))
print("%d+"%1,end=' ')
for i in range(2,n+1):
          print("%d ^ %d+"%(x,i),end=' ')
          sum=sum+x**i
print("\nSum of series is ",sum)
```

OUTPUT:

```
Enter the number of terms
5
Enter the value of x
2
1+ 2 ^ 2+ 2 ^ 3+ 2 ^ 4+ 2 ^ 5+
Sum of series is  61
```

As we have done in all earlier series scripts number of terms is taken in the variable **n.** We do take **value of x** also in the variable **x.** The first term 1 is printed outside the loop and loop runs from 2 to n and n+1 is stopping condition. The sum is initialized to 1 as first term is 1. Rest is self-explanatory.

```
# Script 4.29 To find factorial of a number
fact=1
num=int(input("Enter a +ve integer number\n"))
if num<0:
     print("Enter +ve number only")
     import sys;sys.exit()
for i in range(1,num+1):
     fact=fact*i
```

```
print("The factorial of %d is %d\n"%(num,fact))
```

OUTPUT:

```
Enter a +ve integer number
6
The factorial of 6 is 720
```

The factorial of number say **5** is calculated by multiplying **5*4*3*2*1** or by multiplying **1*2*3*4*5** which will be **120.** Initially **fact is 1.** We run the loop from **1 to num ,** it may be from **num to 1** also. In each iteration value of **fact** is multiplied by **t** and stored back in **fact** which is used in the next iteration. For negative numbers we display a message and terminate the program. For better understanding let's take **num=4**

S1 t=1 fact=1 * 1 => 1

S2 t=2 fact=1 * 2 => 2

S3 t=3 fact=2 * 3 => 6

S4 t=4 fact=6 * 4 => 24 .

If you want to use factorial function without creating your own function, you can use math module's factorial function as:

```
>>> import math
>>> math.factorial(10)
3628800
```

Script 4.30 To check a number is perfect or not

```
sum=1
num=int(input("Enter a +ve integer number\n"))
for i in range(2,num//2+1):
    if num%i == 0:
        sum=sum + i
if(sum==num):
    print("The number is perfect")
else:
    print("The number is not perfect")
```

OUTPUT:

```
Enter a +ve integer number
28
The number is perfect
```

A number is called perfect if sum of its factor is equal to the number itself for e.g. *6* , its factor are *1* ,*2, 3* and sum of its factor is *6* which is equal to the number *6* so it is a perfect number. Similarly **28** is a perfect number (1+2+4+7+14=28). In the loop we initialize **sum** to 1 and run the loop for **num//2** as for any number say **i** which is more than **num//2 , num%i** won't be zero (excluding **num** itself). .

4.6 Nesting of for loop

Nesting of **for** loop is used most frequently in many programming situations and one of the most important usage in displaying various patterns which we will see in the coming programs. The general syntax is :

```
for i in sequence:
  for j in sequence:
       statement(s)
  statement(s)
```

For each iteration of first **for** loop (outer **for** loop, control variable is i) inner **for** loop (control variable j) runs as it is part of the body of outer **for** loop. The inner **for** loop has its own set of statements which executes till the condition for inner **for** loop is true. Outer for loop may or may not other statements as its body other than inner for loop. See number of programs given below.

```
# Script 4.30 Nesting of for loop
for i in range(1,11):
     for j in range(1,11):
          print("%3d"%(i*j),end=' ')
     print("")
```

OUTPUT:

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

The body of the first **for** loop (outer) contains three statements. The inner **for** loop's body has got only one statement. Initially **i is 1** and condition **i<=10** in the outer loop is true . First statement inside outer **for** loop is inner **for** loop which initializes **j=1** . Now this **for** loop runs for **1 to 10** for value of **i=1** . When this loop terminates on reaching a value of **j=11,** control is transferred to third statement **print("");** which leaves a line on the output screen. Now control is transferred to outer loop which increments value of **j by 1** which becomes **2** .The process repeats with value of **j from 1 to 10** for value of **i=2** , till **i<=10** remains true. .

```
# Script 4.31 to print the following pattern, input is number # of lines
#  1
#  2 2
#  3 3 3
#  4 4 4 4
line=int(input("Enter the number of lines\n"))
print("The pattern is ")
for row in range(1,line+1):
   for col in range(1,row+1):
      print(row,end=' ')
   print("")
```

OUTPUT:

```
Enter the number of lines
5
The pattern is
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

We have used two **for** loops in the program. One is to control the number of rows and second to control number of cols. Initially assume **line=5** so outer **for** loop runs four times. In the first run **row=1** and inner loop runs only once. The **print("")** statement is within the inner **for** loop so this leaves new line after printing **1**.When control reaches second time inside outer **for** loop value of row is 2,inner loop starts again by setting the value of **col=1**,this time inner loop runs twice printing the value of **row** which is **2** twice. This continues till **row<=5.**

```
# Script 4.32 to print  the following pattern, input is #number of lines
```

```
#  A
#  B B
#  C C C
#  D D D D
```

```
ch='A'
line=int(input("Enter the number of lines\n"))
print("The pattern is ")
for row in range(1,line+1):
   for col in range(1,row+1):
      print(ch,end=' ')
   ch=chr(ord(ch)+1)
   print("")
```

OUTPUT:

```
Enter the number of lines
5
The pattern is
A
B B
C C C
D D D D
E E E E E
```

For the first run of outer **for** loop **ch='A'** . When first iteration of inner **for** loop finishes value of **ch** is incremented by **1** and becomes **B**. As we have to print one **A** on first row, two **B** on second row and so on. We increment **ch** at the end of every iteration of outer **for** loop. For incrementing character ch we first need to convert into its ASCII representation using ord function then after incrementing convert back to character using chr function.

```
# Script 4.33 to print  the following pattern, input is #number of lines
#  1
#  1 2
#  1 2 3
#  1 2 3 4
line=int(input("Enter the number of lines\n"))
print("The pattern is ")
```

```
for row in range(1,line+1):
   for col in range(1,row+1):
      print(col,end=' ')
   print("")
```

OUTPUT:

```
Enter the number of lines
5
The pattern is
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

This script is similar to **earlier ones** but instead of printing the value of **row** we have printed the value of **col** to get the desired output.

```
# Script 4.34 to print  the following pattern,input is number #of lines
#  1
#  2 3
#  4 5 6
#  7 8 9 10
#
value=0
line=int(input("Enter the number of lines\n"))
print("The pattern is")
for row in range(1,line+1):
    for col in range(1,row+1):
        value=value+1
        print(value,end=' ')
    print("")
```

OUTPUT:

```
Enter the number of lines
4
The pattern is
1
```

```
2 3
4 5 6
7 8 9 10
```

The value of **value** variable is incremented in every iteration of inner **for** loop which is printed through **print.** Thus we get the desired output. As number of lines grows more than 4 display of two digit numbers will not be properly aligned. For that you can make use of **print('%3d'%value,end=' ')** as a replacement option in second line of inner for loop. Give it a try!.

```
# Script 4.35 to print the following pattern, input is number #of lines
#     1
#    22
#   333
#  4444
line=int(input("Enter the number of lines\n"))
print("The pattern is ")
for row in range(1,line+1):
     for space in range(1,line-row+1):
          print(end=' ')
     for col in range(1,row+1):
          print(row,end='')
     print("")
```

OUTPUT:

```
Enter the number of lines
4
The pattern is
    1
   22
  333
 4444
```

To get the above output note before printing **1** we have to leave **3** spaces **(line-row).** Similarly, **2** spaces for **2**, **1** space for printing **b** and no space for printing **4.** This has been achieved by inserting one more **for** loop between outer and inner **for** loop. Prior to printing the values, we leave the spaces through this **for** loop. Note the print statement in the second inner loop has just opening and closing single quotes in the end argument with no space in between. *If you give a space in end argument as:* `print(row,end=' ')`

The output will become:

```
     1
   2 2
  3 3 3
 4 4 4 4
```

And instead of printing row as argument in the above print with space in end,we print '*' then output will be:

```
     *
    * *
   * * *
  * * * *
```

```
# Script 4.36 to print the following pattern, input is number of lines
       *
      *  *
     *  *  *
    *  *  *  *
     *  *  *
      *  *
       *
line=int(input("Enter the number of lines\n"))
print("The pattern is ")
# First half of pyramid
for row in range(1,line+1):
     for space in range(1,line-row+1):
          print(end=' ')
     for col in range(1,row+1):
          print('*',end=' ')
     print("")
# second half of pyramid
for row in range(line,0,-1):
     for space in range(line-row,0,-1):
          print(end=' ')
```

```
        for col in range(row,0,-1):
            if(row==line):
                continue
            print('*',end=' ')
        if(row==line):
            continue
        print("")
```

OUTPUT:

```
Enter the number of lines
4
The pattern is
   *
  * *
 * * *
* * * *
 * * *
  * *
   *
```

The first block of **for** loop prints asterisk in descending order and second block of **for** loop prints them in ascending order. Note the two statements

```
if(row==line)
  continue;
```

ensures that when second block of **for** loop executes we don't get repeat pattern of line no of asterisk in the first iteration of outer **for** loop.

4.7 The pass statement

The **pass** statement is used as null statement in python. A null statement does nothing but its presence in loop ensures that code may be repeated for a fix number of time but it will not execute any statement. It can also be used to make a body as empty in loops,classes etc.

In this section we use pass statement within loops to create delay in execution. Lets see an example:

```
for i in range(1,11):
    for j in range(10000000):
        pass
    print("Hello ",i)
```

OUTPUT:

```
Hello  1
Hello  2
Hello  3
```

```
Hello  4
Hello  5
Hello  6
Hello  7
Hello  8
Hello  9
Hello  10
```

The inner for loop runs for 10^7 times but does nothing due to pass statement. The pass statement is the body of inner loop. It means the print statement that is part of outer for loop executes once inner loop finishes. This cause a delay in printing.

4.8 Ponderable Points

1. Python supports while and for loop.

2. The while loop is general purpose loop but for loop is specially used with sequences or collections.

3. A null statement is represented by a pass statement.

4. Null statements are useful to create time delay.

5. The break keyword causes to come out from loop immediately.

6. The continue statement skips the current iteration and start from new iteration.

7. Nesting of both while and for loop can be easily done.

8. The range object allows us to create different types of ranges but it must be used inside list constructor for generation of list.

5. Functions

5.1 Introduction

A function is a self-contained block of code written once for a specific purpose but can be used again and again. A function is a basic entity for python programming language. They are the basic building blocks for modular/procedural programming. Function allow us to organize our code into small manageable blocks that are written and combined to make a big code. Another modular approach other than functions is class and modules. Functions help us to reuse the code that have been written once instead of repeating same set of code multiple times.

There are number of functions which we have used so far like **print, input, int, float, bin, oct, hex, len** etc. All these functions are library functions or built-in functions i.e. they are already there in the python programming language and we can use them in our programs. All of these functions belong to some module (modules are discussed in chapter 10). For example the default module imported is **builtins** to which **print** and **input** functions are part of.

We can simply use the built-in functions in our program, but we cannot modify them as their code is not available to the programmer. Some of the modules are written in C and they are part of python interpreter, so you cannot see the code of those modules like math or sys module. The others are in in the form python files (.py extensions).

Functions are first-class objects in Python, which means the function names can be assigned to another variable, and they can be used as elements of a list or in the dictionary. It also means that you can use functions as arguments to other functions, store functions as dictionary values, or return a function from another function. This allows to use functions in more meaningful and powerful ways in number of programming situations.

We can write our own functions depending upon requirements and all those functions will be called user-defined functions.

5.2 The Function Syntax

```
def functioname(argument(s)):
    '''
        description of what function
    does
    '''
    statement(s)
        return statement (optional)
```

The **def** keyword is used to create a function in Python. Following **def** keyword we have the *functioname* which can be any name which follows the rules of writing identifiers. The function may or

may not take any arguments. If no arguments are there we just write ( ) (opening and closing parenthesis) which is exactly called the function symbol. If arguments are present they are written inside parenthesis separated by comma. The colon after the right parenthesis start function as a block and all statements are written as body of the function. All these statements must be indented either by 4 spaces or by tab but not a mix of both. If you want to return some value from function than return statement can be used.

For every function python let you create a documentation string where within triple double quotes or triple single quotes you can write a description of function. This may include what the function does and its parameters or how to use the function.

5.3 Examples of function

Let's see our first example of function in python

```
def fun():
    """
    My first function for demo purpose
    """
    print("First function in python")
fun()
print(fun.__doc__)
```

OUTPUT:

```
First function in python
    My first function for demo purpose
```

In the above **fun** is the function name function does not take any arguments. Whenever you have **( )** after any name then that is a function. Obviously you may have arguments within **( ).** The string written within triple double quotes is the doc string for function **fun** and it can be accessed outside the function by the function attribute __doc__. The documentation string is ignored by the python interpreter.

Actual working of the function is done by the definition /body of the function which is just one print statement:

```
print("First function in python")
```

The next line is not indented with **fun** function. Writing just **fun( )** means we are calling the function **fun ( ).** Whenever **fun()** statement is encountered then control is transferred to the body of the function and all the statements written within the body of the function gets executed. When last statement of the function body is executed than function return to the next statement after from where the function was called. In this case it is

```
print(fun.__doc__)
```

In python every function will be called from some other function. Assume default function present is **main** function (*Kind of hidden and will be discussed during module*) where you write all your python code. Here we have called **fun()** from within **main().** So **main()** is called **calling** function and **fun()** is called **called / callee** function.

Let's have another example with a small change from previous one.

```
def fun():
    """
    Demo function fun
    """
    print("demo function fun")
print("In default main")
fun()
print("Back in default main")
```

As the program executes controls goes to **print** which prints **In default main** on the screen. Then when **fun()** is encountered control is transferred to the definition of **fun()** and executes all the statements within its body, here it prints **demo function fun** .As stated in the previous explanation that whenever a function returns it goes back to the next statement after from where it was called and here the next statement is **print** which displays **Back in default main** on to the screen and program terminates.

5.4 Illustrative Examples

```
# Script 5.1  working with two functions
def fun():
    """
    Demo function fun
    """
    print("In function fun")
def show():
    """
    Demo function show
    """
    print("In function show")
print("In default main")
print("calling function fun")
fun()
print("calling function show")
show()
print("Back in default main")
```

OUTPUT:

```
In default main
```

```
calling function fun
In function fun
calling function show
In function show
Back in default main
```

Here we are working with two functions. As clear from the output initially first two **print** statement in our default **main() function** is executed then **fun()** is called and **print** within **fun()** is executed . When **fun()** returns it returns to print statement within default main function that displays ‘calling function show;. The call **show()** calls show function and **print** within **show()** is executed. When **show()** is returned last **print** statement in default **main()** executes and program terminates.

For brevity, the doc string is omitted in all the functions presented next.

```
# Script 5.2 working with three functions
def fun1():
    print("In function fun1")
def fun2():
    print("In function fun2")
def fun3():
    print("In function fun3")


print("In default main")
fun1()
fun2()
fun3()
print("Back in default main")
```

OUTPUT:

```
In default main
In function fun1
In function fun2
In function fun3
Back in default main
```

Not much is there to explain in this program. We have three functions which are called one by one.

```
# Script 5.3  Nesting of functions
def fun2():
  print("In fun2 function, called from fun1")
  print("Returning from fun2")
def fun1():
```

```
  print("In fun1 function")
  print("Going in fun2")
  fun2()
  print("Back in fun1,going back to main")
print("In main")
print("Going in function 1")
fun1()
print("Back in main")
```

OUTPUT:

```
In main
Going in function 1
In fun1 function
Going in fun2
In fun2 function, called from fun1
Returning from fun2
Back in fun1,going back to main
Back in main
```

In this program we have called **fun1()** from default **main()** and inside **fun1()** we have called **fun2().** When **fun2()** returns control returns to the next **print** statement following which **fun1()** returns.

I think at this stage you must have understood the basic idea of functions which does not take any argument and which does not return any value. Now we turn our attention towards functions which accept parameters but does not return any value.

5.5 Passing parameters to functions

The general syntax of this type of functions is as follows:

```
def function_name (arg1,arg2, ):
                doc string
                      statement(s)
```

Basic idea of syntax of function we have covered earlier. In this section we see function codes where parameters are passed to the functions, but no value is returned from function. Let's see some illustrative examples.

```
# Script 5.4 display of single integer through function
def show(a):
print("U entered = ", a)
x=int(input("Enter a number\n"))
show(x)
```

```
OUTPUT:
Enter a number
34
U entered =  34
```

The function **show** accepts just one parameter **x** that can be anything: integer, float, string, list, dictionary etc. but here we have taken an integer from user and stored in **x** in default main function. The same **x** is passed to **show** function during a call as **show(x)**. This gets collected in **a** and **print** statement inside function executes. When we call the function we pass an **integer** value as(though any type value can be passed) shown by statement **show(x).** Here we are calling the function **show** and passing **a** as argument. **This is known as call by value as we are calling the function and passing value of the variable x**. The **x** value sent must be collected in some variable in the function definition. We collect this value in variable **a.** Note that number of arguments must match when defining the function. Variable **x** in function **main** is called **actual argument** and variable **a** in function **show** is called **formal argument.** When we pass **x** to function **show** a copy of **x** is sent which is collected in **a.** In the function **show** we simply print the value of **a**.

Let's modify the above code little bit as:

```
def show(a):
    print("a in show = ", a)
    a=a+10
x=10
show(x)
print("x in main=",x)
```

In the **show** after printing **a** we have incremented it by **10**. When function returns we print the value of **x** in the main . The value of **x** remains same as it was prior to sending to function **show.** ***Any change performed on the formal parameter does not reflect back in the actual parameters when functions are called by value i.e. when value of actual parameters is send to the functions***.

```
# Script 5.5 Function with two arguments
def show(name,age):
    print("Hello ",name," you are",age,"years old")
show('Ajay',15)
OUTPUT:
Hello  Ajay  you are 15 years old
```

The **show** function takes two arguments: **name** and **age**. As python is dynamic type language so anything can be passed to function **show** during the call. Here during call we have just passed a string 'Ajay' and 15 as integer data. In the body of the **show** function the print statement executes and

displays:

```
Hello  Ajay  you are 15 years old
```

```
# Script 5.6 sum of two float numbers using function
def sum(x,y):
  t=x+y
  print("Sum=",t)
a=float(input('Enter first number\n'))
b=float(input('Enter second number\n'))
sum(a, b)
```

OUTPUT:

```
Enter first number
1.4
Enter second number
5.6
Sum= 7.0
```

We input two numbers in **a** and **b** from user and pass the same to function **sum.** The function **sum** takes two parameters and finds their sum. The actual argument **a** and **b** are collected in formal argument **x** and **y**. Inside **sum** we declare a local variable **t** and store the sum **x+y** in **t** . In the end we display **sum** thorough **print**.

The definition may be written using small number of statements as

```
def sum(x,y):
    print("Sum=",x+y)
```

```
# Script 5.7 max of two float numbers using function
def max(a,b):
    m=a if a>b else b
print("max=",m)
a=float(input('Enter first number\n'))
b=float(input('Enter second number\n'))
max(a, b)
```

OUTPUT:

```
Enter first number
23
Enter second number
```

```
45
max= 45.0
```

In the program we are finding max of two numbers using function. We accept two numbers from user and pass to the function. In the function body we have found out maximum using **simple if else** and stored the same in **max** which is displayed through **print.**

One limitation of the above code is that it does not check equality of two numbers. See the next script that removes this limitation.

```
# Script 5.8 Maximum of two numbers with equality option
def max(a,b):
    if a==b:
        print("both are equal")
    elif a>b:
        print("max=",a)
    else:
        print("max=",b)
a=float(input('Enter first number\n'))
b=float(input('Enter second number\n'))
max(a, b)
```

OUTPUT:

```
Enter first number
23
Enter second number
24
max= 24.0
```

The code is easy to understand where we have else-if ladder inside function for finding maximum of two numbers along with checking equality of two numbers.

```
# Script 5.9 sum and average of three numbers
def sumavg(a,b,c):
  s=a+b+c
  avg=s/3
       print("Sum=",s)
  print("Average=%6.2f"%avg)
x,y,z=input('Enter three numbers separated by space\n').split()
x=float(x);y=float(y);z=float(z);
sumavg(x,y,z)
```

OUTPUT:

```
Enter three numbers separated by space
34 51 12
Sum= 97.0
Average= 32.33
```

The function **sumavg** finds sum and average of three numbers passed. The program is self-explanatory.

```
# Script 5.10 To check number is even or odd using function
def evenodd(num):
    s="odd" if num%2 else "even"
print(num," is",s)
n=int(input("Enter an integer number\n"))
evenodd(n)
```

OUTPUT:

```
Enter an integer number
23
23  is odd
```

We input an integer number from user and pass it to function **evenodd** which checks whether number is even or odd. If remainder by 2 is 0 then number is even else number is false. If number is odd then odd is assigned to s else even is assigned to **s**. The function can be used anywhere we want to check number is even/odd.

```
# Script 5.11 Display decorated name using function
def decorate(name):
    l=len(name)+7+1
    line = '+' + '-'*l + '+'
    print(line)
    name="|Welcome "+name+"|"
    print(name)
    print(line)
name=input('Enter your name\n')
decorate(name)
```

OUTPUT:

```
Enter your name
Juhi Jain
+-----------------+
```

```
|Welcome Juhi Jain|
+-----------------+
```

The function **decorate** accepts just one parameter: the name to be decorated. Inside the function we find the length of the name using len function and add 8 to it (length of Welcome is 7 and 1 space for separating Welcome and name. The line variable stores the fancy line to be displayed over top and bottom of the new string. The name is now modified by concatenating **|Welcome** before the name and adding **|** at the end. The resultant name and line are then displayed.

```
# Script 5.12 To find Greatest Common Divisor of two numbers
def gcd(a,b):
    while b:
        a,b=b,a%b
print("GCD=",a)
x,y=input("Enter two numbers\n").split()
x=int(x);y=int(y)
gcd(x,y)
```

OUTPUT:

```
Enter two numbers
99 78
GCD= 3
```

The greatest common divisor of two numbers can easily be find out using Euclid algorithm. The algorithm is :

EUCLID(a,b)

1. While b!=0
2. t=a
3. a=b
4. b=t%a
5. gcd is a

Here the code is shortened using parallel assignment. The code works for as for input:99 78

a	b
99	78
78	99%78=21
21	78%21=15
15	21%15=6

6	15%6=3
3	6%3=0

5.6 Function with parameters and return type

The general syntax of this category of this type of functions is as follows:

```
def function_name (data_type arg, ):                    {
                    doc string
                    Statement(s)
                    return value
```

Here **return** is the keyword and value can be any type of value that is returned by the function. Lets see number of examples to understand this category of functions.

```
 # Script 5.13 compute square of function and return using function
def sqr(n):
     """
     function returns square of n
     input:n
     output:n*n

     """
     s=n*n
     return s
num=int(input("Enter a number\n"))
sn=sqr(num)
print("Square of",num,"is=",sn)
```

OUTPUT:

```
Enter a number
5
Square of 5 is= 25
```

In the definition/body of the function we calculate **n*n** into **s** and return the value of **s** through statement **return s.** When this happens the **s** returns at the place from where the function **sqr** was called so the statement **sn=sqr(num)** becomes **sn=s**. If value of **n** happens to be **5, s** will have **25** when it returns and the same will be assigned to **sn.** The definition of function can be modified as:

```
def sqr(n):
```

```
        return n*n
```

In these types of functions the documentation string is quite helpful where you can clearly state what the function is doing , what is input and what the function returns. You can write this in any function which you think will help the user who wants to get information regarding functioning of the function.

Writing this in main function : **`print(sqr.__doc__)`** gives the following output:

```
 function returns square of n
     input:n
     output:n*n
```

```
# Script 5.14 Area of circle using function
def area(r):
     return 3.14*r**2

num=float(input("Enter the radius\n"))
ans=area(num)
print("Area of circle=",ans)
```

OUTPUT:

```
Enter the radius
2
Area of circle= 12.56
```

The function **area** takes just one argument **r** which represents radius of the circle**.** Inside the function we pass the radius of **float** type which we input from user in the **main.** The function calculates the area of the circle and return the area from the function which is collected in the variable **ans** in the **main. The same is then displayed .**

```
# Script 5.15 Maximum of two numbers using function
def max(a,b):
 return a if a>b else b
a,b=input("Enter two numbers separated by space\n").split()
a=float(a);b=float(b);
m=max(a,b)
print("Maximum=",m)
```

OUTPUT:

```
Enter two numbers separated by space
4.5 6.7
Maximum= 6.7
```

The program is quite simple. It finds max of two number using function **max** which accepts two

arguments of type **float** and return maximum of the two passed *variables*. The advantage of returning value from function is that it can be used as a placeholder in other or same function. Let's see an example in below script.

```
# Script 5.16 Maximum of three numbers from a function which find  maximum of two # numbers
    def max(a,b):
     return a if a>b else b
    a,b,c=input("Enter three numbers separated by space\n").split()
    a=float(a);b=float(b);c=float(c)
    m=max(max(a,b),c)
    print("Maximum=",m)
    OUTPUT:
    Enter three numbers separated by space
    1.2 4.5 3.2
    Maximum= 4.5
```

In the expression **m=max (max(a,b),c)** first **max (a,b)** is called and when this returns the call **max(a, b)** is replaced by the maximum of two either **a or b** say **t** (assume). Then again function **max** is called which returns max of **t and c.** Note how we have calculated maximum of three numbers with a function which finds maximum of two number only. The function can also be called as: **m=max(a,max(b,c))**

On the similar basis you can calculate max of 4 numbers by writing

```
    m=max(max(a,b),max(c,d))                                    .
# Script 5.17 To find factorial of a given number using function
    def fact(num):
        if num<1:
            return 1
        else:
            f=1
            for i in range(1,num+1):
                f=f*i
            return f
    n=int(input("Enter an integer number\n"))
    ans=fact(n)
    print("Factorial of ",n,"is=",ans)
    OUTPUT:
    Enter an integer number
    10
```

```
Factorial of  10 is= 3628800
```

The logic for finding factorial was explained in the **chapter 4.** Here we have put whole of the logic in the function **fact.** In the function we pass the number whose factorial is to be finding out. The function calculates the factorial and returns the factorial of the number.

```
# Script 5.18 to check whether given number is prime or not using function
def prime(n):
    c=2
    flag=True
    while c<=n//2:
        if n%c==0:
            flag=False
            break
        c=c+1
    return flag
num=int(input("Enter the number\n"))
print(num,"is prime") if prime(num) else print(num,"is not prime")
```

OUTPUT:

```
Enter the number
23
23 is prime
```

The number to be checked for prime is passed to function **prime** which checks whether number is prime or not. If it is prime then **flag** which is returned contains **True** and if it is not then **flag** returns **False**. In the **main** using simple if-else this returned value is checked and received in **f** and depending upon **0 or 1** we display appropriate output on to the screen..

```
# Script 5.19 To find reverse of a given number using function
def reverse(n):
    rev=0
    while n!=0:
        r=n%10
        rev=rev*10+r
        n=n//10
    return rev
num=int(input("Enter the number\n"))
ans=reverse(num)
```

```
print("Reverse Number is ",ans)
```

OUTPUT:

```
Enter the number
23456
Reverse number is 65432
```

The logic to find reverse of a number has been given in **chapter 4**. The same is presented over here but within function. We simply pass the number and the function return the reverse of the number.

```
# Script 5.20 To check whether a given number is +ve ,-ve or zero
# using function
def status(n):
    if n>0:
        return "Positive"
    elif n<0:
        return "Negative"
    else:
        return "Zero"

num=int(input("Enter any number\n"))
print("Number is ",status(num))
```

OUTPUT:

```
Enter any number
45
Number is  Positive
```

In the function **status** we return **'Positive', 'Negative'** or **'Zero'** if the number happens to be positive, negative or zero respectively. This returned value is directly printed in the print statement.

5.7 The default return type

*When function does not return any value and yet you use the function as if it is returning a value then default value **None** is returned*. See the code given below:

```
def sum(a,b):
 print("sum=",a+b)
x=sum(10,20)
print(x)
```

OUTPUT:

```
Sum=30
None
```

Here the function **sum** is performing sum of **a** and **b** and displaying only without returning the sum . In the main you are storing the return value in **x** though **sum** function is not returning any value. But here the function **sum** returns a value **None** which is stored in **x** and gets printed. The other way to call the sum function is:

```
print("Sum",sum(10,20))
```

In this case output will be:

```
sum= 30
Sum None
```

The same thing you can also try in Python shell

```
>>> def sum(a,b):
...     print(a+b)
...
>>> x=sum(3,4)
7
>>> x
>>> print(x)
None
```

Note here directly seeing the value of **x** shows you nothing. You have to use **print** statement to see the value **None** stored in **x**.

5.8 Function with default arguments

In Python it is possible for a function not to specify all its arguments. Some of the arguments may be specified as default values . When a function having default argument is called, python checks for the number of argument as well as checks which arguments are default. If the argument was default and was not specified during function call, default value of that argument is assumed. In case we provide a new value, default argument is overridden. For example

```
def show(x, y=20):
statement(s)
```

The function **show** takes two arguments, out of which second argument from left is default. In case function is called as **show(10)** , default value of **y** i.e. **20** is assumed. If function is called as **show(10,100)** than default value of **y** i.e. **20** is overridden.

In a function with default argument, if one argument is default, all successive arguments must be default. We cannot provide default values in the middle of the arguments or towards left side. We provide few examples:

1. **def fun(x, y=20,z=35):**
   function body
2. **def fun(x, y=30,z):**
   function body
3. **def fun(x=45,y):**
   function body

Out of three examples given only **1** is valid and **2** and **3** are invalid. In the **2** middle argument is default and the next argument **z** is not default. In the **3** first argument is default and next argument is not default.

See some examples given below:

```
# Script 5.21 Demo of function with default arguments
def show(y=10):
 print("Function argument=",y)
print("Called show with argument 20")
show(20)
print("Called show without argument")
show()
```

OUTPUT:

```
Called show with argument 20
Function argument= 20
Called show without argument
Function argument= 10
```

The function **show** takes just one argument which is a default argument. In the function call **show(20),** the default argument is overridden and **y** takes the value **20**. In the function call **show()** as no argument was specified , the compiler assumes default value **10** for **y**.

In the above example we have called the function with an integer value. The function can be called with any other value like:

```
show("Hello")
show(23.45)
show(True)
show('A')
show([2,3,4])
```

```
# Script 5.22 Demo of function with two default arguments
def show(x,y=10):
```

```
 print("Function arguments=",x,"and ",y)
show("Hello")
show(23.45,234)
show(True,False)
```

OUTPUT:

```
Function arguments= Hello and  10
Function arguments= 23.45 and  234
Function arguments= True and  False
```

In the function call **show("Hello")**, "**Hello"** is passed to **x** and **y** takes default value **10**. In the function call **show(23.45,234) x** takes value **23.45** and default parameter is overridden so **y** takes value 2**34.** Similar kind of explanation applies to show(True,False).

```
# Script 5.23 Demo of function with three default arguments
def show(x=10,y=20,z="Hello"):
  print("Function arguments=",x,",",y,"and",z)
show("First")
show(23.45,234)
show("First","Second","Third")
show()
```

OUTPUT:

```
Function arguments= First , 20 and Hello
Function arguments= 23.45 , 234 and Hello
Function arguments= First , Second and Third
Function arguments= 10 , 20 and Hello
```

In the function **show** all three parameters are default. In the function call **show(First)** , first default argument is overridden and other two default are used. Next function call **show(23.45,234)** overrides first two default arguments and third default argument is used. The last call **show()** uses all default arguments.

```
# Script 5.24 Demo of function with default arguments, finding bonus for
# an employee
def incr(sal,bonus_pr=10):
    sal = sal*(1+ bonus_pr/100)
    return sal
salary=float(input("Enter the salary\n"))
if salary>=10000:
```

```
    newsalary=incr(salary,15)
else:
    newsalary=incr(salary)
print("Salary =",salary)
print("Bonus=%7.2f"%(newsalary-salary))
print("Gross Salary=%7.2f"%newsalary)
```

OUTPUT:

```
(First Run)
Enter the salary
8000
Salary = 8000.0
Bonus= 800.00
Gross Salary=8800.00
(Second Run)
Enter the salary
12000
Salary = 12000.0
Bonus=1800.00
Gross Salary=13800.00
```

The function **incr** finds the new salary after adding bonus to the original salary. If salary is **<10000** we provide a bonus of **10%** of **sal** else we provide bonus of **15%** of salary. The incremented salary is returned. The code does not check when entered salary is negative.

Script 5.25 To find simple interest using function with default arguments

```
def SI(p,rate=8.5,time=1):
    si=p*rate*time/100
    return si
si=SI(1000)
print("Simple Interest=",si)
si=SI(1000,9.5)
print("Simple Interest=",si)
si=SI(1000,10.5,4)
print("Simple Interest=",si)
```

OUTPUT:

```
Simple Interest= 85.0
```

```
Simple Interest= 95.0
Simple Interest= 420.0
```

The function **SI** calculates simple interest for given principal,rate and time. Default values of time and rate are 1 and 8.5 respectively. In the main we call the function **SI** 3 times with 1, 2 and 3 arguments. Rest is easy to understand.

5.9 Call by name

In general, when we call the function the parameters are passed from left to right. That's why we had the restriction that when one argument is default all other must be default or it must be the right most argument. But when we know what the names of various parameter the function is having, we can call the function by name by passing parameters with name in any order. Let's understand by an example:

```
def SI(p,rate=8.5,time=1):
    return p*rate*time/100
print("Simple Interest=",SI(p=1000))
print("Simple Interest=",SI(time=2,p=5000,rate=5))
print("Simple Interest=",SI(rate=9.5,p=2000,time=2))
```

OUTPUT:

```
Simple Interest= 85.0
Simple Interest= 500.0
Simple Interest= 380.0
```

Note the order of parameters in the second and third call to function **SI**, as we are passing parameters by name we can select any order. For example, we have passed **time** as first parameter, **p** as second and **rate** as third parameter in the function call to **SI**.

Its not necessary to have default arguments when passing parameters by name. See one more simple example:

```
def personinfo(name,age,sex):
    if sex=='Male':
        print("Hello Mr",name,"You are",age,"years old")
    else:
        print("Hello Ms",name,"You are",age,"years old")
personinfo(age=21,name='Tarun',sex='Male')
personinfo(sex='Female',name='Bani',age=19)
```

OUTPUT:

```
Hello Mr Tarun You are 21 years old
Hello Ms Bani You are 19 years old
```

The code is easy to understand.

5.10 Returning more than one value

In all earlier examples of functions where function returns a value, it returned only one value. But in python the function can return more than one value. Just separate the returned values by comma and during call save all returned values in equal number of variables. See one example:

```
def mfun(a,b):
 return a+b,a-b,a*b,a/b,a**b
s,su,m,d,p=mfun(3,7)
print(s,su,m,round(d,3),p)
```

OUTPUT:

```
10 -4 21 0.429 2187
```

The function **mfun** performs addition, subtraction, multiplication, division and power operation of two numbers **a** and **b** and returns the resultant values. The same is stored in variables **s,su,m,d,p** respectively. The **round** function is used to display result upto 3 places after decimal point.

In reality, only value is returned in the form of a **tuple** but these values are unwrapped in equal number of variables on the left hand side. Lets rewrite the previous code as:

```
def mfun(a,b):
 return a+b,a-b,a*b,a/b,a**b
ans=mfun(3,7)
print(ans)
```

OUTPUT:

```
(10, -4, 21, 0.42857142857142855, 2187)
```

If you want you can have access to the above tuple elements as ans[0],ans[1] upto ans[4] or using for loop you can traverse them:

```
for a in ans:
    print(a,end=' ')
```

Other than using the tuple way to return the multiple values from function, you can make use of list and dictionary also. Though they are covered in the coming chapters, but we present examples here as you won't find them difficult to understand.

```
# List version
```

```
def mfun(a,b):
 return [a+b,a-b,a*b,a/b,a**b]
ans=mfun(3,7)
for a in ans:
    print(a,end=' ')


#Dictionary Version
def mfun(a,b):
    d=dict()
    d[0]=a+b;d[1]=a-b
    d[2]=a*b;d[3]=a/b
    d[4]=a**b
 return d
ans=mfun(3,7)
for k in ans.keys():
    print(ans[k],end=' ')
```

List version is easy to understand. In dictionary version for every key we assign the items. For key 0 , value **a+b** is assigned, for key 1 , value is **a-b** and so on. The returned dictionary is stored in **ans**. For every key **k** belongs to **{0,1,2,3,4}** which is returned by **ans.keys()**, value is displayed by writing **ans[k]**.

5.11 Global variables in functions

All variables that are defined within functions as we have seen in earlier programs in this chapter are local variables. These variables have their scope limited to functions in which they were defined. They cannot be used outside the function. The variables which are defined in main are global and they can be used in any function. The keyword **global** is useful in these situations. Let's understand using some examples:

```
x = 10
def func1():
 return x*x
def func2():
 return x**2
print(func1())
print(func2())
print(x)
```

```
OUTPUT:
100
100
10
```

The variable **x** is global and is used in both the functions **func1** and **func2 .** Both work on the global variable **x** *but do not modify them*. That's why the **print** statement prints the original value 10 of **x**. Let's modify the value now in the next code.

