

**THIRD YEAR B. Sc.
COMPUTER SCIENCE
SEMESTER-V**

**NEW SYLLABUS
CBCS PATTERN**

PYTHON PROGRAMMING

Dr. Ms. MANISHA BHARAMBE



SPPU New Syllabus

A Book Of

PYTHON PROGRAMMING

For T.Y.B.Sc. Computer Science : Semester – V

[Course Code CS 3510 : Credits - 2]

CBCS Pattern

As Per New Syllabus, Effective from June 2021

Dr. Ms. Manisha Bharambe

M.Sc. (Comp. Sci.), M.Phil. Ph.D. (Comp. Sci.)

Vice Principal, Associate Professor, Department of Computer Science
MES's Abasaheb Garware College
Pune

Price ₹ 300.00



N5867

PYTHON PROGRAMMING**ISBN 978-93-5451-195-0****First Edition : August 2021****© : Author**

The text of this publication, or any part thereof, should not be reproduced or transmitted in any form or stored in any computer storage system or device for distribution including photocopy, recording, taping or information retrieval system or reproduced on any disc, tape, perforated media or other information storage device etc., without the written permission of Author with whom the rights are reserved. Breach of this condition is liable for legal action.

Every effort has been made to avoid errors or omissions in this publication. In spite of this, errors may have crept in. Any mistake, error or discrepancy so noted and shall be brought to our notice shall be taken care of in the next edition. It is notified that neither the publisher nor the author or seller shall be responsible for any damage or loss of action to any one, of any kind, in any manner, there from. The reader must cross check all the facts and contents with original Government notification or publications.

Published By :**NIRALI PRAKASHAN**

Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate**Printed By :****YOGIRAJ PRINTERS AND BINDERS**

Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041

DISTRIBUTION CENTRES**PUNE****Nirali Prakashan****(For orders outside Pune)**

S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan**(For orders within Pune)**

119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Tel : (020) 2445 2044; Mobile : 9657703145
Email : niralilocal@pragationline.com

MUMBAI**Nirali Prakashan**

Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES**DELHI****Nirali Prakashan**

Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU**Nirali Prakashan**

Maitri Ground Floor, Jaya Apartments,
No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003
Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR**Nirali Prakashan**

Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR**Nirali Prakashan**

New Mahadvar Road, Kedar Plaza,
1st Floor Opp. IDBI Bank
Kolhapur 416 012 Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON**Nirali Prakashan**

34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR**Nirali Prakashan**

R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.pragationline.com**Also find us on  www.facebook.com/niralibooks**

Preface ...

I take an opportunity to present this Text Book on "**Python Programming**" to the students of Third Year B.Sc. (Computer Science) Semester-V as per the New Syllabus, June 2021.

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts. The book covers theory of An Introduction to Python, Control Statements, List, Functions, Tuples, Directories and Sets, Modules, Working with Files and Exception Handling.

A special word of thank to Shri. Dineshbhai Furia, and Mr. Jignesh Furia for showing full faith in me to write this text book. I also thank to Mr. Amar Salunkhe and Mr. Akbar Shaikh of M/s Nirali Prakashan for their excellent co-operation.

I also thank Ms. Chaitali Takle, Mr. Ravindra Walodare, Mr. Sachin Shinde, Mr. Ashok Bodke, Mr. Moshin Sayyed and Mr. Nitin Thorat.

Although every care has been taken to check mistakes and misprints, any errors, omission and suggestions from teachers and students for the improvement of this text book shall be most welcome.

Author



Syllabus ...

1. An Introduction to Python

(3 Lectures)

- **Introduction to Python:** The Python Programming Language, History, Features, Applications, Installing Python, Running Simple Python Program
- **Basics of Python**
- **Standard Data Types:** Basic, None, Boolean (True and False), Numbers, Variables, Constants, Python Identifiers and Reserved Words, Lines and Indentation, Multi-line Statements and Comments, Input/output with Print and Input, Functions Declaration, Operations on Data such as Assignment, Arithmetic, Relational, Logical and Bitwise Operations, Dry Run, Simple Input and Output etc.

2. Control Statements

(4 Lectures)

- **Sequence Control:** Precedence of Operators, Type Conversion
- **Conditional Statements:** if, if-else, nested if-else
- **Looping:** for, while, Nested loops, Loop Control Statements (break, continue, pass)
- **Strings:** Declaration, Manipulation, Special Operations, Escape Character, String Formatting Operator, Raw String, Unicode Strings, Built-in String Methods

3. Lists, Functions, Tuples and Dictionaries, Set

(7 Lectures)

- **Python Lists:** Concept, Creating and Accessing Elements, Updating and Deleting Lists, Traversing a List, Reverse Built-in List Operators, Concatenation, Repetition, in Operator, Built-in List Functions and Methods
- **Functions:** Definitions and Uses, Function Calls, Type Conversion Functions, Math Functions, Composition, Adding New Functions, Flow of Execution, Parameters and Arguments, Variables and Parameters, Stack Diagrams, Void Functions, Anonymous Functions, Importing with From, Return Values, Boolean Functions, More Recursion, Functional Programming Tools: filter(), map(), and reduce(), Recursion, Lambda Forms
- **Tuples and Dictionaries:** Tuples, Accessing Values in Tuples, Tuple Assignment, Tuples as Return Values, Variable-length Argument Tuples, and Basic Tuples Operations, Concatenation, Repetition, in Operator, Iteration, Built-in Tuple Functions, Indexing, Slicing and Matrices
Creating a Dictionary, Accessing Values in a Dictionary, Updating Dictionary, Deleting Elements from Dictionary, Properties of Dictionary Keys, Operations in Dictionary, Built-in Dictionary Functions, Built-in Dictionary Methods
- **Sets:** Definition, Transaction of Set (Adding, Union, Intersection), Working with Sets

4. Modules, Working with files, Exception Handling

(4 Lectures)

- **Modules:** Importing Module, Creating and Exploring Modules, Math Module, Random Module, Time Module
- **Packages:** Importing Package, Creating Package, Examples
- **Working with Files:** Creating Files and Operations on Files (Open, Close, Read, Write), File Object Attributes, File Positions, Listing Files in a Directory, Testing File Types, Removing Files and Directories, Copying and Renaming Files, Splitting Pathnames, Creating and Moving Directories
- **Regular Expression:** Concept of Regular Expression, Various Types of Regular Expressions, using Match Function
- **Exception Handling:** Built-in Exceptions, Handling Exceptions, Exception with Arguments, User-defined Exceptions



Contents ...

1. An Introduction to Python	1.1 – 1.50
2. Control Statements	2.1 – 2.58
3. Lists, Functions, Tuples, Dictionaries and Sets	3.1 – 3.96
4. Modules, Working with Files and Exception Handling	4.1 – 4.84



An Introduction to Python

Objectives...