```
x = 10
def func1():
  x=20
def func2():
  x=30
print(func1())
print(func2())
print(x)
```

```
OUTPUT:
None
None
10
```

Both the functions are not returning the values, so **None** is returned and last **print** statement prints 10. *The global **x** was not modified at all. This is because **x** defined in both the functions is treated as local and not global*. To use the global variable inside a function you have to declare the global variable inside the function as:

```
global globalvariablename
```

Let's modify the above code now and see the result:

```
x = 10
def func1():
    global x
 x=20
def func2():
    global x
 x=30
print(func1())
print(func2())
print(x)
```

```
OUTPUT:
```

```
None
None
30
```

This time we get the results as expected. First the global **x** was modified in **func1** to value **20** and same was modified to **30** in **func2**. Again with a little change in code:

```
x = 10
def func1():
    global x
 x=x+10

def func2():
    global x
 x=x+20
func1()
func2()
print(x)
```

OUTPUT:

```
40
```

The first function modifies global **x** to **20** and same is modified by **func2** to **40**. When we print **x** in the main we get the last value of **x** which was modified by **func2**.

What if we want to use both the local and global variable together in function. See one example:

```
x = "global"
def func1():
    y="local in func1"
    print(y)
    global x
    x=x+y
def func2():
    y="local in func2"
    print(y)
    global x
 x=x+y
func1()
func2()
print(x)
```

OUTPUT:

```
local in func1
local in func2
globallocal in func1local in func2
```

In the code above we have two local variables **y** in each of the function **func1** and **func2**. The global variable **x** is modified in both the functions. In the main the function **func1** is called first. This gives output:

`local in func1`

and sets **x** to the value `globallocal in func1.` When **func2** is called in main it prints

`local in func2`

and **x** now becomes `globallocal in func1local in fun2.` Here the global **x** which was modified by **func2** is used in **main**.

But what if we want to keep local and global variable by the same name. Let's see the next code:

```
x = "global"
def func1():
    x="func1 local"
    print(x)
    global x
    x="global in func1"
def func2():
    x="func2 local"
    print(x)
    global x
 x="global in func2"
func1()
func2()
print(x)
```

OUTPUT:

```
Syntax Error: name 'x' is used prior to global declaration:
```

The above code generates error as python does not allow to use global and local variable having same name and to be used at the same time. But there is a way around. Luckily python provides a global dictionary in the form of symbol table that keeps values of all global variables used within a single python file. This is used as : **globals ( )['globalvariablename'].**

See one small example:

```
x='global'
def fun():
```

```
    x='local'
    print(x)
    print(globals()['x'])
    globals()['x']='modified global'
fun()
print(x)
```

OUTPUT:

```
local
global
modified global
```

As discussed above to access the value of global variable **x** within a function having a local variable **x** we can make use of globals dictionary. In the function fun we can easily use a local **x** and global **x** by writing **globals()['x']**.

One final example before we wrap up this section:

```
x = "global"
def func1():
    x="local in func1"
    print(x)
    globals()['x']="In func1" + globals()['x']+"\n"
def func2():
    x="local in func2"
    print(x)
    globals()['x']="In func2 " + globals()['x']+"\n"
func1()
func2()
print(x)
```

OUTPUT:

```
local in func1
local in func2
In func2 In func1global
```

Try to figure out how the output has come?

5.12 Passing function as argument

A function can also be passed as an argument in another function. The reason behind that a function can easily be given a new name and when passed into another function during function call , it can be

collected into function formal parameters. It is because functions are first class objects in python just like integer, float, list, strings etc. Let's see a small example of function assignment in shell:

```
>>> def fun():
...     print("Hello")
...
>>> fun()
Hello
>>> x=fun
>>> x()
Hello
>>> y=fun
>>> y()
Hello
>>> L=[x,y,fun]
>>> for func in L:
...     func()
...
Hello
Hello
Hello
```

When you write **x=fun,** it means creating new name for the function.In other words you are creating alias for the function. Writing simply function name without () represents function address in memory. For example:

```
>>> x
<function fun at 0x000001AEE30C1EA0>
>>> y
<function fun at 0x000001AEE30C1EA0>
>>> fun
<function fun at 0x000001AEE30C1EA0>
```

Once an alias has been created for the function, the function can be called by that new name. Even function addresses can be stored in list and processed using **for** loop.

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

```
    return a-b
def mul(a,b):
    return a*b
L=[sum,sub,mul]
for fun in L:
    print(fun(10,20))
```

OUTPUT:

```
30
-10
200
```

In each iteration of **for** loop fun represents function **sum(10,20)**, **sub(10,20)** and **mul(10,20)**. It is obvious that number of arguments must be same in all functions stored as elements in the list **L**.

As you have got an idea now that function can be assigned and processed as if it is a simple data type, let's move our attention towards passing function as argument to functions. We will understand this using an example:

```
def mul(a,b):
  return a*b

def po(a,b):
  return a**b

def fun(a,b,f):
  return f(a,b)
print("Multiplication=",fun(2,3,mul))
print("Power=",fun(3,4,po))
```

OUTPUT:

```
Multiplication= 6
Power= 81
```

The function **fun** takes three arguments: first two here we are passing integers and last one is a function itself. In call **fun(2,3,mul)**, the reference of **mul** is stored in **f** and in the body when it executes **f(2,3)** internally it calls function **mul(2,3)**. Similar kind of arguments are applied to **fun(3,4,po)**. See how easy it is to pass function as argument to other functions.

5.13 Passing variable arguments to functions

Python has a special feature which allow us to pass variable number of arguments to functions. That feature is known as passing parameters with asterisk. A python function having single parameter with asterisk (*) as prefix can accept variable number of arguments. The parameter name serve as a tuple with variable number and types of arguments. The arguments cannot be keyword arguments as we will see in a short while. Let's see an example to understand this:

```
def demo(*args):
    for arg in args:
        print(arg,end=',')
       print()
demo(10)
demo(10,"Hello")
demo(True,"Python",123)
demo([1,2],2.34,False,"Last")
```

OUTPUT:

```
10,
10, Hello,
True, Python,123,
[1, 2],2.34, False, Last,
```

The function **demo** takes variable number of arguments (even zero) in the form of a tuple. For each argument in the tuple **args** we simply display it separated by comma. For leaving a new line between each function call we have used **print()** after **for** loop. In the main the **demo** function has been called with 1,2,3 and 4 arguments. The function even be called as: **demo()**.

As said above the arguments passed in the form of a tuple. Let's verify this:

```
def demo(*args):
   print(type(args))
demo(True,"Python",123)
```

OUTPUT:

```
<class 'tuple'>
```

Let's take some more examples of variable number of arguments to functions.

```
def fun(*args):
   sum=0
   for x in args:
      sum=sum+x
   print("Sum of",args,"=",sum)
fun(10)
fun(10,20)
```

```
fun(12,3.4,56,7.8)
```

OUTPUT:

```
Sum of (10,) = 10
Sum of (10, 20) = 30
Sum of (12, 3.4, 56, 7.8) = 79.2
```

The code is easy to understand. We are just summing variable number of arguments passed to functions.

In addition to passing variable number of arguments to functions if you want to pass some fixed number of arguments, you can do that too. Let's see an example:

```
def fun(first, *args):
  print("First argument=",first)
  print("Remaining arguments")
  for arg in args:
print(arg,end=' ')
fun("an","example","of","extra","argument")
```

OUTPUT:

```
First argument= an
Remaining arguments
example of extra argument
```

The first argument **an** to the function **fun** is fixed and rest all other are variable number of arguments. Not just one fixed argument, any number of fixed arguments are allowed but variable number argument syntax (***args**) must be the last one if using.

5.13.1 The keyword arguments (**kwargs)

Apart from passing variable number of non-keyword argument using ***args** syntax, you can also pass variable number of keyword arguments using ****kwargs** syntax. Every keyword has a value. The name of word is treated as keys of dictionary and name value is equivalent to corresponding value of the key. Let's understand using an example:

```
def fun(**args):
  print(args)
fun(name='kuhu',age=21)
fun(name='kuhu',age=21,city='delhi')
```

OUTPUT:

```
{'name': 'kuhu', 'age': 21}
{'name': 'kuhu', 'age': 21, 'city': 'delhi'}
```

First call of the function **fun** passed two keyword arguments: **name** and **age**. The corresponding values

are **'kuhu'** and **21**. Next function call passes one additional keyword argument: **city** with corresponding value **'delhi'**. As mentioned above the keyword arguments are passed in the form of dictionary with keyword name as keys and their values as values. Let's modify the above code and display keys and values from passed arguments.

```
def fun(**args):
  for k in args:
    print(k,":",args[k],end=' ')
  print()
fun(name='kuhu',age=21)
fun(name='kuhu',age=21,city='delhi')
fun(name='kuhu',age=21,city='delhi',job='lecturer')
```

OUTPUT:

```
name : kuhu age : 21
name : kuhu age : 21 city : delhi
name : kuhu age : 21 city : delhi job : lecturer
```

For every **k** which represents a key for the named argument,**args[k]** represent the value. Thus, for **k=name**, **args['name']** gives **'kuhu'** and for **k=age**, **args['age']** gives **21**. Rest is easy to understand.

The other way to write the above function **fun** is :

```
def fun(**args):
  for key,value in args.items():
    print(key,":",value,end=' ')
  print()
```

Here the function items of dictionary **args** gives a (key, value) pair in every iteration of the **for** loop, like **[('name', 'kuhu'), ('age', 21)]** for function call **fun(name='kuhu',age=21)**

To wrap up this section we provide an example to combine fixed argument, variable argument and variable keyword arguments. Let's see the code given below:

```
def fun(a,*args,**kargs):
  print("First argument=",a)
  print("Variable arguments")
  for x in args:
    print(x,end=' ')
  print()
  print("Keyword arguments")
  for k in kargs:
    print(k,kargs[k])
```

```
fun(20,"Hello",30,3.45,name='kuhu',age=21,city='delhi')
```

OUTPUT:

```
First argument= 20
Variable arguments
Hello 30 3.45
Keyword arguments
name kuhu
age 21
city delhi
```

5.14 Recursion

Recursion is a programming technique in which a function calls itself for several times until a condition is satisfied. It's a very important technique to understand and once understood many long listing of code can be reduced to a few number of lines. Recursion basically a word mostly used in mathematics to state a new term in previous term such as

$$X_{n+1} = X_n + 1 \text{ for } n>=1 \text{ and } X_1 = 1$$

Then we can calculate X_2 in terms of X_1 , X_3 in terms of X_2 and so on.

When solving a problem through recursion two conditions must be satisfied.

1. The problem must be expressed in recursive manner.
2. There must be a condition which stops the recursion otherwise there would be a stack overflow.

Let's write few programs which illustrate how recursion works.

```
# Script 5.26 Factorial of a number using recursion
def fact(n):
  if n<=1:
    return 1
  else:
 return n*fact(n-1)
num=int(input("Enter an integer number\n"))
ans=fact(num)
print("Factorial of",num,"is",ans)
```

OUTPUT:

```
Enter an integer number
8
```

```
Factorial of 8 is 40320
```

Assume **n is 4 ,** now recursion works as :

N	**function returns**
4	**4 * fact (3)**
3	**4 * (3 * fact(2))**
2	**4 * (3 * (2 * fact (1)))**
1	**4 * (3 * (2 * 1)))**

When recursion starts each call to function **fact** creates new set of variables (here only one). When ever recursion starts the recursive function calls does not execute immediately (in reality function addresses are put into stack). They are saved inside the stack along with the value of variables. **(A stack is a data structure which grows upward from max_limit to 1. Each new item is pushed in the stack takes its place above the previously entered item. The items are popped out in the reverse order in which they were entered ie last item is popped out first. That's why they are called LIFO (last in first out)).** This process is called **winding** in the recursion context. When recursion is stopped in the above program when **n becomes 1** and function return the value **1, all the function call saved inside the stack are popped out from the stack in the reverse order and get executed ie fact(1) returns to fact(2) ,fact(2) returns to fact(3) and in the end fact (3) returns to fact(4)** which ultimately return to the **main.** This process is called **unwinding.**

```
# Script 5.27 Display the entered number using recursion
def show(n):
  if n//10==0:
      print(n%10,end='')
      return
  show(n//10)
  show(n%10)
num=int(input("Enter a number\n"))
print("u entered")
show(num)
```

OUTPUT:

```
Enter a number
345
u entered
345
```

The program prints the number entered but this is done through recursion. Assume the number is **234.** We understand the process step by step

Step 1 n=234 the **if** condition is false recursion starts here by calling **show (234//10)** *i.e.* **show(23).**

Step 2 n=23 if condition is false, now **show (23//10***)* is called i.e. **show (2)** is called.

Step 3 n=2 if condition is true this time **2** is printed on the screen and function returns. As function returns for the value of **n=23** it returns to next statement from where it was called is **show (23%10)** i.e. **show (3)** is called.

Step 4 n=3 if condition is true and **3** is printed. When function returns it has executed all the statements of show function for **n=23** and it returns to **show(n%10)** statement for **n=234**. This time **show(4)** is called .

Step 5 n=4 if condition is true and **4** is printed and function returns to **main.**.

```
# Script 5.28 Fibbonacci Series using recursion
def fibbo_series(a,b,n):
     if n:
          print(a,end=',')
          n=n-1
          fibbo_series(b,a+b,n)
     else:
          return
num=int(input("Enter the number of terms\n"))
print("Fibonacci series up to",num,"terms")
fibbo_series(0,1,num)
```

OUTPUT:

```
Enter the number of terms
6
Fibonacci series up to 6 terms
0,1,1,2,3,5,
```

The code prints the fibbonacci series up to given **nth** term. In the function **fibbo_series** we pass **3** parameters. First two are the values from which next term of the series will be evaluated and last parameter is the number of term. Assume value of **n is 5**. When the function is called from **main** the **value of a is 0, b is 1 and n is 5.**

Step 1 n=5 if condition is true and value of **a** is printed i.e. **0** and recursion starts by calling **fibbo_series(1,1,4).**

Step 2 n=4 if condition is true and again **1** is printed, **fibbo_series(1,2,3)** is called.

Step 3 n=3 if condition is true and **2** is printed, **fibbo_series(2,3,2)** is called.

Step 4 n=2 **if** condition is true and **3** is printed, **fibbo_series(3,5,1)** is called.

Step 5 n=1 **if** condition is true and **5** is printed, **fibbo_series(5,8,0)** is called.

Step 6 n=0 **if** condition is false and function returns. .

```
# Script 5.29 nth term of Fibonacci series using recursion
def fib_ser_term(n):
    if(n==2 or n==3):
        return 1
    elif (n==1):
        return 0
    else:
        return fib_ser_term(n-1)+fib_ser_term(n-2)
n=int(input("Enter the number of terms: "))
print(n," th term of series is ",fib_ser_term(n))
```

OUTPUT:

```
Enter the number of terms: 6
6  th term of series is  5
```

The program finds **nth** Fibonacci series term through recursion. The function **fib_ser_term** works as follows assume **n is 6** so series will be

0 1 1 2 3 5 and the 6th term will be 5.

Step 1 n=6 recursion starts as **fib_ser_term(5)+fib_ser_term(4);**

Step 2 In the **step 1** the function is called twice one with value of **5** and another with value of **4**. Function call **fib_ser_term(5)** results in **fib_ser_term(4)+fib_ser_term(3)** and **fib_ser_term(4)** *results in* **fib_ser_term(3)+fib_ser_term(2)**

Step 3. Continuing in this manner when call **fib_ser_term(1)** and **fib_ser_term(0)** occurs then function returns **1.**

In short to get the result we show in the following manner

fib_ser_term(2)=fib_ser_term(1)+ fib_ser_term(0)

= 1 + 0

= 1

fib_ser_term(3)=fib_ser_term(2)+ fib_ser_term(1)

= 1 + 1

= 2

fib_ser_term(4)=fib_ser_term(3)+ fib_ser_term(2)

= 2 + 1

= 3

fib_ser_term(5)=fib_ser_term(4)+ fib_ser_term(3)

= 3 + 2

= 5

This last result **5** is returned to the **main** when unwinding finishes.

```
# Script 5.30 hcf of two numbers using recursion
def hcf(a,b):
  if(b==0):
    return a
  else:
    return  hcf(b,a%b)
a=int(input("Enter the first number\n"))
b=int(input("Enter the second number\n"))
ans=hcf(a,b)
print("hcf of two numbers is ", ans)
```

OUTPUT:

```
 Enter the first number
99
Enter the second number
78
hcf of two numbers is  3
```

The recursive method of finding the **hcf** of two numbers was given by renowned **mathematician Euclid**. We have used the same method in our function. The working is as follows for a=99,b=78

Step 1 a=99 , b=78 if condition is false function returns **hcf(78,99%78) is hcf(78,21).**

Step 2 a=78,b=21 if condition is false ,function returns **hcf(21,78%21) is hcf(21,15).**

Step 3 a=21,b=15 if condition is false ,function returns **hcf(15,21%15) is hcf(15,6).**

Step 4 a=15,b=6 if condition is false ,function returns **hcf(6,15%6) is hcf(6,3).**

Step 5 a=6,b=3 if condition is false ,function returns **hcf(3,6%3) is hcf(3,0).**

Step 4 a=3,b=0 if condition is true ,function returns **a ie 3** and this is our desired answer..

```
# Script 5.31 sum of series 1 + 2 + 3 + 4 + 5 + ..  n through recursion
def sum(term):
  if term>0:
    return (term+sum(term -1))
  else:
```

```
    return 0
n=int(input("Enter the number of terms\n"))
s=sum(n)
print("Sum of series up to",n,"terms is",s)
```

OUTPUT:

```
Enter the number of terms
10
Sum of series up to 10 terms is 55
```

The function **sum** is quite easy to understand. Assume **n is 5**

Then function will be returned as

5+sum(4)

5+4 + sum(3)

5+4 + 3+ sum(3)

5+4 + 3 + 2 +sum(1)

5+4 + 3 + 2 +1+sum(0)

5+4+3+2+1+0

The last expression will be summed up and returned to the calling function i.e. **main** when unwinding finishes.

On the similar ground you can find the sum of series: 1+3+5+7+9+. using following function:

```
def sum(term):
  if term>0:
    return (2*term-1+sum(term -1))
  else:
    return 0
```

5.15 The Tower of Hanoi puzzle.

The **Tower of Hanoi** or **Towers of Hanoi** is a mathematical game or puzzle. The puzzle was invented by the French mathematician **Edouard Lucas** in **1883**.It consists of three pegs, and a number of discs of different sizes which can slide onto any peg. The puzzle starts with the discs neatly stacked in order of size on one peg, smallest at the top, thus making a conical shape.

The object of the game is to move the entire stack to another peg, obeying the following rules:

- only one disc may be moved at a time
- no disc may be placed on top of a smaller disc

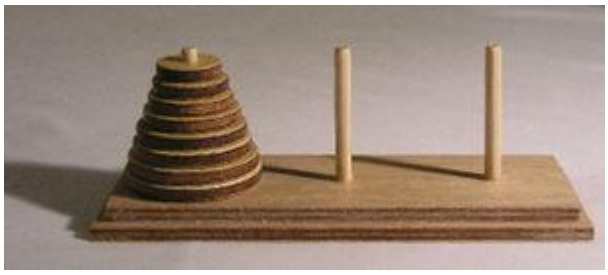


Fig 5.1:Tower of Hanoi (3 pegs,8 discs)

5.15.1 Solution as Recursive algorithm

1. label the pegs A, B, C -- these labels may move at different steps.
2. let ***n*** be the total number of discs.
3. number the discs from 1 (smallest, topmost) to ***n*** (largest, bottommost)
4. To move n discs from peg A to peg C using auxiliary peg B:

    4.1 Move **n-1** discs from A to B. This leaves disc **#n** alone on peg A.

    4.2 Move disc **#n** from A to C.

    4.3 Move **n-1** discs from B to C so they sit on disc **#n**.

The above is a recursive algorithm: to carry out steps 1 and 3, apply the same algorithm again for **n−1.** The entire procedure is a finite number of steps, since at some point the algorithm will be required for **n = 1**. This step, moving a single disc from peg A to peg C, is trivial.

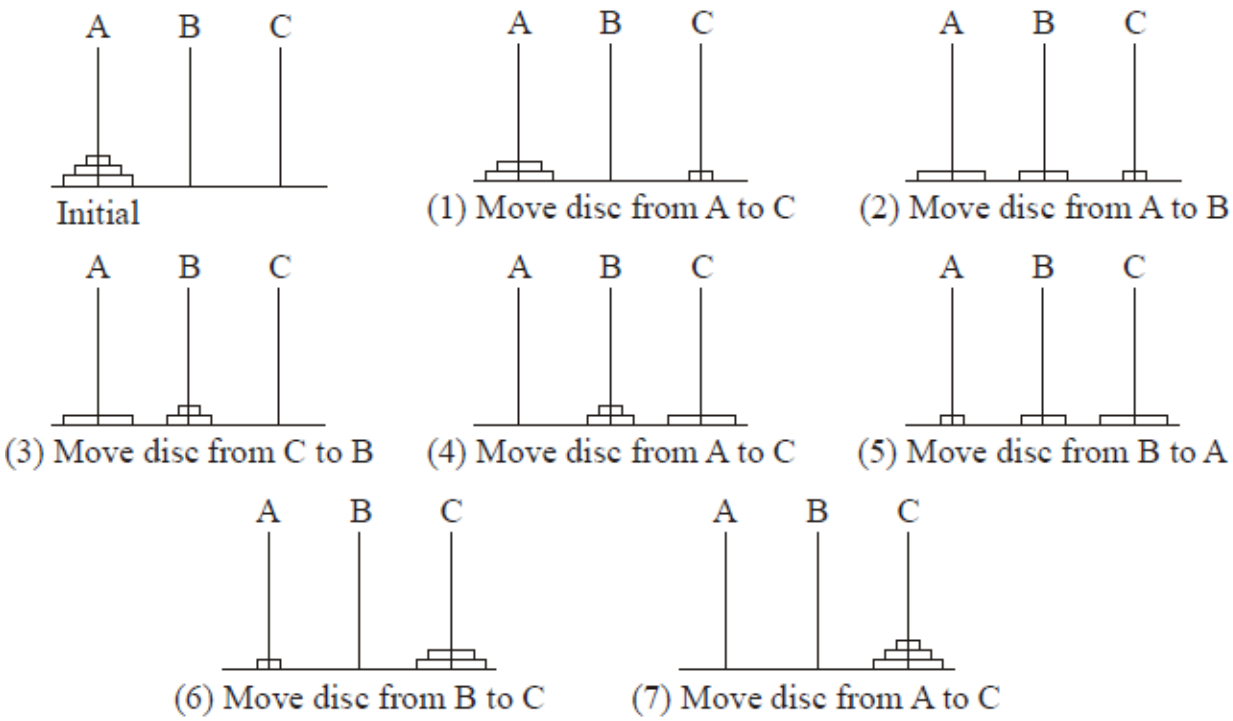


Fig 5.2 : All steps to move discs from A to C , B as auxiliary peg.

```
def TOW_HON(A,C,B,n):
  if(n<=0):
    print("Error, invalid disc number\n")
  if(n==1):
    print("Move disc from %s to %s"%(A,C))
  else:
    TOW_HON(A,B,C,n-1)
    TOW_HON(A,C,B,1)
    TOW_HON(B,C,A,n-1)

n=int(input("Enter the number of discs\n"))
print("The steps for the solution are")
TOW_HON("A","C","B",n)
```

OUTPUT:

```
Enter the number of discs
3
```

```
The steps for the solution are

 Move disc from Peg1 to Peg2
 Move disc from Peg1 to Peg3
 Move disc from Peg3 to Peg1
 Move disc from Peg1 to Peg2
 Move disc from Peg2 to Peg3
 Move disc from Peg2 to Peg1
 Move disc from Peg1 to Peg2
```

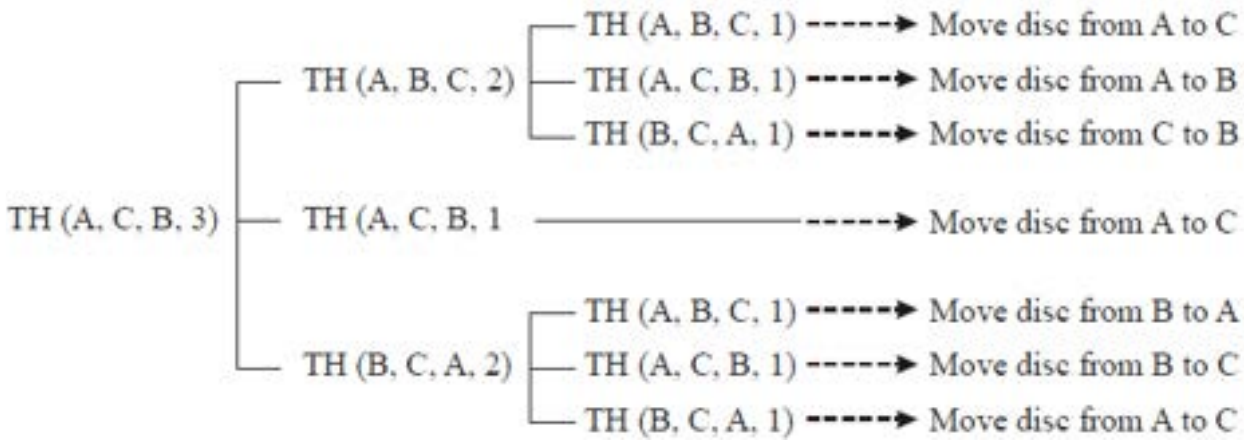


Fig 5.3 : Recursion unrolled for 3 discs

5.16 The lambda function

We have seen number of examples of creating functions in this chapter. All these functions are created using the def keyword and has a name. It is possible in python to create functions without any name. This can be done using the **lambda** keyword. Using this keyword, we can create some functions which are small, without any name and used only once. These functions are known as **lambda** function. The function can take any number of arguments, but the body must contain only a single expression. These are used whenever we need a function object in any expression or in function call. Further they can be passed to other functions also.

The syntax of creating lambda function is:

```
lambda arguments: expression
```

Here arguments are passed which the are evaluated by the expression and returned. The colon separates arguments and expression. You can notice that return statement is not required when using lambda function. Lambda functions are sometimes known as lambda operator. Let's understand this using an example in shell:

```
>>>fun=lambda x:x*x
>>> fun(2)
4
```

Here the lambda function is created that takes just single argument **x** and return **x*x** which is expression here. This is assigned to **fun** and function is called as **fun(2)**. This assigns **2** to **x** and expression **x*x** changes to **2*2** and **4** is returned. The function is equivalent to

```
def  fun(x):
  return x*x
```

The lambda function if required do not need to be assigned to any variable.

```
>>> (lambda x,y:x+y)(2,3)
5
```

5.16.1 The map function

As stated earlier lambda function can be used as an argument to other functions. Some of the most common built-in functions where this is useful are **map** and **filter**. The function **map** will be discussed here and filter in next section. Syntax of the **map** function is :

```
map(func,iterables)
```

The **map** function takes two arguments: first is the function and second is any iterables such as list, set, tuple. The function **func** is appled on every element of the iterable. Let's see in effect using an example both in shell and as a script:

(In Shell)

```
>>> def sqr(x):
...     return x*x
...
>>> L=list(range(10,16))
>>> L
[10, 11, 12, 13, 14, 15]
>>> LS=map(sqr,L)
>>> list(LS)
[100, 121, 144, 169, 196, 225]
```

#(Python Script)

```
def sqr(x):
    return x*x
L=list(range(10,16))
Ls=map(sqr,L)
print(list(Ls))
```

OUTPUT:

```
[100, 121, 144, 169, 196, 225]
```

Here we have a function **sqr** that finds square of a number supplied as argument. We create a list **L=[10,11,12,13,14,15]** using range and **list** function. The **map** function takes **sqr** as first argument and

L as second argument. The function **sqr** is called for every element of the list **L** and result is stored in **LS** (an new map object). The returned map object is to be converted into an iterable using list function so **list(LS)** gives us **`[100, 121, 144, 169, 196, 225]`**

Now we have seen a simple example of map function let's see how can be write the above code using **lambda** function. Just replace the function **sqr** in map function with the code: **lambda x: x*x**. So our script becomes:

```
L=list(range(10,16))
Ls=map(lambda x:x*x,L)
print(list(Ls))
```

Here **x** takes on values from the list **L** and performs **x*x** and returns.

In the above example the lambda function was taking just one argument. What if it takes more than one argument. Let's find sum of two numbers for every element at same position in the list as an illustrative example.

```
LS=map(lambda x,y:x+y,[10,20,30,40,50],[23,34,45,56,67])
print(list(LS))
```

`OUTPUT:`

```
[33, 54, 75, 96, 117]
```

For the first iteration **x** takes value **10** and **y** takes **23**, element at 0th position from both the list. For second iteration **x** and **y** takes element at 1st position that is **20** and **34**. In the similar manner all respective elements at all other positions are summed up and stored in the variable **LS**. The same is displayed after converting it into list.

Its not necessary that when lambda function takes two arguments length of two supplied lists in the map function should match. Number of elements can differ. Arguments equal to the smaller length are considered and extra elements from larger list are ignored. See an example:

```
>>> print(list(map(pow,[2,3,4],[3,2,5,5])))
[8, 9, 1024]
>>> print(list(map(pow,[2,3,4,5],[3,2,5])))
[8, 9, 1024]
```

5.16.2 The filter function

As the name suggests the filter function perform filtering of elements. The syntax is :

`filter(func,iterable)`

The function takes function func and only one iterable as argument. It returns all the elements for which the func returns true. Let's understand using an example:

```
>>> L=[2,4,7,9,12,34,57]
```

```
>>> LO=filter (lambda x:x%2,L)
>>> list(LO)
[7, 9, 57]
```

Here the lambda function checks x to be an odd number. For every element x in L if x%2 is not zero then x is returned. To check for even number just change expression to **x%2==0** or **not x%2.** This time we show as a script.

```
L=[2,4,7,9,12,34,57]
LE=filter (lambda x:not x%2,L)
print(list(LE))
```

OUTPUT:

```
[2, 4, 12, 34]
```

One more small example is to remove all zero elements from an iterable and keep non zero elements. See the next script

```
L=[2,4,0,9,0,34,0]
LE=filter (lambda x:x!=0,L)
print(list(LE))
```

OUTPUT:

```
[2, 4, 9, 34]
```

5.16.3 The reduce function

The reduce function was part of python 2 but removed in python 3. I think it's a useful function and we will cover this here. As it is not directly available we need to import **functools** to use it. It can be imported as: **from functools import reduce**

The **reduce** function is used to repeatedly apply a function to an iterable. Assume your iterable contains elements as [e1,e2,e3,e4,e5] then function func is applied to e1 and e2, result is say A. Then func is applied to A and e3,result is say B. Then func is applied to B and e4, result is say C. Finally the func is applied to e5 and C and result is returned. An example will clarify the above:

```
>>> from functools import reduce
>>> reduce(lambda x,y: x+y, [10,5,2,11])
28
```

Here lambda function calculates 10+5 and returns 15. Next lambda function calculates 15+2 and returns 17, then operation 17+11 is done and 28 is returned. That is the above **reduce** function is performing cumulative sum of all the elements.

If instead of + , you use * then it gives you cumulative multiplication of all the elements. Let's see some more examples:

```
# Finding maximum of a list of elements
>>> L=[4,56,23,45,78,89,12]
>>> x=reduce(lambda x,y:x if x>y else y,L)
```

```
>>> x
89
# Sum of values in a given range
>>> x=reduce(lambda x,y:x+y,range(1,11))
>>> x
55
# Script Finding factorial of a given number
from functools import reduce
num=int(input('Enter a number\n'))
if num<=0:
    print("Factorial is 1")
import sys;sys.exit()

fact=reduce(lambda x,y:x*y,range(1,num+1))
print("Facotiral of ",num,"is",fact)
```

5.17 Ponderable Points

1. Based on the nature of creation there are two categories of functions : built-in and user defined.
2. The **def** keyword is used to create a function.
3. The functions which are predefined and supplied along with the compiler are known as built-in
4. functions.
5. The function main ( ) is present in python but kind of invisible.
6. A function that performs no action is known as dummy function. It is a valid function. Dummy
7. functions may be used as a place-holder, which facilitates adding new functions later on. For
8. Example : **def dummy( ) :**
   pass
9. Advantages of recursion
   (i) Easier understanding the program logic.
   (ii) Helpful in implementing recursively defined data structures.
   (iii) Compact code can be written.
10. Use of **return** statement helps in early exiting from a function apart from returning a value from a function.
11. In a function with default argument, when one argument is default, then all successive arguments (towards right) must be default.

12. Function description can be written inside function in triple quotes and can be used as function_name.__doc__
13. Python supports calling function by name where order of argument name does not matter.
14. When function does not return any value and yet you use the function as if it is returning a value then default value **None** is returned
15. A function can return more than one value in the form of tuple, list or dictionary.
16. Use of global keyword allow us to use global variable inside a function.
17. python provides a global dictionary in the form of symbol table that keeps values of all global variables used within a single python file. This is used as : **globals ( )[‘globalvariablename’].**
18. A function is a first class object in python so function name can be assigned to another variable can be stored in list ,in dictionary or in any collection.
19. A python function having single parameter with asterisk (*) as prefix can accept variable number of arguments.
20. Apart from passing variable number of non-keyword argument using ***args** syntax, you can also pass variable number of keyword arguments using ****kwargs** syntax
21. Anonymous function can be created using lambda keyword.
22. The map function is used to apply a function onto any iterable.
23. The filter function is used to apply a function onto any iterable and filters elements when condition as stated in function is true.

6. Strings in Python

6.1 What is Python String ?

Strings in Python are sequence **of characters, digits, symbols enclosed within single or double quotes.** . Strings are used for the manipulation of words and sentences. Few examples of Python strings are **"A book", "xyz$56" 'This is demo'** etc. We see number of examples of strings in our daily life. Name of a person, subject name, course name, color of any object etc. It's not necessary that a string should only contain characters, it may contain any symbol but only within single or double quotes. Strings can also be enclosed within triple quotes. This is specially used for documentation string or multiline strings.

The encoding used for representation of string characters is Unicode as it can represent almost any character present in any language of the world.

6.2 Creating Strings

The strings can easily be created by enclosing any type of data within single, double or triple quotes. See this in shell:

```
>>> s="Hello";print(s)
Hello
>>> s='Hello';print(s)
Hello
```

Without print the strings are printed with quotes :

```
>>> s='Hello';s
'Hello'
>>> s="Hello";s
'Hello'
```

For creating multiline strings just continue the string with backslash character "\"

```
>>> s="This is an \
... example of \
... multiline \
... string"
>>> s
'This is an example of multiline string'
>>> print(s)
```

```
This is an example of multiline string
```

Another way to create multiline strings is using triple quotes.

```
>>> s="""This is an example
... of multiple lines
... using triple quotes
... """
>>> s
'This is an example\nof multiple lines\nusing triple quotes\n'
>>> print(s)
This is an example
of multiple lines
using triple quotes
```

6.3 Accessing String

As we have seen in the previous section just writing name of the string in shell and in **print** function displays the contents of the string. But for accessing individual elements subscript notation [index] can be used. The first element of the string is at index 0 and not 1. Even indexing can be done from the last element. The last element is at index -1, second last is at -2 and so on. The length of the string is the total number of characters present in it. Lets see some examples:

```
>>> s='Python'
>>> s[0]
'P'
>>> s[2]
't'
>>> s[-1]
'n'
>>> s[-2]
'o'
>>> type(s)
<class 'str'>
>>> len(s)
6
```

The index for a string can be an integer literal, variable or even any expression but all must yield an integer. See shell in action:

```
>>> s='Python';s
'Python'
>>> x=1
>>> s[x]
'y'
>>> s[x-2]
'n'
>>> x=2
>>> s[x*2+1]
'n'
```

```
>>> s[x*10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

The last expression gives error as s[20] is out of the range of 0 to 5.

6.4 String Slicing

Python also permits us to select a portion of a string using index. This can be done by specifying range of characters and their indexes. The syntax is

```
string[start:last:step]
```

Default value for start is 0, stop is len(string) and step is 1.

See shell in action with proper explanation. Assuming string **s='Strings in Python'**

Sr.No	Shell Code	Explanation
1.	>>> s[0:7] 'Strings'	Select string slice from index 0 to 6(7-1)
2.	>>> s[2:7] 'rings'	Select string slice from index 2 to 6(7-1)
3.	>>> s[8:] 'in Python'	Select string slice from index 8 to the end of the string s
4.	>>> s[11:-1] 'Pytho'	Select string slice from index 11 to the end without including last character at index -1.
5.	>>> s[-6:-1] 'Pytho'	Select string slice from index -6 to -1 non inclusive
6.	>>> s[0::2] 'Srnsi yhn' >>> s[0:len(s):2] 'Srnsi yhn' >>> s[::2] 'Srnsi yhn'	Select string slice from index 0 to end of the string with a step size of 2. First and last have default value as 0 and length of the string: len(s).
8.	>>> s[::-1] 'nohtyP ni sgnirtS'	Reverses the string. From last value to 0 index with a step size of -1. When negative index is used then default initial and last value reverses.
9.	>>> s[:-1] 'Strings in Pytho'	All but last character.
10.	>>> s[::-2] 'nhy isnrS'	String reversal with step size of 2 from last to beginning.
11.	>>> s[6:0:-1] 'sgnirt'	From index 6 to 0 non inclusive in reverse order.
12.	>>> s[:] 'Strings in	Takes default values of start, stop and step.

	Python'	

6.5 String operators

Strings in python supports three operators: + for concatenation, * for repetition and in, not in for membership. They are shown in table:

Table 6.1 : String Operators

Sr.No	Operator	Remarks
1.	+	String concatenation, “Hello”+”World” gives “HelloWorld”
2.	*	String repetition, “Hi”*3 gives “HiHiHi”.
3.	in, not in	Checks for membership of element to string, (a) ‘n’ in ‘python’ returns True (b) ‘i’ not in ‘python’ returns True

Let’s see some examples:

```
>>> s1="Hello";s2="Python"
>>> s3=s1+" "+s2
>>> print(s3)
Hello Python
>>> s4="An"+"Example in "+s2
>>> s4
'AnExample in Python'
>>> "Hi"*4
'HiHiHiHi'
>>> 'i' in s2
False
>>> 'i' not in s2
True
```

String concatenation can only be performed between two strings. You cannot mix any other type with the string. If you try to do so you will get error:

```
>>> s="Hello"+123
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

The solution is to simply type cast or convert non-string to string data type by str function.

```
>>> s="Hello"+str(123)
>>> s
```

```
'Hello123'
>>> x=10
>>> y=20
>>> s="Sum of "+str(x)+"and"+str(y)+"is"+str(x+y)
>>> print(s)
Sum of 10and20is30
```

6.6 String traversal

As string is nothing but a sequence of characters, it can be easily traversed using a loop. Using a loop each character can be analyzed and processed depending upon the problem at hand. For example, vowels can be counted, any specific character can be counted, character case can be changed, or any other required processing can be performed. Let's write some good scripts to illustrate string traversal in python.

```
# Script 6.1 to count number of vowels in a given string using while #loop
    s=input('Enter a string\n')
    i=0
    count=0
    while i<len(s):
        if s[i] in 'aeiouAEIOU':
            count+=1
        i=i+1
    print("Number of vowels=",count)
    OUTPUT:
    Enter a string
    THIS IS test
    Number of vowels= 3
```

Input string is taken from user and stored in s. The character of the string **s** are from **s[0]** to **s[len(s)-1]**. The **while** loops runs for all the characters in the string **s** starting from **0** and upto **len(s)-1**. This is done by loop control variable **i**. The variable **count** is incremented whenever a vowel is found in string **s**. This is checked using membership operator **in**. At the end of the loop number of vowels are displayed.

The same code without much effort can be written using for loop too. In fact string processing using for loop is favored by most python programmers simply because of its ease and simplicity.

```
# Script 6.2 to count number of vowels in a given string using for loop

    s=input('Enter a string\n')
    count=0
    for char in s:
        if char in 'aeiouAEIOU':
            count+=1
    print("Number of vowels=",count)
    OUTPUT:
    Enter a string
    THIS IS test
```

```
Number of vowels= 3
```

Here the **char** variable takes on values from string **s** one by one directly without using any index. That is first time char has value **'T'**, second iteration it has **'H'** and so on.

```
# Script 6.3 to find length of a string without built in function
s=input('Enter a string\n')
count=0
for char in s:
      count+=1
print("Length of string=",count)
OUTPUT:
Enter a string
python
Length of string= 6
```

The **for** loop increments **count** for every character in the string, thus when for loop ends, **count** stores length of the string. This can also be put inside a function as:

```
def mylen(s):
      count=0
      for char in s:
            count+=1
      return count
```

It can be used as

```
print("Length=",mylen("python"))
```

Let's see one more script which checks two strings and tells which string is greater in length. We make use of the function **mylen** created above.

```
# Script 6.4 String comparison without built in function

def mylen(s):
      count=0
      for char in s:
            count+=1
      return count

s1=input("Enter first string\n")
s2=input("Enter second string\n")
if mylen(s1)>mylen(s2):
      print(s1,"is greater in length than",s2)
elif mylen(s1)<mylen(s2):
      print(s2,"is greater in length than",s1)
else:
      print(s1,"is equal in length to",s2)
OUTPUT:
```

```
(First Run)
Enter first string
cat
Enter second string
cats
cats is greater in length than cat
(Second Run)
Enter first string
python
Enter second string
cooler
python is equal in length to cooler
```

We simply find the length of two strings and compare them using **else if** ladder. To conclude this section let's write a script which changes the case of letters from upper to lower and vice versa, leaving non-alphabets as it.

```
# Script 6.5 Function to convert case of alphabets in a given string

def togglecase(s):
      import string
      ns=''
      for char in s:
            if char in string.ascii_uppercase:
                  convert=chr(ord(char)+32)
            elif char in string.ascii_lowercase:
                  convert=chr(ord(char)-32)
            else:
                  convert=char
            ns=ns+convert

      return ns

print(togglecase('ThIs WoRkS'))
```

The important point here is the use of string module that defines number of built in constants. We will cover string module in detail later in this chapter but **string.ascii_uppercase** and **string.ascii_lowercase** are simply strings containing all uppercase and lowercase characters respectively.

```
>>> import string
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Every character from string is checked whether it belongs to any of the constant strings used from string module. ASCII values for uppercase alphabets is from 65-90 and for lowercase alphabets it is 97-122. The difference between ASCII values of upper case and lower case character is 32.

For conversion from lower to upper we subtract 32 and converting from upper case to lower case we add 32. But even a single character is a string in python and arithmetic of integer and string is not allowed in python:

```
>>> 'a'-32
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

For that character is converted to its ASCII value, arithmetic operation is performed and again converted back to character. For example say **char='t', ord('t')** gives **116**, subtracting **32** from it gives **84** and **chr(84)** gives **'T'**.

6.7 String is immutable

The string object does not support item assignment. That is once a string is initialized you cannot change its content. It is possible to extract some portion of a string and create new string but in now way original string can be mutated. See shell in action:

```
>>> s="Attitude"
>>> s[0]='L'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Here we try to change the first character of the string **s,** which the python does not allow. It throws a **TypeError**. But what if instead of changing just one character we try to assign a new string literal to variable s. Let see what happens:

```
>>> s="Positive"
```

This happened without any error as now a new string literal **"Positive"** is created in memory and **s** points to that. The old variable s is now lost. Another concept which is worth mentioning here is through the following example:

```
>>> s="Positive"
>>> id(s)
2093790809072
>>> s=s+" Attitude"
>>> s
'Positive Attitude'
>>> id(s)
2093790894456
```

Here you may think that writing : **`s=s+" Attitude"`** is modifying the original string **s** but the contents of **s** and **" Attitude"** are concatenated together and assigned to new variable **s**. The old variable **s** no longer exists containing value "Positive".

Following shell code let you create a new string by using some slice from existing string:

```
>>> s="Positive"
>>> ns="Nega"+s[4:]
>>> ns
'Negative'
```

Immutability also applies on deletion of one or more characters from an existing string but again whole string can be deleted easily:

```
>>> del ns[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion
>>> del ns
>>> ns
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ns' is not defined
```

6.8 String Comparison

In section 6.6 we saw one python script for comparing two strings based on their length. But actual comparison is based on ASCII values of the individual characters present in the string. Let's see some example and then we'll write one script for the same.

```
>>> s1='Apple'
>>> s2='Banana'
>>> s1==s2
False
>>> s1>s2
False
>>> s2>s1
True
>>> s1=='Apple'
True
>>> s1=='apple'
False
>>> s1>'Applet'
False
```

Python compares two strings and returns a **+ve** value if first string is greater than second string, **-ve** value if first string is smaller than second string and a zero value if both strings are equal . What is the **+ve** or **-ve** value? Actually **it** returns the ASCII difference of the first unmatched character.

The ascii value of 'A' is 65 and of 'B' is 66 so string **s2>s1** returns True. If string **s1='Cot'** and **s2='Cop'** are compared than first mismatched character is **'t'** in **s1** and **'p'** in **s2**. The difference in their ASCII values are **ord('t')-ord('p')= 4**. This means string **s1** is greater than **s2**.

```
# Script 6.6 to compare two string by contents
s1=input('Enter first string\n')
s2=input('Enter second string\n')
if s1>s2:
    print(s1,'is greater than',s2)
elif s1<s2:
    print(s2,'is greater than',s1)
else:
    print(s1,'and',s2,'is equal')
OUTPUT:
Enter first string
apple
Enter second string
banana
banana is greater than apple
```

The slight problem with the above script is that it is case sensitive. For example, **'Apple'** is smaller than **'apple'** because of ASCII value difference. To make our code case insensitive either we can make string to be compared all in upper case or lower case. We here use lower method of string class (not module string) prior to checking string for comparison.

```
s1=input('Enter first string\n')
s2=input('Enter second string\n')
s1=s1.lower()
s2=s2.lower()
if s1>s2:
    print(s1,'is greater than',s2)
elif s1<s2:
    print(s2,'is greater than',s1)
else:
    print(s1,'and',s2,'is equal')
OUTPUT:
Enter first string
Apple
Enter second string
apple
apple and apple is equal
```

6.9 Methods of string class

Every string belongs to class 'str' and every string is an object.

```
>>> type('xyz')
<class 'str'>
>>> type('xyz') is str
True
```

The various methods of 'str' class can easily be used by any string. One such example we have just seen in the script above. To find out all the methods of 'str' class just type **dir(str)** in python shell:

```
>>> dir(str)
['__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']
```

Help for any method can easily be seen in the shell by writing : **help(str.methodname)**. For example:

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

replace(...)
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

Some of the commonly used methods are discussed here.

1.find: the find method finds index of the substring in a string. Default index are from start(0) to end(length of string non inclusive), it returns -1 when substring is not found.

```
>>> s='positive'
>>> s.find('i')
# finds first occurrence of  'i' in s (from index 0 to 7)
3
>>> s.find('i',4)
# finds first occurrence of  'i' in s (from index 4 to 7)
5
>>> s.find('i',2,4)
# finds first occurrence of  'i' in s (from index 2 to 3)
3
>>> s.find('i',4,5)
# finds first occurrence of  'i' in s (from index 4 to 4)
-1
>>> s.find('it')
3
```

2. capitalize: Capitalize the first character of string

```
>>> s='positive'
>>> s.capitalize()
'Positive'
```

3. center: The method center a string. The arguments are width and fill character. Default fill character is space.

```
>>> s.center(40,'*')
'****************positive****************'
```

4. count: The method counts number of occurrences of a given substring.

```
>>> s='aabrakadabra'
>>> s.count('a')       # ‘a’ appears 6 times in string s
6
>>> s.count('g')       # ‘g’ does not appear in string
0
>>> s.count('ab')    # ‘ab’ appear 2 times in string s
2
```

5. endswith: The method checks if a given string ends with a specified suffix.

```
>>> 'position'.endswith('tion')
# string ‘position’ has ‘tion’ as suffix
True
>>> 'position'.endswith('tive')
# string ‘position’ does not have ‘tive’ as suffix
False
```

6. startswith: The method checks if a given string begins with a specified prefix.

```
>>> 'preempt'.startswith('pre')
True
>>> 'dislike'.startswith('dis')
True
>>> 'sosweet'.startswith('soo')
False
```

7. index: Similar to **find** method but raises error when string not found.

```
>>> 'positive'.index('g')
# find method returns -1 when substring not found
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

8. isalnum: The methods checks whether string contains only alphabets and digits, returns True if so else return False.

```
>>> 'abc123'.isalnum()
True
>>> 'a+b'.isalnum()
False
>>> 'ab'.isalnum()
True
```

Similar to this **isalpha** checks for all characters to be alphabets only, **isdigit**, **isdecimal**, **isnumeric** checks for all digits to be only between 0-9, there are minor differences among these functions but its not necessary to discuss here. Inquisitive user is encouraged to google the differences.

9. istitle: The method checks string for title case. Every word's first character must be capital and all other characters of a word be in lowercase.

```
>>> 'Fun'.istitle()
# 'F' in 'Fun' is capital
True
>>> 'FunF'.istitle()
# 'F' in 'FunF' is capital but 'F' after 'n' is also capital
False
>>> 'funF'.istitle()
# 'f' in 'funF' is small
False
>>> 'Demo Of Title'.istitle()
# Every word is capitalized
True
```

10. isupper: checks the string is in upper case or not

```
>>> 'HELLO'.isupper()
True
>>> 'Hi'.isupper()
False
```

Similar to this we have **islower** to check string for all lowercase characters. Two more methods are their for conversion from lower to upper and vice versa. These are **lower** and **upper**.

11.join: The **join** method takes an iterable (list, tuple etc) and joins all elements of the iterable using a string which can be space or any other special character.

```
>>> ' '.join(('An','example','of','join'))
'An example of join'
>>> ' '.join(['An','example','of','join'])
'An example of join'
>>> '$'.join(['An','example','of','join'])
'An$example$of$join'
```

In the first example we have a tuple of 4 strings which are joined by space character. Second example replaces tuple by list and third example uses '$' as joining character.

12.strip: The strip function removes extra whitespaces(space and tabs) from beginning and end of the string. For removing either from left or right you can use **lstrip** and **rstrip** functions.

```
>>> s='     Strip demo     '
>>> s.strip()
'Strip demo'
>>> s.rstrip()
'     Strip demo'
>>> s.lstrip()
'Strip demo     '
```

13. partition: The method separates the string based on some separator, supplied as argument to partition function. *The function returns three strings: string before the separator, separator itself, string after the separator.* When separator is not found than the whole string is returned as first string and remaining strings are treated as empty. See shell in action:

```
>>> s='It is an example'
>>> s.partition('an')
# breaks s on 'an' and returns 3 strings as tuple
('It is ', 'an', ' example')
>>> s.partition('just')
# 'just' is not in s so second and third strings are empty
('It is an example', '', '')
>>> s='I just love python' # breaks on 'love'
>>> s.partition('love')
('I just ', 'love', ' python')
>>> s='key:value'  # breaks on ':'
>>> s.partition(':')
('key', ':', 'value')
```

One big issue with partition method is that the supplied separator works with embedded string also. For example:

```
>>> s='This is another example'
>>> s.partition('an')
('This is ', 'an', 'other example')
```

This is not desirable in most of the programming situations.

Another issue is that the partition method breaks the string only at the first occurrence of the separator. If you have more than one separator appearing in the string than they are not considered and become the part of third string post separator.

```
>>> s='Keep the spirit on and on and on'
>>> s.partition('on')
```

```
('Keep the spirit ', 'on', ' and on and on')
>>> s.partition('and on')
('Keep the spirit on ', 'and on', ' and on')
```

In first example the separator is ‘on’ and in second example the separator is ‘and on’.

The advantage of **partition** returning three strings in a single tuple is that string can be easily recreated back using **join** method.

```
>>> s='Keep the spirit on and on and on'
>>> part=s.partition('on')
>>> part
('Keep the spirit ', 'on', ' and on and on')
>>> s1=''.join(part)
>>> s1
'Keep the spirit on and on and on'
```

14. split: The method splits the strings on the given delimiter supplied as first argument. The default is space. The method returns a list of separated words based on delimiter. The second argument is **maxsplit** which specifies maximum number of splits. An important point to note that during splitting the delimiter is removed from the string and does not become part of the returned list. Let’s understand it using some examples:

```
>>> s='This is an example'
>>> s.split() # splits on space, the default delimiter
['This', 'is', 'an', 'example']
>>> s='This$is$an$example'
>>> s.split('$') # splits on '$'
['This', 'is', 'an', 'example']
>>> s='This is an example of       split      method'
>>> s.split()  # splits on space and remove extra whitespace
['This', 'is', 'an', 'example', 'of', 'split', 'method']
```

Let’s some more example where delimiter is some word in the string:

```
>>> s="keep your spirit on and on and on"
>>> s.split('on')
['keep your spirit ', ' and ', ' and ', '']
>>> s='an another an example'
>>> s.split('an')
['', ' ', 'other ', ' example']
```

In the first example the split is done on string “on” and list of 3 strings is returned. Second example is interesting where “an” is also in “another”. In the returned list first string is empty because of first “an” in string **s**, second is space ‘ ‘ because of “an” in “another” and third one is because of “an” before “example”.

Now let's see how the second parameter **maxsplit** of **split** function affects the output. The default value of this parameter is -1. See some examples:

```
>>> s='An example of split function'
>>> s.split(maxsplit=1)
['An', 'example of split function']
>>> s.split(maxsplit=2)
['An', 'example', 'of split function']
>>> s.split(maxsplit=3)
['An', 'example', 'of', 'split function']
>>> s.split(maxsplit=5)
['An', 'example', 'of', 'split', 'function']
>>> s.split()
['An', 'example', 'of', 'split', 'function']
>>> s.split(maxsplit=15)
['An', 'example', 'of', 'split', 'function']
```

Setting maxsplit=1 gives two strings in the returned list. For maxsplit=2 gives 3 strings in the returned list. It depends upon how many occurrences of delimiter are present in the string. For a small example string say 'split function' maxsplit=1 will work and any value of maxsplit more than 1 will have no effect if length of string is less than maxsplit. Try with a long string of 10 to 15 words.

15. maketrans and translate: These two methods are very useful in programming situations where in a given string some characters are to be replaced by some other characters and some unwanted characters to be removed. Let's understand both the functions.

Function **maketrans** takes three string as parameters: first parameter consists of string of characters which are to replaced, second string parameter consist of those characters from which replacement is to be done and third string parameter consist of characters in original string which are to be removed. The function returns a table in the form of dictionary where keys are the characters to be replaced and values are characters from which replacement is to be done. For characters that are to be deleted value is None.

The function **translate** take this returned table as argument and translates the string onto which it is called.

Let's understand using an example:

```
>>> xstr='abc'
>>> ystr='xyz'
>>> zstr='d'
>>> s='aabbcda'
>>> tab=s.maketrans(xstr,ystr,zstr)
>>> tab
{97: 120, 98: 121, 99: 122, 100: None}
>>> s.translate(tab)
'xxyyzx'
```

Here **xstr** is the string containing characters to be replaced; **ystr** contains characters which are to be substituted for characters in string **xstr** and **zstr** contains characters which are to be deleted. The string we work on in this example is **s='aabbcda'**. Call to function **maketrans** using string **s** passes parameters **xstr,ystr** and **zstr**. From these passed parameters **maketrans** creates a dictionary table and returns. The same is stored in variable **tab**.

As can be seen from the output of running tab in shell it returns a dictionary. All entries except None are ASCII values of the characters. This can be interpreted as: character 'a'(97) is to be replaced by 'x'(120), character 'b'(98) is to be replaced by 'y'(121), character 'c'(99) is to be replaced by 'z'(122), and character 'd' is to be removed because of None.

This dictionary table is then passed to **translate** function which performs translation as par table tab and returns a new string. Original string remains unchanged.

It is also possible to do the translation without using **maketrans** function. We have seen that **maketrans** gives us a translation table; what if we create our own translation table for replacement and deletion of characters from original string. Let's see how this can be done:

```
    s="peel on rovings"
    # want to replace 'p' by 'k','l' by 'p','r' by 'm' and remove 's'
    >>>tab={ord('p'):ord('k'),ord('l'):ord('p'),ord('r'):ord('m'),ord('s'):N
one}
    >>>tab
    {112: 107, 108: 112, 114: 109, 115: None}
    >>> s2=s1.translate(tab)
    >>> s2
    'keep on moving'
```

You can see it is possible to use translate without **maketrans** but its takes time to manually create the mapping dictionary table.

The main use of **translate** and **maketrans** is in removing punctuations from a string which is required in most programming situations. Let's understand using an example:

```
    >>> s='this;+$is%:an#example'
    >>> tab=s.maketrans(';+$%:#',' '*6,'')
    >>> tab
    {59: 32, 43: 32, 36: 32, 37: 32, 58: 32, 35: 32}
    >>> s1=s.translate(tab)
    >>> s1
    'this   is  an example'
    >>> import re
    >>> s1=re.sub(' +',' ',s1)
    >>> s1
    'this is an example'
```

In the string **s** we have 6 different punctuation symbols which we want to remove but removing may remove any spacing between the words of string s. See this:

```
>>> tab=s.maketrans('','',';+$%:#')
>>> s.translate(tab)
'thisisanexample'
```

The other way to replace the punctuation symbols which we want to remove by spaces. This approach we have followed above. But the problem with this approach is that it leaves extra spaces in between words. To remove this we have taken help of module **re** (stands for regular expression). The **sub** function of this module replaces one or more spaces by a single space.

16. replace: The replace method replaces one string by another string. The method takes three arguments: old string, new string and an optional count argument. The count argument specifies how many occurrences of old string are to be replaced by new string. By default all occurrences are replaced.

```
>>> s="It is fun"
>>> s.replace("is","was")
'It was fun'
>>> s="off and off"
>>> s.replace("off","on")
'on and on'
>>> s.replace("off","on",1)# replace only first occurrence of "off"
'on and off'
>>> s="off and offer"
>>> s.replace("off","on") # embedded words are also matched
'on and oner'
```

The matching done by replace method is case sensitive. See an example:

```
>>> s.replace("OFF","on")
'off and offer'
```

As "OFF" is not present in string **s**, the original string is returned.

17. swapcase: The function as name suggests swap the cases of alphabets present in the string. See an example:

```
>>> "This is An Example".swapcase()
'tHIS IS aN eXAMPLE'
>>> s="AABBccdd"
>>> s1=s.swapcase()
>>> s1
'aabbCCDD'
```

18 zfill: The function zfill is used for padding zeros to the left of the string. Useful in programming situations where a binary number is to be padded or represented in a fixed length. For example, if you number 10 and requirement is to represent this number into binary of length 16 digits. This can easily be done using **zfill**. See below:

```
 >>> x=str(bin(10))
>>> x
'0b1010'
>>> y=x[2:].zfill(16) # x[2:] is to remove '0b'
>>> y
'0000000000001010'
```

6.10 The format method

The format method of **str** class has so much to offer that's why we have devoted an entire section for this. As the name suggest the **format** method does the task of formatting the output. There are number of formatting options which you can use for various data types. The function takes variable number of arguments and keyword arguments that act as substitution for the placeholders represented using { }. Let's take a simple example first:

```
name='chinmay'
print("Hello {}".format(name))
s="Hello {}"
print(s.format(name))
OUTPUT:
Hello chinmay
Hello chinmay
```

Here {} is the placeholder where the value of the variable name will be substituted. In the example two different ways have been shown to use **format** method. On with string literal and another with string variable. You can use any method which you prefer.

Lets see next example with multiple variables/literals.

```
print("Hello {} , you are {} years old".format("aksh",10))
print("{} plus {} is {} ".format(10,10,10+10))
a=2.3;b=3.45
print("{} x {} ={}".format(a,b,a*b))
OUTPUT:
Hello aksh , you are 10 years old
10 plus 10 is 20
2.3 x 3.45 =7.935
```

The two examples seen above used the concept of automatic numbering where the placeholders were simply {}. The number of placeholders must match with the number of variables/literals in the **format** function. Runtime error may ensue if the preceding line is not true.

It is also possible and useful way to provides numbers or keywords inside braces to bind them with the positional or keyword arguments in format function. Let's start with this in next sub section.

6.10.1 Positional Arguments in format function

The variables/literals in the **format** function are numbered from left to right starting from 0,1,2….The position numbers can be placed inside the {} to clearly specify the binding of variables/literals with the placeholders.

```
name='chinmay'
age=21
salary=75565.56
print("Hello{0}your age is {1} and salary is {2}".format(name,age,salary))
```

Here {0} is the placeholder for name, {1} for age and {2} for salary. You cannot change this positional numbering i.e. instead of 0,1,2 inside {} you would like to use {1},{2} and {3} then python will generate error.

builtins.IndexError: tuple index out of range

Instead of using variables in format function you can also write literals directly:

print("Hello {0} your age is {1} and salary is {2}".format("chinmay",21,75565.56))

Further if you change the positions, nothing harms but the meaning of string changes like:

print("Hello {1} your age is {2} and salary is {0}".format("chinmay",21,75565.56))

6.10.2 Number Formatting

For formatting numbers, the format command provides number of options. All the options are listed in the table x.1:

Table 6.2: Number formatting options in format function

Sr.No.	Format symbol	Meaning
1.	d	Decimal numbers
2.	b	Binary numbers
3.	o	Octal numbers
4.	x/6	Hexadecimal numbers (lower/uppercase)
5.	e/E	Exponent (lower/uppercase)
6.	%	Multiplies the argument by 100 and put % as suffix.
7.	c	Converts the integer to Unicode character
8.	f/F	Floating point numbers (default precision is 6 digits)

Let's understand the above format specifiers using an example:

```
print("decimal number {:d}".format(275))
print("float number {:f}".format(275.345))
print("Scientific notation of {0:f} is {0:E}".format(.0023))
```

```
print("octal of {0:d} is {0:o}".format(123))
print("hex  of {0:d} is {0:x}".format(324))
print("Ascii of {0:c} is {0:d}".format(65))
print("My percentage={:.2f}%".format(97.6789))
```

OUTPUT:

```
decimal number 275
float number 275.345000
Scientific notation of 0.002300 is 2.300000E-03
octal of 123 is 173
hex  of 324 is 144
Ascii of A is 65
My percentage=97.68%
```

The code is easy to interpret by seeing the output. The thing which require detail explanation is floating point formatting. The placeholder **{:0.2f}%** means 2 digits after decimal point are to be displayed, % means multiply by 100 and put a % sign after it. While displaying octal, hex or binary numbers with prefix 0o,0x and 0b respectively, '#' can be used. See an example:

```
print("int {0:d} is binary {0:#b}".format(45))
print("int {0:d} is octal {0:#o}".format(45))
print("int {0:d} is hex {0:#x}".format(45))
```

OUTPUT:

```
int 45 is binary 0b101101
int 45 is octal 0o55
int 45 is hex 0x2d
```

6.10.3 Number Padding

The **format** command lets you format the numbers by specifying the display width for each number. If the display width is smaller than number of digits, the whole number is displayed (you cannot shrink the number!). If display width is more than number of digits than extra padding is displayed in output. This is useful for aligning the numbers of different lengths.

For integers display width can be specified by {p:wd}, where p is the position of the variable to be formatted , w is width in decimal and d is the format specifier for integers. For example:

```
>>>print("{0:6d}".format(275))
   275
print("{0:06d}".format(275))
000275
```

The placeholder **{0:6d}** for number **275** assigns a width of 6 and displays number 275 from right which takes just 3 places and remaining 3 places are left vacant. But as such these are not visible over here. The next placeholder **{0:06d}** puts 0's at the empty places from left.

Let's see one more example in a script:

```
print("{0:2d}".format(12345))
print("{0:02d}".format(12345))
```

```
print("{0:012d}".format(12345))
OUTPUT:
12345
12345
000000012345
```

As stated earlier when display width is smaller than number of digits in number, the whole number is displayed as it is.

We have discussed display width for integer numbers. Let's discuss now how display width affect formatting of floating-point numbers. The placeholder **{:w.pf}** reserves display width of **w** spaces including one space for decimal point, **p** spaces for digits after floating point and remaining for digits before floating point. For example **{:010.3f}** means a display width of 10 spaces, out of which 1 for decimal point and 3 for digits after decimal point will be allocated, rest (10-(3+1))=6 will be allocated for digits before decimal point with zero padding if require. See tiny code snippet below:

```
print("{:010.3f}".format(4567.89767))
print("{:010.3f}".format(234567.89767))
OUTPUT:
004567.898
234567.898
```

The code is easy to understand. let's have one more example:

```
x=1234.567896
print("{:06.2f}".format(x))
print("{:07.2f}".format(x))
print("{:08.2f}".format(x))
print("{:09.2f}".format(x))
print("{:010.2f}".format(x))
OUTPUT:
1234.57
1234.57
01234.57
001234.57
0001234.57
```

The reader is encouraged to figure out the output. Zero padding has been used to clearly see the padding done.

When dealing with signed numbers, sign can also be shown along with the numbers. Just add the + symbol as prefix after : as shown in the example below. The + symbol tells format function to show the sign before the number , be it +ve or -ve. See example:

```
print("{0:+d},{1:+d}".format(10,-12))
print("{0:+4.2f},{1:+4.2f}".format(10.34,-12.12))
OUTPUT:
+10,-12
+10.34,-12.12
```

We have seen the concept of padding above and its by default onto the left side of the number to be displayed. Format function allows us to control the direction of padding too. The default is right alignment with left padding. For that it provides some symbols that can be used for controlling alignment. These are given in the table below:

Table 6.3: Symbols for controlling alignment

Symbol	Purpose
<	Left aligned/right padding of the remaining width
^	Center aligned of the remaining width
>	Right aligned/left padding of the remaining width
=	Sign appear left side

Let's see an example that make use of all of the above symbols :

```
print("Default padding |{:6d}|".format(123))
print("Default padding with zero |{:06d}|".format(123))
print("Right padding |{:<6d}|".format(123))
print("Right padding with zero |{:<06d}|".format(123))
print("Center padding  |{:^7d}|".format(123))
print("Center padding with zero |{:^07d}|".format(123))
print("With = for sign |{:=5d}|".format(-123))
print("With + for sign |{:+5d}|".format(-123))
```

OUTPUT:

```
Default padding |   123|
Default padding with zero |000123|
Right padding |123   |
Right padding with zero |123000|
Center padding  |  123  |
Center padding with zero |0012300|
With = for sign |- 123|
With + for sign | -123|
```

Few points to observe from the above output:

(i) Right padding changes the meaning of the number.
(ii) Center padding puts number in center and fills either side with zero.
(iii) Try with even number width and odd length number.
(iii)= symbol puts sign towards left and + just before the number.

We have shown the example with integers numbers. The same can be tried with floating point numbers.

6.10.4 String Formatting

Like numbers, string can also be formatted using format function. Padding and alignment symbols remain same for strings too. The difference is that default padding for numbers is left and right alignment but for strings default padding is right and left alignment. Here no special character is used for symbol. Just width is written in integer. See one example:

```
print("|{0:7}|".format("money"))
print("|{0:<7}|".format("money"))
print("|{0:>7}|".format("money"))
print("|{0:^7}|".format("money"))
print("|{0:$^7}|".format("money"))
OUTPUT:
|money  |
|money  |
|  money|
| money |
|$money$|
```

In all print statements width is chosen as 7. First two print statement prints string as left align and right padding. The third print statement prints string as right align with left padding. Fourth prints center align and last one is center align with character '$' filling the empty spaces.

The format function can be used to shrink the string or display only few starting characters from string. The syntax for doing it is: {:0.n} where n is the number of characters to be extracted. The number n must be an integer literal. See an example:

```
x="chinmay"
print("{:.1}".format(x))
print("{:.2}".format(x))
print("{:.3}".format(x))
print("{:.4}".format(x))
print("{:.5}".format(x))
print("{:.6}".format(x))
print("{:.7}".format(x))
OUTPUT:
c
ch
chi
chin
chinm
chinma
chinmay
```

Note that the above output can easily be achieved with string indexing and a for loop but here we have used multiple lines. As stated above n cannot be a variable. Even alignment symbol can be used while extracting starting few characters:

```
x="chinmay"
print("|{:>8.4}|".format(x))
```

```
print("|{:$^8.4}|".format(x))
OUTPUT:
|    chin|
|$$chin$$|
```

6.10.5 Keyword arguments

Like positional arguments, keyword arguments can also be used with format function. This becomes quite handy as you do not need to remember the positions. The name itself denotes the position. See a small example:

```
s="Hello {name} your salary is {salary}"
print(s.format(name="chinmay",salary=76897))
s="Hello {name:.7} your salary is {salary:0.2f}"
print(s.format(name="chinmay jain",salary=76897.8978))
OUTPUT:
Hello chinmay your salary is 76897
Hello chinmay your salary is 76897.90
```

As you can see keyword arguments are nothing special than positional arguments. The numbers have been replaced by names rest everything remains same.

Positional and keyword arguments can be combined too. See an example:

```
s="My name is {0},I live in {city}"
print(s.format("Purvi",city="Gurgaon"))
OUTPUT:
My name is Purvi,I live in Gurgaon
```

The keyword argument always appears later than positional arguments. If you don't follow the above error will be flashed by python. That is the following will result into error:

```
s="My name is {1},I live in {city}"
print(s.format(city="Gurgaon","Purvi"))
OUTPUT:
Syntax Error: positional argument follows keyword argument
```

6.11 Ponderable Points

1. Strings in Python are sequence **of characters, digits, symbols enclosed within single or double quotes.**
2. For creating multiline strings just continue the string with backslash character "\"
3. Python also permits us to select a portion of a string using index as: string[start:last:step]
4. The + operator is used for string concatenation and * for string repetition.
5. String can easily be traversed character by character using loop.
6. The string object does not support item assignment as string is immutable in python.
7. String can be compared using ASCII characters and relational operators.

8. The format method is most useful method of string class.
9. All properties and methods of class string can be easily seen using dir(str)

7. LIST

7.1 Introduction

A list is a data type in python. Like string it is also a sequence of values of varied types. Elements of a list can be of any type even another list, tuples, dictionary, or any other type of sequence. List is a heterogeneous data structure where all the elements can be of any data type. We have seen number of examples involving list but that was a cursory treatment. In this chapter we are going to have a deep dive into this most powerful and useful data structure. We'll learn how to create list, perform various operations on list, work with various methods of list and many other stuff that can be done with list.

7.2 Creation of List

A list is created with elements placed within square brackets [ ]. When no element is placed then list is an empty list. A list is an ordered collection and elements can be added, removed, and modified. Thus list is a mutable data structure/collection/type.

To create a list just put elements within square brackets.

```
>>> [1,2,3,12,0]
[1, 2, 3, 12, 0]
>>> ['python','java','c++']
['python', 'java', 'c++']
>>> ['python','java',12,True]
['python', 'java', 12, True]
>>> []
[]
>>> [12,34.56,'Hello',['a','b'],False]
[12, 34.56, 'Hello', ['a', 'b'], False]
```

First list consists of all integer, second all strings, third is a mix of data types. The fourth list is an empty list. The final example is a list where other list is an element of the main list.

7.3 Accessing elements from list

The examples seen above in the previous section were all literals. The list can simply be assigned to variables and manipulated. Once we have a list variable indexing can be done onto list. The first element is at index 0, second on 1 and so on. Like string , list also supports negative indexing: last element at index -1, second last at -2 and so on. Let's see some examples:

```
>>> L=[2,12,45,12]
>>> L[0]
2
>>> L[-1]
12
>>> L=['python','java','c++']
>>> L[1]
```

```
'java'
>>> L=['python',[12,34],True]
>>> L[1]
[12, 34]
>>> L[1][0]
12
>>> S=L[1]
>>> S[0]
12
```

All examples are easy to understand. In the last example **L[1]** is a list and to access its elements we need to perform double indexing . Even **L[1]** can be stored in some other variable and we have done the same, storing it in **S**. Then S can be treated as another list.

An index which is not in the range of list generates error:

```
>>> L
['a', 'b', 'c', 'd', 'e']
>>> L[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> L[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

7.4 List slicing

We have seen slicing of strings in the previous chapter. Slicing can also be done with the list. Just to remind you slicing is done as: **list[start:stop:step]**. Default values for start=0, stop=len(list), step=1. See some examples:

`>>> L=['a','b','c','d','e']` `>>> L[1:3]` `['b', 'c']`	Select elements from 1 to 2 (inclusive both)
`>>> L[-1:-3]` `[]`	Empty list because default step size is 1
`>>> L[-1:-3:-1]` `['e', 'd']`	This works as step size -1 is given
`>>> L[:]` `['a', 'b', 'c', 'd', 'e']`	Default values of start,stop and step are taken
`>>> L[::]` `['a', 'b', 'c', 'd', 'e']`	Default values of start,stop and step are taken
`>>> L[:-1]` `['a', 'b', 'c', 'd']`	From index 0 to last element (non inclusive)

>>> L[::-1] ['e', 'd', 'c', 'b', 'a']	Reverse the list.

7.5 Modifying List

When we say modifying list we are concerned with addition , deletion and medication of existing elements within the list. This is possible because list is a mutable sequence. It means that elements within the list can be modified, new elements can be added or removed.

7.5.1 Updating List

To modify the contents without any addition or removal of new elements, just assign new item to individual index or assign multiple items using slicing operation.

```
>>> L
['a', 'b', 'c', 'd', 'e']
>>> L[0]=1
>>> L[-1]=4
>>> L
[1, 'b', 'c', 'd', 4]
>>> L[2:4]=[2,3]
>>> L
[1, 'b', 2, 3, 4]
```

7.5.2 Adding Elements to List

To add elements to list either **append** or **extend** method can be used. The difference is that **append** method can be used for adding a single element or a list but **extend** can be used for adding multiple items. Let's see example of **append** first.

```
>>> L=['a','b','c']
>>> L.append('d')
>>> L
['a', 'b', 'c', 'd']
>>> L.append('e','f')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
>>> L.append(['e','f'])
>>> L
['a', 'b', 'c', 'd', ['e', 'f']]
```

As can be seen from above you cannot add multiple elements to list using **append** method. When you try to add a whole list to an existing list the list is added as an element and not the elements from the list are added. The last example exemplifies this.

Let's see examples of extend method now:

```
>>> L=['a','b','c']
>>> L.extend('d')
```

```
>>> L
['a', 'b', 'c', 'd']
>>> L.extend(['e','f'])
>>> L
['a', 'b', 'c', 'd', 'e', 'f']
>>> L1=[1,2,3]
>>> L.extend(L1)
>>> L
['a', 'b', 'c', 'd', 'e', 'f', 1, 2, 3]
>>> L.extend([L1])
>>> L
['a', 'b', 'c', 'd', 'e', 'f', 1, 2, 3, [1, 2, 3]]
```

As can be seen from above examples that using **extend** method single or multiple elements can be added to an existing list. For adding a new list (as list and not element of list) to an existing list just put the list itself in square bracket.

Another way of adding elements to list is use of + operator. The operator can be used for extending an existing list or concatenating two lists. See some examples:

```
>>> L=['a','b','c']
>>> L1=['d','e']
>>> L=L+L1
>>> L
['a', 'b', 'c', 'd', 'e']
>>> L=L+['f']
>>> L
['a', 'b', 'c', 'd', 'e', 'f']
>>> L=L+['g','h']
>>> L
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Note even for adding single element to list you must put square brackets around it as + operator is only for concatenation of two lists only. You cannot combine list with string, integer or any other data type.

Elements can also be added to the list using the **insert** method. The signature of the method is:

```
L.insert(index, object) -- insert object before index
```

The method inserts object at given index. See some examples:

```
>>> L=[1,4,5,7,8]
>>> L.insert(1,2) # insert element 2 at index 1
>>> L
[1, 2, 4, 5, 7, 8]
>>> L.insert(2,3) # insert element 3 at index 2
>>> L
[1, 2, 3, 4, 5, 7, 8]
>>> L.insert(len(L),[9,10]) # insert at the end
>>> L
```

```
[1, 2, 3, 4, 5, 7, 8, [9, 10]]
>>> L[-1]=9
>>> L
[1, 2, 3, 4, 5, 7, 8, 9]
>>> L[4:len(L)]=[5,6,7,8]
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

The new thing you must have observed apart from **insert** method is replacing of elements using index and slice notation. Single or multiple elements of list can be replaced using this notation.

7.5.3 Removing elements from list

For removing elements from list you can make use of **del, remove** and **pop** functions. Let's understand how to use them and how they differ ?

```
>>> L=[2,6,8,10,0]
>>> L.pop(2)
8
>>> L
[2, 6, 10, 0]
>>> x=L.pop(1)
>>> x
6
>>> L
[2, 10, 0]
>>> L.pop()
0
>>> L
[2, 10]
>>> L.pop(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

The **pop** method of list class removes the element from the list at supplied index. The signature is :

```
L.pop([index]) -> item -- remove and return item at index (default last).
```

If no index is supplied than by default **pop** returns last element. If index is not between **range(0,len)**than it raises error.

The other method for deleting elements from list is **remove**. The signature is:

```
L.remove(value) -> None -- remove first occurrence of value.
```

As can be seen from signature the method does not take index as argument instead it takes value (an element in list) as argument and removes it from list. Further note that return type is None, it means that removed element is not returned as it with the **pop** method. See some examples:

```
>>> L=['a','b','c','d','e']
>>> L.remove('d')
>>> L
['a', 'b', 'c', 'e']
>>> x=L.remove('b')
>>> x
>>> print(x)
None
>>> L.remove('f')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

The last method for removing elements from list is **del**. The method can be used to delete any variable in the shell or python script. We take advantage of this to remove list elements. See some examples

```
>>> L=['a','b','c','d','e']
>>> del L[0]   # remove first element ‘a’
>>> L
['b', 'c', 'd', 'e']
>>> del L[1:3]  # remove elements ‘c’ ,’d’
>>> L
['b', 'e']
>>> del L[2]   # index out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> x=10
>>> x
10
>>> del x  # example of deleting variable x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

7.6 Operations on List

Number of operators can be applied onto list like **+, *, in, not in** etc. Lets understand them using some examples:

7.6.1 Operator + with list

The + operator can be used for string concatenation as well as for concatenation of two lists . See some examples:

```
>>> L=[1,2,3]+[4,5]
>>> L
[1, 2, 3, 4, 5]
>>> L=L+list(range(6,11))
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

7.6.2 Operator * with list

The * operator works same as we have seen with string. If used as **L*3** (L is a list) tt simply creates 3 copies of the list **L**. See some examples:

```
>>> L=['a','b']
>>> L*3
['a', 'b', 'a', 'b', 'a', 'b']
>>> list('aba')*2   # creates list from string and create 2 copies
['a', 'b', 'a', 'a', 'b', 'a']
>>> []*3   # no copies of empty list.
[]
>>> L[0]*3 # L[0] is string so 3 copies are created
'aaa'
>>> L[1]=10
>>> L
['a', 10]
>>> L[1]*3 # normal arithmetic multiplication
30
```

7.6.3 The membership operator on list

The membership operators **in** and **not in** we have discussed in chapter 2. The same works with list and can be used to check for membership testing on list. See some examples.

```
>>> L=['a',2,3,'amit']
>>> 'amit' in L
True
>>> 'c' in L
False
>>> L.insert(1,['b','c'])
>>> L
['a', ['b', 'c'], 2, 3, 'amit']
>>> ['b','c'] in L
True
```

7.7 List traversal

List traversal means accessing each element of list exactly once and performing some operation on them that may be display, checking for some conditions or any other which the programming problem demands. List can be traversed easily using any loop either for or while loop . But the easiest one is using for loop. We will show one example of list traversal using while loop and rest all examples we will do using for loop.

7.7.1 List traversal using while loop

For list traversal using **while** loop we need to have the length of the list that can be obtained using **len** function. Then each element of the list can be accessed using the index notation. See some examples:

```
# Script 7.1 displaying all elements of list one by one using while
  # loop
  L=[1,4,7,8,10]
  i=0
  print("List elements")
  while i<len(L):
      print(L[i],end=' ')
      i=i+1
  print("\nOut of loop")
  OUTPUT:
  List elements
  1 4 7 8 10
  Out of loop
```

The code is easy to understand. We have started from index **0 to len(L)-1** and displayed all elements of list **L** using index notation **L[i]**. The value of **i** is incremented in every iteration of **while** loop. When the condition **i<len(L)** becomes false control comes out from loop.

Let's see one more example where we double every element of list store in new list and add 10 to every element of list and store into new list. Thus, code generated two new list along with original list.

```
  # Script 7.2 Generating new list from existing using while loop
  L=[1,4,7,8,10]
  L1=[]
  L2=[]
  i=0
  while i<len(L):
      L1.append(L[i]+10)
      L2.append(L[i]*2)
      i=i+1
  print("List L=",L)
  print("List L1=",L1)
  print("List L2=",L2)
  OUTPUT:
  List L= [1, 4, 7, 8, 10]
  List L1= [11, 14, 17, 18, 20]
```

```
List L2= [2, 8, 14, 16, 20]
```

Two new list are initialized to empty. As the list L1, L2 have zero length so you cannot assign element to them using index notation. You'll have to use the append method for adding element to the list. The same we have done in first two lines of while loop. Outside the while loop we display all three list.