- To learn Basic Concepts in Python
 - To study Installation of Python
 - To understand Data Types, Variables, Constants etc. in Python
-

1.0 INTRODUCTION

- Python is a high-level, interpreted, interactive and object-oriented programming language. Today, Python is the trendiest programming language programming.
 - There are several reasons for why Python programming language is the preferable choice of the programmers/developers over other popular programming languages like C++, Java and so on.
 - Python is popular programming language because of it provides more reliability of code, clean syntax of code, advanced language features, scalability of code, portability of code, support object oriented programming, broad standard library, easy to learn and read, support GUI mode, interactive, versatile and interpreted, interfaces to all major commercial databases and so on.
 - There are two major Python versions namely, Python 2 and Python 3. Python 3.0 was developed with the same philosophy as in prior versions.
 - Python 3.9.6 is the newest/latest major release of the Python programming language. Python scripts normally have the file extension .py.
 - Some common **applications of Python Programming** are listed below:
 1. Google's App Engine web development framework uses Python as an application language.
 2. Maya, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
 3. Linux Weekly News, published by using a web application written in Python programming.
 4. Google makes extensive use of Python in its Web Search Systems.
 5. The popular YouTube video sharing service is largely written in Python programming.
-


6. The NSA (National Security Agency) uses Python programming for cryptography and intelligence analysis.
7. iRobot uses Python programming to develop commercial and military robotic devices.
8. The Raspberry Pi single-board computer promotes Python programming as its educational language.
9. Netflix and Yelp have both documented the role of Python in their software infrastructures.
10. Industrial Light and Magic, Pixar and others uses Python programming in the production of animated movies.

1.1 PYTHON PROGRAMMING LANGUAGE





- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.
- Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Why Python?

1. Python can be treated in a procedural way, an object-oriented way or a functional way.
 2. Python's simple, easy to learn syntax emphasizes readability of code.
 3. Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
 4. Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Today, there are many free and commercial editors available for Python. Following table lists Python editors:

Sr. No.	Editor	Description	Icon/Logo
1.	IDLE	<ul style="list-style-type: none"> • IDLE is a popular Integrated Development Environment written in Python and it has been integrated with the default language. • Mainly used by the beginner level developers who want to practice on Python development. 	

contd. ...

2.	PyCharm	<ul style="list-style-type: none"> PyCharm is one of the widely used Python IDE which was created by Jet Brains. With PyCharm, the developers can write a neat and maintainable code. It helps to be more productive and gives smart assistance to the developers. It takes care of the routine tasks by saving time and thereby increasing profit accordingly. 	
3.	Spyder	<ul style="list-style-type: none"> It was mainly developed for scientists and engineers to provide a powerful scientific environment for Python. It offers an advanced level of edit, debug, and data exploration feature. It is very extensible and has a good plugin system and API. 	
4.	PyDev	<ul style="list-style-type: none"> PyDev is an outside plugin for Eclipse. It is basically an IDE that is used for Python development. It is linear in size. It mainly focuses on the refactoring of python code, debugging in the graphical pattern, analysis of code etc. It is a strong python inter-preter. 	
5.	Jupyter Notebook	<ul style="list-style-type: none"> The Jupyter Notebook is a browser-based graphical interface to the IPython shell. Allows us to create and share documents that contain live code, equations, visualizations and narrative text. 	

- In this text book, IDLE is used for Python programming. IDLE (Integrated Development and Learning Environment) is an Integrated Development Environment (IDE) for Python.
- To start IDLE interactive shell, search for the IDLE icon in the start menu and double click on it and we will get the following window, (See Fig. 1.1).
- In Python IDLE shell not only we can execute commands one by one like in Python Command Prompt but also can create .py files and see execution of those files.

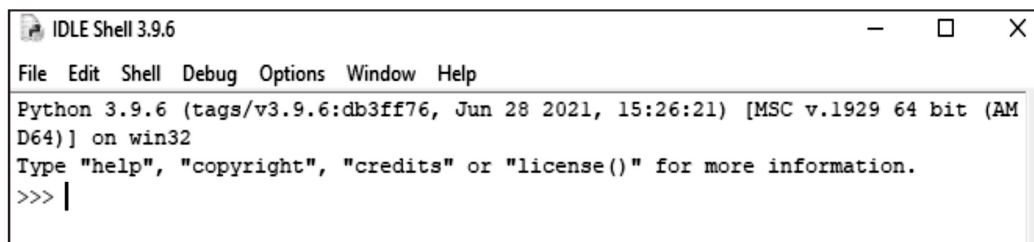


Fig. 1.1: Python Shell

1.1.1 History

- Python laid its foundation in the late 1980s. Python was developed by Guido Van Rossum at National Research Institute for Mathematics and Computer Science in Netherlands in 1990.
- Inspired by Monty Python's Flying Circus, a BBC comedy series, he named the language Python.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- ABC programming language is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Like Perl, Python source code is now available under the GNU General Public License (GPL). In February 1991, Guido Van Rossum published Python 0.9.0 (first release). In addition to exception handling, Python included classes, lists and strings.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce which aligned it heavily in relation to functional programming.
- Python 2.0 added new features like list comprehensions, garbage collection system and it supported Unicode.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language. In Python 3.0 the print statement has been replaced with a print() function.
- Python widely used in both industry and academia because of its simple, concise and extensive support of libraries.
- Python is available for almost all operating systems such as Windows, Mac, Linux/Unix etc. Python can be downloading from <http://www.python.org/downloads>.

1.1.2 Features

- Features of Python programming language are given below:
 1. **Simple and Easy-to-Learn:** Python is a simple language with few keywords, simple structure and its syntax is also clearly defined. This makes Python a beginner's language. In Python programming programs are easy to write and execute as it omits some cumbersome, poorly understandable and confusing features of other programming language such as C++ and Java.
 2. **Interpreted and Interactive:** Python is processed at runtime by the interpreter. We need not compile the Program before executing it. The Python prompt interact with the interpreter to interpret the programs that we have written. Python has an option namely interactive mode which allows interactive testing and debugging of code.

3. **Object-Oriented:** Python supports Object Oriented Programming (OOP) concepts that encapsulate code within objects. All concepts in OOPs like data hiding, Operator Overloading, inheritance etc. can be well written in Python. It supports functional as well as structured Programming.
4. **Platform Independence/Cross Platform Language/Portable:** Python can run on a wide variety of hardware and software platforms and has the same interface on all platforms, such as Windows, Unix, Linux and Macintosh. Fig. 1.2 shows execution of Python code by interpreter.