7.7.2 Traversing list using for loop

List traversal using for loop is the most convenient way and most python programmers often use this method of list traversal. Let's rewrite the same code as we have seen in the previous section using for loop.

```
# Script 7.3 displaying all elements of list one by one using for loop
L=[1,4,7,8,10]
print("List elements")
for x in L:
    print(x,end=' ')
print("\nOut of loop")
OUTPUT:
List elements
1 4 7 8 10
Out of loop
```

As can be seen from above code you only need element in the **for** loop with in operator and there is no need of index. Each time the **for** loop runs the loop variable **x** takes a new element from list **L** and displays it. When list becomes empty the **for** loop exits.

The need of index arises only when we have to perform some modification onto list elements. Even though we don't need index here, the above code can also be written as (we show just changed **for** loop):

```
for i in range(len(L)):
    print(L[i],end=' ')
```

Let's write the code where we generated two list from an existing list using for loop.

```
# Script 7.4 Generating new list from existing using for loop
L=[1,4,7,8,10]
L1=[]
L2=[]
for x in L:
    L1.append(x+10)
    L2.append(x*2)
print("List L=",L)
print("List L1=",L1)
print("List L2=",L2)
OUTPUT:
List L= [1, 4, 7, 8, 10]
```

```
List L1= [11, 14, 17, 18, 20]
List L2= [2, 8, 14, 16, 20]
```

As you must have noticed index notation has gone from the above code and simple for loop has resulted into simple code.

See some new scripts that make use of **for** loop for list processing

```
# simple list traversal using for loop
games=['cricket','tennis','badminton']
for game in games:
    print("I like",game)
```

The code is quite simple. The game variables take values from games list and print statement executes for each element of list games.

```
# Script 7.5 count odd and even elements in list
L=[4,5,7,9,12,3,14,89,11,10]
counteven,countodd=0,0
for x in L:
    if x%2==0:
        counteven+=1
    else:
        countodd+=1
print("Number of even elements=",counteven)
print("Number of odd elements=",countodd)
OUTPUT:
Number of even elements= 4
Number of odd elements= 6
```

The code finds count of odd and even elements of the list. For this we have taken two variables counteven and countodd and initialized them to 0. The list is traversed using for loop and each element is examined for even or odd. The corresponding variable is incremented. Outside the loop we display the count using print statement.

What if in the above code we want to separate the odd and even elements into new list and destroy the original list? Let's see how to do this:

```
# Script 7.6 Separating odd and even elements and removing original
#list
L=[4,5,7,9,12,3,14,89,11,10]
evenL=[];oddL=[]
for x in L:
    if x%2==0:
        evenL.append(x)
    else:
        oddL.append(x)
del L
```

```
print("Even elements List=",evenL)
print("Odd elements List=",oddL)
OUTPUT:
Even elements List= [4, 12, 14, 10]
Odd elements List= [5, 7, 9, 3, 89, 11]
```

We have created two empty list **oddL** and **evenL**. The list is traversed using **for** loop. If it is even element than it is appended into **evenL** list else it is appended into **oddL** list. Outside the **for** loop the original list is deleted.

In final script we process a list of strings and check whether they are palindrome or not. A palindrome string is same on reversal.

```
# Script 7.7 Checking string for palindrome
def palindrome(s):
    s1=s[::-1]
    if s==s1:
        return True
    else:
        return False

L=['peep','novia','keep','maam','malayalam']
for x in L:
    if palindrome(x):
        print(x,"is palindrome")
    else:
        print(x,"is not palindrome")
OUTPUT:
peep is palindrome
novia is not palindrome
keep is not palindrome
maam is palindrome
malayalam is palindrome
```

For checking string is palindrome or not we have written a function **palindrome** that reverses a string and compare it with original. If both strings are same than **True** is returned from function else **False** is returned from function. In the main function the list is processed using **for** loop and every list element is passed to **palindrome** function that tells whether string is palindrome or not. This is checked in **if** block and corresponding result is displayed.

7.8 Other List methods

This section discusses some other list methods that we have not covered so far. Let's discuss them one by one.

7.8.1 The copy method

The **copy** method makes a copy of the list on which it is called upon. The reason we would like to use **copy** method on number of occasions is that merely copying one list to another using assignment operators creates a reference and not a copy. See this:

```
>>> L=[2,4,7,9,1]
>>> L1=L
>>> L1[0]=34
>>> L
[34, 4, 7, 9, 1]
>>> L1.pop(2)
7
>>> L
[34, 4, 9, 1]
```

When you write **L1=L**, a new reference of **L** is created, and all changes done onto list either by **L** or by **L1** seen by both. To avoid this, we can create a copy of **L** and assign to **L1** so operations performed on **L** will have no effect on **L1** and vice versa.

```
>>> L
[34, 4, 9, 1]
>>> L1=L.copy()
>>> L1[0]='changed'
>>> L
[34, 4, 9, 1]
>>> L1
['changed', 4, 9, 1]
>>> L==L1
False
>>> L2=L
>>> L==L2
True
```

The other way to make a copy of the list is by using **L1=L[:]**.

7.8.2 The sort method

The sort sorts the elements of the string in place. It means original list is modified. Because of in place sorting the function returns None. So in case you want to keep the original unsorted list intact better make a copy of the list.

```
>>> L=[2,3,12,1,15]
>>> L.sort()
>>> L
[1, 2, 3, 12, 15]
>>> L=[2,3,12,1,15]
>>> L1=L.sort()
>>> print(L1)
None
>>> L
```

```
[1, 2, 3, 12, 15]
>>> L=[2,3,12,1,15]
>>> L.sort(reverse=True)
>>> L
[15, 12, 3, 2, 1]
```

Setting **reverse** keyword parameter to **True** in **sort** function sorts the elements in descending order.

7.8.3 The clear and count method

The **clear** method clears the list and make it empty. The **count** method counts occurrences of any specific element passed as argument. Returns 0 if element is not in list. See examples of both:

```
>>> L=list('positive')
>>> L.count('i')
2
>>> L
['p', 'o', 's', 'i', 't', 'i', 'v', 'e']
>>> L.clear()
>>> L
[]
>>> L=list('positive')
>>> L.count('x')
0
```

7.9 General methods applied on list

There are number of predefined functions into the **builtins** modules that can be used to work with list. In this section we are going to look at all those functions, so you can use them directly without writing any explicit function for that.

7.9.1 The max,min,sum functions

The functions **max, min** and **sum** finds maximum, minimum and sum of all elements of list respectively. See the examples:

```
>>> L=[2,5,8,11,34,78]
>>> max(L)
78
>>> min(L)
2
>>> sum(L)
138
>>> L=['a','c','b']
>>> max(L)
'c'
>>> min(L)
```

```
'a'
>>> sum(L)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

7.9.2 The all and any function

The function **all** returns true if all the elements of the list are nonzero or True else the function returns False. The **any** function returns True when any element of the list is True. It returns false only when all the elements of the list are zero. See examples:

```
>>> L=[2,5,8,11,34,78]
>>> all(L)
True
>>> L=[0,0,0,3]
>>> all(L)
False
>>> any(L)
True
>>> L=[0,0,0,0]
>>> any(L)
False
>>> L=[False,True]
>>> all(L)
False
>>> any(L)
True
```

7.9.3 The sorted method

The **sorted** method like **sort** method of list class also does sorting. The difference between **sort** and **sorted** is *that **sorted** does not do in place sorting instead it returns a newly sorted list keeping original list intact.* See examples:

```
>>> L=[12,5,18,1,3,7]
>>> sorted(L)
[1, 3, 5, 7, 12, 18]
>>> L1=sorted(L)
>>> L1
[1, 3, 5, 7, 12, 18]
>>> L
[12, 5, 18, 1, 3, 7]
```

7.9.4 The enumerate method

The **enumerate** method returns a list of tuples where every tuple has two values: the first is index and corresponding element at that index. The index is not the actual index of the element in the list and can be changed during display. By default, it is 0. See some examples:

```
>>> L1=['juhi','akshay','namit']
>>> enum=enumerate(L1)
>>> enum
<enumerate object at 0x000001A11EEB6438>
>>> L=list(enum)
>>> L
[(0, 'juhi'), (1, 'akshay'), (2, 'namit')]
>>> enum=enumerate(L1,10)
>>> list(enum)
[(10, 'juhi'), (11, 'akshay'), (12, 'namit')]
```

The **enumerate** function returns an object that needs to be converted into list using **list** function. As can be seen from the output the list **L** contains list of tuples where second element of tuple is corresponding element from the list. In the next example we have passed second parameter as 10 so now first element of the returned list has index at 10.

As the returned value of enumerate is a list of tuples it can easily be processed using **for** loop. See an example:

```
# In Shell
>>> L1=['juhi','akshay','namit']
>>> for index,name in enumerate(L1):
...     print(index,'==>',name)
...
0 ==> juhi
1 ==> akshay
2 ==> namit
```

The enumerate(L1) returns [(0, 'juhi'), (1, 'akshay'), (2, 'namit')]. First time when for loop runs index has value 0 and name has ‘juhi’, second time index takes 1 and names takes ‘akshay’ and so on.

```
# Script 7.8 enumerate function in action
L1=['juhi','akshay','namit']
for index,name in enumerate(L1):
    print(index,'==>',name)
OUTPUT:
0 ==> juhi
1 ==> akshay
2 ==> namit
```

7.10 List Input

In all the earlier sections we have initialized the list directly or statically. There are occasions where you want to take input from files or keyboard (standard input). Files we will cover later in this book. But here we see how to assign elements to list using keyboard input.

Let's write a simple code to take float inputs from users and append into an empty list. Later we find maximum, minimum and sum of all elements of the list.

```
# Script 7.9 List input demonstration from keyboard version 1
L=list()
while(True):
    x=float(input('Enter a float,-99 to stop\n'))
    if x==-99:
        break
    L.append(x)
print("List L=",L)
print("max element=",max(L))
print("min element=",min(L))
print("sum of all elements=%5.2f"%(sum(L)))
OUTPUT:
Enter a float,-99 to stop
2.4
Enter a float,-99 to stop
3.4
Enter a float,-99 to stop
5.4
Enter a float,-99 to stop
-99
List L= [2.4, 3.4, 5.4]
max element= 5.4
min element= 2.4
sum of all elements=11.20
```

The loop runs infinite times because of **while(True)**. The loop asks for float number to be entered by the user for indefinite number of times. To stop user enters -99. Every element entered by user is appended into the list L except -99. When control comes out from loop than we display the whole list, maximum, minimum element of the list and sum of all elements of the list.

If you decide to input a fixed number of elements, the same you can ask from user and run a loop for that number of time. See the below script:

```
# Script 7.10 List input demonstration from keyboard version 2
L=list()
n=int(input('Enter how many elements\n'))
for i in range(n):
    x=float(input("Enter float element no."+str(i+1)+"\n"))
    L.append(x)
print("List L=",L)
print("max element=",max(L))
print("min element=",min(L))
print("sum of all elements=%5.2f"%(sum(L)))
```

OUTPUT:

```
Enter how many elements
4
Enter float element no.1
2
Enter float element no.2
4
Enter float element no.3
5
Enter float element no.4
7
List L= [2.0, 4.0, 5.0, 7.0]
max element= 7.0
min element= 2.0
sum of all elements=18.00
```

What if you want to add different types of elements taken from user ? This will create problem as input method reads all input as string. Later you can convert them into corresponding type. One thing we will have to make sure that user enters correct type of element else runtime error will occur. Let's see how this can be done? Let's write the script

```
# Script 7.11 Converting data types of list elements
import ast
dtype=['int','string','bool','list']
L=list()
for x in range(len(dtype)):
    x=input('Enter '+dtype[x]+' element\n')
    L.append(x)
print("Simple input=",L)
for i in range(len(L)):
        if dtype[i]=='int':
            L[i]=int(L[i])
        elif dtype[i]=='bool':
                L[i]=bool(L[i])
        elif dtype[i]=='list':
            L[i]=ast.literal_eval(L[i])
print("After converting=",L)
```

OUTPUT:

```
Enter int element
23
Enter string element
cool
Enter bool element
True
Enter list element
[3,4]
Simple input= ['23', 'cool', 'True', '[3,4]']
After converting= [23, 'cool', True, [3, 4]]
```

We have a list **dtype** where few data types are written in string form. This list we use for reading a specific type of element from the user. Each such element is stored at same index in the list **L**. This means that as index of 'int' in dtype is 0 so integer is stored at index 0 in L. We assume that when we ask for integer , user enters integer and not any other type. Erroneous input can be handled using exception handling mechanism that we will see later in this book.

We have taken element of each data type from user and stored in list **L** but all are in string form. After all elements have been appended into list **L**, the list **L** is traversed again and every element is converted into its real type i.e. 'True' is converted from string to bool, '23' is converted from string to integer and so on.

Finally to wrap up this section lets use function **eval** that can be used to take all elements of a list at once from user:

```
L=eval(input("Enter elements: "))
print(L)
OUTPUT:
Enter elements : 4,5,6,7,12,True,"demo",[4,5]
(4, 5, 6, 7, 12, True, 'demo', [4, 5])
```

As you can see from the above code,using **eval** function you can input any type of elements at once into list but they all must be separated by comma. Not just list even tuple , dictionary or set can also be input using **eval** method.

7.11 List and Functions

As we pass simple types like integer, float , string to functions, list can also be passed to functions. Whenever a list is passed to function it is passed by reference. The concept of this we have seen in **copy** function. As a reference of list is passed to the function , the changes performed inside the function onto list are visible outside the list. Let's start with a simple example.

```
# Script 7.12 demo of passing list to function
def demo(Lf):
 Lf[0]='modified'

L=[1,'hello',23]
print("List before function call=",L)
demo(L)
print("List after function call=",L)
OUTPUT:
List before function call= [1, 'hello', 23]
List after function call= ['modified', 'hello', 23]
```

As can be seen from the above code the first element of the passed list is modified in the function **demo** and same affects the original list. Here **L** is our actual parameter and **Lf** is formal parameter.

In the above we saw an example of passing list as an argument to function. We can also return a list from a function. Let's see with an example:

```
# Script 7.13 Removing first element of list
def removefirst(Lf):
 return Lf[1:]

L=[1,'hello',23]
print("Original list=",L)
L1=removefirst(L)
print("List with first element removed=",L1)
OUTPUT:
Original list= [1, 'hello', 23]
List with first element removed= ['hello', 23]
```

The first element of the list is at index **0**, so we return the list starting with index **1** to the end of the list. This returned list is stored in **L1** in the main function. The output clearly justifies what i've said in previous lines.

But the inquisitive mind can figure out that same code can be written without returning list from function. Yes ! But it creates some problem. Let's see about this:

```
def removefirst(Lf):
 Lf=Lf[1:]

L=[1,'hello',23]
print("List L=",L)
removefirst(L)
print("List with head removed =",L)
OUTPUT:
List L= [1, 'hello', 23]
List with first element removed = [1, 'hello', 23]
```

The issue with the code is that slice operation on list creates a new list and does not return the reference of original list. That's why the **Lf** that is in function argument list and **Lf** in first line of function **removefirst** are two different list. To prove what I just said, we uses **id** function in **removefirst** and compare the **ids** of two **Lf**. Let's see this modified code (only for purpose of id comparison)

```
def removefirst(Lf):
    print(id(Lf))
    Lf=Lf[1:]
 print(id(Lf))

L=[1,'hello',23]
print("List L=",L)
removefirst(L)
print("List with first element removed =",L)
OUTPUT:
List L= [1, 'hello', 23]
```

```
2429924851976
2429924851848
List with first element removed = [1, 'hello', 23]
```

As you can see, the id of two Lf are different. It means they represent two different list and not just two references pointing to same list.

Continuing with the above we write one more function to remove last element of the list and then we combine the two functions to remove first and last element of the list.

```
# Script 7.14 Removing last element of the list
def removelast(Lf):
    return Lf[:-1]

L=[1,'hello',23]
print("List L=",L)
L1=removelast(L)
print("List with last element removed =",L1)
OUTPUT:
List L= [1, 'hello', 23]
List with last element removed = [1, 'hello']
```

To remove the last element start with 0 as first index (which is default) and set last index is -1 (signifies index of last element) . Let's combine the two to remove both first and last element of the list.

```
  def removeFirstLast(Lf):
return Lf[1:-1]

  L=[1,'hello',23]
  print("List L=",L)
  L1=removeFirstLast(L)
  print("List with first n last element removed =",L1)
  OUTPUT:
  List L= [1, 'hello', 23]
  List with first n last element removed = ['hello']
```

It's quite easy to understand the function **removeFirstLast**. Now if want to combine the previous two functions in a new function you can do as (bit lengthy and require three functions !)

```
def removeFirstLast(Lf):
    return removeFirst(removeLast(Lf))
def removeFirst(Lf):
    return Lf[1:]
def removeLast(Lf):
    return Lf[:-1]

L=[1,'hello',23]
print("List L=",L)
L1=removeFirstLast(L)
print("List with first n last element removed =",L1)
```

7.12 List Comprehension

List comprehension is a python way to make a new list by applying an expression to every element in a sequence for example list or string. They make use of for loop and some expression. Following **for,** you can also have some **if** clauses. The output of the list comprehension is a new list obtained by **for** loop and **if** clauses (optional). The general syntax is :

```
newlist= [ expression for element in list if condition ]
```

The **for** loop and **if** conditions can be more than one depending upon what you want to achieve with list comprehension.

They are undoubtedly resulting in much simpler code and quite easy. Lets understand with a simple example where we want to square each element of an integer list. Without list comprehension the code will be:

```
L=list(range(1,11))
L1=[]
for i in range(len(L)):
    L1.append(L[i]*L[i])
print(L1)
```

With list comprehension the code can be shortened as:

```
L=list(range(1,11))
L1=[x*x for x in L]
print(L1)
```

The second line in the code is list comprehension. The expression is written first and **for** loop later. The whole is put inside square brackets. See how easy is to use list comprehension in python. Lets see some more example:

We want to generate a list of random integers between 1 and n and then separate them into odd and even number list as we have seen earlier. But this time we do using list comprehension.

```
# Script 7.15 Even odd separation using list comprehension
import random
L=[random.randint(1,30) for i in range(10)]
LE=[x for x in L if x%2==0]
LO=[x for x in L if x%2!=0]
print("Main List=",L)
print("Even List=",LE)
print("Odd List=",LO)
OUTPUT:
Main List= [30, 7, 5, 26, 17, 24, 7, 5, 18, 22]
Even List= [30, 26, 24, 18, 22]
Odd List= [7, 5, 17, 7, 5]
```

The module **random** has a **randint** function to generate random integer between given range **a** and **b** (inclusive both). Here we generate 10 random numbers between 1 and 30 and store in list **L**. Than using list comprehension we store the even and odd elements of the list into new lists.

Like if, if-else can also be used inside list comprehension. The general syntax is:

```
[expression if condition else expression for item in list]
```

See an example where the list is elements containing random values between 1000 for all values greater than 5000 in a list we add 15% of value when value is >=5000 and 10% of value when value is less than 5000. The values can be treated as salary of workers.

```
# Script 7.16 Random salary and bonus using list comprehension
import random
L=[random.randint(1000,10000) for i in range(10)]
print("Salary without bonus=",L)
L1=[round(x*1.15) if x>=5000 else round(x*1.10,2) for x in L]
print("Salary with bonus=",L1)
OUTPUT:
Salary without bonus= [7986, 8830, 7467, 5457, 3313, 3468, 5823, 5484,
8641, 2197]
Salary with bonus= [9184, 10154, 8587, 6276, 3644.3, 3814.8, 6696, 6307,
9937, 2416.7]
```

As you can see from the above code the list comprehension makes the code easy and compact.

Functions can also be used for processing of list elements. We write a function **bonus** that takes salary from the list **L** and calculated the salary with **bonus** using the condition as discussed above.

```
import random
L=[random.randint(1000,10000) for i in range(10)]
def bonus(sal):
    if sal>=5000:
        return round(sal*1.15,2)
    else:
        return round(sal*1.10,2)

L1=[bonus(sal) for sal in L]
print("Salary without bonus=",L)
print("Salary with bonus=",L1)
OUTPUT:
Salary without bonus= [5426, 9141, 7799, 4255, 9015, 3124, 6593, 4460,
7214, 9522]
Salary with bonus= [6239.9, 10512.15, 8968.85, 4680.5, 10367.25, 3436.4,
7581.95, 4906.0, 8296.1, 10950.3]
```

In line `L1=[bonus(sal) for sal in L]` we apply function **bonus** on every element **sal** of the list **L**. The function returns the salary with bonus calculated as discussed above.

Let's write one more function where we filter out only prime elements from a list. For that we write a function that checks number is prime or not and returns True or False accordingly.

```
# Script 7.17 Prime number checking using list comprehension
import random
L=[random.randint(1,50) for i in range(10)]
def prime(n):
    flag=True
    for c in range(2, n//2+1):
        if n%c==0:
            flag=False
            break
    return flag

PL=[x for x in L if prime(x)]
print("Original List=",L)
print("Prime List=",PL)
OUTPUT:
Original List= [46, 19, 38, 20, 11, 33, 41, 36, 18, 35]
Prime List= [19, 11, 41]
```

We randomly generated 10 numbers between 1 and 50 and checked for primeness of every element of the generated list. The logic for prime numbers was discussed in *chapter 5: Functions.*

List comprehension can also be done with multiple list. Let's see one example where every element of both the list is added at same index.

```
# Script 7.18 List comprehension with multiple list ver 1
import random
L1=[random.randint(1,30) for i in range(5)]
L2=[random.randint(1,30) for i in range(5)]
print("Zipped List=",list(zip(L1,L2)))
SUML=[x+y for x,y in zip(L1,L2)]
print("Original List L1=",L1)
print("Original List L2=",L2)
print("Sum of two List=",SUML)
OUTPUT:
Zipped List= [(29, 3), (4, 17), (13, 5), (12, 30), (26, 27)]
Original List L1= [29, 4, 13, 12, 26]
Original List L2= [3, 17, 5, 30, 27]
Sum of two List= [32, 21, 18, 42, 53]
```

The zip function returns a list of tuples or pairs. It takes more than one list as argument and combine the elements at same index. This is the first line of the output. In list comprehension every pair is picked up and its elements are copied into x and y. For performing sum the list comprehension can also be written as:

```
SUML=[sum(pair) for pair in zip(L1,L2)]
```

A little change and we can find maximum of elements at same index position among two list.

```
# Script 7.19 List comprehension with multiple list ver 2
import random
L1=[random.randint(1,30) for i in range(5)]
L2=[random.randint(1,30) for i in range(5)]
print("Zipped List=",list(zip(L1,L2)))
MAX=[x if x>y else y for x,y in zip(L1,L2)]
print("Original List L1=",L1)
print("Original List L2=",L2)
print("Sum of two List=",MAX)
OUTPUT:
Zipped List= [(3, 27), (12, 28), (12, 21), (7, 2), (27, 30)]
Original List L1= [3, 12, 12, 7, 27]
Original List L2= [27, 28, 21, 2, 30]
Sum of two List= [27, 28, 21, 7, 30]
```

List comprehension can also be done with multiple for loops. This is equivalent to nesting of two or more for loops. Let's understand it with an example. This time we work with list of strings and find out a new list with common names in both the list.

```
# Script 7.20 List comprehension with two for loops
import random
L1=['jiya','anu','lavi','piya','pari']
L2=['pari','chinu','jiya','piya']
COM=[x for x in L1 for y in L2 if x==y]
print("Original List L1=",L1)
print("Original List L2=",L2)
print("Common names list=",COM)
OUTPUT:
Original List L1= ['jiya', 'anu', 'lavi', 'piya', 'pari']
Original List L2= ['pari', 'chinu', 'jiya', 'piya']
Common names list= ['jiya', 'piya', 'pari']
```

The list comprehension is equivalent to :

```
COM=[]
for x in L1:
    for y in L2:
        if x==y:
            COM.append(x)
```

Let's take one last example with 3 lists. We try to find out if sum of any element in the two list is equal to any element in the third list.

```
# Script 7.20 List comprehension with two for loops
import random
L1=[random.randint(1,10) for i in range(10)]
L2=[random.randint(1,10) for i in range(10)]
L3=[random.randint(1,20) for i in range(10)]
Res=[(x,y,z) for x in L1 for y in L2 for z in L3 if x+y==z]
print("Original List L1=",L1)
```

```
    print("Original List L2=",L2)
    print("Original List L3=",L3)
    print("Result List=",Res)
    OUTPUT:
    Original List L1= [7, 3, 7, 1, 7, 5, 5, 7, 9, 6]
    Original List L2= [5, 10, 2, 10, 6, 5, 4, 1, 1, 4]
    Original List L3= [7, 6, 2, 1, 2, 8, 20, 11, 8, 13]
    Result List= [(7, 6, 13), (7, 4, 11), (7, 1, 8), (7, 1, 8), (7, 1, 8),
(7, 1, 8), (7, 4, 11), (3, 5, 8), (3, 5, 8), (3, 10, 13), (3, 10, 13), (3,
5, 8), (3, 5, 8), (3, 4, 7), (3, 4, 7), (7, 6, 13), (7, 4, 11), (7, 1, 8),
(7, 1, 8), (7, 1, 8), (7, 1, 8), (7, 4, 11), (1, 5, 6), (1, 10, 11), (1,
10, 11), (1, 6, 7), (1, 5, 6), (1, 1, 2), (1, 1, 2), (1, 1, 2), (1, 1, 2),
(7, 6, 13), (7, 4, 11), (7, 1, 8), (7, 1, 8), (7, 1, 8), (7, 1, 8), (7, 4,
11), (5, 2, 7), (5, 6, 11), (5, 1, 6), (5, 1, 6), (5, 2, 7), (5, 6, 11),
(5, 1, 6), (5, 1, 6), (7, 6, 13), (7, 4, 11), (7, 1, 8), (7, 1, 8), (7, 1,
8), (7, 1, 8), (7, 4, 11), (9, 2, 11), (9, 4, 13), (9, 4, 13), (6, 5, 11),
(6, 2, 8), (6, 2, 8), (6, 5, 11), (6, 1, 7), (6, 1, 7)]
```

Its quite obvious from the output that nesting of three for loops in our list comprehension have resulted in total execution of if condition 10*10*20 times and have generated lots of triplets tuples having sum of first two elements is equal to third element. But the resultant list contains lots of duplicate tuples. To remove them and keep the list as list , what we can do is to convert the list into set (set can only have unique elements) and convert back to list. Just add one line in the code above as:

```
    import random
    L1=[random.randint(1,10) for i in range(10)]
    L2=[random.randint(1,10) for i in range(10)]
    L3=[random.randint(1,20) for i in range(10)]
    Res=[(x,y,z) for x in L1 for y in L2 for z in L3 if x+y==z]
    Res=list(set(Res))
    print("Original List L1=",L1)
    print("Original List L2=",L2)
    print("Original List L3=",L3)
    print("Result List=",Res)
    OUTPUT:
    Original List L1= [9, 1, 2, 10, 4, 9, 7, 4, 6, 1]
    Original List L2= [9, 9, 10, 2, 9, 4, 10, 2, 2, 10]
    Original List L3= [9, 2, 6, 12, 17, 15, 10, 17, 11, 2]
    Result List= [(7, 2, 9), (6, 4, 10), (4, 2, 6), (7, 4, 11), (2, 10, 12),
(2, 4, 6), (6, 9, 15), (1, 9, 10), (1, 10, 11), (7, 10, 17), (10, 2, 12),
(2, 9, 11), (9, 2, 11)]
```

7.13 Ponderable Points

1. A list is a mutable data type in python.
2. List indexing using [] operator with first element at index 0 and last element at -1.
3. Updation, insertion, deletion, appending all types of operations can be performed on list.
4. Operators +, * and membership operators can be used with list.
5. List can be easily traversed using any loop but for loop is preferred.
6. List has number of methods to use in any programming situations.

7. Like other data types list can be passed to functions and returned also.
8. General syntax for list comprehension is: newlist= [expression for element in list if condition]

8. Dictionary

8.1 Introduction

A dictionary is a collection of key-value pair. Here key means index and value means value at the given index. Unlike list the index can be anything in dictionary. It can be integer, string or any other object. The dictionary is an unordered, mutable collection. The dictionary can be thought of mapping from keys to values. Every key is mapped to some value. This pair of key-value is treated as one element or item of the dictionary. The good thing about dictionary and its usefulness in number of programming situations is searching for an element through key. You just need to remember the key and dictionary will return the value associated with the key. The keys for the dictionary must be immutable and cannot be repeated. That is keys must be unique and nonmodifiable. The values can be repeated.

8.2 Creating Dictionaries

The dictionary in python can be easily created with curly braces **{ }** or using built-in function **dict()**. Let's see how to create an empty dictionary.

```
>>> x={}
>>> type(x)
<class 'dict'>
>>> x=dict()
>>> type(x)
<class 'dict'>
```

The key value pairs can be added to an empty dictionary as:

```
dictionaryname[key]=value
```

Let's add some items into the empty dictionary created above:

```
>>> x[1]='one'
>>> x[2]='two'
>>> x[5]='five'
>>> x
{1: 'one', 2: 'two', 5: 'five'}
```

As you can see from above 3 items have been added to the dictionary **x** with key written in square brackets and value assigned at that key using assignment operator. To display the dictionary just write the dictionary name at the shell prompt. The dictionary can be created also in just one single line as:

```
>>> x={1:'one',2:'two',3:'three'}
>>> x
{1: 'one', 2: 'two', 3: 'three'}
```

Another ways to create dictionary is to have a list of key-value pair in the form of a tuple. See an example:

```
>>> x=dict([(1,'one'),(2,'two'),(3,'three')])
>>> x
{1: 'one', 2: 'two', 3: 'three'}
```

An example of dictionary with strings as keys is given below:

```
>>> d={'name':'pihu','age':'18','sex':'female'}
>>> d
{'name': 'pihu', 'age': '18', 'sex': 'female'}
```

Another way to write the above one is using keyword argument.

```
d=dict(name='pihu',age='18',sex='female')
```

Even though the preceding dictionary creation syntax is quite easy but for accessing values of an individual item , the key must be placed in single or double quotes. See the error caused:

```
>>> d[name]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
```

To correct the above use as:

```
>>> d['name']
```

'pihu'

The value for a key can also be a list. For example in the above if we want to add a new key sports a person like with more than one option can be added as:

```
>>> d['sports']=['cricket','tennis','soccer']
>>> d
{'name': 'pihu', 'age': '18', 'sex': 'female', 'sports': ['cricket', 'tennis', 'soccer']}
```

A dictionary can also be a dictionary for a given key. This is nesting of dictionary. Let's see how to do this. We add new key in the above dictionary:

```
>>> d['address']={'colony':'bank colony','hno':'b-62','city':'ajmer'}
```

Here the address key's value is a dictionary. To access individual key inside this dictionary (which is a value) we need to perform two way indexing. Left one is for main dictionary and right for dictionary as value:

```
>>> d['address']['city']
'ajmer'
```

8.2.1 Restriction on keys

There is some restriction on the type of keys that can be used while creating dictionaries in python. Only types that are immutable can be used as keys. We know that string, numbers, tuples are immutable data types in python so they can be used as keys. Mutable types like list and sets cannot be used as keys. If you try to do so error is flashed. See some examples

```
>>> d={[1,2]:'one two'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d={{2,3}:'one two'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> d={(1,2):'one two'}
>>> d[(1,2)]
'one two'
```

8.3 Accessing elements

The dictionary elements are accessed by simply providing their key as index. The other way is to use get method. Both are discussed here. Assuming dictionary d from previous section still exist in our python shell:

```
>>> d['name']
{'first': 'pari', 'last': 'jain'}
>>> d.get('name')
{'first': 'pari', 'last': 'jain'}
>>> d['age']
'18'
>>> d.get('name')['first']
'pari'
>>> d['name']['first']
'pari'
```

What if you try to access an item in the dictionary whose key is not present ? An error is generated:

```
>>> d['job']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'job'
>>> d.get('job')
>>> print(d.get('job'))
None
```

The **get** method returns **None** and does not generate any error when key is not present in dictionary. The **None** was displayed only when we used **print** statement. To avoid this , the **get** method also takes an additional argument which is returned only when key is not present. See few examples:

```
>>> d={1:98.3,2:95.4,3:95.4}
>>> d.get(4)
>>> d.get(4,'key does not exist')
# second argument used when key is not present
'key does not exist'
>>> d.get(4,0)
0
>>> d.get(1,0)
98.3
```

8.4 Adding and Modifying elements in Dictionary

We have seen earlier that a new element can be added to the dictionary just by adding key and value as:

```
Dname[key]=value
```

And we have seen number of examples of this earlier. To modify the elements just change the value at key . See some examples:

```
>>> d={1:'puru',3:'luv',5:'kuhu'}
>>> d
{1: 'puru', 3: 'luv', 5: 'kuhu'}
>>> d[4]='riya'
>>> d[1]='pari'
>>> d
{1: 'pari', 3: 'luv', 5: 'kuhu', 4: 'riya'}
```

8.5 Removing elements from dictionary

For removing items from dictionary three methods can be used: **pop, popitem** and **del**. The **pop** method's signature is as follows:

```
D.pop(k[,d]) -> v,
```

Here **D** is the dictionary object, **k** is the key , **v** is the corresponding value at key **k** and **d** is any type value which is returned when key is not found. This type value **d** is optional. When key is not present in dictionary and **d** is not specified then **KeyError** is raised. See some examples:

```
>>d={1: 'pari', 3: 'luv', 5: 'kuhu', 4: 'riya'}
>>> d.pop(1)
'pari'
>>> d
{3: 'luv', 5: 'kuhu', 4: 'riya'}
```

```
>>> d.pop(2,'Key not found')
'Key not found'
>>> d.pop(5,'Key not found')
'kuhu'
>>> d.pop(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
```

The **popitem** method returns any randomoly chosen values from the dictionary. The order does not matter here. It is quite useful when you have to delete an entry from dictionary regardless of any key. The method errors only when dictionary has no items. See some examples:

```
>>> d={3:'aa',4:'bb',1:'cc'}
>>> d
{3: 'aa', 4: 'bb', 1: 'cc'}
>>> d[2]='dd'
>>> d
{3: 'aa', 4: 'bb', 1: 'cc', 2: 'dd'}
>>> d.popitem()
(2, 'dd')
>>> d.popitem()
(1, 'cc')
>>> d.popitem()
(4, 'bb')
>>> d.popitem()
(3, 'aa')
>>> d.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

The other way of deleting items from the dictionary is **del** command. The **del** command can be used to delete an item from dictionary or entire dictionary. See some examples:

```
>>> d={3:'aa',4:'bb',1:'cc'}
>>> del d[4]
>>> d
{3: 'aa', 1: 'cc'}
>>> del d[2]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
>>> del d
>>> d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
```

The final method for removing items from dictionary is clear that removes every item from dictionary and make it empty.

```
>>> d={3:'aa',4:'bb',1:'cc'}
>>> d.clear()
>>> d
{}
```

8.6 Traversing Dictionaries

The dictionary can be easily traversed using **for** loop. The methods **keys, values, items** are of importance while traversing dictionary in python. Let's first see these methods.

```
>>> d={3:'aa',4:'bb',1:'cc'}
>>> d.keys()
dict_keys([3, 4, 1])
>>> d.values()
dict_values(['aa', 'bb', 'cc'])
>>> d.items()
dict_items([(3, 'aa'), (4, 'bb'), (1, 'cc')])
```

The **keys** method returns a list of all keys of dictionary on which it is applied. The **values** returns a list of all values of dictionary on which it is applied. The **items** method returns a list of all tuples in the form of key-value pair.

Now see how can we traverse a dictionary. Let's define a dictionary **students** where key is student name and value is their aggregate percentage.

```
 >>> students={'pari':98,'koyal':93,'mouni':93.5,'lavi':94}
>>> for name,per in students.items():
...     print(name," has got ",per,'%')
...
pari  has got  98 %
koyal  has got  93 %
mouni  has got  93.5 %
```

```
lavi  has got  94 %
```

The above code we have written in python shell. The same can be written as a script:

```
students={'pari':98,'koyal':93,'mouni':93.5,'lavi':94}
for name,per in students.items():
    print(name,"has got",per,'%')
```

OUTPUT:

```
pari has got 98 %
koyal has got 93 %
mouni has got 93.5 %
lavi has got 94 %
```

I discussed the methods **keys()** and **values()** above. Let's use them here:

```
>>> students.values()
dict_values([98, 93, 93.5, 94])
>>> students.keys()
dict_keys(['pari', 'koyal', 'mouni', 'lavi'])
```

The above traversal of dictionary items can also be done using the **keys()** methods . For every key in keys list we can find the value as: **students[key]**. This is what we have done in the following modified script:

```
students={'pari':98,'koyal':93,'mouni':93.5,'lavi':94}
for key in students.keys():
    print(key,"has got",students[key],'%')
```

The output remains same.

8.7 Methods of Dictionary class

Number of methods of dictionary class we have seen in previous sections. Here we discuss the remaining methods that are important and frequently used.

1. Copy method

References of a dictionary can be created and original dictionary can be easily modified using the reference. See this:

```
>>> students
{'pari': 98, 'koyal': 93, 'mouni': 93.5, 'lavi': 94}
>>> s1=students
>>> s1
{'pari': 98, 'koyal': 93, 'mouni': 93.5, 'lavi': 94}
>>> s1['pari']=99
>>> students
```

```
{'pari': 99, 'koyal': 93, 'mouni': 93.5, 'lavi': 94}
```

To prevent this a copy of the original dictionary can be created and operations performed on copied dictionary does not affect the original dictionary. See an example:

```
>>> students
{'pari': 99, 'koyal': 93, 'mouni': 93.5, 'lavi': 94}
>>> s1=students.copy()
>>> s1['mouni']=97
>>> students
{'pari': 99, 'koyal': 93, 'mouni': 93.5, 'lavi': 94}
>>> s1
{'pari': 99, 'koyal': 93, 'mouni': 97, 'lavi': 94}
```

2.update method

The update method of dictionary class merges the two dictionaries. While merging if key is already presents in the other dictionary then old value of that key is updated by new value. See some examples:

```
>>> d={1: 'navi', 2: 'lavi', 3: 'ravi'}
>>> d.update({2:'pari',4:'tanu'})
>>> d
{1: 'navi', 2: 'pari', 3: 'ravi', 4: 'tanu'}
```

The value =’lavi’ for key=2 was updated to ‘pari’ and 4:’tanu’ was added as new item in dictionary d.

The parameters for the update method can also be an iterable where keys and values can be separated using assignment operator. See an example:

```
>>> d={'english':98,'maths':90}
>>> d.update(science=89,hindi=95,sscience=97)
>>> d
{'english': 98, 'maths': 90, 'science': 89, 'hindi': 95, 'sscience': 97}
```

3.fromkeys method

The fromkeys method takes an iterable and use that as keys for the creation of new dictionary. The second parameter is the value for all the keys generated from iterables. The function returns the generated dictionary. See some examples:

```
>>> d={}
>>> d.fromkeys([1,2,3,4,5])
{1: None, 2: None, 3: None, 4: None, 5: None}
>>>d= d.fromkeys([1,2,3,4,5],0)
>>>d
{1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
```

```
>>> d.fromkeys('aeiou',0)
{'a': 0, 'e': 0, 'i': 0, 'o': 0, 'u': 0}
>>> d.fromkeys('aabra',0)
{'a': 0, 'b': 0, 'r': 0}
```

4.setdefault method

The setdefault method takes two parameters, the second parameter is optional. First parameter is treated as key to the dictionary. If key is present in the dictionary, its value is returned. If key is not present and second parameter is not provided then key is added to the dictionary with value None. If second parameter is also provided than this parameter is taken as value for the key and inserted into the dictionary. In the example shown below we are working with a dictionary *spn2eng (Spanish to English)* having some Spanish words as keys and their corresponding English translations.

```
>>spn2eng={'soy':'I am','bien':'good','el hombre':'The man'}
>>> spn2eng.setdefault('amigo')
>>> print(spn2eng.setdefault('amigo'))
None
>>> spn2eng.setdefault('gracias','thank you')
'thank you'
>>> spn2eng
{'soy': 'I am', 'bien': 'good', 'el hombre': 'The man', 'amigo': None, 'gracias': 'thank you'}
```

5.len method

This built-in method returns number of entries in the dictionary. For the above example **len(spn2eng)** returns 5.