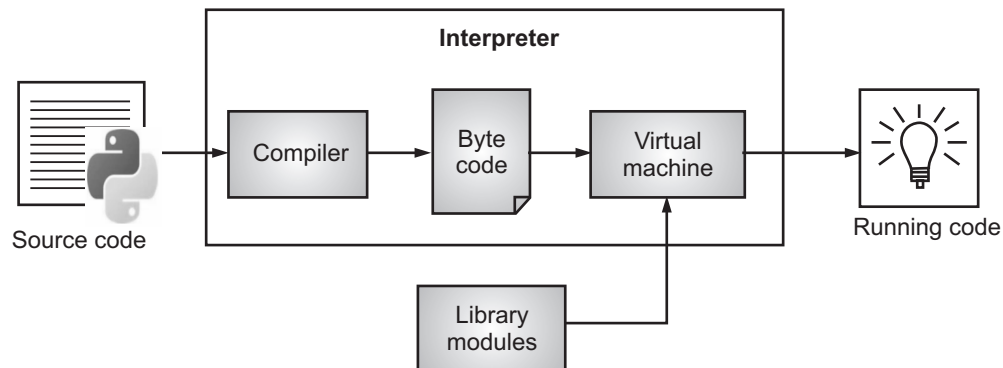


Fig. 1.2: Execution of Python Code

- Python source code goes through Compiler which compiles the source code into a format known as byte code. Byte code is a lower level, platform independent, efficient and intermediate representation of the source code. As soon as source code gets converted to byte code, it is fed into PVM (Python Virtual Machine). The PVM is the runtime engine of Python; it's always present as part of the Python system, and is the component that truly runs the scripts. Technically, it's just the last step of what is called the Python interpreter.
5. **Scalable:** Python provides a better structure and support for large Programs than shell scripting. It can be used as a scripting language or can be compiled to byte code {intermediate code that is platform independent} for building large applications.
 6. **Extendable:** You can add low-level modules to the Python interpreter. These modules enable Programmers to add to or customize their tools to be more efficient. It can be easily integrated with C, COM, ActiveX, CORBA, and Java.
 7. **Dynamic:** Python provides very high-level dynamic data types and supports dynamic type checking. It also supports automatic garbage collection.
 8. **GUI Programming and Databases:** Python supports GUI applications that can be created and ported to many libraries and windows systems, such as Windows Microsoft Foundation Classes (MFC), Macintosh, and the X Window system of Unix. Python also provides interlaces to all major commercial databases.

9. **Broad Standard Library:** Python's library is portable and cross platform compatible on UNIX, Linux, Windows and Macintosh. This helps in the support and development of a wide range of applications from simple text processing to browsers and complex games.
10. **Free and Open Source:** Python programming language is developed under an OSI approved open source license making it freely available at official web address. The source code is also available for use. The Python software can be freely distributed and any one can use and read its source code make changes/modifications to it and use the pieces in new free programs.

1.1.3 Applications

- Python is known for its general purpose nature that makes it applicable in almost each domain of software development.
- Here, we are specifying applications areas where python can be applied:
 1. **Web Applications:** We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML, XML and JSON, etc. It also provides Frameworks such as Django, Pyramid, Flask etc. to design and develop web based applications. Some important developments are PythonWikiEngines, PythonBlogSoftware etc.
 2. **GUI Based Desktop Applications:** Python has simple syntax, modular architecture, rich text processing tools and the ability to work on multiple operating systems which make it a desirable choice for developing desktop-based applications. There are various GUI toolkits like wxPython, PyQt or PyGtk available which help developers create highly functional Graphical User Interface (GUI).
 3. **Scientific and Numeric Applications:** Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, Ipython, Numeric Python etc. SciPy is group of packages of engineering, science and mathematics.
 4. **Software Development:** Python is helpful for software development process. It works as a support language and can be used for build control and management, testing etc.
 5. **Business Applications:** With features that include special libraries, extensibility, scalability and easily readable syntax, Python is a suitable coding language for customizing larger applications. Python is used to build Business applications like ERP and e-commerce systems. Tryton is a high level application platform.
 6. **Console Based Application:** We can use Python to develop console based applications. For example, IPython.

7. **Audio or Video based Applications:** Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are TimPlayer, cplay etc.
8. **3D CAD Applications:** To create CAD application Fandango is a real application which provides full features of CAD.
9. **Enterprise Applications:** Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are OpenErp, Tryton, Picalo etc.
10. **Applications for Images:** Using Python several application can be developed for image. Applications developed are VPython, Gogh, imgSeek etc.
11. **Education:** Python is a superb language for teaching programming, both at the introductory level and in more advanced courses.

1.1.4 Installing Python

- Python distribution is available for a wide variety of platforms such as Unix, Linux, Macintosh and Windows. We need to download only the binary code applicable for the platform and install Python.
- The most up-to-date and current source code, binaries, documentation, news, etc. is available on the official website of Python <https://www.python.org/>.

Installing Python in Windows:

Step 1 : Open any internet browser then type <http://www.python.org/downloads/> in address bar and Enter. The Home page will appear, (See Fig. 1.3).



Fig. 1.3: Home Page

Step 2 : Click on download to get the latest version of Python.

Looking for a specific release?
Python releases by version number:

Release version	Release date		Click for more
Python 3.7.1	2018-10-20	Download	Release Notes
Python 3.6.7	2018-10-20	Download	Release Notes
Python 3.5.6	2018-08-02	Download	Release Notes
Python 3.4.9	2018-08-02	Download	Release Notes
Python 3.7.0	2018-06-27	Download	Release Notes
Python 3.6.6	2018-06-27	Download	Release Notes
Python 2.7.15	2018-05-01	Download	Release Notes

[View older releases](#)

Fig. 1.4: Python Release Versions

Step 3 : Once Python version is downloaded in this case 3.9.6. Open the Python 3.9.6 version pack and double click on "Install Now", option for to start installation (See Fig. 1.5).

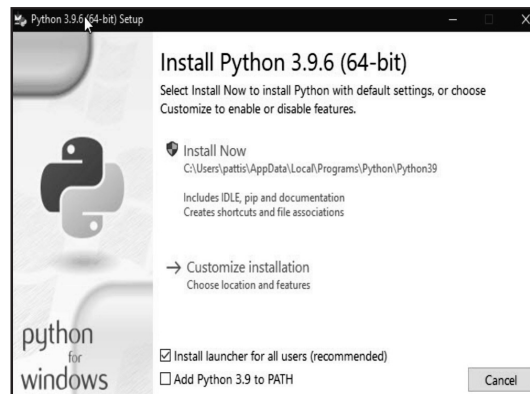


Fig. 1.5

Step 4 : When installation of starts it shows Setup Progress as shown in Fig. 1.6.

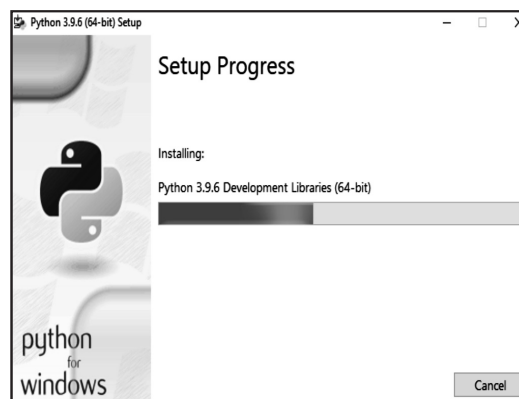


Fig. 1.6

Step 5 : After complete the installation click close button to finish installation.