6.sorted method

The built-in sorted method sorts the dictionary using keys and returns the sorted keys as list. This can be used to display the entries of dictionary in sorted manner.

```
>>> d={'a':34,'c':22,'r':11,'e':10}
>>> sorted(d)
['a', 'c', 'e', 'r']
>>> for k in sorted(d):
...     print(k,d[k])
...
a 34
c 22
e 10
r 11
```

The other way to sort the dictionary on keys is to apply **sort** method on keys of dictionary.

```
>>> k=list(d.keys())
>>> k.sort()
>>> k
['a', 'c', 'e', 'r']
```

8.8 Membership testing in dictionary

The in and not in operator can be applied over dictionary for testing whether a key is present in the dictionary or not.

```
>>> if 'a' in d:print ("key exist,value=",d['a'])
...
key exist,value= 34
>>> if 'b' in d:print ("key exist,value=",d['b'])
...
```

8.9 Scripting examples

```
# Script 8.1 to count frequencies of letters in string
string='aabrakadabra'
d={}
for x in string:
    d[x]=d.get(x,0)+1
print('Letters with Frequencies are')
print(d)
```

OUTPUT:

```
Letters with Frequencies are
{'a': 6, 'b': 2, 'r': 2, 'k': 1, 'd': 1}
```

The **get** method returns value of **x** if it is present in dictionary as key else return 0 (second parameter). So first time say when **'a'** is encountered then **d['a']** is set to **1** because **d['a']=d.get('a',0)+1** becomes **d['a']=0+1=1**,Thus **d['a']** is set to **1**. For next **'a'**, we have **d['a']=d.get('a',0)+1**. As **d['a']=1** so **d.get('a',0)** returns **1** and **d['a'=1+1** is set to **2**. The same logic is applied to all other characters.

```
# Script 8.2 to count frequencies of vowels in a given string
string='this is an example of dictionary'
vowels="aeiou"
d={}
d=d.fromkeys(vowels,0)
for x in string:
    if x in vowels:
```

```
        d[x]=d.get(x,0)+1
for x in d:
    print(x,d[x])
```

OUTPUT:

```
a 3
e 2
i 4
o 2
u 0
```

The code is identical to the previous script but here first the dictionary has been created using **fromkeys** function with vowels **'aeiou'** as keys and initial frequency to **0**. Rest is simple to understand. One issue with the above code is case sensitivity of vowels. Try to modify the code to include uppercase also or make it case insensitive.

Script 8.3 switch case implementation using dictionary (simple #arithmetic calculator)

```
def plus(a,b):
    return a+b
def minus(a,b):
    return a-b
def div(a,b):
    if b==0:
        return "not possible"
    else:
        return a/b
def mul(a,b):
    return a*b
def power(a,b):
    return a**b

dic={'+':plus,'-':minus,'/':div,'*':mul,'**':power}
a,b=input("Enter two numbers separated by space\n").split()
a=int(a);b=int(b)
op=input("Select operators:+ | - | * | / | **\n")
if op in dic:
    ans=dic[op](a,b)
    print("Answer=",ans)
```

```
else:
    print("Unknown operator")
```

OUTPUT:

```
(First Run)
Enter two numbers separated by space
3 0
Select operators:+ | - | * | / | **
/
Answer= not possible
(Second Run)
Enter two numbers separated by space
4 5
Select operators:+ | - | * | / | **
=
Unknown operator
```

The code is interesting to understand. To make use of dictionary as an alternative to switch case we first need to write functions for every case. In this example here, we want to achieve five arithmetic operations so we have written five functions for performing addition, subtraction, multiplication, division and power. All functions takes two parameters and return the respective result. That portion of code does not need any explanation. What is interesting in the code is that we have keys as various arithmetic symbols in the form of strings as keys. The functions names are used as values for their corresponding keys. *The expression dic['+'] returns a function name plus and expression dic['+'](10,20) is equivalent to plus(10,20).*

We first ask two numbers from user and operation which he/she wants to perform. If operator is in list of operators (as keys) than we call the dictionary with supplied operator as key and two numbers as argument. As explained in the previous para this is equivalent to calling the respective function determined by operator supplied.

8.10 Dictionary Comprehension

We have seen examples of list comprehension in previous chapter. The dictionary comprehension is same as list comprehension but here instead of just elements of list we have keys and values serving one item of the dictionary. The syntax of dictionary comprehension is also same with the difference that we are dealing with key:value and not just elements. See some examples

Sr.No	Shell Code	Explanation
1.	>>> sqr={x: x*x for x in range(1,6)} >>> sqr {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}	Keys as element from 1:6 and square of 1:6 is as values
2.	>>> d={key:value for key,value in zip('abcde',list(range(5)))} >>> d	Using zip method one key,value taken at a time from 'abcde' and 12345 and

	{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}	dictionary is created.
3.	>>> fruits={'kiwi','guava','pear','mango'} >>> fdict={k:len(k) for k in fruits} >>> fdict {'mango': 5, 'pear': 4, 'guava': 5, 'kiwi': 4}	Fruits and its length as key and dictionary elements
4.	>>>keys=[1,4,5,7] >>> values=['naman','kuntal','riya','nupur'] >>> di={k:v for k,v in zip(keys,values)} >>> di {1: 'naman', 4: 'kuntal', 5: 'riya', 7: 'nupur'}	Dictionary created from two different list:keys and values.
5.	>>> import string >>> ascii={k:ord(k) for k in string.ascii_lowercase} >>> ascii {'a': 97, 'b': 98, 'c': 99, 'd': 100, 'e': 101, 'f': 102, 'g': 103, 'h': 104, 'i': 105, 'j': 106, 'k': 107, 'l': 108, 'm': 109, 'n': 110, 'o': 111, 'p': 112, 'q': 113, 'r': 114, 's': 115, 't': 116, 'u': 117, 'v': 118, 'w': 119, 'x': 120, 'y': 121, 'z': 122} >>> ascii['v'] 118	Generating dictionary of lowercase alphabet and corresponding ASCII letter.
6.	>>> d={x:x*x for x in range(1,30) if x%5==0} >>> d {5: 25, 10: 100, 15: 225, 20: 400, 25: 625}	Dictionary comprehension with if condition.

8.11Ponderable Points

1. A dictionary is a collection of key-value pair. Here key means index and value means value at the given index.
2. The dictionary in python can be easily created with curly braces **{ }** or using built-in function **dict()**.
3. A dictionary can also be a dictionary for a given key. This is nesting of dictionary.
4. Only types that are immutable can be used as keys.
5. The dictionary elements are accessed by simply providing their key as index.
6. Elements can be added, removed, and updated easily in a dictionary.
7. The dictionary can be easily traversed using **for** loop.
8. The methods **keys, values, items** are of importance while traversing dictionary in python.
9. Plenty of useful methods can be used in any programming situation.
10. Like list comprehension, dictionary comprehension can also be achieved.

9. Tuple

9.1 Introduction

A tuple is a sequence of objects which can be anything : integer, string, boolean, etc. They are just like list with integers as indexes for accessing individual items. Tuples are just comma separated values with optional parenthesis. The difference is that list is mutable and tuple is immutable. Because of this tuple objects does not support item assignments. Another difference is that list elements are enclosed within square brackets and tuple elements are enclosed within parenthesis.

9.2 Creating Tuples

A tuple can easily be created as:

```
>>> t=1,2,3
>>> t
(1, 2, 3)
```

As mentioned earlier parenthesis is optional to surround elements. See some more examples:

```
>>> t1=()
>>> type(t1)
<class 'tuple'>
>>> t1=("a","b","man","woman")
>>> t1
('a', 'b', 'man', 'woman')
>>> t1=(10,20,True,[2,3])
>>> t1
(10, 20, True, [2, 3])
>>> t1=(1,2,{'a':20})
```

As can be seen from above examples a tuple can contains any type of python objects even list and dictionary. An empty tuple is created using (). A tuple with just one element is created as:

```
t1=(10,)
```

A comma is necessary when tuple contains just one element else the type will not be a tuple.

```
>>> t1=(10,)
>>> type(t1)
<class 'tuple'>
>>> t1=(10)
>>> type(t1)
```

```
<class 'int'>
>>> t1=('a')
>>> type(t1)
<class 'str'>
```

Python also provides tuple constructor for creating empty tuples or creating tuple object from string. See examples:

```
>>> t=tuple()
>>> type(t)
<class 'tuple'>
>>> t=tuple('example')
>>> t
('e', 'x', 'a', 'm', 'p', 'l', 'e')
```

Even list can be easily converted into a tuple. This is required when you do not want item to be modified by user. See one example:

```
>>> L=list('example')
>>> L
['e', 'x', 'a', 'm', 'p', 'l', 'e']
>>> t=tuple(L)
>>> t
('e', 'x', 'a', 'm', 'p', 'l', 'e')
```

The above code can be written in just one line of code:

```
t=tuple(list('example'))
```

9.3 Accessing Tuple elements

Accessing tuple elements are like accessing list elements. Simple indexing within square brackets and slicing both are allowed. See some example:

```
>>> t=tuple('example')
>>> t[0]
'e'
>>> t[0:4]
('e', 'x', 'a', 'm')
>>> t[-1]
'e'
>>> t[-4:-1]
('m', 'p', 'l')
>>> t[-4:]
```

```
('m', 'p', 'l', 'e')
```

To combine elements back and form a string (only in case of string elements) you can make use of **join** method of string class. See one example:

```
>>> print('you have an','').join(t[0:4]))
you have an exam
```

9.4 Modifying tuple elements

As we know that tuples are immutable, so modifying tuple elements are not allowed. Doing so will raise an error. But we can combine some slice of tuple and form new tuples or even new strings. See some examples:

```
>>> t=tuple('example')
>>> t[0]='E'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t1="My "+''.join(t[0:4])
>>> t1
'My exam'
>>> t1=(10,20)+t[0:4]
>>> t1
(10, 20, 'e', 'x', 'a', 'm')
```

9.5 Deleting tuple elements

Deleting tuple elements is not permitted because of its immutable nature. However entire tuple can be deleted at once using **del** method.

```
>>> t=(1,2,'hello',True)
>>> del t[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> del t
>>> t
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 't' is not defined
```

9.6 Operations on tuple

Tuples in python can be combined using + operator, can be compared using relational operators, membership test can be performed using in and not in operator and * operator can be used for repetition of tuple elements. Let's understand all operators using examples:

9.6.1 Tuple concatenation

Two tuples can be concatenated using the + operator. See some example:

```
>>> t=(1,2,3)
>>> t1=t+(4,5)
>>> t1
(1, 2, 3, 4, 5)
>>> t=(1,2)+('hello',)
>>> t
(1, 2, 'hello')
```

9.6.2 Tuple Repetition

The * operator as we have seen earlier with list and strings can be used with tuple also. The purpose remain same for performing repetition of elements.See some examples

```
>>> ('a',)*5
('a', 'a', 'a', 'a', 'a')
>>> t=(1,2)
>>> t*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> t=('a',)
>>> t=t+('b',)*3
>>> t
('a', 'b', 'b', 'b')
```

9.6.3 Tuple Membership

The membership operator **in** and **not in** checks existence/ non existence of an element in the tuple. They are same as used with list , string and dictionary. See some example.

```
>>> t=tuple('abcd')
>>> 'e' in t
False
>>> 'a' in t
True
>>> 'e' not in t
True
```

```
>>> 'a' not in t
False
```

9.6.4 Tuple comparison

The relational operators can be used for comparison of two tuples. The comparison is done element wise . First element of each tuple is compared, if they differ than either one of the tuple is greater. If they are not then subsequent elements are compared. See some examples:

```
>>> t=tuple('abc')
>>> t1=tuple('abc')
>>> t2=tuple('abde')
>>> t1==t2
False
>>> t1>t2
False
>>> t2>t1
True
```

In the tuple **t1** and **t2** third element is differ and because ASCII value of **'d'** is more than **'c'** so **t2 >t1** turns out to be true. See one more example

```
>>> (3,5,6)<(6,9,10)
True
>>> (13,5,6)>=(6,19,10)
True
```

This comparison of tuple is quite useful in certain situations where sorting of element is to be performed. Let's understand it with the aid of an example. Assuming you have a dictionary where key elements are names of students (assuming unique names) and values are their marks. You want to sort the elements of dictionary on the basis of their marks, but marks are not key part , they are value part of dictionary. To do this we can have a list of tuples of two elements and while making list of tuples we put value as first element of each tuple.

```
>>> stu={'a':98,'b':97,'c':87,'d':99}
>>> L=list()
>>> for name,marks in stu.items():
...     L.append((marks,name))
...
>>> L
[(98, 'a'), (97, 'b'), (87, 'c'), (99, 'd')]
>>> L.sort(reverse=True)
>>> L
[(99, 'd'), (98, 'a'), (97, 'b'), (87, 'c')]
```

You can notice that each element of the list is a tuple and in each tuple the first element is marks and second element of each tuple is student name. The list is than sorted in descending order. The script version of the above is presented below with a minor modification:

```
    # Script 9.1 To generate merit list of students using dictionary and
#tuple
    stu={'a':98,'b':97,'c':87,'d':99}
    L=list()
    for name,marks in stu.items():
        L.append((marks,name))
    L.sort(reverse=True)
    print("Student Merit list")
    for i in range(0,len(L)):
        print(L[i][1],"\t",L[i][0])
```

OUTPUT:

```
Student Merit list
d   99
a   98
b   97
c   87
```

Because L has tuple as each of its element so first element of the tuple is accessed as L[0][0] and second element of tuple as L[0][1] (first index represent list element and second index tuple element).

One more example where this comparison of tuples is useful is arranging words of a string according to their length. For example : “This is an example”. If the preceding string is the input to our script or shell the output will be: [(7,’example’),(4,’This’),(2,’is’),(2,’an’].

```
    # Script 9.2 Sorting string using their length
    s='I just love working in python'
    L=s.split()
    L1=list()
    for word in L:
        L1.append((len(word),word))
    L1.sort(reverse=True)
    print("Sorted Words (by length)")
    for i in range(0,len(L1)):
        print(L1[i][1])
```

OUTPUT:

```
Sorted Words (by length)
working
python
love
just
in
I
```

9.5 Ponderable Points

1. A tuple is a sequence of objects which can be anything : integer, string, boolean, etc.
2. The list is mutable and tuple is immutable. Because of this tuple objects does not support item assignments.
3. A tuple with just one element is created as: t1=(10,)
4. Accessing tuple elements are like accessing list elements: using []
5. Deletion and modification of tuple elements are not permitted.
6. Operations using +, * and membership operators are allowed.

10. Modules in Python

10.1 Introduction

A Python module is a Python file itself where constants, functions, and classes are defined. It allows us to group related objects into a single python file.For example, all math-related functions can be easily placed in a module math or mymath. The name of the module is recognized by the python file itself. Python modules provides the idea of re-usability so that code once written in one python file can be used by importing that file as python module. Each module written is usually independent of each other and can be used within other module using import statement. Use of python modules make python programs and projects more manageable. We have seen usage of standard modules in number of codes in previous chapters.

In this chapter we are going to study module creation and loading them in other python files.

10.2 The first Python Module

The best to understand about module is to create one and use it. So lets get started !

Create a file in your current working directory by the name : mymath.py. You can use any of your favorite editor for writing the python code.

```
""" Example of module in Python-mymath module"""
pi=3.14
def area(r):
    """ input: radius r
        output: area of circle
    """
    global pi
    return pi*r*r
def perimeter(r):
    """ input: radius r
        output: perimeter of circle
        """
    global pi
    return 2*pi*r
```

In the above code written in “mymath.py” file we have first line as documentation for the module written in triple double quotes. The module is having one global variable pi and two functions to calculate area and perimeter of a circle. Both the function have their own doc string.

Once done , open python shell from the same working directory. In order to use the module we have just created we need to import it. Without importing you cannot access the variable and functions of a module. If you try you are going to get NameError exception as:

```
>>> area
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
NameError: name 'area' is not defined
```

(Same type of error will be shown for pi and perimeter). So let's import the module and use it:

```
>>> import mymath
>>> mymath.pi
3.14
>>> mymath.area(4)
50.24
>>> mymath.perimeter(4)
25.12
```

When you import module the python system searches for the file modulename.py (here mymath.py) and loads it. But you must prefix module name before any object of the module. You can see that after importing mymath module you can use the variable and functions using dot notation.

There is also an easier way where you don't have to prefix the module name before any module contents (variable,function,class).

```
>>> from mymath import area,perimeter
>>> area(2)
12.56
>>> perimeter(2)
12.56
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
```

As you can see from the above code you can import specific functions or any constant from the module using syntax: from modulename import functions,constants,.....

Here we imported area and perimeter from the module mymath so in calling them we do not need to prefix module name mymath. But as you can notice as didn't import pi from the mymath module using this syntax we cannot use pi without prefixing mymath module name. You have to use pi as mymath.pi. But if you want you also import pi along with functions as:

```
from mymath import area,perimeter,pi
```

Lastly if you want you can also see docstrings of both modules and functions:

```
>>> mymath.__doc__
' Example of module in Python-mymath module'
>>> print(area.__doc__)
```

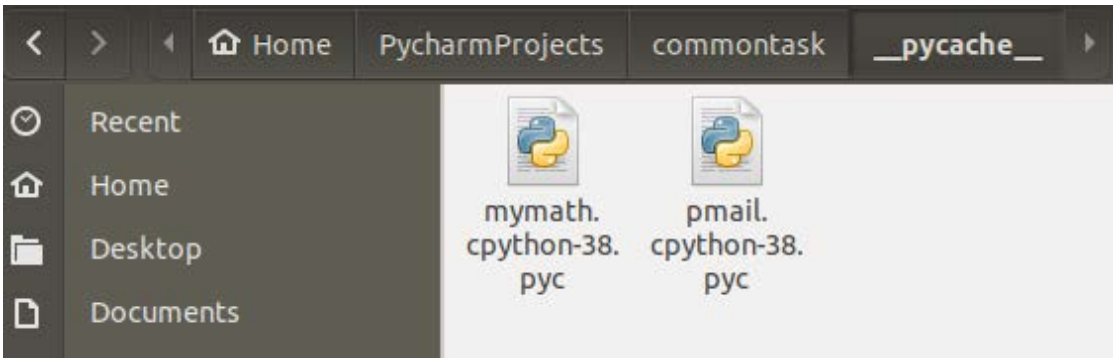
```
    input: radius r
           output: area of circle
>>> print(perimeter.__doc__)
    input: radius r
           output: perimeter of circle
>>>
```

10.3 Reloading the module

On times you want to do some changes to your module file and want to use again in your code. But there is an issue over here. After doing some changes in the file if you try to load it again using import statement it does not work. Let's add the following code to our mymath.py file

```
def max3(a,b,c):
    """ returns max of 3 numbers a,b,c"""
    return max3(max3(a,b),c)
```

Now if you import the module again as you have modified your module file it won't include the new function max3. The simple reason behind this is that when a module is first imported a byte code file having .pyc extension is created and used when functions/constants inside the module are used. When the module is changed or modified importing module second time does not change the byte code file. Only reloading the module can cause byte code file to be recreated or if you shut down the python shell and restart it again.



Shutting down the python shell and opening it again is a bit time consuming and you usually don't want when you changing your module frequently. The easy way to reload your module is using reload function that is part of importlib module.

```
>>> import importlib
>>> importlib.reload(mymath)
<module 'mymath' from '/home/drvikasthada/PycharmProjects/commontask/mymath.py'>
>>> mymath.max3(2,3,5)
5
```

10.4 Importing in another script

We saw simple example of creating a module and using it python shell. Lets create one more file in the current working directory to use this mymath module into another python script.

```
import mymath
r=float(input("Enter radius: "))
ar=mymath.area(r)
pr=mymath.perimeter(r)
print("Radius=",r)
print("Circle Area=",round(ar,2))
print("Circle Perimeter=",round(pr,2))
    OUTPUT:
    Enter radius: 2.3
    Radius= 2.3
    Circle Area= 16.61
    Circle Perimeter= 14.44
```

The code is self-explanatory.

10.5 Understanding import

We have seen examples of import in the previous section. We explained how to use import keyword for importing modules and from syntax for importing selective contents from module. Then a different section on import. Well in this section I want to show you something more about import that we did not discuss in the previous section.

For importing everything from mymath module we can use the following:

```
from mymath import *
```

The * notation imports everything from the module mymath except ones that start with underscore. To understand this lets take an example. Lets add a new function definition to mymath module along with new constant __MAX=1000:

```
def __myadd__(a,b):
    return a+b
__MAX=1000
```

You can either reload the module or start a new python shell. Now if you write

```
from mymath import *
```

Then above import does not import _myadd_ method and __MAX and gives you error:

```
>>> area(2)
12.56
>>> perimeter(2)
12.56
>>> _myadd_(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name '_myadd_' is not defined
>>> pi
3.14
>>> __MAX
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '__MAX' is not defined
```

Using from method also allows you to create alias. See an example:

```
>>> from mymath import perimeter as pr
>>> pr(2)
12.56
>>> from mymath import area as A,perimeter as pr
>>> A(2)
12.56
>>> pr(3)
18.84
```

In the first we create an alias pr for perimeter and in second example we create A as alias for area method and alias pr for perimeter. Alias work fine not just for functions, they also work for module also like:

```
>>> import mymath as mm
>>> mm.area(2)
12.56
```

10.6 The Search Path for Module

Ever wondered where python look for all the built-in modules ? This information is stored in the path variable that is part of the sys module. Lets see the same in my laptop:

```
>>> import sys
>>> sys.path
['',                    '/home/drvikasthada/anaconda3/lib/python38.zip',
'/home/drvikasthada/anaconda3/lib/python3.8',
'/home/drvikasthada/anaconda3/lib/python3.8/lib-dynload',
'/home/drvikasthada/.local/lib/python3.8/site-packages',
'/home/drvikasthada/anaconda3/lib/python3.8/site-packages']
```

As you can see sys.path returns a list containing all different paths where python look for all the built-in modules. Interesting to note is the first element which is empty string. This empty string represents current directory to look for any module files. The contents of sys.path is created during installation of python or anaconda.

Having understood the concept of sys.path, lets understand how can we place our module in search path so that our module can be searched not from current directory but from anywhere within the system. There are two ways to achieve this:

10.6.1 The PYTHONPATH variable

The PYTHONPATH environment variable is used to set the path for any directory where you want the system to look for the module files. We will demonstrate this using an example in ubuntu 18.04

We want that out module definitions stored in file mymath.py is available everywhere. The path to directory is : /home/drvikasthada/PycharmProjects/commontask. We have to open file .bashrc and make an entry into this file as shown below:

Once done, save the file and restart the shell. Now each time you use your own created modules then the module will be searched in the path as set in PYTHONPATH environment variable. To also check that this new path has been added to the path as returned by sys module see below:

```
>>> import sys
>>> sys.path
['',                    '/home/drvikasthada/PycharmProjects/commontask',
'/home/drvikasthada/anaconda3/lib/python38.zip',
'/home/drvikasthada/anaconda3/lib/python3.8',
'/home/drvikasthada/anaconda3/lib/python3.8/lib-dynload',
'/home/drvikasthada/.local/lib/python3.8/site-packages',
'/home/drvikasthada/anaconda3/lib/python3.8/site-packages']
```

Second element of this list is our PYTHONPATH. First element is emptystring.

10.6.2 Adding Path to sys.path

The second way of setting the module path is adding the path to the list as returned by sys.path. Support we have a path where our modules reside, and we want to have access to these modules from anywhere within the system. The easy but not so efficient way is to append that path to sys.path as shown below:

```
>>> import sys
>>> sys.path
['',                    '/home/drvikasthada/PycharmProjects/commontask',
'/home/drvikasthada/anaconda3/lib/python38.zip',
'/home/drvikasthada/anaconda3/lib/python3.8',
'/home/drvikasthada/anaconda3/lib/python3.8/lib-dynload',
'/home/drvikasthada/.local/lib/python3.8/site-packages',
'/home/drvikasthada/anaconda3/lib/python3.8/site-packages']
>>> sys.path.append('/home/drvikasthada/mymodules/')
>>> sys.path
```

```
   ['',                    '/home/drvikasthada/PycharmProjects/commontask',
'/home/drvikasthada/anaconda3/lib/python38.zip',
'/home/drvikasthada/anaconda3/lib/python3.8',
'/home/drvikasthada/anaconda3/lib/python3.8/lib-dynload',
'/home/drvikasthada/.local/lib/python3.8/site-packages',
'/home/drvikasthada/anaconda3/lib/python3.8/site-packages',
'/home/drvikasthada/mymodules/']
```

But the problem with this approach is that once you exit from shell and open it again the path vanishes. So effect of this approach is only temporary and first approach using PYTHONPATH is recommended.

10.7 The default module

The default module in python3 is buitlins. The module has number of functions that are commonly used by programmer to perform basic python scripting. As it is default module there is no need to import this module. See the contents of this module in python shell:

```
   >>> dir(builtins)
```

```
   ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError',   'BrokenPipeError',   'BufferError',   'BytesWarning',
'ChildProcessError',      'ConnectionAbortedError',      'ConnectionError',
'ConnectionRefusedError',   'ConnectionResetError',   'DeprecationWarning',
'EOFError',   'Ellipsis',   'EnvironmentError',   'Exception',   'False',
'FileExistsError',        'FileNotFoundError',        'FloatingPointError',
'FutureWarning',      'GeneratorExit',      'IOError',      'ImportError',
'ImportWarning',   'IndentationError',   'IndexError',   'InterruptedError',
'IsADirectoryError',   'KeyError',   'KeyboardInterrupt',   'LookupError',
'MemoryError',      'ModuleNotFoundError',      'NameError',      'None',
'NotADirectoryError',  'NotImplemented',  'NotImplementedError',  'OSError',
'OverflowError',      'PendingDeprecationWarning',      'PermissionError',
'ProcessLookupError',          'RecursionError',          'ReferenceError',
'ResourceWarning',  'RuntimeError',  'RuntimeWarning',  'StopAsyncIteration',
'StopIteration',     'SyntaxError',     'SyntaxWarning',     'SystemError',
'SystemExit',    'TabError',    'TimeoutError',    'True',    'TypeError',
'UnboundLocalError',      'UnicodeDecodeError',      'UnicodeEncodeError',
'UnicodeError',  'UnicodeTranslateError',  'UnicodeWarning',  'UserWarning',
'ValueError',   'Warning',   'ZeroDivisionError',   '_',   '__build_class__',
'__debug__',    '__doc__',    '__import__',    '__loader__',    '__name__',
'__package__',  '__spec__',  'abs',  'all',  'any',  'ascii',  'bin',  'bool',
'breakpoint',  'bytearray',  'bytes',  'callable',  'chr',  'classmethod',
'compile',  'complex',  'copyright',  'credits',  'delattr',  'dict',  'dir',
'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input',  'int',  'isinstance',  'issubclass',  'iter',  'len',  'license',
'list',  'locals',  'map',  'max',  'memoryview',  'min',  'next',  'object',
'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

In Python2 the name of default module was __builtins__. It has been changed to builtins in version 3. But inside directory function dir both __builtins__ and builtins works.

10.8 The main program

Every python script has a top-level script known as main program. The module name for the main program is always __main__ . Further __name__ global variable also has value as __main__ when the module is running as main program instead of being imported in some other module.

The code to check if the module is being used as the main program or not can be easily done by writing:

```
if __name__ == '__main__':
```

Lets understand this using a simple example:

```
# mymath.py
pi=3.14
def area(r):
    global pi
    return pi*r*r

def sum2(a,b):
    return a+b

a=10
b=20
r=2.0
print(f"Sum of {a} and {b} is {sum2(a,b)}")
print(f"Area of Circle with radius {r} is {area(r)}")
```

When you run the above python module (every python module is a script and executable) you will get the output as:

```
Sum of 10 and 20 is 30
Area of Circle with radius 2.0 is 12.56
```

Now lets say you want to use the module example.py in some other python file and use the function sum2 and area.

```
# newfile.py
import mymath
print("Area= ",mymath.area(3.0))
print("Sum= ",mymath.sum2(1,6))
OUTPUT:
Sum of 10 and 20 is 30
Area of Circle with radius 2.0 is 12.56
Area=  28.259999999999998
Sum=  7
```

The first two lines of output you did not expect. The output is because of import mymath line. When you want to use some code to be executed in a module when running as python script you must use the if __name__==”__main__”: line. Now modify the code in mymath.py as:

```
pi=3.14
def area(r):
    global pi
    return pi*r*r

def sum2(a,b):
    return a+b

if __name__=="__main__":
    a=10
    b=20
    r=2.0
    print(f"Sum of {a} and {b} is {sum2(a,b)}")
    print(f"Area of Circle with radius {r} is {area(r)}")
```

After adding the if block if you import mymath in other python script, the code within if block in mymath doesn't execute. It will only execute when you run mymath as python script. This is technique which you should always follows in any python script.

10.9 Ponderable Points

1. A Python module is a python file itself where constants, functions and classes are defined.
2. A module can be imported using import keyword followed by module name.
3. An alias for the module can be created as: import module as alias.
4. An example is : import numpy as np
5. Specific functions can be created from module as: from module import fun1,fun2.
6. An example is: from mymath import area, perimeter
7. The reload method of importlib can be used for reloading the module after module has been modified.
8. Example is: import importlib; importlib.reload(mymath).
9. The sys.path returns list of search path where various modules are functions are looked for access to your code.
10. The PYTHONPATH environment variable is used to set the path for any directory where you want the system to look for the module files.
11. The default module in python3 is buitlins
12. The code to check if the module is being used as the main program or not can be easily done by writing:
    if __name__ == '__main__':

11. Classes and Objects

11.1 Introduction

A class is the basic unit of encapsulation and abstraction. The class binds together data and methods which work on data. The class is an abstract data type (ADT) so creation of class simply creates a template. The data members of the class are called fields and member function are known as methods. As compare to other programming language it is very easy to create and use a class. Lets start by creating our first class.

```
class demo:
pass
```

Just two lines of object-oriented programming in Python. You might be surprised as where are data and methods of this class named as **demo**. Hold on, as Python does not force as to declare methods and function in class. We can create an empty class by just typing **pass** statement inside the class.

Now lets have some basic theory. The **class** is a keyword. Following this keyword class, **demo** represents name of the class. The class name must adhere rules of writing identifier as **class_name** is nothing but an identifier. It is recommended that classes should be named using CamelCase notation (start with a capital letter; any subsequent words should also start with a capital).The class is opened by using **:** and writing any code indented by 4 spaces or tab but not both. To make use of the class we need to create variable of type class. *A variable of class type is known as an object*. The class is loaded into memory when first object of the class is created.

Creation of object creates memory space for the object which depends upon size of the data members of the class. In python this is pure dynamic and no prior allocation is required. Further members of a class can be created dynamically. For each object separate copy of the data members is created. But only one copy of the member function is created which is shared by all the objects. The objects call member functions of the class using operator **.** which is known as period or membership operator. Functions and method example will be covered in a short while.

But how do we work with this class. Well its so easy!. Lets use this class in either Python shell or you can create a complete Python script. I'll show you in Python shell:

```
>>> class demo:
...     pass
...
>>> obj1=demo()
>>> obj2=demo()
>>> obj1
<__main__.demo object at 0x7f285523a3a0>
>>> obj2
<__main__.demo object at 0x7f28552fc040>
>>> obj1.name="Harsh"
>>> obj2.name="Koyal"
>>> obj2.age=21
```

Make sure you have **pass** statement indented by 4 spaces. Objects are created by treating class name as function as written as **demo()**. Just by writing demo() creates an object and same is stored in obj1 and obj2. On printing both the objects display their memory addresses. The **__main__** is default module name in the output. Note that an object can also be created without assigning to any object as shown:

```
>>> demo()
<__main__.demo object at 0x7f28552fc340>
>>> demo()
<__main__.demo object at 0x7f2853f1d280>
```

But each time you create an object in this manner a new memory address will be printed and but there is no way to track the location of objects because we have not assigned the newly created objects into any variables. This way of creating objects sometimes useful.

If you notice carefully we have created two members for object 2: name,age and just one member for object 1. This may seems strange to you if you have some prior experience of OOP in C++ and Java and objects from same class cannot have different members. But Python allow this !.

11.2 Adding members

Lets create a new class **Person** with data members and methods to fully understand the class in Python. This code we create in file **Person.py** using PyCharm Community Edition.

```
class Person:
    definput(self,name,age):
        self.name=name
        self.age=age
    defshow(self):
        print(f"Name={self.name}\tAge={self.age}")
defmain():
    p1=Person()
```

```
    p1.input("Naman",21)
    p2=Person()
    p2.input("Chhaya",19)
    p1.show()
    p2.show()
if__name__=="__main__":
    main()
OUTPUT:
Name=Naman    Age=21
Name=Chhaya   Age=19
```

Lets understand the code. We start with the methods created inside the class **Person**. A method is fancy name given to function in the context of OOP. It starts with the keyword **def** followed by a space and the name of the method. This is followed by a set of parentheses containing the parameter list separated by comma(we'll discuss that **self** parameter in a short while), and terminated with a colon. The next line is indented to contain the statements inside the method. The **self** argument to a method is simply a reference to the object that the method is being invoked on or reference to the current object (equivalent to **this** in C++ and Java). We can access attributes and methods of that object as if it were any another object. The data members are created dynamically using **self** and assigned values passed to method as done in **input** method. Remember ! **self** is not a keyword and any other name can be used. The data members of class **name** and **age** are always accessed using **self** parameter that must always be first parameter in any class method. When method **input** is called using **p1** object **"Naman"** and **21** is assigned to **name** and **age** parameters in method **input** and in method body they are assigned to **name** and **age** of object **p1**. The **self** here represent current object **p1** *that is passed automatically along with the two arguments*. Next when the same method is called by **p2**, **self** now represent **p2** in **input** method and **name** and **age** is for **p2** is set to **"Chhaya"** and **19** respectively.

Similarly when **show** method is called,reference of current object is passed to **self**.

The main method is not part of the class and is the standard practice when you want to run your code from the current module itself. That's why the **if** block checks if we name of the module is **__main__**.

Lets modify the script **Person.py** by removing everything except class code. Now you can easily check the functionality of the code inside Python shell. Run your code on command prompt as: **python -i Person.py**. The preceding code interprets the code in **Person.py** and start the shell:

```
(base) drvikasthada@MyLapi:~/PycharmProjects/commontask$ python -i Person.py
>>> p1=Person()
>>> p1.input("Naman",21)
>>> p1.show()
Name=Naman      Age=21
```

As you can see from the above figure, once script is interpreted correctly(shell will also be invoked if code has error)you can check class functionality inside the Python shell the way you have done inside the script.

11.3 Initializing Object

Most programming languages like C++ and Java have a concept of constructor that is used to create and initialize the object. Python has the concept of constructor but in practice it make use of **__init__** method(double underscore to either side of **init**) for initializing the object. The constructor method is known as **__new__** and accepts exactly one argument; the class that is being constructed (it is called before the object is constructed, so there is no self argument). In practice it is rarely used and **__init__** is mostly used for initializing the object so most of authors hide the notion of **__new__** and explain **__init__** as constructor in Python.

Lets see how we can initialize the object using **__init__** method. We replace the input method in **Person** class by **__init__** method as:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show(self):
        print(f"Name={self.name}\tAge={self.age}")
```

Once the class has initializer method the object can be created as:

```
p1=Person("Juhi",20)
```

Here we don't have to call the **__init__** method,it is called as we write **Person("Juhi",20)**. The **self** is passed secretly and arguments are collected in **name** and **age**. Inside the body of the initializer they are assigned to members of **p1**.

If you want to make it more compact, even **show** method can be called from inside the **init** method. Just add **self.show()** inside **init** method. Now when you create object of class **Person**,after initializing it will call **show** method and display the members.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.show()

    def show(self):
        print(f"Name={self.name}\tAge={self.age}")
```

```
Person("Navin",20)
```

11.3.1 Default Constructor

When you initialize objects using init method as explain in the previous section,you cannot objects without passing any arguments. That is if I create an object of Person class as: obj=Person() it will give you error as shown below:

```
(base) drvikasthada@MyLapi:~/PycharmProjects/commontask$ python -i Person.py
>>> obj=Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

In other programming languages you can overload the constructor (here init method)but in Python you can only have just one init method. Then how you can make a call like Person().

The solution is simple. Just set the parameters in init method as default. See an example.

```
class Person:
    def __init__(self, name="", age=0):
        self.name = name
        self.age = age

    def show(self):
        print(f"Name={self.name}\tAge={self.age}")
```

In the init method we have set name and age as default parameters. This allows us to create an object of Person class in many number of ways as:

```
Person()
Person("Mohit")
Person("Mohit",23)
Person(23) # will set name to 23 and age will be 0
Person(age=23) # name is "" and age is 23
Person(age=21,name="juhi")
```

See the code in execution:

```
(base)    drvikasthada@MyLapi:~/PycharmProjects/commontask$    python    -i
Person.py
>>> p1=Person()
>>> p1.show()
```

```
Name=   Age=0
>>> p2=Person("Mohit")
>>> p2.show()
Name=Mohit      Age=0
>>> p3=Person(23)
>>> p3.show()
Name=23 Age=0
>>> p3=Person(age=23,name="juhi")
>>> p3.show()
Name=juhi       Age=23
>>> p3=Person(age=23)
>>> p3.show()
Name=   Age=23
```

11.4 Passing and Returning objects to functions

Similar to returning and passing arguments to functions of basic types like int,char, double, float, string,list, tuple etc, we can pass objects of class to functions and even return objects from functions. See one small example first then we will make a bit bigger.

```
class Person:
    def __init__(self,name=""):
        self.name=name
    def copy(self,P):
        self.name=P.name
def main():
    p1=Person("Naman")
    p2=Person()
    p2.copy(p1)
    print(f"{p1.name},{p2.name}")
if __name__=="__main__":
    main()
```

The init method in class Person has just one member name. For object p1 this name is set to “Naman” and initially for p2 it is empty. The method copy takes one object of class Person type and assigns it to P. The call is by reference so if try to change the value of name member of p1 it is possible. As copy method was called by p2, self represents p2 in the copy method. The statement self.name=P.name copies name of p1 to name of p2.

Let’s see one more example where we pass and object and return an object.

```
class Person:
    def __init__(self,name="",sal=0):
        self.name=name
        self.sal=sal
    def compare(self,P):
        if self.sal > P.sal:
            return self
        else:
            return P
def main():
    p1=Person("Naman",25000)
    p2=Person("Bina",27000)
    p3=Person()
    p3=p1.compare(p2)
    print(f"{p3.name},{p3.sal}")
if __name__=="__main__":
    main()
```

Here the Person class has two members: name and sal. In the compare method we check which person's salary is higher. In statement p1.compare(p2), p1 calls the function and pass p2 as object. Inside function self is p1 and p2 is P. The salary of both the Person object is compared and that person is returned. In the example it is object p2. The returned object is assigned to p3 and its members are printed.

As a final example of passing and returning objects we work with complex numbers.

```
class Complex:
    def __init__(self,real=0,imag=0):
        self.real=real
        self.imag=imag
    def add(self,C):
        c1=Complex()
        c1.real=self.real+C.real
```

```
        c1.imag=self.imag+C.imag
        return c1

    def mul(self,C):
        r=self.real*C.real-self.imag*C.imag
        i=self.real*C.imag+self.imag*C.real
        return Complex(r,i)
    def __str__(self):
        return f"{self.real}+i{self.imag}"
def main():
    c1=Complex(2,3)
    c2=Complex(4,5)
    c3=c1.add(c2)
    c4=c1.mul(c2)
    print("Complex Number c1=",c1)
    print("Complex Number c2=",c2)
    print("Addition =",c3)
    print("Multiplication =", c4)
if __name__=="__main__":
    main()
```

Assume two complex numbers are (x1+jy1) and (x2+jy2) . Here j is called iota and j is $\sqrt{-1}$ so j * j will be -1. Now multiplication will be:

(x1 + j y1) * (x2 + j y2) = x1 x2 + j x1 y2 + j y1 x2 - y1 y2

= (x1 x2 - y1 y2) + j(x1 y2 + x2 y1)

For addition we can simply add the real part together and imaginary part together. In method add we pass an object of Complex class that is stored in C. Inside the body we find sum of real part of self and C and store in c1.real (c1 is temporary object created of Complex class). Similarly imaginary parts are added together. This object is then returned. In the case of multiplication the temporary object is not created. Here we store the real part and imaginary part (after calculation) in r and i. The last line creates an object of Complex class using this calculated r and i and returned.

One important method you can notice in this code is __str__ method. This method allows us to treat object as built-in data type and use in the print method for printing. You can note that we have simply printed all objects using print method without defining and show method. Whenever the __str__ method is defined in a class and that object is used for printing, __str__ method gets called automatically whenever the object is accessed. The __str__ method simply has to return a string.

If you want real feeling of complex numbers using arithmetic operators, instead of add and mul method you can override built-in methods __add__ and __mul__. See the modified code below (only changed portion)

```
def __add__(self,C):
    c1=Complex()
    c1.real=self.real+C.real
    c1.imag=self.imag+C.imag
    return c1
def __mul__(self,C):
    r=self.real*C.real-self.imag*C.imag
    i=self.real*C.imag+self.imag*C.real
    return Complex(r,i)
```

And in calling:

```
c3=c1+c2
c4=c1*c2
```

11.5 Array of objects

Similar to array of any basic data types we can create array of objects of any class. This comes handy when we want to process say salary of number of employees, processing accounts of persons, records of students or age of students etc. In all these situation array of objects makes our work easier and makes processing faster. Simple intuition is to create objects of a class and keep adding to an empty list. For example if Item is a class with item_name and price as its member fields then an array of objects of Item class can be created as:

```
ItemArray=[ ]

ItemArray.append(Item( “Mouse” ,560))
ItemArray.append(Item( “Pendrive” ,300))
ItemArray.append(Item( “Monitor” ,2500))
ItemArray.append(Item( “Keyboard” ,750))
```

The ItemArray above is now a list of objects where first object is at index 0 and last is at index 3. All the list operations can easily be applied to it. Lets see how can we create an array of objects using an example.

```
import operator
class Kid:
    def __init__(self, name="", age=0):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Name={self.name}\tAge={self.age}"

kids=[]
kids.append(Kid("chiku",8))
kids.append(Kid("nonu",12))
kids.append(Kid("tiku",9))
kids.append(Kid("chiya",11))
kids.append(Kid("chiky",6))

kids.sort(key=operator.attrgetter('age'),reverse=True)
for kid in kids:
    print(kid)

OUTPUT:
Name=nonu     Age=12
Name=chiya    Age=11
Name=tiku     Age=9
Name=chiku    Age=8
Name=chiky    Age=6
```

The Kid class has name and age has member fields. As explained above 5 different objects are added to the list kids. If you just want to display all objects just ignore sort method applied. In sort method we are sorting the objects on the field "age" and reverse=True means sorting in descending order. The attrgetter method of operator module does the job over here. The default sorting is in ascending order by removing reverse=True.

The other to perform sorting is using sorted method where key is assigned a function. Lets replace the line having sort method by :

kids=sorted(kids, key=lambda kid: kid.age)

The sorted method takes list as first argument and named parameter key as second argument. This named argument must be defined a function that returns atrribute on which sorting is to be done.

Lets see one more example of array of objects where we sort the geometric points based on their distance from origin.

```
from math import sqrt
```

```
class Point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y

    def dist(self):
      return sqrt(self.x*self.x+self.y*self.y)

    def __str__(self):
        return f"({self.x},{self.y})"

points=[]
points.append(Point(3,4))
points.append(Point(4,5))
points.append(Point(-2,-3))
points.append(Point(7,5))
points.append(Point(8,6))

points=sorted(points,key=lambda point: point.dist())

print("Sorted Points (nearer to origin first)")
for point in points:
    print(point)
```

The Point class has two data members: x and y that represents x and y coordinates of point. To find distance from origin we have simply written function dist that returns distance of the point from origin. The simple formula used is euclidean distance: $\sqrt{}(x2-x1)^2+(y2-y1)^2$ f which is distance between two points (x1,y1) and (x2,y2). One point is (0,0) because of origin. Rest of the code is simple to understand. The main code to understand is sorted function. The argument is points array and named argument key is lambda function. Note that sorting is done on distance, that's why dist function returns distance between current point and origin.

11.6 Static members in Class

We know that whenever an object is created separate copies of data members are created for each object. But in case of static data members only one copy of static data members is available which is shared among all the objects created. The static data member is member of class and therefore it is also known as class variable or class member.

Note several points about static data members:

1. They are created by placing variable inside the class and not within any method of the class.

2. There is one single copy of the static data member is created which is shared among all objects. Changes made by one object on a static data member are reflected back to all other objects.

3. They are used when you have to keep one value common to whole class.

4. They can be accessed either by object name or by class name

Lets understand this new concept with the help of a small program.

```
class demo:
    s=10
    def __init__(self,var):
        self.ns=var
        demo.s=demo.s+1

    def __str__(self):
        return f"Static var={demo.s}\tNon Static var={self.ns}"

d1=demo(10)
print(d1)
d2=demo(20)
print(d2)
d3=demo(30)
print(d3)
OUTPUT:
Static var=11 Non Static var=10
Static var=12 Non Static var=20
Static var=13 Non Static var=30
```

The variable s created as first statement inside the class demo and is a class variable or static variable. Inside the init method ns is non-static variable of object variable. The ns variable is accessed using self and s is accessed using classname(it can also be accessed using object). Each time init method is called this static variable s is increased by 1, thus keeping track of number of objects created. The output clearly shows that only a single copy of class variable is shared among all objects.

Lets have another example to understand more about class variables.

```
class University:
    pcount = 0
    def __init__(self, stream, pcount):
        self.stream = stream
        self.pcount = pcount
        University.pcount = University.pcount + pcount

    def __str__(self):
        return f"University Faculty={University.pcount}" \
               f"\t{self.stream} Faculty ={self.pcount}"

def main():
    engineering = University("Engineering", 36);
    print(engineering)
    biotech = University("BioTech", 21);
    print (biotech)

if __name__ == '__main__':
    main()
OUTPUT:
University Faculty=36       Engineering Faculty =36
University Faculty=57       BioTech Faculty =21
```

The pcount in University class represents counts of total faculties in various schools. The pcount as object variable represent count of faculties in individual schools. Here two different schools are taken: "Engineering" and "BioTech". Both the schools are represented by two different objects. For engineering object stream is set to "Engineering" and pcount is set to 36 during call to init method and University's pcount increased by 36. Similarly for biotech object stream is set to "BioTech" and pcount is set to 21 during call to init method and University's pcount increases by 21.

The class variable has many more ways to be accessed. In the above example we can write:

```
print (engineering.__class__.pcount)
print (type(biotech).pcount)
```

Both will print the pcount of University as engineering.__class__ and type(biotech) both will print: <class '__main__.University'>

Before wrapping up this section let's see one final example where we access the class variable using object.

```
class demo:
    svar = 10

print ("class variable using class=",demo.svar)
obj = demo()
print("class variable using object=",obj.svar)
obj.svar = 20
print("object variable using object=",obj.svar)
print("class variable using class=",demo.svar)
OUTPUT:
class variable using class= 10
class variable using object= 10
object variable using object= 20
class variable using class= 10
```

The class demo has just one class variable svar. First two print statements are easy to understand. One main point to understand is that when you create and initialize svar for the object obj, it is treated different despite having the same name. This is clarified by the last two print statements.

11.7 Static methods

Static methods in Python are bound to a class rather than the objects for that class. This means that you do not need any object to call a static. Further as there is no self argument is passed for static methods, static methods cannot modify the state of an object. This also implies that static method knows nothing about the class and just deals with the parameters if any. They can be called both by the class and its object.

There are two ways to create static methods in python:

- Using staticmethod()
- Using @staticmethod

We will see both the ways to understand creation of static methods.

Having said enough about static method, let's see our first example of static method in python using staticmethod().

```
class demo:
    def show():
        print("Hello from show")

# create show static method
demo.show = staticmethod(demo.show)
demo.show()
demo().show()
OUTPUT:
Hello from show
Hello from show
```

The function show() has no self parameter and its not an instance method. The staticmethod() function takes a function as input (function to be converted to static method) and return a static method. The last two lines calls the static method. First using class and second using object.

That was a very trivial example of static methods in python using staticmethod(). Lets take one more example where we have some arguments to the static method.

```
class demo:

    def add(a,b):
        return a+b

# create add static method
demo.add = staticmethod(demo.add)
print("Called using class=",demo.add(1,4))
print("Called using object=",demo().add(3,4))
OUTPUT:
Called using class= 5
Called using object= 7
```

Not much change from the previous example. We have just one static method that add two numbers.

Using @staticmethod

The staticmethod() approach is less popular as it require a staticmethod function to convert our internal class method to static method. The @staticmethod is a decorator (don't worry much about it at this time) and its use is much simple as compared to staticmethod() way. Lets rewrite the previous code using @staticmethod decorator.

```
class demo:
    @staticmethod
    def add(a,b):
```

```
        return a+b

print("Called using class=",demo.add(1,4))
print("Called using object=",demo().add(3,4))
OUTPUT:
Called using class= 5
Called using object= 7
```

As you can see in the code, the @staticmethod decorator must be placed just before the function definition and that's it. Much simpler that using staticmethod() way.

Why Static Methods

Static methods are useful when you want some utility or helper method having logic pertaining to class and not bound to any object. Let's take an example to understand this.

```
class Emp:
    @staticmethod
    def check_full_name(name):
        names = name.split(' ')
        return len(names) > 1

name1="Ravi Sharma"
name2="Ravi"
print(Emp.check_full_name(name1))
print(Emp.check_full_name(name2))
OUTPUT:
True
False
```

The static method check_full_name checks any name is full or just first name. It splits the input name on space and if the length of returned list is is more than 1 then name is full name else not a full name.

Quick Recap

Lets conclude this topic by revising about static method. When we need some functionality not w.r.t an Object but w.r.t the complete class, we can make a method static. This offers advantage in terms of creating helper methods which are not bind to any object. Finally, note that in a static method, we don't need the self to be passed as the first argument.

11.8 Class Method

The class method is similar to static method with the difference that it belongs to a class as whole and can modify the properties of class members. The parameter that is passed automatically when class method is called is popularly written as cls and this is uninstantiated class itself. It will be clear in example.

There are two methods to create class methods:

- Using classmethod()

- Using @classmethod

Lets see one example using classmethod().

```
class demo:
    count = 0
    def show(cls):
        print('The count is:', cls.count)

# create show class method
demo.show = classmethod(demo.show)
demo.show()
demo().show()
OUTPUT:
The count is: 0
The count is: 0
```

The show method in class is converted to class method using classmethod function and last two lines calls the show method: one using class and second using object. The difference from static method is that class method show is able to use class member count using cls argument that was not possible in static method.

That was a trivial example of classmethod. Lets see one more practical example, but this time using @classmethod decorator. It is similar to @staticclass decorator.

```
class Emp:

        def __init__(self,fname,lname):
          self.fname=fname
          self.lname=lname

        @classmethod
        def from_string(cls, name):
            first_name, last_name = map(str, name.split(' '))
            emp = cls(first_name, last_name)
            return emp

        def __str__(self):
            return f"{self.fname} {self.lname}"

e1 = Emp.from_string('Ravi Sharma')
e2=Emp("pari","singhal")
print(e1)
print(e2)
OUTPUT:
Ravi Sharma
pari Singhal
```

I'll concentrate mainly on class method from_string. The method takes two parameters:cls and name. This method is a kind of factory method for the production of objects of class Emp. That is instead of using init method for object creattion, we can use this method. The name argument method supplied to

this method is a fullname. Inside the method the name is split, firstname and last name is stored in local variables. Using these first_name and last_name and cls (represent Emp here) an object is constructed : cls(first_name,last_name) and returned as emp. Internally the line cls(first_name,last_name) is equivalent to Emp(first_name,last_name).

Outside the class the usage is shown as: e1 = Emp.from_string('Ravi Sharma'). The advantage of using factory method is that If we decide to rename this class at some point we won't have to remember updating the constructor name in a classmethod factory methods. Note internally this syntax make call to the **init** method and without init method in the class with required number of arguments, it won't work.

Quick Recap

The @classmethod decorator is used to create factory methods as any input can be passed to them and it can create objects based on the provided inputs. Using this decorator, it is possible to create as many constructors for a class which behaves as factory constructors. Finally classmethod takes a reference to class as cls that can be used to access class members also.

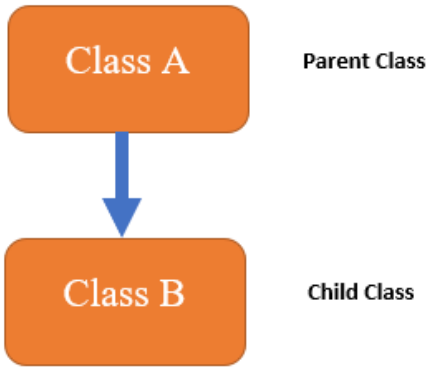
11.9 Ponderable Points

1. A class is the basic unit of encapsulation and abstraction. The c**lass** binds together data and methods which work on data.
2. An empty class in python can be created by just writing pass as body of the class.
3. *A variable of class type is known as an object.*
4. Creation of object creates memory space for the object which depends upon size of the data members of the class.
5. The **self** argument to a method is simply a reference to the object that the method is being invoked on or reference to the current object (equivalent to **this** in C++ and Java).
6. . Python has the concept of constructor but in practice it make use of **__init__** method(double underscore to either side of **init**) for initializing the object.
7. The init method (initializer) cannot be overloaded.
8. Objects can be passed to functions and can be returned also.
9. Like array of any basic data types, we can create array of objects of any class.
10. Only one copy of static data members is available which is shared among all the objects created.
11. Static methods in Python are bound to a class rather than the objects for that class.
12. there is no self argument is passed for static methods, static methods cannot modify the state of an object.
13. The class method is similar to static method with the difference that it belongs to a class as whole and can modify the properties of class members

12. Inheritance

12.1 Introduction

The term inheritance refers to the fact that one class can inherit part or all its structure and behavior from another class. Inheritance provides the idea of re-usability i.e. code once written can be used again & again in number of new classes. The class that does the inheriting is said to be a **subclass** of the class from which it inherits. If **class B** is a **subclass** of **class A**, we also say that **class A** is a **super class** of **class B**. (Sometimes the terms **derived** class and **base** class are used instead of **subclass** and **super class**.) A **subclass** can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and super class is sometimes shown by a diagram in which the subclass is shown below, and connected to, its super class.



In Python, when you create a new class, you can declare that it is a subclass of an existing class. If you are defining a class named **"B"** and you want it to be a subclass of a class named **"A",** you will be writing:

class B(A):

#functions and fields

12. 2 Types of Inheritance

In general inheritance is of 5 types.

1. Single level Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance

5. Hybrid Inheritance

The syntax of deriving a new class from an already existing class is as shown:

class newclassname(oldclassname):

#functions and fields

where **class** is the keyword used to create a class, **new_class_name** is the name of new derived class. **old_class_name** is the name of an already existing class. It may be a user defined or a built-in class.

12.2.1 Single level inheritance

In single level inheritance we have just one base class and one derived class. It is represented as:

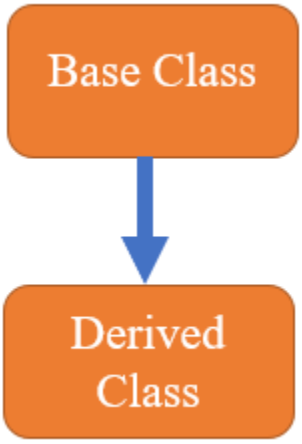
In Python code this can be written as:

class base:

#data members and functions

class derived(base):

#data members and functions



12.2.2 Multilevel inheritance

In multilevel inheritance we have one base class and one derived class at one level. At the next level the derived class becomes base class for the next derived class and so on. This is as shown below:

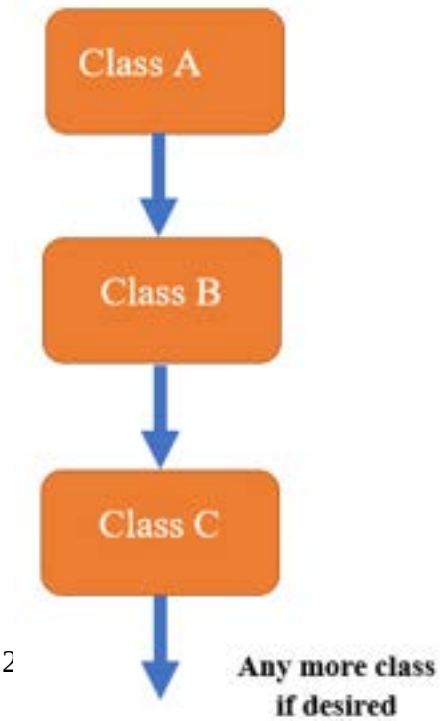
class A:

pass

class B(A):

pass

class C(B):



```
pass


class D(C):
pass
```

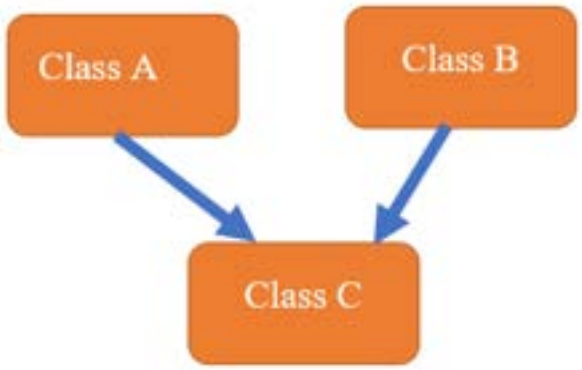
An example program is given above: The class A and class B together forms one level, class B and class C together forms another level and so on. For class B, class A is the parent and for class C, class B is the parent thus in this inheritance level we can say that A is the grandparent of class C and class C is the grandchild of class A.

12.2.3 Multiple Inheritance

In a multiple inheritance a child can have more than parent i.e. a child can inherit properties from more than one class. Diagrammatically this is as shown:

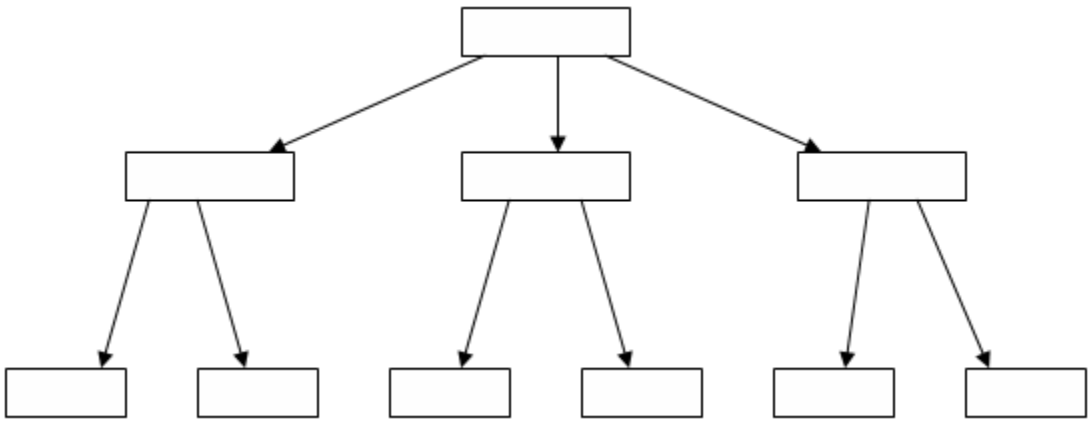
That is in Python you can write.

```
class A:
pass
class B:
pass
 class C(A,B):
pass
```



12.2.4 Hierarchical Inheritance

In this type of inheritance multiple classes share the same base class. That is number of classes inherits the properties of one common base class. The derived classes again may become base class for other classes. This is as shown:



For example a university has number of colleges under its affiliation. Each college may use the university name, the chairperson name, its address, phone number etc.

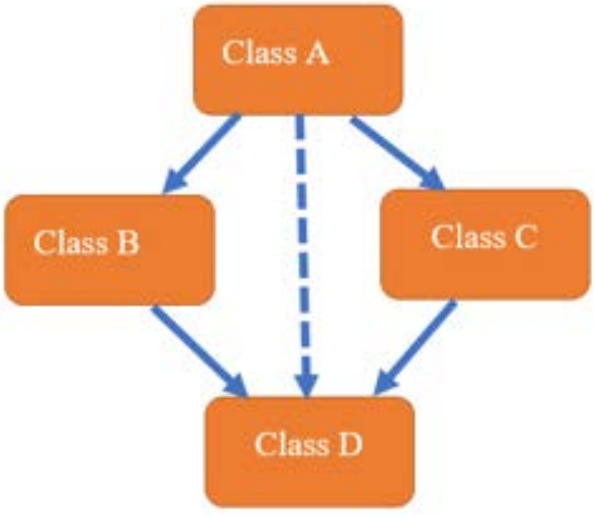
There are number of properties or features which a vehicle posses. The common properties of all the vehicle may be put under one class vehicle and different classes like two-wheeler, four-wheeler, three-wheeler can inherit the vehicle class.

As another example in an engineering college various department be termed as various classes which may have one parent class common, the name of engineering college. Again For each department there may be various classes like Lab_staff, Faculty class etc. In Python code the first level can be seen as follows:

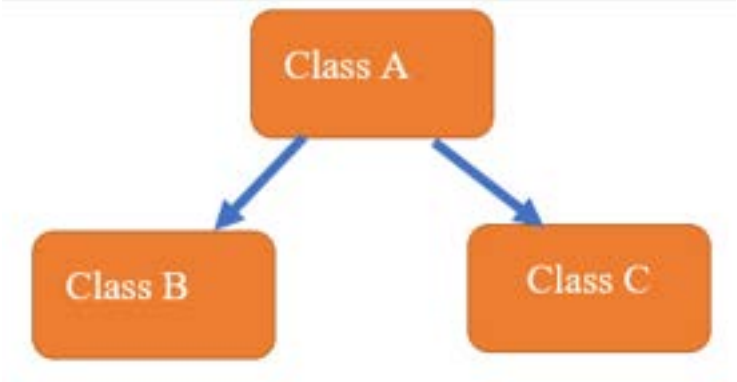
```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(A):
    pass
```

12.2.5 Hybrid Inheritance

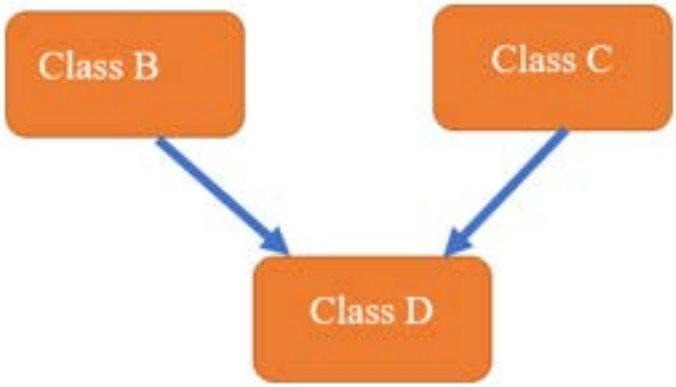
Consider the figure as shown below:



For the first half of the figure, we have hierarchical inheritance as shown by breaking the figure:

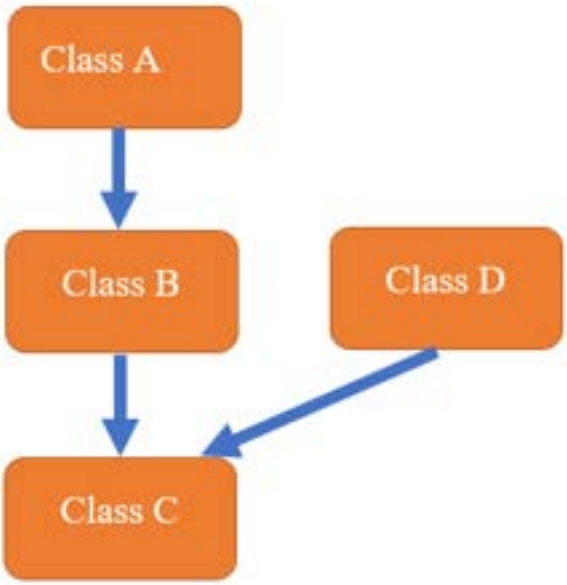


In the second half we have multiple inheritance as shown in the figure:



Combining them as shown in fig 1 we have hybrid inheritance.

The second figure for hybrid inheritance may be viewed as :



12.3 Single and Multilevel Inheritance in Python

The concept of single level and multilevel have been discussed earlier. In this section we see practical examples in python. Lets start with a very basic example of single level inheritance.

```
class A:
  def showA(self):
    print("Show from A")

class B(A):
  pass

if __name__=="__main__":
  obj=B()
    obj.showA()
```

OUTPUT:

```
Show from A
```

We have a class **A** in which a single function **showA** has been defined**. The line class B(A)** inherits class A into class B**.** That is it tells Python interpreter that **B** is a new class and we are inheriting class **A** in class **B**. This makes **A** as a parent of class **B**. The class **B** is also known as derived class and class **A** as base class. In the class **B** we have not defined any data or function, so it contains only inherited members from class **A** (in this case only function **showA**). In the **main** we create an object of class **B** and call the function **showA** which was inherited from class **A**. The if condition is as stated in previous chapter is only when this file is treated as for execution and not for importing.In case you don't like this if now it can be safely removed.

Lets modify the above code to add a member function in class B also.

```
class A:
    def showA(self):
        print("Show from A")

class B(A):
    def showB(self):
        print("Show from B")

if __name__=="__main__":
    obj1=B()
    obj1.showA()
    obj1.showB()
```

OUTPUT:

```
Show from A
Show from B
```

The class B now has two member function. One of its own (showB) and other is inherited from its parent(class A). Outside the class in if block we create an object of class B and call both the functions using this object. Note in case you want to create object of class A and call its method, you can. But in inheritance, object of usually derived class is created and methods are called using this object.

NOTE: If you have come from C++/Java background, you may be surprised to see that there is no notion of visibility modifier like: public/private/protected. Just plain inheritance.

In the previous two examples we had worked with just methods inside the class. Lets include some data members in both the class.

```
class A:
    def inputA(self,x):
        self.x=x

    def showA(self):
        print("From A x=",self.x)

class B(A):
    def inputB(self, y):
        self.y = y

    def showB(self):
        self.showA()
        print("From B y=", self.y)

if __name__=="__main__":
    obj=B()
    obj.inputA(10)
    obj.inputB(20)
    obj.showB()
```

OUTPUT:

```
From A x= 10
From B y= 20
```

Both the class A and B have one data members x and y respectively. In both the classes we have methods for taking input and displaying the data member. In the showB method we have called the showA method of class A because of inheritance. In the main we have just created one object of class B and called all required methods.

The showA can be removed if desired and showB can be modified as:

```
def showB(self):
    print("From A x=", self.x)
    print("From B y=", self.y)
```

The next example is a real-world example where from Area we find Volume of a room or any similar object.

```
class Area:
    def input1(self):
        self.l=int(input("Enter length: "))
        self.b=int(input("Enter breadth: "))

    def area(self):
        return self.l*self.b
```

```
class Volume(Area):

    def input2(self):
        self.input1()
        self.h = int(input("Enter height: "))

    def show(self):
        print("Area= ",self.area())
        print("Volume= ",self.area()*self.h)

if __name__=="__main__":
    V=Volume()
    V.input2()
    V.show()
```

OUTPUT:

```
Enter length: 12
Enter breadth: 10
Enter height: 5
Area=  120
Volume=  600
```

In the class **Area** we have two data members **l** and **b**. Function **input1** takes input from keyboard into these data items directly. The area function returns area of the rectangle. The class **Volume** inherits class **Area**. This class **Volume** contains one data member **h**. The class calculates area directly by calling the inherited area method. The volume is calculated by just multiplying area by height h. Note we have called only the derived class function **input2** inside the if block. This function in turn calls **input1** of the **Area** class and takes input from keyboard. The function **show** calculates area and volume and displays the same.

12.4 Multilevel Inheritance

Multilevel inheritance was discussed earlier in this chapter. Here we see some practical examples in python.

```
class A:
    def show1(self):
        print("show of A")

class B(A):
    def show2(self):
        print("show of B")

class C(B):
    def show3(self):
        self.show1()
        self.show2()
        print("show of C")
obj=C()
obj.show3()
```

OUTPUT:

```
show of A
show of B
show of C
```

Class A and B together make one level of inheritance as A is a parent class for B. Similarly class C and class B make one level of inheritance as class B is a parent class for class C. It makes class A as grandparent for class C. Through this two level of inheritance class C can have access to functions of class A and class B.

Last example of this type of inheritance is:

```
class Bird:
    def speak(self):
        print("Bird is Chirping")

class Cuckoo(Bird):
    def color(self):
        print("Cuckoo color is black")

class CuckooChild(Cuckoo):
    def home(self):
        print("In another bird's nest")
c = CuckooChild()
c.color()
c.speak()
c.home()
```

OUTPUT:

```
Cuckoo color is black
Bird Chirping
In another bird's nest
```

Class Bird is parent class for class Cuckoo and from Cuckoo class we derive one class CuckooChild. This creates two level of inheritance. The functions inside the class are self explanatory. An object of CuckooChild class have access to class Cuckoo and class Bird (indirectly).

12.5 Multiple Inheritance

Multiple inheritance has been explained in the beginning of the chapter. Recall In a multiple-inheritance graph, the derived classes may have several direct base classes to access the functionality of parent classes. In practice it is not recommended to use this type of inheritance because of problems associated with it that we will see.

Here we present few examples of multiple inheritance. Lets start with a very basic example:

```
class A:
    def show1(self):
        print("show of A")

class B:
    def show2(self):
        print("show of B")
```

```
class C(A,B):
    pass
obj=C()
obj.show1()
obj.show2()
```

OUTPUT:

show of A

show of B

The code is simple. Here we have two base classes: class A and class B. Both the classes have one member function each. The class C inherits both the classes A and B. As it has two parents the function show1 and show2 can easily be accessed by an object of class C. Outside the classes (in virtual main function) an object of class C is created and both the functions are called.

It was a trivial example of multiple inheritance. Lets change the above code a bit and keep both the function name same: show.

```
class A:
    def show(self):
        print("show of A")

class B:
    def show(self):
        print("show of B")

class C(A,B):
    pass
obj=C()
obj.show()
obj.show()
```

OUTPUT:

```
show of A
show of A
```

When both the parent classes has same function name, only one copy of the function from first inherited class (here A) is used in the class C. That's why the output. If you change the code as:

```
class C(B,A):
   pass
```

Then show function of class B is called. This is a kind of ambiguity in function call and that's why multiple inheritance is not recommended by many programmers. Further Java programming language doesn't support multiple inheritance through classes. One specific problem associated with this type of inheritance known as diamond problem will be covered later in this chapter.

Further if you have a function by the same name in class C (method overriding will be covered later) then show function of class C will be called and if we try to use the function show of class A or B, lets see what happens?

```
class A:
    def show(self):
        print("show of A")
```

```
class B:
    def show(self):
        print("show of B")

class C(B,A):
    def show(self):
        self.show()
        print("show of C")
obj=C()
obj.show()
   OUTPUT:

   (Some code not shown intentionally)

   RecursionError: maximum recursion depth exceeded
```

In the show function of class C, call self.show() calls to show function of C class and causes uncontrollable recursion. During recursion function addresses are stored in stacks and as this continues till the stack becomes full. Once stacks become full error is raised.

```
class base:
    def input1(self):
        self.b=int(input("Enter the base: "))

    def show1(self):
        print("base=" , self.b)

class exponent:
    def input2(self):
        self.e=int(input("Enter the exponent: "))

    def show2(self):
        print("exponent=",self.e)

class power(base,exponent):
    def input(self):
        self.input1()
        self.input2()

    def pow(self):
        po=1
        for i in range(1,self.e+1):
            po=po*self.b
        return po

    def show(self):
        self.show1()
        self.show2()
        print(self.b,"^",self.e,"= ",self.pow())

if __name__=="__main__":
```

```
    obj=power()
    obj.input()
    obj.show()
```

OUTPUT:

```
Enter the base: 3
Enter the exponent: 5
base= 3
exponent= 5
3 ^ 5 =  243
```

The code is simple. In the base class we have a data member b which represent base and in the class exponent we have a data member e which represent exponent. These two classes are inherited by class power which has two functions input and show. In the input of this class the two input functions of class base and exponent is called. In the pow function we find the power using for loop. This functions is used in show function along with show1 and show2 functions of class base and exponent respectively.

The other example which is our last example for multiple inheritance is where we have two classes for Internal and External Marks for the student. These two are combined to calculate final marks.

```
import sys
class Internal:
    def input1(self):
        self.i_marks=int(input("Enter the marks (max 30): "))
        if (not (self.i_marks >= 0 and self.i_marks <= 30)):
            print("Invalid Marks")
            sys.exit(0)

    def show1(self):
        print("Internal Marks=" , self.i_marks)


class External:
    def input2(self):
        self.e_marks = int(input("Enter the marks (max 70): "))
        if (not (self.e_marks >= 0 and self.e_marks <= 70)):
            print("Invalid Marks")
            sys.exit(0)

    def show2(self):
        print("External Marks=", self.e_marks)

class Final(Internal,External):
    def input(self):
        self.input1()
        self.input2()

    def total(self):
        return self.i_marks+self.e_marks

    def show(self):
```

```
        self.show1()
        self.show2()
        print("Final Marks= ",self.total())

if __name__=="__main__":
    obj=Final()
    obj.input()
    obj.show()
```

OUTPUT:

```
Enter the marks (max 30): 27
Enter the marks (max 70): 67
Internal Marks= 27
External Marks= 67
Final Marks=  94
```

For student's internal marks we have a class Internal and for student's external marks we have a class External. The internal marks must be between 0 and 30 and external marks must be between 0 and 70. In class Internal internal marks are stored in i_marks and in class External external marks are stored in e_marks.These two classes are inherited by class Final which finds the total marks and display all three marks: internal, external and total.

12.6 Hierarchical Inheritance

This type of inheritance was explained earlier in the chapter. When number of classes has a direct access to one common class, then this type of inheritance is visible. The main base class can be modified alone if required and all derived classes will see that effect. Here we present few examples of this type of inheritance.

Lets start with a very basic example where one class is a parent for three other classes:

```
class A:
    def showA(self):
        print("show of A")

class B(A):
    def showB(self):
        print("show of B")

class C(A):
    def showC(self):
        print("show of C")

class D(A):
    def showD(self):
        print("show of D")

if __name__=="__main__":
```

```
    print("From object of B class")
    obj1=B()
    obj1.showA()
    obj1.showB()
    print("From object of C class")
    obj1 = C()
    obj1.showA()
    obj1.showC()
    print("From object of D class")
    obj1 = D()
    obj1.showA()
    obj1.showD()
   OUTPUT:
From object of B class
show of A
show of B
From object of C class
show of A
show of C
From object of D class
show of A
show of D
```

This is a very trivial example where we have one class A that is parent of all other classes B,C,D. For this reason the function showA can be used by all other classes along with their own functions.

The other example we see is where number of colleges are affiliated to a single university. Thus single university will act as a parent to all affiliated colleges.

```
class University:
    uname="Rajasthan Technical University"

class college1(University):
    cname="SVC"
    def show_college1(cls):
        print("College Name=",cls.cname)
        print("Affiliated to=", cls.uname)

class college2(University):
    cname="LIET"
    def show_college2(cls):
        print("College Name=",cls.cname)
        print("Affiliated to=", cls.uname)

if __name__=="__main__":
    c1=college1()
    c1.show_college1()
    c2=college2()
    c2.show_college2()
   OUTPUT:
College Name= SVC
```

```
Affiliated to= Rajasthan Technical University
College Name= LIET
Affiliated to= Rajasthan Technical University
```

The program is so simple. We have University class which has just one class data member uname that stores name of the university . This class is inherited by two classes college1 and college2. The two classes display their name and the university to which they are affiliated.

12.7 Method Overriding

In method overriding a base class method is overridden in the derived class. That is the same method is written with the same signature as of the base class method but different implementation. Method overriding in often used in inheritance to override the base class implementation of the method and providing its own implementation. In method overloading arguments and type of arguments are used to distinguish between two functions, but in method overriding no such distinction can be made.In method overriding the signature of the two methods must match. This is shown in the program given below.

```
class A:
    def show(self):
        print("show of A")

class B(A):
    def show(self):
        print("show of B")

class C(A):
    pass

if __name__=="__main__":
    obj1 = B()
    obj1.show()
    obj1=C()
    obj1.show()
OUTPUT:
   show of B
   show of A
```

In the **class B** function **show** is overridden. The function **show** of class A is now hidden. From the main (in if block) when statement **obj1.show** executes it calls the **show** function of the **class B**. The class A is also inherited by class C but it does not overiride the show method of class A so when obj1.show() executes in last line it search for the method show in class C but it is not there so it look for the same in in its immediate parent class. It finds in class A and show method of class A gets called.

Lets have one more example with two functions in parent class and two derived class.

```
class A:
    def show(self):
        print("show of A")
    def display(self):
        print("display of A")
class B(A):
    def show(self):
```

```
        print("show of B")

class C(A):
    def display(self):
        print("display of C")

if __name__=="__main__":
    Lobjects =[B(),C()]
    for obj in Lobjects:
        obj.show()
        obj.display()
```

OUTPUT:

```
show of B
display of A
show of A
display of C
```

The class A has two methods : show and display. In the **class B** function **show** is overridden but display is not. The function **show** of class A is now hidden. In class C display function is overridden but show is not. In the main (in if block) we have a list of two objects of class B and class C. In the first iteration the obj represent object of class B and show and display method are called. The show function of class B and display function of class A is called. In the next iteration obj represent object of class C and show function of class A and display function of class C is called. The explanation of the same was given in previous code.

12.8 The super( ) method

In cases where you want to call the base class version of the function which is overridden in the derived class, you can use the **super( )** method in python. The **super()** actually returns a temporary object of base class object. Using this object you can call the methods of base class. For example consider the following code:

```
class A:
    def show(self):
        print("show of A")

class B(A):
    def show(self):
        super().show()
        print("show of B")

if __name__=="__main__":
        obj=B()
        obj.show()
```

In the above case inside **show** of class **B** if you simply write **show (as written)**, then it will call itself and recursion will follow(as discussed earlier). To call the base class version we can write **super( ).show** which means **show** of class **A** will be called. You can also use **super( )** to refer to any of the method or field of the base class or can even use it for calling constructor.