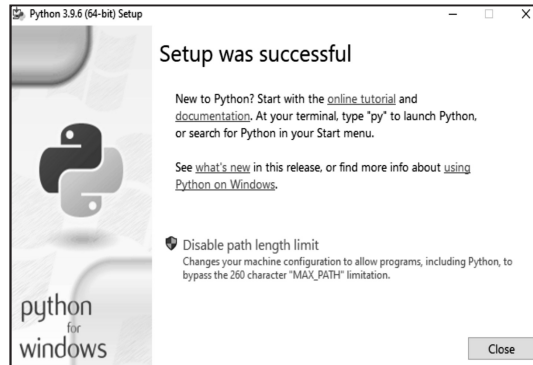


Fig. 1.7

Starting Python in different Modes:

1. Starting Python (Command Line):

- A Python script can be executed at command line also. This can be done by invoking the interpreter on the application.
- In command line mode, we type the Python programming program on the Python shell and the interpreter prints the result. The steps are given below:

Step 1 : Press Start button, (See Fig. 1.8).

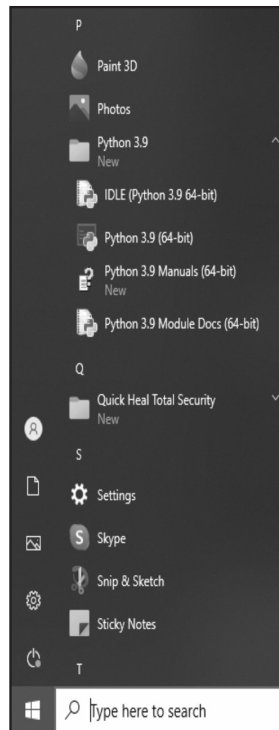
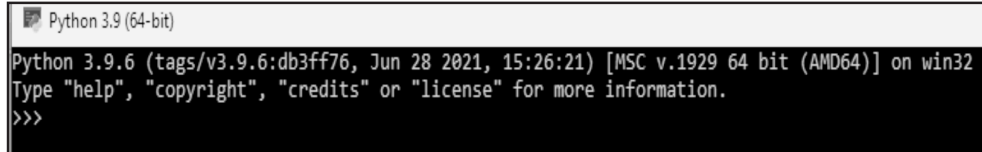


Fig. 1.8

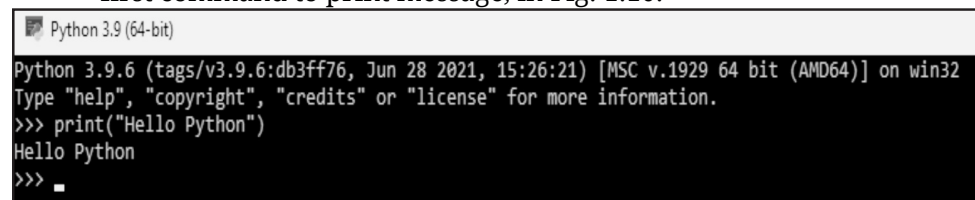
Step 2 : Click on All Programs and then click on Python 3.9 (64 bit) as shown in Fig. 1.9. We will see the Python interactive prompt in Python command line.



```
Python 3.9 (64-bit)
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Fig. 1.9

Python command prompt contains an opening message >>> called command prompt. The cursor at command prompt waits for to enter Python command. A complete command is called a statement. For example check first command to print message, in Fig. 1.10.