Lets see one more example with some data members in each class.

```
class A:
    def input1(self,num):
        self.num1=num

class B(A):
    def input2(self,num1,num2):
        super().input1(num1)
        self.num2=num2

    def show(self):
        print("num of class A=",self.num1)
        print("num of class B=",self.num2)
if __name__=="__main__":
    obj=B()
    obj.input2(10,20)
    obj.show()
```

OUTPUT:

```
num of class A= 10
num of class B= 20
```

In the class B the input2 function takes two integer parameters: num1 and num2. The parameter num1 is passed to input1 function of class A using super() and num2 assigned to num2 of class B. In the show we display both num1 of class A and num2 of class B.

More examples of super method will be shown in the next section.

12.9 Constructor (Initializer) & Inheritance

When constructors are present both in base and derived classes then how they are called, how values are passed from derived class to base class, that we will see in this section. We will also make use of super method in calling constructors of base class. Assume a small example of single level inheritance in which class **A** is inherited by class **B.** Both the classes have their default constructors. When an object of class **B** is created , it calls the constructor of class **B**, but as this class **B** has got **A** as its parent class, constructor of class **A** will be called first , then constructor of class **B** will be called. Why this is so? The reason is simple. When derived class has inherited base class, then obviously it will be using the data members from base class. Now without calling the constructor of base class, data members of base class will not have been initialized and if derived class uses the uninitialized data members of base class, unexpected results may follow. Calling a constructor of base class first allows base class to properly set up its data members so that they can be used by derived classes.

See few programs.

```
class A:
   def __init__(self):
       print("construtor of class A")

class B(A):
    def __init__(self):
        print("construtor of class B")

if __name__=="__main__":
```

```
        B()
```
OUTPUT:
```
constructor of class B
```

The init method act as constructor in python. Both the classes B and A have their own init method and class B has class A as its parent. When an object of class B is created in if block it calls only the constructor of class B and not of class A. To make a call to constructor of class A, we have to make use of super method as shown below in modified code.

```
class A:
    def __init__(self):
        print("construtor of class A")

class B(A):
    def __init__(self):
        super().__init__()
        print("construtor of class B")

if __name__=="__main__":
        B()
```
OUTPUT:
```
construtor of class A
construtor of class B
```

The code is easy to understand. We are just calling init method of parent class using super() method.

The real use of super method comes into picture when we want to use some inherited members in derived class and those members of parent has not set up. In that case we have to set them up using a call to init method in the init method of derived method with the help of super method. See an example below:

```
class A:
    def __init__(self,a):
        self.a=a

class B(A):
    def __init__(self,a,b):
        super().__init__(a)
        self.b=b
    def show(self):
        print("a=",self.a)
        print("b=", self.b)
        print("c=", self.a+self.b)

if __name__=="__main__":
        obj=B(10,20)
        obj.show()
```
OUTPUT:
```
a= 10
b= 20
```

```
c= 30
```

In the main we call the init method of class B using syntax: obj=B(10,20). Inside the constructor of B class, value a is passed into the init method of class A using super() and b is assigned to self.b. Once this initialization is done members a and b can be used inside the show method.

What if you try to use the data member a of class A without initializing it using a call super().__init__(a)? It will be give you error that B class has no data member a. See the code without show method.

```
class A:
   def __init__(self,a):
       self.a=a

class B(A):
    def __init__(self,a,b):
        self.b=b
        c=self.a+self.b
        super().__init__(a)
        print("c=",c)

if __name__=="__main__":
        obj=B(10,20)
```

OUTPUT:

```
AttributeError: 'B' object has no attribute 'a'
```

We are using data member a in second line of init method of class B without properly setting up using super() method. That's why the error.

Having understood basics of super method in calling the init method through inheritance. Lets see an example of super method in multilevel inheritance.

```
class A:
   def __init__(self):
       print("construtor of class A")

class B(A):
    def __init__(self):
        super().__init__()
        print("construtor of class B")

class C(B):
    def __init__(self):
        super().__init__()
        print("construtor of class C")

if __name__=="__main__":
       C()
```

OUTPUT:

```
construtor of class A
```

```
construtor of class B
construtor of class C
```

class A is inherited by class B and class B is inherited by class C. In both the init method of class B and class C, init method of their immediate parent class is called. That's why the output.

In the case of multiple inheritance, first inherited class's init method will be called first.

```
class A:
   def __init__(self):
       print("construtor of class A")

class B:
    def __init__(self):
        print("construtor of class B")

class C(A,B):
    def __init__(self):
        super().__init__()
        print("construtor of class C")

if __name__=="__main__":
       C()
```

OUTPUT:

```
construtor of class A
construtor of class C
```

When class C is created it inherits class A first then class B. So when an object of class C is created as C() in if block, it first calls its own init method and then using super it calls init method of class A. After printing it comes back and displays the content of print method.

As a final example of inheritance with super, lets take a practical example of rectangle and sqaure class.

```
class Rect:
    def __init__(self, l, w):
        self.l = l
        self.w = w

    def area(self):
        return self.l * self.w

class Sqr(Rect):
    def __init__(self, l):
        super().__init__(l, l)

if __name__=="__main__":
       r=Rect(10,20)
       s=Sqr(10)
       print("Area of Rectangle=",r.area())
```

```
        print("Area of Square=", s.area())
OUTPUT:
Area of Rectangle= 200
Area of Square= 100
```

The Rect class has two data members l and w that represents length and width and one method area that calculates area of the rectangle. The square is a special type of rectangle where length and width are same. Thus Sqr class inherits Rect class and make call to init method of Rect class using super(). In the main if block two objects each of Rect and Sqr is created and both calls the same method area. But for object r in calculating area l=10 and w=20 is used whereas for object s l=w=10 is used.

12.10 Abstract base class

An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists only to express the common properties of all its subclasses. An abstract class is a class that leaves one or more method implementations unspecified by declaring one or more methods abstract. An abstract method has no body i.e., no implementation (partially true) A subclass is required to override the abstract method and provide an implementation. Hence, an abstract class is incomplete and cannot be instantiated, but can be used as a base class.

An abstract class usually has one or more abstract methods. This is a method that is incomplete; it has only a declaration and no method body. Python provides module abc for abstract base classes. From this module we need to import ABC class and abstractmethod as decorator. Here is the syntax for an abstract method declaration in abstract class demo:

```
from abc import ABC,abstractmethod

class demo(ABC):
        @abstractmethod
        def show(self):
                pass
```

A class containing abstract methods is called an abstract class. If a class contains one or more abstract methods, the class itself must be qualified as abstract. (Otherwise, the interpreter gives you an error message.) Objects of an abstract class cannot be created so an abstract class must be inherited by some other class. The derived class must provide the implementation of all abstract methods declared in the abstract class. If the derived class does not provide implementation for the abstract functions, the it must be declared as abstract.

It's possible to create a class as abstract without including any abstract methods. This is useful when you've got a class in which it doesn't make sense to have any abstract methods, and yet you want to prevent any instances of that class.

If an abstract class is incomplete, what is the interpreter supposed to do when someone tries to make an object of that class? It cannot safely create an object of an abstract class, so you get an error message

from the compiler. This way, the interpreter ensures the purity of the abstract class, and you don't need to worry about misusing it.

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you may choose not to), then the derived class is also abstract,

It's possible to create a class as abstract without including any abstract methods. This is useful when you've got a class in which it doesn't make sense to have any abstract methods, and yet you want to prevent any instances of that class.

Let's see a small example program of abstract class.

```
from abc import  ABC, abstractmethod
class A(ABC):
    @abstractmethod
    def show(self):
        pass

class B(A):
    def show(self):
        print("Show of B")

class C(A):
    def show(self):
        print("Show of C")

if __name__=="__main__":
    B().show()
    C().show()
```

OUTPUT:

```
Show of B
Show of C
```

To make a class as abstract it must have class ABC as its parent class. An abstract class can have abstract or non abstract method or both. Here we have just one abstract method show in class A. The method is made abstract by using the decorator @abstractmethod. Once a method is become abstract it must be overridden by the derived class else error will be flashed. Both class B and C override the show method. In the main if block we just create dynamic objects of class B and C and call the show method of each class.

Lets modify the above code a bit so that C class does not override the show method.

```
from abc import  ABC, abstractmethod
class A(ABC):
    @abstractmethod
    def show(self):
        pass
```

```
class B(A):
    def show(self):
        print("Show of B")

class C(A):
    pass

if __name__=="__main__":
    B().show()
    C().show()
```

OUTPUT:

```
TypeError: Can't instantiate abstract class C with abstract methods show
```

As explained earlier the class C must override the show function inherited from class A but it is not done in class C and that's why error is flashed.

Lets modify the code again and this time we remove the decorator @abstractmethod in class A before show function.

```
from abc import  ABC, abstractmethod
class A(ABC):
    def show(self):
        print("Show of A")

class B(A):
    def show(self):
        print("Show of B")

class C(A):
    pass

if __name__=="__main__":
    B().show()
    C().show()
```

OUTPUT:

```
Show of B
Show of A
```

We stated earlier that an abstract class may or may not have abstract methods. Here the class A is abstract but it does not have any abstract method. It is now up to derived class as to whether override the show method or not. B class override the method and C class does not. So when B().show() call is made it calls the show method of class B and when C().show() call is made it calls the show method of class A. If you make show method is class A as empty by writing just pass, C.show() will call the method of A class but will not display anything.

Having understood concept of abstract class and abstract method, lets write a code where we have both abstract and non abstract method in parent class.

```
from abc import  ABC, abstractmethod
```

```
class A(ABC):
    def show(self):
        print("Show of A")

    @abstractmethod
    def display(self):
        print("display of A")

class B(A):
    def display(self):
        print("display of B")

class C(A):
    def display(self):
        print("display of C")

    def show(self):
        print("Show of C")

if __name__=="__main__":
    L=[B(),C()]
    for obj in L:
        obj.show()
        obj.display()
```

OUTPUT:

```
Show of A
display of B
Show of C
display of C
```

I leave it to you to figure out the output of the above code.

The above few python script must have given you clear idea of what is an abstract class and abstract method is and how to use them in python. But we have covered some real world scenario where abstract class and method would be useful. Let's write some python script code where this concept would be applicable.

```
from abc import  ABC, abstractmethod
class Insect(ABC):
    @abstractmethod
    def flystatus(self):
        pass

class Cockroach(Insect):
    def flystatus(self):
        print(self.__class__.__name__," can fly")

class Termite(Insect):
    def flystatus(self):
        print(self.__class__.__name__," cannot fly")
```

```
class Grasshopper(Insect):
    def flystatus(self):
        print(self.__class__.__name__," can fly")

class Ant(Insect):
    def flystatus(self):
        print(self.__class__.__name__," cannot fly")

if __name__=="__main__":
    L=[Cockroach(),Termite(),Grasshopper(),Ant()]
    for obj in L:
        obj.flystatus()
```

OUTPUT:

```
Cockroach  can fly
Termite  cannot fly
Grasshopper  can fly
Ant  cannot fly
```

The abstract class Insect declares one abstract function flystatus. Any class which inherits this class has to redefine this function and tell his/her flying status i.e. whether he/she can fly or not. This class Insect is inherited by 4 different classes viz. Cockroach, Termite, Grasshopper and Ant. Each class does provide implementation of function flystatus. In the main we create a list of objects of each of the derived class. Using for loop we call the flystatus method of each class. First iteration the object is of Cockroach class , next it is of Termite() and so on. Thus flystatus method of corresponding class in each iteration is called.

One final example of abstract class is also a real world example where we create one abstract class Figure. Each figure has to tell how many sides it has along with its area. Lets write the code:

```
from abc import ABC, abstractmethod

class Figure(ABC):
    def __init__(self, s1, s2):
        self.s1 = s1
        self.s2 = s2

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def tellsides(self):
        pass


class Rectangle(Figure):
    sides=4
    def __init__(self,s1,s2):
```

```
        super().__init__(s1,s2)

    def tellsides(self):
        return self.sides

    def area(self):
        return self.s1*self.s2

class Triangle(Figure):
    sides=3
    def __init__(self,s1,s2):
        super().__init__(s1,s2)

    def tellsides(self):
        return self.sides

    def area(self):
        return self.s1*self.s2*0.5

class Square(Figure):
    sides=3
    def __init__(self,s):
        super().__init__(s,s)

    def tellsides(self):
        return self.sides

    def area(self):
        return self.s1*self.s2

if __name__ == "__main__":
    L = [Rectangle(5,6),Triangle(5,8),Square(5)]
    for obj in L:
        print(obj.__class__.__name__," has ",obj.tellsides()," sides")
        print("Area of ",obj.__class__.__name__," is " ,obj.area())
```

OUTPUT:

```
Rectangle  has  4  sides
Area of  Rectangle  is  30
Triangle  has  3  sides
Area of  Triangle  is  20.0
Square  has  3  sides
Area of  Square  is  25
```

In the program we have an abstract base class **Figure**. The class has two data members **s1** and **s2** which stands for two sides of the figure. The class has two abstract methods **area** and **tellsides**. Each geometrical figure which inherits this class must calculate the area and have to tell the number of sides it has. The class also has constructor which takes two parameters **s1** and **s2 and assign to class members s1 and s2.**

The class is inherited by three classes **Rectangle** ,**Trinagle** and **Square** which provide the implementation of **area** and **tellsides**. All three uses a class member **sides** which represents number of sides a figure has.

In the **main** if block object of class **Rectangle, Triangle and Square** are created using constructors and dimension of sides is passed.All objects are placed as element of list L. In the call **Rectangle(5,6)** and **Triangle(5,8), Square(5)** constructor of **Figure** class is called and **s1** and **s2** gets their value. Each iteration of for loop takes one of these objects and calls their respective methods. The syntax: obj.__class__.__name__ has been used to find class name of the object in current iteration.

12.11 Visibility Modifiers in Python

Most object-oriented programming languages like C++,Java C# have a concept of access control through visibility modifiers: private, protected and public. Data members are usually marked as private meaning only the objects of the class can access them within the class. Protected members are available inside the class and in any of the derived classes. The public members are accessible everywhere. *But all this is not available in python. Python has no concept of visibility modifier or access control.*

But there are some python secrets which if others have no idea than the concept of private members may work in python. First lets see how can we make a member as private (not in true sense).

```
class demo:
    def __init__(self,a,b):
        self.__a=a
        self.b=b

d=demo(10,20)
print(d.__a)
```

OUTPUT:

```
AttributeError: 'demo' object has no attribute '__a'
```

Prefixing a class member by __ (double underscore) prevents it from accessing it outside the class. The member __a is now become secret. In Pycharm editor if you write d. it will show you a drop down box but will not show you __a member. It is only possible if you remember the private data member name is __a.

One more thing you must understand with respect to this double underscore notation is that it allow us to set the values for __a without error but it doesn't get reflected back. Confused ! don't worry. See the code below with output.

```
class demo:
    def __init__(self,a,b):
        self.__a=a
        self.b=b

    def show(self):
        print(self.__a,self.b)
d=demo(10,20)
```

```
d.__a=100
d.show()
```
OUTPUT:

```
10 20
```

As you can see even after we try to change the value of __a by writing d.__a=100, it actually does not change the value of __a. This is visible when we call show function to print the values of __a and b. Further no error is flashed.

Now we reveal the secret as to how the to get access to that hidden member __a.

```
class demo:
    def __init__(self,a,b):
        self.__a=a
        self.b=b
    def show(self):
        print(self.__a,self.b)
d=demo(10,20)
d._demo__a=100
d.show()
```
OUTPUT:

```
100 20
```

The syntax to use the hidden member __a outside the class is: d._demo__a (objectname._classname__attributename). That's it ! now you can have access to that hidden member.

In python terminology it is known as name mangling. If you use hidden member inside the method of class, the hidden members are unmangled and accessible but outside the class they are not unmangled by system interpreter and not visible. The programmer must unmangle them by clearing writing them in proper syntax. Thus __a can be called as name mangled variable.

In practice most of the programmers have not much to do with single and double underscore and they rarely make use of these features. In situations only when required use this feature of python otherwise you can safely ignore it.

12.12 Final class

A **final** class has the property that it cannot be inherited. When you say that an entire class is **final** you state that you don't want to inherit from this class or allow anyone else to do so. In other words, for some reason the design of your class is such that there is never a need to make any changes, or for safety or security reasons you don't want sub classing.

A simple way to make a class as final class using the module: final_class. The module can be installed using pip as: pip install final-class. Once installed it can be used as:

```
from final_class import final
@final
class demo:
    def show(self):
        print("Show of final class demo")
```

```
class derived(demo):
    pass
```

OUTPUT:

```
TypeError: Subclassing final classes is restricted
```

By placing @final decorator the class demo becomes final and it cannot be inherited. When you try to inherit it as in derived class error will be generated.

12.13 The Diamond Problem

Consider the situation where we have one class **A**. This class **A** is inherited by two other classes **B** and **C**. Both these classes are inherited in a new class **D**. This is as shown in figure given below. In dummy code form this is as shown below:

class A:

pass

class B(A):

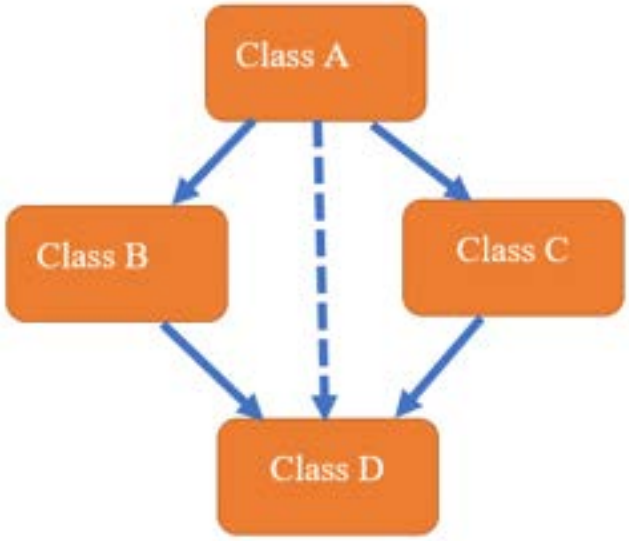
pass

class C(A):

pass

class D(B,C):

pass



This is called diamond inheritance because of the diamond shape of the class diagram. Why this is also called as diamond problem ? As can be seen from the figure that data members/functions of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data /function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called. One inherited through **B** or the other inherited through **C**. This may create problems in real life situations. For examples transaction can be called twice and money can be debited twice from the account. Lets understand this using a simple example:

```
class A:
    Acalls=0
    def show(self):
        print("A class called")
        self.Acalls+=1

class B(A):
    Bcalls=0
    def show(self):
```

```
        A.show(self)
        print("B class called")
        self.Bcalls+=1

class C(A):
    Ccalls=0
    def show(self):
        A.show(self)
        print("C class called")
        self.Ccalls+=1


class D(B,C):
    Dcalls=0
    def show(self):
        B.show(self)
        C.show(self)
        print("D class called")
        self.Dcalls+=1

if __name__=="__main__":
    d=D()
    d.show()
    print("Number of calls in each Class")
    print(d.Acalls,d.Bcalls,d.Ccalls,d.Dcalls)
```

OUTPUT:

```
A class called
B class called
A class called
C class called
D class called
Number of calls in each Class
2 1 1 1
```

Class A is inherited by both class B and class C. Both the classes are then inherited by class D. This creates a diamond inheritance. Each class has one class member to count how many times the function show has been called. The function show of class A is overridden by each of the derived classes. If you observe closely the output, you will see that show method of A class has been called twice: One by class B and another by class C. It is also visible from the count of calls of each class, The value of Acalls is 2 whereas all others show method call is 1.

To resolve the issue of this multiple calls in case of diamond inheritance, python provides next method and super method. Lets rewrite the code using super and see the output. Pay close attention to explanation that follows:

```
class A:
    Acalls=0
```

```
    def show(self):
        print("A class called")
        self.Acalls+=1

class B(A):
    Bcalls=0
    def show(self):
        super().show()
        print("B class called")
        self.Bcalls+=1

class C(A):
    Ccalls=0
    def show(self):
        super().show()
        print("C class called")
        self.Ccalls+=1


class D(B,C):
    Dcalls=0
    def show(self):
        super().show()
        print("D class called")
        self.Dcalls+=1

if __name__=="__main__":
    d=D()
    d.show()
    print("Number of calls in each Class")
    print(d.Acalls,d.Bcalls,d.Ccalls,d.Dcalls)
```

OUTPUT:

```
A class called
C class called
B class called
D class called
Number of calls in each Class
1 1 1 1
```

The d.show() calls show method of D class. The class has two parents B and C. Inside show method of D class super.show() make call to show method of B class. Now inside show method of B class first statement is super().show. This call is next method and calls show method of class C (even though C is not immediate parent of class B). Inside show method of class C super.show() calls show method of class A. Then print statement inside show method of class C executes. Control then returns to B class show method. The print statement within B class show method executes. Control then returns to D class show method and print statement inside D class executes.

Thus next method (internally) together with super method make sure then only copy of show method of A class called and problem of diamond inheritance is resolved.

Resolving the order of method is known as method resolution order. This can be achieved by calling method mro on class D as: D.mro(). The preceding call prints:

[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]

Here the last class is object class that is parent class of all classes in python.

12.14 Composition or Containership

We can create object of one class into another and that object will be a member of the class. This type of relationship between classes is known as **has_a** relationship or containership as one class contains object of another class. The inheritance which we have seen till now is considered **kind_of** or **is_a** relationship. Lets see an example of composition:

```
class A:
    def show(self):
        print("A class called")

class B:
    def __init__(self):
        self.objA=A()

b=B()
b.objA.show()
```

OUTPUT:

```
A class called
```

In the constructor of A class we have objA as member. This member is assigned an object of class B. This is composition in python. An object is used directly inside another class without inheritance. Note how the show method of A class was called.

As another example lets rewrite our Rect and Sqr program using composition.

```
class Rect:
    def __init__(self, l, w):
        self.l = l
        self.w = w

    def area(self):
        return self.l * self.w

class Sqr:
    def __init__(self, l):
        self.sr=Rect(l,l)
```

```
if __name__=="__main__":
        r=Rect(10,20)
        s=Sqr(10)
        print("Area of Rectangle=",r.area())
        print("Area of Square=", s.sr.area())
```

OUTPUT:

```
Area of Rectangle= 200
Area of Square= 100
```

The Sqr class constructor is now has sr its data member and that represents an object of Rect class. Note how the area method has been called: s.sr.area(). As sr is member of Sqr class and s is an object of Sqr class, we can call s.sr that represents an object of Rect class.

But what is the need of composition? In situations when you just must use the existing functionality of base class without any alteration, we can use composition. Otherwise, inheritance would be ideal choice.

12.15 Ponderable Points

1. Inheritance provides the idea of re-usability i.e., code once written can be used again & again in number of new classes.
2. There is no notion of visibility modifier like public, private, protected.
3. Python supports multiple inheritance where one class can have multiple parents.
4. In method overriding a base class method is overridden in the derived class. That is the same method is written with the same signature as of the base class method but different implementation.
5. In cases where you want to call the base class version of the function which is overridden in the derived class, you can use the **super( )** method in python
6. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses.
7. An abstract class usually has one or more abstract methods.
8. The ABC class and @abstractmethod from module abc are must for creating an abstract class.
9. Prefixing a class member by __ (double underscore) prevents it from accessing it outside the class.
10. A **final** class has the property that it cannot be inherited.
11. Multiple inheritance and hierarchical inheritance together can cause diamond problem.
12. There are two kinds of relationship: is_a and has_a.

13. EXCEPTION HANDLING

13.1 Introduction

In python errors can be divided into two categories: syntax errors and exceptions. Syntax errors (also known as parse errors) are errors that occurs during the writing of the program. Most common examples of syntax errors are missing comma, missing double or single quotes etc. They occur mainly due to poor understanding of language or writing program without proper concentration to the program.

There are logical errors which are mainly due to improper understanding of the program logic by the programmer. Logical errors cause the unexpected or unwanted output. Exceptions are runtime errors which a programmer usually does not expect.

They occurs accidentally which may result in abnormal termination of the program. Python provides exception handling mechanism which can be used to trap this exception and running programs smoothly after catching the exception.

Common examples of exceptions are division by zero, opening file which does not exist, insufficient memory, violating array bounds etc.

13.2 Basis for exception handling

A Program is correct if it accomplishes the task that it was designed to perform. It is robust if it can handle illegal inputs and other unexpected situations in a reasonable way. For example, consider a program that is designed to read some numbers from the user and then print the same numbers in sorted order. The program is correct if it works for any set of input numbers. It is robust if it can also deal with non-numeric input by, for example, printing an error message and ignoring the bad input. A non-robust program might crash or give nonsensical output in the same circumstance.

Getting a program to work under ideal circumstances is usually a lot easier than making the program robust. A robust program can survive unusual or “exceptional” circumstances without crashing.

One approach to writing robust programs is to anticipate the problems that might arise and to include tests in the program for each possible problem. For example, a program will crash if it tries to use an array element ARR[i], when i is not within the declared range of indices for the array ARR. A robust program must anticipate the possibility of a bad index and guard against it. This could be done with an if statement.

```
if (i < -len(L) or i>len(L)):
        print("invalid")
        # do something to handle
else:
    # process the list element
    print("valid")
```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It’s not always clear what to do when an error is detected.

Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straight forward program into a messy tangle of if statements.

Python provides a neater, more structured alternative method for dealing with errors that can occur while a program is running. The method is referred to as exception-handling. The word “exception” is meant to be more general than “error.” It includes any circumstance that arises as the program is executed which is meant to be treated as an exception to the normal flow of control of the program.

Exception handling is the process to handle the exception if generated by the program at run time. The aim of exception handling is to write a code which passes exception generated to a routine which can handle the exception and can take suitable action.

13.3 Exception Hierarchy

The exception hierarchy of exceptions in python is shown in the figure 1.

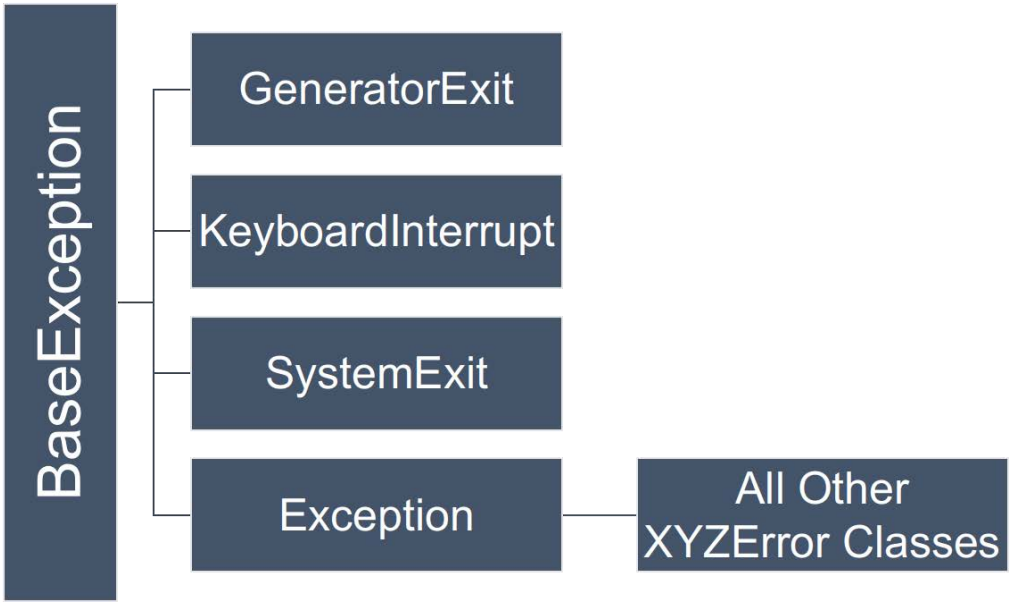


Figure 13.1: Exception hierarchy in Python

As you can see from figure, the root class of all exceptions is BaseException. The 4 classes that are derived from this class is GeneratorExit, KeyboardInterrupt, SystemExit and Exception. All the classes that we use for handling runtime errors are derived from the class Exception. Even user defined exception class has to make Exception class as its parent class. The reason why child classes of Exception class is not named as exception classes and as error classes it because of PEP8 (Python Enhancement Project) convention.

The 3 main classes except Exception are directly derived from BaseException as they present errors that are usually handled by the python system itself. For example SystemExit is used during a function call sys.exit to gracefully exit the python program. Similarly KeyboardInterrupt is caused when user presses Ctrl+C in command line programs. GeneratorExit is also used when there is no more element is there to generate. Some of derived classes from Exception classes are: LookupError, ZeroDivisionError, ArithmeticError, NameError, TypeError, SyntaxError etc.

The complete exception hierarchy would be too long to cover here. For details please see python documentation at this URL. BuiltIn Exceptions and Exception Hierarchy

13.4 Some Examples Of Exceptions

Having said a lot about exceptions, let's see some examples of exceptions in python without handling them in python shell. This is shown in table along with explanation.

Python code Examples with output	Exception Generated with reason
>>>a,b=10,0 >>> a/b Traceback (most recent call last): File "<stdin>", line 1, in <module> ZeroDivisionError: division by zero	ZeroDivisionError: division by zero Value of b is 0 and you cannot divide a number by 0.
>>> L=[1,2,3,4] >>> L[5] Traceback (most recent call last): File "<stdin>", line 1, in <module> IndexError: list index out of range	IndexError: list index out of range Possible index values for L is -4 to 4.
>>> x=int("23.45") Traceback (most recent call last): File "<stdin>", line 1, in <module> ValueError: invalid literal for int() with base 10: '23.45'	ValueError: invalid literal for int() with base 10: '23.45'. Trying to parse a float into integer.
y=x+10 Traceback (most recent call last): File "<stdin>", line 1,	NameError: name 'x' is not defined. Variable x is not initialized and does not exist.

in <module> NameError: name 'x' is not defined	
>>> '10'+20 Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: can only concatenate str (not "int") to str	TypeError: can only concatenate str (not "int") to str. You are trying to add two different kinds of literals: one is string and second is integer.

The table must have given you some feel of exceptions generated during execution of python code. We will see detailed python code with full fledged python programs in the coming sections. Lets move on now to understand the mechanism of exception handling.

13.5 Exception handling mechanism

In python exceptions are objects which are thrown. All such code which might throw an exception during the execution of the program must be placed in the try block. We have seen usage of try block in number of programs earlier. The exception thrown must be caught by the except block. If not caught your program may terminate abnormally. An exception can be thrown explicitly or implicitly. Exception can be explicitly using keyword throw. In explicit exception a method must tell what type of exception it might throw. This can be done by using throws keyword. All other exceptions thrown by the java run time system are known as implicit exceptions. A finally clause may be put after the try except block which executes before the method returns. A try block must have a corresponding except block, though it may have number of except blocks.

This try block is also known as exception generated block or guarded region. The except block is responsible for catching the exception thrown by the try block. It is also known as exception handler block. When try throw an exception, the control of the program passes to the except block and if argument matches, exception is caught. In case no exception is thrown the except block is ignored and control passes to the next statement after the except block.

The general syntax of exception-handling construct is shown as:

```
try:
# code that may generate exception
except ExceptionClass1 as e1:
# code to handle exception
except ExceptionClass2 as e2:
# code to handle exception
.
```

```
    .
    .
    .
    else:
    # (Optional)
       # executes when no exception is caught
    finally:
    # (Optional)
    # for some cleanup action
    # always executes
```

When you think that your code may throw some exception, you must put that code in try block. Depending upon code the exception objects thrown may be of different types. To except different types of exception objects there are different exception classes. These can be specified in except blocks which can be more than one. In case no exception is thrown the else optional block executes. The finally block is optional and it always executes regardless of whether an exception is thrown or not.

13.6 Python stack trace

A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown. Python prints a stack trace whenever your code throws an exception.

Lets see an example to understand stack trace in python.

```
def demo(index):
    L=[2,3,4]
    print(L[index])
demo(4)
```

OUTPUT:

```
Traceback (most recent call last):
  File "quickstart.py", line 4, in <module>
    demo(4)
  File "quickstart.py", line 3, in demo
    print(L[index])
IndexError: list index out of range
```

A stack trace report contains the function calls made in your code right before the error occurred. When your program raises an exception, it will print the stack trace. If you clearly the output from the code, it gives you lots of information. It tells you that when demo function was called with argument 4 and error was generated. Further it was because of line print(L[index]) and lastly it prints the actual exception thrown.

If you go from last line to top line, you will come to know that exception was generated because of line print(L[index]) and that because of function call demo(). Further that call was made in main function (default function).

Different types of stack traces will be generated depending upon what type of exceptions are generated in your code. Lastly stack traces are not visible when you handle the exceptions using try-except block.

13.7 Exception handling using try and except

In this section we see programmatically how exceptions are thrown within try block and handled by the except block. Let's start with our first program on exception handling.

```
# First Run of Script
try:
    L=[1,2,3,4]
    x=L[-6]
except IndexError as e:
    print("Exception caught=",e)
else:
    print("else block x=",x)
```

OUTPUT:

```
Exception caught= list index out of range
```

In second line of try block we are using index as -6 for list L which is out of bounds. Permissible range is from -4 to 4 as length of L is 4. So line x=L[-6] throws an exception of IndexError type and this exception is caught by except block in object 3. The same is displayed through print. The object is internally converted to string by either calling __str__ method or __repr__ method.

Let's change the line x=L[-6] to L[-2] (valid index) and see the output:

```
# First Run of Script
try:
    L=[1,2,3,4]
    x=L[-2]
except IndexError as e:
    print("Exception caught=",e)
else:
    print("else block x=",x)
```

OUTPUT:

```
else block x= 3
```

This time the index -2 is valid and second last element from list L is displayed in else block as else block executes only no exception is thrown.

But what if we don't use else block and write statement outside try-except block. Let's see what happens?

```
try:
    L=[1,2,3,4]
    x=L[-2]
except IndexError as e:
```

```
    print("Exception caught=",e)
print("else block x=",x)
```

OUTPUT:

```
else block x=3
```

The output remains same as we got in the previous script as no exception was thrown. But will the result remain same if exception is thrown? Let's see.

```
try:
    L=[1,2,3,4]
    x=L[-6]
except IndexError as e:
    print("Exception caught=",e)
print("else block x=",x)
```

OUTPUT:

```
Traceback (most recent call last):
  File "quickstart.py", line 6, in <module>
    print("else block x=",x)
NameError: name 'x' is not defined
Exception caught= list index out of range
```

Exception was caught by the except block but the line(x=L[-6]) because of exception was generated didn't execute, variable x was not defined. As x was not defined we get one more exception NameError. This exception was not handled so script crashed.

The morale of the story is that else block is always useful when exception is not thrown and should be used.

Further if you are only interested in catching the exception and ignoring the caught exception object, you can just write exception class name as: except IndexError. The rewritten except block is shown below:

```
except IndexError:
    print("Exception caught=")
```

Lets have some more examples of handling exceptions of different types.

```
try:
    x=input("Enter an integer number: ")
    x=int(x)
    s=x*10
except ValueError as e:
    print("Exception caught=",e)
else:
    print("else block s=",s)

print("out of try-except-else block")
```

```
OUTPUT:
(First Run Exception not generated)
Enter an integer number: 12
else block s= 120
out of try-except-else block
(Second Run Exception was thrown)
Enter an integer number: 2.3
Exception caught= invalid literal for int() with base 10: '2.3'
out of try-except-else block
```

In the try block we take an integer using input method and try to parse string integer to integer using int method. If input is integer, the code multiply the integer value x by 10 and store in s. In case int method cannot parse the input value into integer it will throw an exception of ValueError type. This exception will be caught by except block. The last statement is outside the try-except-else block and will always executes.

There are number of exceptions and all of them cannot be covered in this chapter as python programs. Instead of this, we explain them in tabular form with brief explanation.

Python code Examples with output	**Use**
`try:` `  d = {'a':10, 'b':20, 'c':30}` `  print (d['d'])` `except KeyError as e:` `  print ("KeyError Exception caught= ",e)` `else:` `  print ("else block no exception")`	KeyError exception is generated when a key in dictionary is not found. NOTE: The LookupError exception is base class for KeyError and IndexError and can be used for handling both types of exception.
`try:` `  print (x)` `except NameError as e:` `  print ("NameError Exception=",e)` `else:` `  print ("else block all ok")`	Name Error is generated when a local or global name is not found.
`try:` `  import xyz` `except ImportError as e:` `  print ("ImportError`	ImportError is generated when a module is not loaded or system having trouble to load the module.

Exception=",e) else: print ("else block all ok")	
try: x=10/0 except ArithmeticError as e: print("exception caught=",e) else: print("worked well")	ArithmeticError is the base class of errors: OverflowError, ZeroDivisionError, FloatingPointError.
try: with open("file.txt") as f: print(f.read()) except IOError as e: print("exception caught=",e) else: print("worked well")	IOError exception is generated when some error related to input/output happens. Here file does not exit so an exception is generated.

13.7.1 Try-Except with Multiple Exceptions

Lets see an example where our code can generate multiple exceptions and we handle these exceptions using either a single except block or multiple blocks. The general syntax of combining multiple exceptions in a single except block is:

```
except (Exceptionclass1,Exceptionclass2,....):
   # handle exception here
```

```
try:
    a=input("Enter first integer number: ")
    a=int(a)
    b=input("Enter second integer number: ")
    b=int(b)
    c=a/b
except (ValueError,ZeroDivisionError) as e:
    print("Exception caught=",e)
else:
    print("else block division= ",c)

print("out of try-except-else block")
```

```
OUTPUT:

(First Run)

Enter first integer number: 10
```

```
Enter second integer number: 5
else block division=  2.0
out of try-except-else block
(Second Run)
Enter first integer number: 10
Enter second integer number: 0
Exception caught= division by zero
out of try-except-else block
(Third Run)
Enter first integer number: 10
Enter second integer number: 2.5
Exception caught= invalid literal for int() with base 10: '2.5'
out of try-except-else block
```

In the program within the try block 2 different types of exception may be thrown: First exception of type ZeroDivisionError which may be generated when division by zero operation is performed. Second when instead of integer you input some other data type like float, double, string or even just press Enter; exception of type ValueError type is thrown. In the first run of the program we simply get the division of two number as output; No exception is generated and else block executes. In the second run when 0 is input for denominator, python run time system checks that division by zero operation is performed which is an illegal operation, so it construct an object of ZeroDivisionError type and throws it. This thrown exception is caught by the except block which displays the description of the exception thrown. In the third run of the program we intentionally input float input. As the int method cannot parse a float string into integer an exception object of ValueError class is generated and thrown. The thrown object is caught by the except block. Note in both run after handling the exception by the except block, statements after the try-except block executes as if nothing has happened.

In case you want single except block can be split into multiple except blocks as shown into following code snippet.

```
except ZeroDivisionError as e:
    print("Division Error Exception caught=",e)
except ValueError as e:
    print("Value Error Exception caught=",e)
```

13.7.2 Catching Exceptions with Empty Except

We saw earlier that to except exceptions we have to mention exception classes after the except keyword. This is one way of catching exceptions. In case you only want to except exception without worrying what type of exception will be thrown, then there are two different ways. One is covered in this section and another will be covered in next section. Lets understand this by slightly modifying the example that we saw in the previous section.

```
import sys
try:
    a=input("Enter first integer number: ")
    a=int(a)
    b=input("Enter second integer number: ")
    b=int(b)
    c=a/b
except:
    print("Exception caught of type=",sys.exc_info()[0])
else:
    print("else block division= ",c)
```

OUTPUT:

```
(First Run)
Enter first integer number: 10
Enter second integer number: 2.3
Exception caught= <class 'ValueError'>
(Second Run)
Enter first integer number: 10
Enter second integer number: 0
Exception caught of type= <class 'ZeroDivisionError'>
```

The different from the previous code is clearly visible. We have imported sys module for the function exc_info. The exc_info function of sys module returns a tuple containing three pieces of information: class name, classname with reason and traceback object at given address. This is shown below for the current example:

(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero'), <traceback object at 0x7f36daf22c40>).