```
Python 3.9 (64-bit)
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello Python")
Hello Python
>>> █
```

Fig. 1.10

Step 3 : To exit from the command line of Python, use Ctrl+z or quit() followed by Enter.

2. Starting Python IDLE:

- When we install Python 3, we also get IDLE (Integrated Development Environment).
- IDLE includes a color syntax-highlighting editor, a debugger, the Python Shell, and a complete copy of Python 3's online documentation set.
- The steps are given below:

Step 1 : Press Start button and click on IDLE (Python 3.9, 64-bit) options.

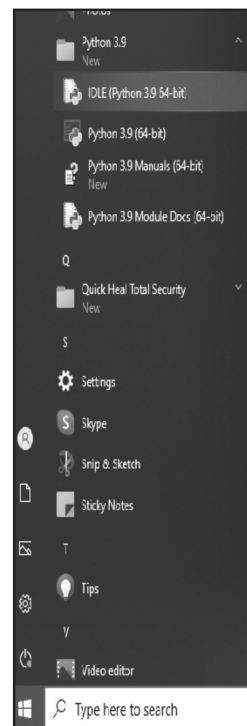


Fig. 1.11

Step 2 : We will see the Python interactive prompt i.e. interactive shell.

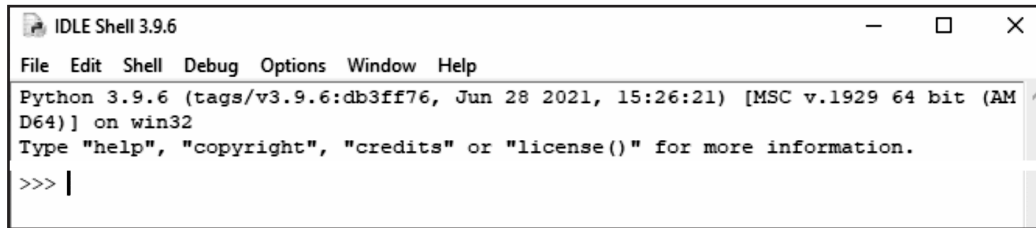


Fig. 1.12

Python interactive shell prompt contains opening message >>>, called shell prompt. A cursor is waiting for the command. A complete command is called a statement. When we write a command and press enter, the python interpreter will immediately display the result.

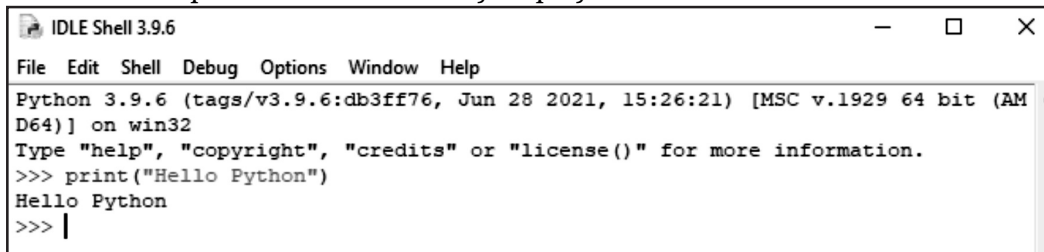


Fig. 1.13

Executing Python Programs Scripts:

- In Python IDLEs shell window, click on File, and select the New File or press Ctrl+N.

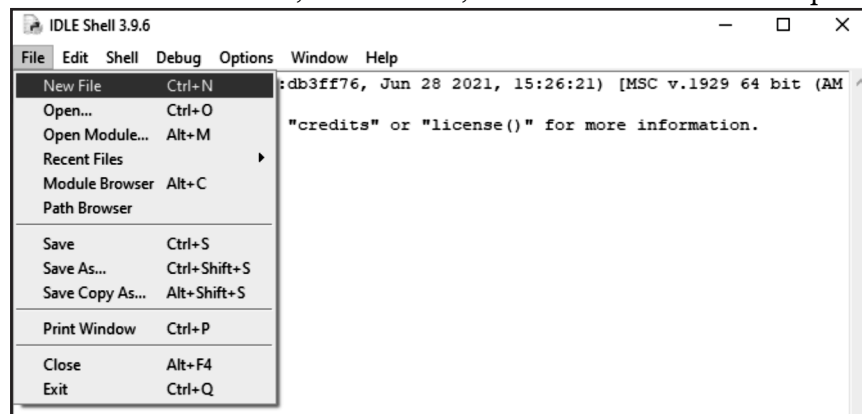


Fig. 1.14

- As soon as we click on New File, the window appears as shown in Fig. 1.15.

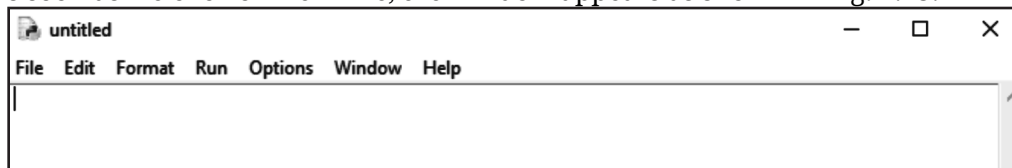


Fig. 1.15

- Write Python program in script mode, (See Fig. 1.16).

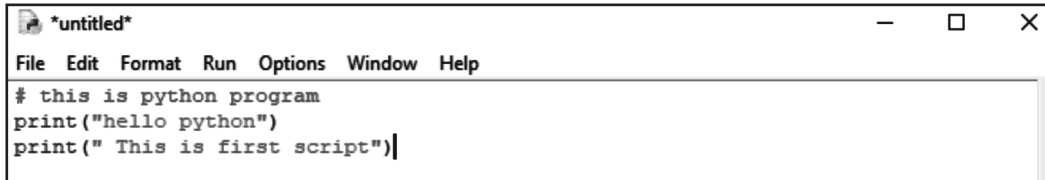


Fig. 1.16

- Save the above code with filename. By default, python interpreter will save it using the filename.py. Here, we save the script with file name test.py.

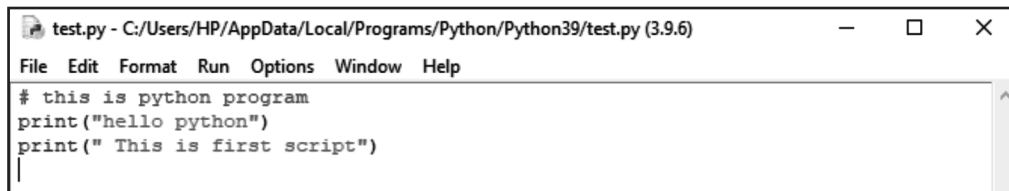


Fig. 1.17

- To run the Python program, click on Run and then Run Module option or we can press Ctrl+F5.

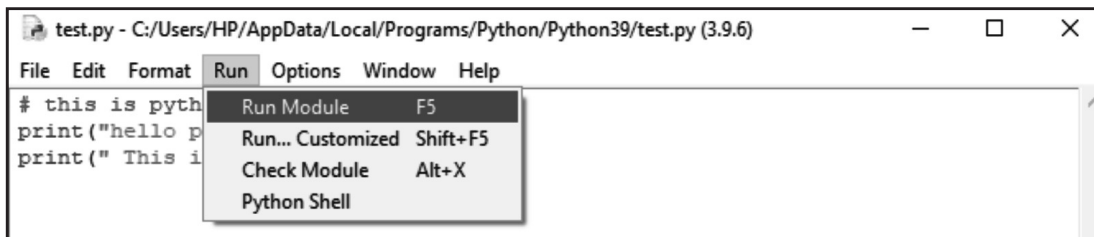


Fig. 1.18

- After clicking Run Module, we will get the output of program on Python shell.

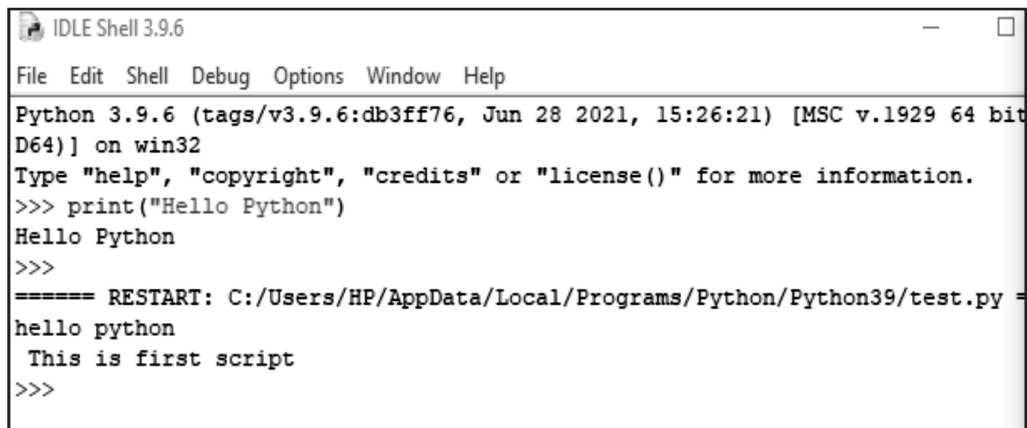


Fig. 1.19

1.1.5 Running Simple Python Program

- Python program is set of instructions. Python programs are typically organized with one statement per line. Before to write and run a Python program we understand structure of a Python program.
- Fig. 1.20 shows a typical program structure of Python programming.
- Python programming programs are structured as a sequence of statements. A Python statement is smallest program unit.
- Statements are the instructions that are written in a program to perform a specific task. A Python statement is a complete instruction executed by the Python interpreter.
- By default, the Python interpreter executes all statements sequentially, but we can change order of execution using control statements.

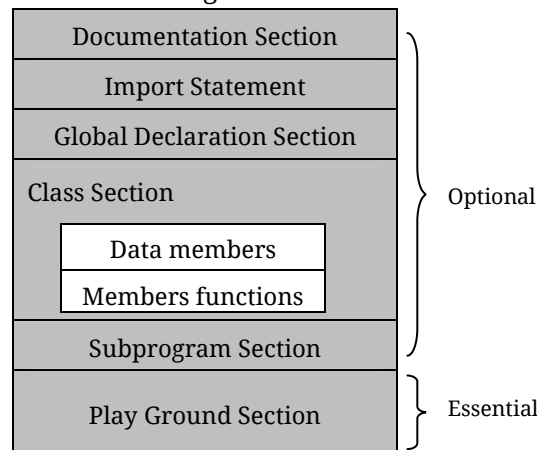


Fig. 1.20: Typical Program Structure of Python Programming

- Program structure of Python programming contains following sections:
 1. **Documentation Section** includes the comments that specify the purpose of the program.
A comments that is a non-executable statement which is ignored by the compiler while program execution. Python comments are written anywhere in the program.
 2. **Import Section** is used includes different built in or user defined modules.
 3. **Global Declaration Section** is used to define the global variables for the programs.
 4. **Class Section** describes the information about the user defined classes in the Python program. A class is a collection of data members and member functions called method that operate on data members.
 5. **Sub Program Section** includes use defined functions. The functions include the set of statements that need to be executed when the function is called form anywhere.

6. **Pay Ground Section** is the main section of Python program and the main section starts where the function calling.

- Following simple Python program shows structure of Python program which calculates area and circumference of circle using class:

```
#Python Program Structure.# Documentation section
import math                # import statement
radius=5                   # global declaration Section
class Circle():            # class section
    def getArea(self):
        return math.pi*radius*radius
    def getCircumference(self):
        return radius*2*math.pi
    def showradius():       # sub Program section
        print("Radius =",radius)
showradius()               # Playground Section
c=Circle()
print("Area =",c.getArea())
print("Circumference=",c.getCircumference())
```

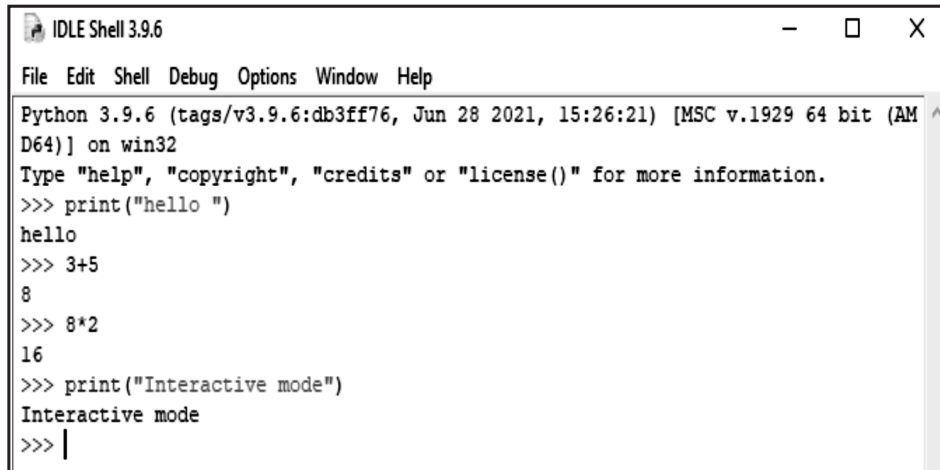
Output:

```
Radius = 5
Area = 78.53981633974483
Circumference = 31.41592653589793
```

- Python has two basic modes namely, normal and interactive.
 1. The **normal script mode** is the mode where the scripted and finished .py files are run in the Python interpreter.
 2. The **interactive mode** is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory.
- As new lines are fed into the interpreter, the fed program is evaluated both in part and in whole.

1. Interactive Mode:

- Interactive mode is used for quickly and conveniently running single line or blocks of code. Here's an example using the python shell that comes with a basic python installation.
- The ">>>" indicates that the shell is ready to accept interactive commands. For example, if we want to print the statement "Interactive Mode", simply type the appropriate code and hit enter.

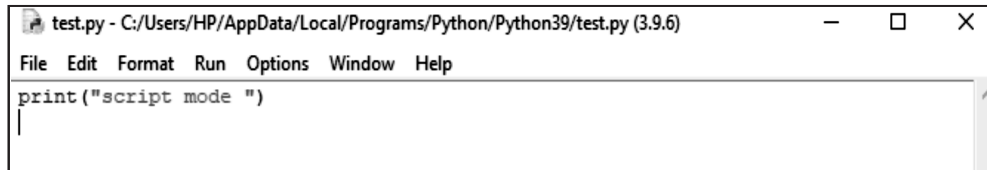


```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("hello ")
hello
>>> 3+5
8
>>> 8*2
16
>>> print("Interactive mode")
Interactive mode
>>> |
```

Fig. 1.21

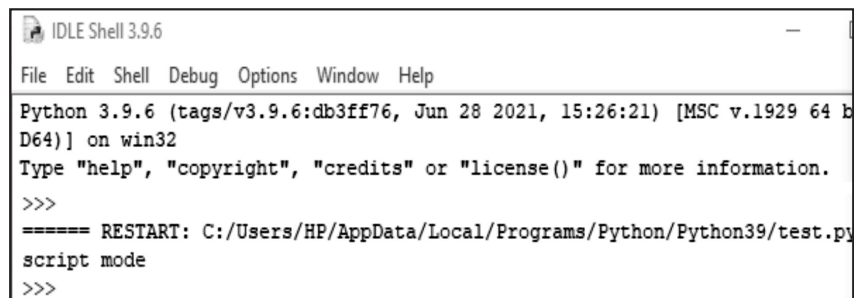
2. Script Mode:

- In the standard Python shell we can go to “File” → “New File” (or just hit Ctrl + N) to pull up a blank script to write the code. Then save the script with a “.py” extension.
- We can save it anywhere we want for now, though we may want to make a folder somewhere to store the code as we test Python out.
- To run the script, either select “Run” → “Run Module” or press F5.
- We should see something like the following, (See Fig. 1.22 (a) and 1.22 (b)).



```
test.py - C:/Users/HP/AppData/Local/Programs/Python/Python39/test.py (3.9.6)
File Edit Format Run Options Window Help
print("script mode ")
|
```

Fig. 1.22 (a)



```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/HP/AppData/Local/Programs/Python/Python39/test.py
script mode
>>>
```

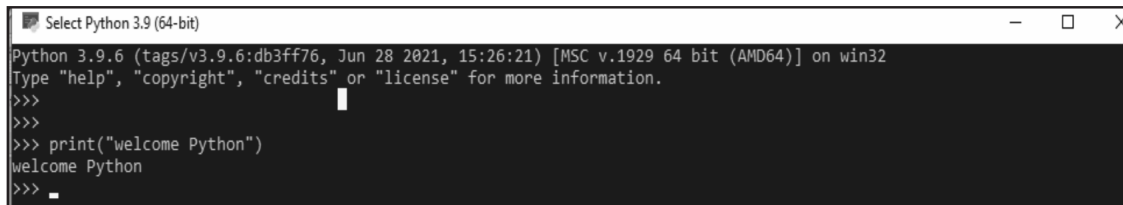
Fig. 1.22 (b)

Running Simple Python Script/Program to Display ‘Welcome’ Message:

- There are two modes for executing Python program namely Interactive mode programming and Script mode programming.
- In **interactive mode programming**, interpreter is invoked and the programmer can code statements directly to the interpreter without passing a script file as a parameter.
- In **script mode programming**, the complete script is written in an editor such as Notepad in Windows and then interpreter is invoked with a script parameter. It begins execution of the script and continues until the script is finished.

1. Interactive Mode Programming:

- Click on All Programs and then click on Python 3.9 (64-bit). We will see the Python interactive prompt in Python command line.
- This method invokes the interpreter without passing a script file and brings up the following prompt, (See Fig. 1.23).