In the code we have used the element only at index 0 to show type of exception thrown. For other type of exception thrown output is different as shown in first run of the code. The advantage of this type of exception handling is that you don't have to worry about what kind of exception may be thrown by your code.

13.7.3 Catching all exceptions using exception class

The other way of handling all types of exception is using parent classes of all exceptions to handle any exception thrown. The code is shown below:

```
try:
    a=input("Enter first integer number: ")
    a=int(a)
    b=input("Enter second integer number: ")
    b=int(b)
    c=a/b
except Exception as e:
    print("Exception caught of type=",e)
```

```
else:
    print("else block division= ",c)
```

OUTPUT:

```
(First Run)
Enter first integer number: 10
Enter second integer number: 3.4
Exception caught of type= invalid literal for int() with base 10: '3.4'
(Second Run)
Enter first integer number: 10
Enter second integer number: 0
Exception caught of type= division by zero
```

The code is self-explanatory. In case you want to display the class of exception caught just replace e by e.__class__ in except block.

13.7.4 Raising Exception

In all the programs of exception handling seen earlier the python run time system was responsible for throwing the exceptions. All those exceptions come under the category of implicit exceptions. If we want we can throw exceptions manually or explicitly. For that python provides the keyword raise. The raise keyword can be used to throw an exception explicitly. This is also known as raising an exception. The raise statement requires a single argument: an object of any exception class. Lets see a simple example:

```
try:
    raise ValueError("Demo of raising exception")
except Exception as e:
    print("Exception caught=",e)
```

OUTPUT:

```
Exception caught= Demo of raising exception
```

In the try block we are raising an exception explicitly. The exception object is of ValueError type and it has a string message. This exception is caught by except block in e and is display through print. The message passed to ValueError constructor is displayed when we print the caught object in except block using e.

The constructor can be empty and can be an exception of any type python support. Lets see another example:

```
try:
    raise LookupError()
except LookupError:
    print("Exception caught")
```

OUTPUT:

```
Exception caught
```

Raising exception is useful when you create your own exceptions and use them to throw explicitly in programming situations. One such situation would be to check stack is either full or empty.

```
top=-1
def pop():
    global top
    if (top == -1):
        raise EmptyStackException()
    else:
        obj = stack[top]
        top=top-1
        return obj;
```

The above is a tiny code snippet (not complete example). When top is -1 we raise an EmptyStackException. This exception class is already created as our exception class. Note there is no try and except block shown in the example. As the exception will be raised inside the function. The function call must be placed inside the try block. Exceptions inside functions are covered next.

As a final example we raise different types of exception inside a for loop.

```
for i in range(1,4):
    try:
        if(i==1):
            raise ArithmeticError("Arithmetic error exception raised")
        elif(i==2):
            raise RuntimeError("Runtime error exception raised")
        elif(i==3):
            raise NameError("Name error exception raised")

    except Exception as e:
        print("Exception raised=",e.__class__)
```

OUTPUT:

```
Exception raised= <class 'ArithmeticError'>
Exception raised= <class 'RuntimeError'>
Exception raised= <class 'NameError'>
```

The code is self-explanatory. In each iteration of for loop a different type of exception is raised. For each thrown exception we have just one class to except exception: Exception.

13.7.5 Nesting Of Try Except Block

Just like the multiple except blocks, we can also have multiple try blocks. These try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-except block within another try-block. The program structure for nested try statement is:

```
try:
    # outer try
    pass
    # statements
    try:
        # inner try
```

```
        pass
        # statements
    except ExceptionClass1 as e:
        pass
        # statements
except ExceptionClass2 as e:
    pass
    # statements
```

Here outer try block may have some exception to be thrown. That is caught by the outer block. After processing in outer try block, some codes are to be processed inside the try block. These code may also throw some exception and will be handled by inner try. That is if outer try catches exception early no need to process the further statements that may also cause some exception to be thrown. Lets clarify this using an example.

```
try:
    a = input("Enter first integer number: ")
    a = int(a)
    b = input("Enter second integer number: ")
    b = int(b)
    try:
        c = a / b
        print("Division=",c)
    except ZeroDivisionError as e:
        print("Exception caught=", e)
except ValueError as e:
    print("Exception caught=",e)
```

OUTPUT:

```
(First Run)
Enter first integer number: 10
Enter second integer number: 0
Exception caught= division by zero
(Second Run)
Enter first integer number: 10
Enter second integer number: 2.3
Exception caught= invalid literal for int() with base 10: '2.3'
```

In the program we are accepting two numbers from the user. After that, the input which are in the string format, are converted to integers. If the numbers were not received properly in a number format, then during the conversion a ValueError exception is raised otherwise the control goes to the next try block. Inside this second try-except block the first number is divided by the second number, and during the calculation if there is any division error, it is caught by the inner except block

13.8 The Finally Block

There is often some piece of code that you want to execute whether or not an exception is thrown within a try block. This usually pertains to some operation that is often referred to cleaning operation.

Like flushing memory buffers, closing files etc. To achieve this effect, you use a finally clause at the end of all the exception handlers. The syntax of try-except-finally is presented here once again :

```
try:
pass
except Exception1:
        pass
except Exception2:
        pass
.
.
.
finally:
        pass
```

The finally clause similar to except and try is a block of code by the name finally. The finally executes all the time irrespective of whether an exception is thrown or not; Even if an exception is thrown and handled or not handled finally will execute. The finally block is guaranteed to execute after the try-except block. The finally clause is an optional. Its usage is up to the programmer. If you have placed try-except block inside a method, then before returning from the method either maturely or prematurely finally clause will execute. The finally clause is necessary in some kind of cleanup like an open file or network connection, something you've drawn on the screen etc.

Lets see some examples of finally clause:

```
try:
    x=[1,2,3][5]
except Exception as e:
    print("Exception caught=",e)
finally:
    print("finally executes")
```

OUTPUT:

```
Exception caught= list index out of range
finally executes
```

The code is self explanatory. An exception is thrown as we are try to use index 5 for a list having just three elements. At the last finally block also executes.

Lets change the code so that exception is not thrown.

```
try:
    x=[1,2,3][2]
except Exception as e:
    print("Exception caught=",e)
finally:
    print("finally executes")
```

OUTPUT:

```
    finally executes
```

As said earlier, finally always executes even if the exception is not thrown. So the output. When the exception is not thrown and code is static, do we need except block? Lets remove except block.

```
try:
    x=[1,2,3][2]
finally:
    print("finally executes")
```

OUTPUT:

```
finally executes
```

It worked ! . try-finally together works without except block. But what if an exception is thrown in the above code. Pretty obvious, it will not be caught, and program is going to crash but finally will execute. Try yourself.

Lets see an example now where we use try and finally within a function.

```
def fun():
    try:
        return
    finally:
        print("finally executes")
fun()
```

OUTPUT:

```
finally executes
```

In the function fun we have written only the try and finally block and not except block. In the try block we simply returned from the function. But note before returning from the function the finally block executes.

13.9 Creating Your Own Exceptions

To create your own exceptions, you will need to make Exception class as the base class of the class which you are going to create. As Exception class will be the parent class, all methods and attributes of Exception class can be used inside this newly created exception class. Lets see few examples:

Our first example is a basic one:

```
class MyException(Exception):
    def __init__(self,message):
        super().__init__(message)

try:
    raise MyException("Demo of user defined exception")
except MyException as e:
    print("exception caught=",e.__class__)
```

OUTPUT:

```
exception caught= <class '__main__.MyException'>
```

Class MyException is our new exception class that is derived from Exception class. Now we can raise exception of this class this with a message. For that in the init method we pass a message string and

same we pass to the parent class's init method (Exception) using super() method. In the main (default is main method) we raise an exception of this class and except method catches it.

See its so simple to create a user defined exception and raise it.

Lets see another example (not a basic one) where we check sign of a number and if its negative we throw a user defined exception.

```
class MyException(Exception):
    def __init__(self,num):
        self.num=num
    def __str__(self):
        return "MyException object thrown with num="+str(self.num)

def check(x):
    print("check called with x=",x)
    if(x<0):
        raise MyException(x)
    print("Returning from function check")

try:
    check(10)
    check(-10)
except MyException as e:
    print("exception caught=",e)
```

OUTPUT:

```
check called with x= 10
Returning from function check
check called with x= -10
exception caught= MyException object thrown with num=-10
```

The MyException class has one int argument constructor which initializes the num. The function has __str__ method overridden which is called automatically when displaying an object of MyException class to convert it into String form. In the main when method check is called with negative argument exception of MyException is thrown. This thrown exception is caught by the except block in e which is displayed. As object cannot be displayed in this manner, but as we have __str__ method defined in the MyException class , this method is called and string "MyException Thrown with num=-10" is returned.

Final example in this section is number guessing game where a number will be guessed by user and on wrong guess we throw exceptions.

```
class SmallNumber(Exception):
    def __init__(self, message):
        super().__init__(message)

class LargeNumber(Exception):
    def __init__(self, message):
        super().__init__(message)

num=9
while True:
```

```
    try:
        n = int(input("Enter a guess(1 to 20): "))
        if n < num:
            raise SmallNumber("number is small, Try again")
        elif n > num:
            raise LargeNumber("number is large, Try again")
        break
    except SmallNumber as e:
        print(e)
    except LargeNumber as e:
        print(e)

print("Congrats! You guessed it right.")
```

OUTPUT:

```
Enter a guess(1 to 20): 12
number is large, Try again
Enter a guess(1 to 20): 6
number is small, Try again
Enter a guess(1 to 20): 10
number is large, Try again
Enter a guess(1 to 20): 9
Congrats! You guessed it right.
```

There are two classes that act as user defined exception classes: SmallNumber and LargeNumber. When the guessed number is smaller than actual number an exception of SmallNumber type will be thrown. In case the guessed number is larger than the actual number an exception of LargeNumber type will be thrown.

In the code the actual number is stored in num and is 9. Using while loop we keep asking user to guess the number and give hint in the form of thrown exceptions by displaying messages.

13.10 Ponderable Points

1. Syntax errors (also known as parse errors) are errors that occurs during the writing of the program.
2. Exceptions are runtime errors which a programmer usually does not expect.
3. The root class of all exceptions is BaseException.
4. In python exceptions are objects which are thrown.
5. The try block is also known as exception generated block or guarded region.
6. The except block is responsible for catching the exception thrown by the try block.
7. The finally block is optional and it always executes regardless of whether an exception is thrown or not.
8. A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred.
9. The Exception class can catch all types of exceptions.
10. Raising an exception using raise keyword is an example of explicit exception.

11. To create your own exception, you'll have to make Exception class as your base class.

14. FILE HANDLING

14.1 Introduction

We all know that RAM is a volatile memory and any data stored in RAM is lost when PC is turned off. All the programs we have seen so far have made use of RAM. Any data variable that we define in our program is destroyed when the program execution is over. Also, the outputs generated by the program are lost.

One solution may be to take printouts of the program and outputs. They may help up to a certain extent but that is not appropriate for the practical purpose. Therefore in most real word applications data is stored in text files which is stored permanently on to the hard disk , floppy , Compact Disk or in any other persistent storage media. These files can be read back again, and can be modified also.

A data file is a collection of data items stored permanently in persistent storage area. The python language provides the facility to create these data files, write data into them, read back data, modify them and many more operations. The program data or output can be stored in these files and that persists even after program execution is over. The data can be read whenever necessary and can be placed back into the file after modification. The data remain safe provided storage media does not crash or corrupt.

From permanent storage point of view, a file is a region of memory space in the persistent storage media and it can be accessed using the built-in library functions available python moduels or by the system calls of the Operating Systems. **High level files** are those files which are accessed and manipulated using standard library functions. For transfer of data they make use of streams. A stream is a pointer to a buffer of memory which is used for transferring the data. In general stream can be assumed as a sequence of bytes which flow from source to destination. An I/O stream may be text stream or binary stream depending upon in which mode you have opened the file. A text stream contains lines of text and the characters in a text stream may be manipulated as per the suitability. But a binary stream is a sequence of unprocessed data without any modification. The standard I/O stream or stream pointers are (for reading) , **stdout** (for output), and **stderr**(for error). By default **stdin** represent **keyboard** , **stdout** and **stderr** represents **monitor** or **VDU.**

Low level files make use of the system-calls of the Operating System under which the program is run.

14.2 File opening, reading and closing

Python provides open() function to open a file. By default, the file is opened in read mode. This function returns a file object, also called a handle. Using the file handle, we can any operation onto the file. It is used as:

```
file_handle=open(filename)
```

file_handle is any identifier name to handle the file, filename can be a file name in current directory or complete path to the filename. Lets see some examples:

```
f1=open("data.txt")
f2=open("/home/drvikasthada/file.txt")
```

First example is filename present in the current directory and other is absolute path to the file.

Once file is opened without any error (it may throw exception when file trying to read is not present) the content of file can be read with number of functions. The simplest function is read() that read all contents of the file in one go.

```
contents1=f1.read()
contents2=f2.read()
```

Once operations are done with the file, it can be closed easily by calling close method on file handle as:

```
f1.close()
f2.close()
```

Closing a file free up all system resources attached to file and allow the same file to be used by some other applications or programs. It is a good practice to always close the file after you have done with it.

Having understood the basic concepts of file opening, reading and closing, lets see a complete example using python script.

```
f1=open("data.txt")
data=f1.read()
print("**File contents are**")
print(data)
f1.close()
```

OUTPUT:

```
**File contents are**
This is demo of file handling.
This is second line.
```

The output is self-explanatory but there is a small problem with the above approach. In case the file is not found or has been deleted or some other issues related to input/output happens, it may crash the program. The best way to use try-except block as shown below:

```
try:
    f1=open("data1.txt")
except Exception as e:
    print("Error in file opening=",e)
else:
    data=f1.read()
    print("**File contents are**")
```

```
    print(data)
    f1.close()
```

OUTPUT:

```
(File does not exist)
Error in file opening= [Errno 2] No such file or directory: 'data1.txt'
```

As you can see we have put the file-opening code in the try-block. In case there is some error in file opening an exception will be thrown. Otherwise else block will read from the file, display it and close it.

The most used way to open, read and close the file preferred by most authors and python programmers is using with clause. See the code below:

```
with open("data.txt") as f:
    data=f.read()
    print("**File contents are**")
    print(data)
```

OUTPUT:

```
**File contents are**
This is demo of file handling.
This is second line.
```

The advantage of with statement is that it closes the file automatically as soon as contents within with are executed even if an exception is raised at some point.

14.3 File Opening Modes

A file can be opened in different types of modes passed as second argument to the open function. Various modes determine how the file will be opened (reading/writing/appending). Further the file can be plain ASCII text file or a binary file. All the modes with meaning is shown below:

Table 14.1 : File Opening Modes

S.N	Mode	Meaning
1.	r	Opens text file in read mode only and read from the beginning of the file. It is the default mode.
2.	rb	Opens binary file(images,video) in read mode only and read from the beginning of the file. Data is read in the form of bytes.
3.	r+	Opens text file for reading and writing. Places the pointer in the beginning of the file.
4.	rb+	Opens binary file(images,video) for reading and writing. Places the pointer in the beginning of the file. Data is read and written in the form of bytes.
5.	w	Opens text file in write mode. Any existing file with the same name will be overwritten. Pointer is placed in the beginning of the file.

6.	wb	Opens binary file(images,video) in write mode. Any existing file with the same name will be overwritten. Pointer is placed in the beginning of the file. Data is written in the form of bytes.
7.	w+	Opens text file for reading and writing. Places the pointer in the beginning of the file.
8.	wb+	Opens binary file(images,video) for reading and writing. Places the pointer in the beginning of the file. Data is read and written in the form of bytes.
9.	a	Opens text file for appending new data to it. Places the pointer at the end of the file. File is created if doesn't exist.
10.	ab	Opens binary file(images,video) for appending new data to it. Places the pointer at the end of the file
11.	a+	Opens text file for appending new data to it and reading . Places the pointer at the end of the file.
12.	ab+	Opens binary file(images,video) for appending new data to it and reading. Places the pointer at the end of the file
13.	x	Open the file in exclusive mode. It creates file when doesn't exist and gives error if exist.

Most of these modes will be used in examples coming up later in this chapter.

14.4 Reading from file

A simple example of reading from a file was covered earlier. In this section we discuss in details all the methods to read from the file (text or binary).

14.4.1 The read function

The read function in its simplest form (no argument supplied) reads entire file at once. If you want to read specific number of characters, then you can supply. Further if your file contains some special characters then you must supply encoding type as encoding="utf8". See few examples:

```
with open("data.txt","r") as f:
    print("** whole file read**")
    print(f.read())
```

OUTPUT:

```
** whole file read**
This is demo of file handling.
This is second line.
```

```
Wow ! I earned $2000.
```

This kind of example we have seen earlier. Nothing to explain much.

In the next example we read 10 characters at once and then read 20 characters later.

```
with open("data.txt","r") as f:
    print("**First 10 character read**")
    print(f.read(10))
    print("**Next 20 character read**")
    print(f.read(20))
```

OUTPUT:

```
**First 10 character read**
This is de
**Next 20 character read**
mo of file handling.
```

14.4.2 The function readline and readlines

The two functions are used for readling line of text from the file. The function readline reads one line at a time and readlines returns a list of lines. Lets see there usage in examples:

```
with open("data.txt","r") as f:
    print(f.readline())
```

OUTPUT:

```
This is demo of file handling.
```

The above code reads just one line from the input file and prints it. If you want to read complete file line by line just use loop:

```
f=open("data.txt","r")
line=f.readline()
while line!="":
    print(line,end='')
    line=f.readline()
```

OUTPUT:

```
This is demo of file handling.
This is second line.
Wow ! I earned $2000.
```

The function readline reads a single line from file. When end of file reaches then it returns empty string. In the code we open the file in read mode and read first line from the file. The while loop condition checks the end of file as readline function will return empty string if nothing is remaining in file to read. Inside while loop we print the line read before the while loop and keep reading line again.

The method readline also takes an integer parameter: number of characters to read from the current line. Try it out.

There is an easy and efficient way to read a file line by line using for loop without using readline method. See below:

```
with open("data.txt","r") as f:
    for line in f:
        print(line,end='')
```

OUTPUT:

```
This is demo of file handling.
This is second line.
Wow ! I earned $2000.
```

The for loop reads file line by line and using print we display the same. Named parameter end is set to empty string otherwise extra newline will be inserted.

The readlines method

The readlines function reads all lines from a file and returns a list containing each line as element of the list. See the method in action:

```
with open("data.txt","r") as f:
    print(f.readlines())
OUTPUT:
['This is demo of file handling.\n', 'This is second line.\n', 'Wow ! I earned $2000.\n']
```

As you can see, readlines method returns a list of lines present in the file.

14.5 Writing to File

To write into a file in Python, we need to open it in write ,append or exclusive creation mode. Lets open/create a file in write mode and write some data into it. As discussed earlier in file opening modes, if you open a file in write mode and file already exist, all its contents will be destroyed. So be careful in using this mode.

There are two methods available in python for writing to file:write and writelines. The write is used to write one single line to the file and writelines can take a list of lines to write to the file. The write method returns the number of characters written while writelines method returns None.

See some examples to see these methods in action:

```
with open("newfile.txt","w") as f:
    print(f.write("First rule never give up\n"))
    print(f.write("Second rule never give up\n"))
    print(f.write("third rule remember first two rules"))
```

OUTPUT:

```
25
26
35
```

The file was not present in the current directory and it gets created. The three write method writes their contents to the file one per line and returns number of characters written.

In the next example we write the complete list to the new file “newfile1.txt” in just one single line using writelines method.

```
L=["First Line\n","Second Line\n","Third Line\n"]
with open("newfile1.txt","w") as f:
    print(f.writelines(L))
   OUTPUT:

   None
```

14.5.1 Reading and writing

Lets see few examples where we create a new file and write into it. After that we open the same file again but in reading mode and display the contents that was written.

```
print("**File Writing and Reading**")
print("*Writing to File*")
with open("newfile.txt","w") as f:
    f.write("First rule never give up\n")
    f.write("Second rule never give up\n")
    f.write("third rule remember first two rules")
print("*Reading from File*")
with open("newfile.txt","r") as f:
    for line in f:
        print(line,end='')
```

OUTPUT:

```
**File Writing and Reading**

*Writing to File*

*Reading from File*

First rule never give up

Second rule never give up

third rule remember first two rules
```

In the code above we create a new file “newfile.txt” and write some lines into it. The file get auto closed after with block is over. The file is opened again in read mode and contents are displayed.

If you don’t want to close the file, there is another way around. The file can be opened in “w+” mode so writing and reading both are possible. Lets see this in action.

```
print("**File Writing and Reading**")
print("*Writing to File*")
f=open("newfile.txt","w+")
f.write("First rule never give up\n")
f.write("Second rule never give up\n")
f.write("third rule remember first two rules")
print("*Reading from File*")
f.seek(0)
for line in f:
    print(line,end='')
```

OUTPUT:

```
**File Writing and Reading**
```

```
*Writing to File*
*Reading from File*
First rule never give up
Second rule never give up
third rule remember first two rules
```

The file is opened in “w+” mode so file can be used for both reading and writing. Once the contents are written to the file, file handle points to the end of the file. To read from file the handle has to be brought back to the beginning. This is done using seek function. The parameter is the offset from which position. Default is beginning. So the line f.seek(0) bring back the file handle to the beginning of the file. After that file contents are read and displayed.

14.5.2 Appending data to file

The new contents can be added to an existing file by opening in append mode. Here we see an example to open a file in “a+” mode so file can be opened for appending and reading.

```
print("**File Appending and Reading**")
print("*Appending to File*")
f=open("newfile.txt","a+")
f.write("New Contents added")
f.seek(0)
print("*Contents read from file*")
for line in f:
    print(line,end='')
f.close()
```

OUTPUT:

```
*Appending to File*
*Contents read from file*
First rule never give up
Second rule never give up
third rule remember first two rules
New Contents added
```

14.6 Working with multiple files

At times we require to work with multiple files in different modes. Here in this section we see two examples where more than one is used at the same time to carry out required task.

```
def file_copy(src,dest):
    try:
        f1=open(src,"r")
        f2=open(dest,"w")
    except Exception as e:
        print("File error=",e)
    else:
        f2.write(f1.read())
        f1.close()
```

```
        f2.close()
        print("File Copied Successfully")

if __name__=="__main__":

    file_copy("demo.txt","newdemo.txt")
```

OUTPUT:

```
File Copied Successfully
```

We have written a function for copying one file to another file. The source file is opened in read mode and destination file in write mode. The source file must exist else exception will be thrown. In the else block the file is copied simply by:

```
f2.write(f1.read()
```

The other example is where we have two different files:names.txt and snames.txt. First file will be having 3 names and other file the surnames corresponding to the names stored in first file. The names and surnames will be first written to the files. Later both the files will be opened and full names will be shown. See the code in action:

```
f1=open("names.txt","w+")
f2=open("sname.txt","w+")
names=["Namit\n","Juhi\n","Purvi\n"]
snames=["Sharma\n","Jain\n","Jindal\n"]
print("*Writing names to first file*")
f1.writelines(names)
print("*Writing surnames to second file*")
f2.writelines(snames)
f1.seek(0)
f2.seek(0)
print("*Displaying fullnames*")
for (name,sname) in zip(f1,f2):
    print(str(name).strip()+" ",end='')
    print(str(sname).strip())
```

OUTPUT:

```
*Writing names to first file*
*Writing surnames to second file*
*Displaying fullnames*
Namit Sharma
Juhi Jain
Purvi Jindal
```

We first create two files and write names and surnames to respective files. After writing names and surnames we set the file pointer back to position 0 from beginning. To display fullnames the following code has been used:

```
for (name,sname) in zip(f1,f2):
    print(str(name).strip()+" ",end='')
    print(str(sname).strip())
```

In each iteration of the for loop a line is read from each file and stored in name and sname. The zip method zipps the file handles so data from both the files can be read together. The print method deserve

some explanation. During writing we have used \n after every name and same create issues while reading back from file. See the output without using strip method when for loop is used as:

```
for (name,sname) in zip(f1,f2):
    print(name," ",sname,end='')
```

OUTPUT:

```
Namit
   Sharma
Juhi
   Jain
Purvi
   Jindal
```

To remove extra newline character when reading back from file we have used strip method.

14.7 Random Access in File

Random access means reading data randomly from any where in the file. For this purpose, we need to set the position of the file pointer first in the file and then read the data. One example using seek function we have seen in few examples. Python provide two functions: seek and tell. The seek function is for the manipulation of file pointer anywhere in the file and tell function returns the current position of the pointer. With the help of these functions, we can access data in a random fashion.

1. The seek function

The syntax of seek function is:

```
seek(offset, where)
```

The method sets the file pointer to offset relative to parameter where. The offset can be positive or negative integer. The where parameter can have 0(begining of file stream or constant SEEK_SET), 1(current state of file stream or constant SEEK_CUR) and 2(end of file stream or constant SEEK_END). To use constants mentioned in the previous line you must import os module and use them as: os.SEEK_SET

In python 3 onwards seeking to text files is allowed only from beginning and not from current position or end. For more details please refer python documentation. There is a way to achieve negative offset when where is SEEK_CUR or SEEK_END but that has its own issues. The method is to open file in binary mode.

2. The tell function

The tell function returns the current position of the file pointer. If you open the file in read or write mode and print f.tell() it will print 0 and if you open file in append mode if will print number of characters in the file.

Lets see one example to understand both the functions:

```
f=open("alphabets.txt","w+")
f.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ\n")
name=''
# move pointer to position 15 (P) and read 1 character
f.seek(15)
```

```
name=name+f.read(1)
print("Current Position after f.seek(15)=",f.tell())
# move pointer to position 0 (A) and read 1 character
f.seek(0)
name=name+f.read(1)
print("Current Position after f.seek(0)=",f.tell())
# move pointer to position 17(R) and read 1 character
f.seek(17)
name=name+f.read(1)
print("Current Position after f.seek(17)=",f.tell())
# move pointer to position 8 (I) and read 1 character
f.seek(8)
name=name+f.read(1)
print("Current Position after f.seek(8)=",f.tell())
print("Name read through random access= ",name)
f.close()
```

OUTPUT:

```
Current Position after f.seek(15)= 16
Current Position after f.seek(0)= 1
Current Position after f.seek(17)= 18
Current Position after f.seek(8)= 9
Name read through random access=  PARI
```

We have created a file and written capital letter alphabets into it. Using seek we set the position of pointers to read the characters of string "PARI". After setting position for each character we read character at that position. Note after reading one character the file pointer moves to next position. This was clarified using tell function. Finally after reading all the characters of our desired name, we display the same.

14.8 Working with numbers

So far we have worked with only text data. At times we may have some numeric data stored in file and we want to process that data. Python has no specialized file functions to work with numeric data. The functions that we have seen are to be used for dealing with numeric data. In this section we work with numeric data with the files and see how can we process and manipulate the data stored in files.

Lets work with our first example:

```
f=open("mynums.txt","w")
f.writelines(["10\n","20\n","30\n","50\n"])
f.close()
f = open("mynums.txt", 'r')
data = f.readlines()
sum=0
for line in data:
    sum=sum+int(line.strip())
print("Sum=",sum)
f.close()
```

OUTPUT:

```
Sum=110
```

We have created a file "mynums.txt" and put some numbers into the file but note that numbers have been inserted into string format. This is required as write and writelines method only work with string data. Once the data has been written file is closed and opened again in read mode. All the lines within the file are read using readlines method are processed using for loop. Each line read from the file is in string format. The extra newline character is removed using strip method and converted to integer. In case the file has a mix of float and integer data then use float method for conversion as it can handle both int and float type data in string format.

But what if the file has string, integer and float type data like shown below:

```
# contents of file newfile.txt
10
20
40
4.5
5.5
True
2.5
hello
```

In this case we must make sure string data does not crash our program or gives error. The solution is to use exception handling when float cannot convert string data. See code below:

```
def check_data(x):
        try:
            val = float(x)
            return (val,True)
        except ValueError:
            val=x
            return (val,False)

f = open("newfile.txt", 'r')
data = f.readlines()
sum=0
for line in data:
    (v,status)=check_data(line.strip())
    if status:
        sum=sum+v
print("Sum=",sum)
f.close()
OUTPUT:
Sum=82.5
```

We have written a method check_data to check data is in float or integer string. If it is then it is parsed to float and (val,True) is returned. In case exception ValueError is generated then it means data is pure string and (val,False) is returned.

In the for loop the returned values are stored in v and status. If status is True then v value is added to sum else nothing is done. In the end the sum is displayed.

The next example is to deal with when we more than one numeric data in different lines like as shown below:

```
# file newfile.txt
10 20 40
4.5 5.5 True
2.5 hello
```

Here the numbers are separated by space but it can be some other value also. How to read this type of file and find sum of each line. Lets see the next code:

```
def check_data(x):
        try:
            val = float(x)
            return (val,True)
        except ValueError:
            val=x
            return (val,False)

f = open("newfile.txt", 'r')
data = f.readlines()
sum=0
linec=1
for line in data:
    for x in line.split():
        (v,status)=check_data(x.strip())
        if status:
            sum=sum+v
    print("Sum of line ",linec,"=",sum)
    linec+=1
    sum=0
f.close()
```

OUTPUT:

```
Sum of line  1 = 70.0
Sum of line  2 = 10.0
Sum of line  3 = 2.5
```

Here to process each line we have one more for loop that splits the line using split method. Default to split on space. If your file has some other separator like : or ; then pass the same as argument to split function. The same function check_data is used to check each element after line split is a number or string. Accordingly we find sum and display the result for each line.

14.9 Working with binary mode

Binary files are those files where data is stored in terms of sequence of bytes either in 8 bit or 16 bits. are not human readable but they are well understood by their respective applications. Text files can be easily opened and read but its difficult to comprehend binary files. All video,audio, image files are binary files.

Working with binary mode is simple in python. You just need to used “rb” or “br” for reading or “wb” or “bw” is for writing. They use the same function as worked with text files but the data is read and written in the form of bytes.

Lets see some examples:

```
data =[65,66,67,68,127,128]
buffer = bytes(data)
print(buffer)
f = open("binary.txt", "bw")
f.write(buffer)
f.close()
f = open("binary.txt", "br")
print(f.read())
f.close()
```

OUTPUT:

```
b'ABCD\x7f\x80'
b'ABCD\x7f\x80'
```

The data list contains some ASCII code and converted into bytes. A file "binary.txt" is opened in binary mode for writing and byte contents are written to the file. After the same file is read back and contents are displayed. As seen from the output the contents are in bytes and last two bytes are not readable.

Lets have one more example:

```
f=open("myfile","wb+")
f.write(b"hello to binary")
f.seek(0)
data = f.read()
print(data.decode())
f.close()
```

OUTPUT:

```
hello to binary
```

When a file is opened in binary mode you can only write bytes to it. To convert string to bytes either user bytes function or use b as prefix to string. While reading back from file use decode function to get back in string form else output will remain in byte form.

If you remove decode function above, output will be:

```
b'hello to binary'
```

In most networking protocols or communication data is transferred in the form of bytes so it is recommended to use bytes function with proper encoding and decode function while decoding (reading back). See the modified code:

```
f=open("myfile","wb+")
buffer=bytes("hello to binary",encoding="utf8")
f.write(buffer)
f.seek(0)
data = f.read()
print(data.decode())
f.close()
def check_data(x):
    try:
        val = int(x)
```

```
        return True
    except ValueError:
        try:
            val = float(x)
            return True
        except ValueError:
            val=x
            return False

f = open("newfile.txt", 'r')
data = f.readlines()
sum=0
for line in data:
    if check_data(line.strip()):
        sum=sum+float(line.strip())
print(sum)
f.close()
```

14.10 Files and Objects

Python allow you to store virtually any type of objects into the file be it integer, string, float,list, dictionary or even user defined objects of classes. In this section we are going to see some of the modules that help you to achieve this persistence of objects. Storing objects into the file is known as serialization and reading them back is de-serialization.

14.10.1 Picking objects

The very common module for storing the objects into file and reading them back is pickle module. This module is part of the standard python library and very popular for serialization and deserialization.

To store any object into file just use dump method and to read back just use load method. Lets have a very simple example:

```
import pickle
f=open("pick1.pkl","wb+")
a="Hello"
pickle.dump(a,f) # dumping to file
f.seek(0)
del a
a=pickle.load(f) # reading back from file
f.close()
print("Data read from file")
print(a)
OUTPUT:
Data read from file
Hello
```

In the first line the module pickle is imported. A file “pick1.pkl” is created. Its not necessary to have pkl extension but file mode must be binary. In the example we have just dumped one string into the file using dump method and read the same from the file using load method. Before reading the data back

from file we have deleted the variable a, just to make sure that a is initialized with the data read from the file.

Note in the above example we have dumped the data and load in the same program. Usually we dumped the data and load it later from the same file in another script. Try it yourself.

In another code shown below instead of just one object we are dumping multiple objects and loading them back.

```
import pickle
f=open("pick1.pkl","wb+")
a="hello"
b=[1,2,3,4]
c=12.34
d={"a":200,"b":234}
pickle.dump(a,f)
pickle.dump(b,f)
pickle.dump(c,f)
pickle.dump(d,f)
f.seek(0)
del a,b,c,d
a=pickle.load(f)
b=pickle.load(f)
c=pickle.load(f)
d=pickle.load(f)
f.close()
print("Data read from file")
print(a,b,c,d,sep=",")
```

OUTPUT:

```
Data read from file
hello,[1, 2, 3, 4],12.34,{'a': 200, 'b': 234}
```

The code is self-explanatory.

The next example is dumping and loading user-defined objects.

```
import pickle
class Employee:
    def __init__(self,name,sal,job):
        self.name=name
        self.sal=sal
        self.job=job
    def __str__(self):
        s="Name="+self.name
        s=s+"\t"+"Salary="+str(self.sal)+"\t"
        s=s+"Job="+self.job
        return s
if __name__=="__main__":
    f=open("objects.pkl","wb+")
    e1=Employee("Naman",30000,"Trainer")
    e2=Employee("Juhi",60000,"Analyst")
    print("Dumping to file")
    pickle.dump(e1,f)
```

```
    pickle.dump(e2,f)
    del e1,e2
    f.seek(0)
    print("Loading from file")
    e1=pickle.load(f)
    e2=pickle.load(f)
    print("Employee 1")
    print(e1)
    print("Employee 2")
    print(e2)
```

OUTPUT:

```
Dumping to file
Loading from file
Employee 1
Name=Naman Salary=30000     Job=Trainer
Employee 2
Name=Juhi  Salary=60000     Job=Analyst
```

We have a class Employee with three fields: Name, sal and job. We create objects of this class and dump to file using pickle module. After loading back from file the same objects we delete them and read them back. To display objects as string we have written __str__ method that is called automatically when the object is displayed using print method.

There is an issue with the code above. If you create the class and dump the objects in one file and read them in another file you are going to have error. The error will be because class definition is not present in the second file. This is a serious issue as the code to read the pickle file "objects.pkl" may be read any where on any system and programmer may not aware about class definition except the members of the class.

For example after creation of file "objects.pkl" if we try to read it in some other file and load object of Employee class see what happens:

```
import pickle
f=open("objects.pkl","rb")
e1=pickle.load(f)
```

OUTPUT:

```
AttributeError: Can't get attribute 'Employee' on <module '__main__' from
'quickstart.py'>
```

The solution is discussed in the next section.

14.10.2 The dill module

The dill module extends upon pickle module with some extra functionality. The dill module is not part of standard python module and must be installed as: pip install dill.

The dill module has same dump and load method and same syntax but extra functionality the dill module provides it that it saves an entire session of the python interpreter using one single command in a pickle file and same file can be used anywhere in another python session again using a single python command.

Lets change out previous code using dill module. In the left column we have the code that writes two employee objects in "objects.pkl" file using dill module. In the right column (a different python script) is written that is totally independent of first python script. This script loads the object from file and display the two employee objects without any error.

```
import pickle
class Employee:
    def __init__(self,name,sal,job):
        self.name=name
        self.sal=sal
        self.job=job
    def __str__(self):
        s="Name="+self.name

s=s+"\t"+"Salary="+str(self.sal)+"\t"
        s=s+"Job="+self.job
        return s
if __name__=="__main__":
    f=open("objects.pkl","wb+")

e1=Employee("Naman",30000,"Trainer")

e2=Employee("Juhi",60000,"Analyst")
    pickle.dump(e1,f)
    pickle.dump(e2,f)
f.close
```

```
import dill
f=open("objects.pkl","rb")
print("Loading from file")
e1=dill.load(f)
e2=dill.load(f)
print("Employee 1")
print(e1)
print("Employee 2")
print(e2)
OUTPUT:
Loading from file

Employee 1

Name=Naman Salary=30000
     Job=Trainer

Employee 2

Name=Juhi  Salary=60000
     Job=Analyst
```

The other way to save everything and read it later you can save the entire session of python shell and load it later. See an example:

```
>>> class demo:
...     pass
...
>>> d1=demo()
>>> d1.x=20
>>> d1.y=30
>>> d2=demo()
>>> d2.x=2.4
>>> d2.y=3.6
>>> import dill
>>> dill.dump_session("demo.pkl")
>>> exit()
```

As you can see the complete session has been dumped into file "demo.pkl" using dump_session function. The last line is exit() to come out from python shell. Now load the session stored in "demo.pkl" in a new python session:

```
>>> import dill
>>> dill.load_session("demo.pkl")
>>> (d1.x,d1.y)
(20, 30)
>>> (d2.x,d2.y)
(2.4, 3.6)
```

One more good feature of using dill is that it allow us to store lambda functions as property of class objects or can store them directly. See our final example:

```
import dill
class demo:
    pass
d=demo()
d.sum=lambda x,y:x+y
f=open("demo.pkl","wb+")
sqr=lambda x:x*x
dill.dump(sqr,f)
dill.dump(d,f)
f.seek(0)
sqrf=dill.load(f)
obj=dill.load(f)
print(sqrf(2))
print(obj.sum(2,3))
f.close()
```

OUTPUT:

```
4
5
```

Two lambda functions are created in this script. One is general and second as property of the class. Both are dumped using dump function of dill module. After dumping file position is set to 0 using seek both the functions are loaded and used in print function.

14.11 Ponderable Points

1. A data file is a collection of data items stored permanently in persistent storage area.
2. The standard I/O stream or stream pointers are (for reading) , **stdout** (for output), and **stderr**(for error). By default **stdin** represent **keyboard** , **stdout** and **stderr** represents **monitor** or **VDU.**
3. Python provides open() function to open a file. By default, the file is opened in read mode.
4. The function read() that read all contents of the file in one go.
5. The function readline reads one line at a time and readlines returns a list of lines.
6. To write into a file in Python, we need to open it in write ,append or exclusive creation mode.
7. Python provide two functions: seek and tell. The seek function is for the manipulation of file pointer anywhere in the file and tell function returns the current position of the pointer.

8. Binary files are those files where data is stored in terms of sequence of bytes either in 8 bit or 16 bits. are not human readable but they are well understood by their respective applications.
9. Storing objects into the file is known as serialization and reading them back is de-serialization.
10. For reading and writing objects module pickle and dill can be used.

Contents

1. STARTING WITH PYTHON 1
2. OPERATORS & EXPRESSIONS 58
3. DECISION MAKING 94
4. LOOPING 112
5. FUNCTIONS 140
6. STRINGS IN PYTHON 185
7. LIST 211
8. DICTIONARY 237
9. TUPLE 250
10. MODULES IN PYTHON 257
11. CLASSES AND OBJECTS 266
12. INHERITANCE 283
13. EXCEPTION HANDLING 316
14. FILE HANDLING 335