```
Select Python 3.9 (64-bit)
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print("welcome Python")
welcome Python
>>> _
```

Fig. 1.23**2. Script Mode Programming:**

- Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.
- Let us write a simple Python program in a script. Python files have extension .py. Type the following source code in a test.py file:

```
print"Welcome, Python!"
```

- We assume that we have Python interpreter set in PATH variable. Now, try to run this program as follows:

```
$ python test.py
```

Output:

```
Welcome, Python!
```

- On Linux OS to execute a Python script modified test.py file:

```
#!/usr/bin/python
print"Hello, Python!"
```

- We assume that we have Python interpreter available in /usr/bin directory. Now, try to run this program as follows:

```
$ chmod +x test.py      # This is to make file executable
$ ./test.py
Hello, Python!
```

1.2 BASICS OF PYTHON

- In this section we will study various basic concepts of Python programming such as data types, variables, identifiers, control flow statements and so on.

1.2.1 Standard Data Types

- The type of data value that can be stored in an identifier/variable is known as its data type.
- The data type determines how much memory is allocated to store data and what operations can be performed on it.
- The data stored in memory can be of many types and are used to define the operations possible on them and the storage method for each of them.
- Python handles several data types to facilitate the needs of programmers and application developers for workable data.
- The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.
- Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.
- Data types in Python programming includes:
 1. **Numbers:** Represents numeric data to perform mathematical operations. Numbers can be integers (like 1 and 2), floats (like 1.1 and 1.2), fractions (like 1/2 and 2/3), or even complex numbers.
 2. **String:** Represents text characters, special symbols or alphanumeric data. String is sequence of Unicode characters.
 3. **List:** Represents sequential data that the programmer wishes to sort, merge etc.
 4. **Tuple:** Represents sequential data with a little difference from list.
 5. **Dictionary:** Represents a collection of data that associate a unique key with each value.
 6. **Boolean:** Represents truth values (true or false).
 7. **Set:** Set is an unordered collection of unique data items.
 8. **Dictionary:** Dictionary is an unordered collection of key-value pairs.
- To determine a variable's type in Python programming we can use the `type()` function. The value of some objects can be changed. Objects whose value can be changed are called mutable and objects whose value is unchangeable (once they are created) are called immutable.
- Let us see above data types in detail:
 1. **Boolean (bool Data Type):**
 - The simplest built-in type in Python is the bool data type, it represents the two values, True and False. Internally the true value is represented as 1 and false is 0.

Example:

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

2. None Data Type:

- Python defines a special variable None denoting a “null object”, which is convenient to use when a variable is available but its value is considered undefined.
- None in Python is used for defining null variables and objects. None is an instance of the NoneType class.
- None object is accessed through the built-in name None. It is a data type of the class NoneType object.

```
answer = None
    <may update answer from other data...>
if answer is None:
    quit = True
elif answer == 'quit':
    quit = True
else:
    quite = False
```

- To check if a variable answer is None or not, always use if answer is None or if answer is not None. Testing just if not answer is dangerous, because the test is true if answer is an empty string.
- The difference between the is and == operators: is tests for object identity, while == tests if two objects have the same value (i.e., the same content).

3. Number Data Types:

- Number data types store numeric values. Integers, floating point numbers and complex numbers are falls under Python numbers category. They are defined as int, float and complex in Python.

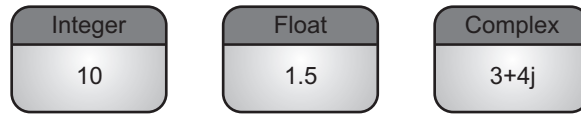


Fig. 1.24: Types of Numbers Data Type

(i) Integers (int Data Type):

- An int data type represents an integer number. An integer number is a number without any decimal or fractional point.
- For example, `a = 57`, here `a` is called the int type variable and stores integer value 57.
- These represent numbers in the range -2147483648 to 2147483647 .
- An integer is a whole number that can be positive (+) or negative (-). Integers can be of any length, it is only limited by the memory available.

Example: For number data types are integers.

```
>>> a=10
```

```
>>> a
```

```
10
```

- To determine the type of a variable `type()` function is used.

```
>>> type(a)
```

```
<class 'int'>
```

- In Python programming one can write integers in Hexadecimal (base 16), Octal (base 8) and Binary (base 2) formats by using one of the following prefixes to the integer.

Sr. No.	Prefix	Interpretation	Base
1.	'0b' or '0B'	Binary	2
2.	'0o' or '0O'	Octal	8
3.	'0x' or '0X'	Hexadecimal	16

Example: Integers in binary, octal and hexadecimal formats.

```
>>> print(0b10111011) # binary number
```

```
187
```

```
>>> print(0o10)      # octal number
```

```
8
```

```
>>> print(0xFF)      # hexadecimal number
```

```
255
```

(ii) Floating Point Numbers (Float Data Type):

- The float data type represents the floating point number. The floating point number is a number that contains a decimal point.
- Examples of floating point numbers, 0.5, -3.445, 330.44. For example, `num = 2.345`.

- Floating-point number or Float is a positive or negative number with a fractional part.
- A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. For example, 1 is integer, 1.0 is floating point number.
- One can append the character e or E followed by a positive or negative integer to specify scientific notation.

Example: Floating point number.

```
>>> x = 10.1
>>> x
10.1
y = -10.5
>>> y
-10.5
>>> print(72e3)
72000.0
>>> print(7.2e-3)
0.0072
```

(iii) Complex Numbers (Complex Data Type):

- A complex number is a number that is written in the form of $a+bj$. Here, a represents the real part of the number and b represents the imaginary part of the number.
- The suffix J or j after b represents the square root value of -1 . The part a and b may contain the integers or floats. For example, $3+5j$, $0.2+10.5j$ are complex numbers.
- Complex numbers are written in the form, $x + yj$, where x is the real part and y is the imaginary part.

Example: Complex number.

```
>>> x = 3+4j
>>> print(x.real)
3.0
>>> print(x.imag)
4.0
```

4. String Data Type:

- String is a collection of group of characters. Strings are identified as a contiguous set of characters enclosed in single quotes (' ') or double quotes (" ").
- Any letter, a number or a symbol could be a part of the string. Strings are unchangeable (immutable). Once a string is created, it cannot be modified.

- Strings in python support Unicode characters. The default encoding for Python source code is UTF-8. So, we can also say that String is a sequence of Unicode characters.
- Strings are ordered. Strings preserved the order of characters inserted.

Example: For string data type.

```
>>> s1="Hello"           # string in double quotes
>>> s2='Hi'              # string in single quotes
>>> s3="Don't open the door" # single quote string in double quotes
>>> s4='I said "yipee"'    # double quote string in single quotes
>>> s1
'Hello'
>>> s2
'Hi'
>>> s3
"Don't open the door"
>>> s4
'I said "yipee"'
>>>
```

5. List Data Type:

- Lists are the most versatile of Python's compound data types. List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible.
- List can contain heterogeneous values such as integers, floats, strings, tuples, lists and dictionaries but they are commonly used to store collections of homogeneous objects.
- The list data type in Python programming is just like an array that can store a group of elements and we can refer to these elements using a single name.
- Declaring a list is pretty straight forward. Items separated by commas (,) are enclosed within square brackets [].

Example: For list.

```
>>> first=[10, 20, 30]           # homogenous values in list
>>> second=["One","Two","Three"] # homogenous values in list
>>> first
[10, 20, 30]
>>> second
['One', 'Two', 'Three']
```

```
>>> third=[10,"one",20,"two"]           # heterogeneous values in list
>>> third
[10, 'one', 20, 'two']
>>> first + second                       # prints the concatenated lists
[10, 20, 30, 'One', 'Two', 'Three']
```

- Lists are mutable which means that value of elements of a list can be altered by using index.

Example: For list with updation/alteration/modification.

```
>>> first=[10, 20, 30]
>>> first[2]                             # print second value in the list
30
>>> first[2]=50                           # change second value in the list
>>> first
[10, 20, 50]
>>> first[2]                             # print second value in the list
50
>>> print (first*2)                       # prints the list two times
[10, 20, 30, 10, 20, 30]
```

List and Strings:

- A string is a sequence of characters and list is a sequence of values, but a list of characters is not same as string. We can convert string to a list of characters.

Example: For conversion of string to a list.

```
>>> p="Python"
>>> p
'Python'
>>> l=list(p)
>>> l
['P', 'y', 't', 'h', 'o', 'n']
```

6. Tuple Data Type:

- Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified.
- Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically. It is defined within parentheses () where items are separated by commas (,).

- A tuple data type in python programming is similar to a list data type, which also contains heterogeneous items/elements.

Example:

- ```
tuple_obj=(70,2.50, "tybca")
```
- Tuples are immutable objects whose contents cannot be altered:

```
words = ('flower', 'homophone', 'flour')
words[1] = 'and' # illegal - Python issues an error message
```

**7. Dictionary Data Type:**

- Dictionary data type in Python is an unordered collection of key-value pairs. Dictionaries in Python work like associative arrays or hashes found in Perl and consist of key-value pairs.
- When we have the large amount of data, the dictionary data type is used. The dictionary data type is mutable in nature which means we can update modify/update any value in the dictionary.
- Items in dictionaries are enclosed in curly braces { } and separated by the comma (.). A colon (:) is used to separate key from value. Values can be assigned and accessed using square braces ([]).

**Example:** For dictionary data type.

```
>>> dic1={1:"First","Second":2}
>>> dic1
{1: 'First', 'Second': 2}
>>> type(dic1)
<class 'dict'>
>>> dic1[3]="Third"
>>> dic1
{1: 'First', 'Second': 2, 3: 'Third'}
>>> dic1.keys()
dict_keys([1, 'Second', 3])
>>> dic1.values()
dict_values(['First', 2, 'Third'])
>>>
```



---

## 1.2.2 Variables

---

- A variable is like a container that stores values that we can access or change. It is a way of pointing to a memory location used by a program.
- We can use variables to instruct the computer to save or retrieve data to and from this memory location.
- A variable is a name given to a location in the computer's memory location, where the value can be stored that can be used in the program.
- When we create a variable, some space in the memory is reserved or allocated for that variable to store a data value in it.
- The size of the memory reserved by the variable depends on the type of data it is going to hold. The period of time that a variable exists is called its lifetime.
- The variable is so called because its value may vary during the time of execution, but at a given instance only one value can be stored in it.

### Variable Declaration:

- A variable is an identifier that holds a value. In programming, we say that we assign a value to a variable. Technically speaking, a variable is a reference to a computer memory, where the value is stored.
- Basic rules to declare variables in python programming language:
  1. Variables in Python can be created from alphanumeric characters and underscore ( `_` ) character.
  2. A variable cannot begin with a number.
  3. The variables are case sensitive. Means Amar is differ the 'AMAR' are two separate variables.
  4. Variable names should not be reserved word or keyword.
  5. No special characters are used except underscore ( `_` ) in variable declaration.
  6. Variables can be of unlimited length.
- Python variables do not have to be explicitly declared to reserve memory space. The variable is declared automatically when the variable is initialized, i.e., when we assign a value to the variable first time it is declared with the data type of the value assigned to it.
- This means we do not need to declare the variables. This is handled automatically according to the type of value assigned to the variable. The equal sign ( `=` ) i.e., the assignment operator is used to assign values to variables.
- The operand to the left of the `=` operator is the name of the variable and the operand to the right of the `=` operator is the literal value or any other variable value that is stored in the variable.

**Syntax:** `variable=value`

**Example:** For variable.

```
>>> a=10
>>> a
10
>>>
```

- Python language allows assigning a single value to several variables simultaneously.

**Example:** a=b=c=1

All above three variables are assigned to same memory location, when integer object is created with value 1.

- Multiple objects can also have assigned to multiple variables:

**Example:** a, b, c = 10, 5.4, "hello"

In above example Integer object a assigned with value 10, float object b assigned with value 5.4 and string object c assigned with value "hello".

**Example:** If x, y, z are defined as three variable in a program, then x = 10 will store the value 10 in the memory location named as x, y = 5 will store the value 5 in the memory location named as y and x + y will store the value 15 in the memory location named as z (as a result after computation of x + y).

```
>>> x = 10
>>> y = 5
>>> name = "Python"
>>> z = x + y
>>> print(x); print(y); print(name); Print (z)
10
5
Python
15
```

### 1.2.3 Constants

- A constant is a type of variable whose value cannot be changed during the execution of program.
- In Python, constants are usually declared and assigned in a module. For declaring and assigning value to a constant consider following example:
  - Create a constant.py:

```
PI = 3.14
GRAVITY = 9.8
```
  - Create a main.py:

```
import constant
print(constant.PI)
print(constant.GRAVITY)
3.14
9.8
```

**Literals in Python:**

- Literals are also constant values in Python, which are fixed. A literal refers to the fixed value that directly appears in the program. Literals can be defined as, a data that is given in a variable or constant.
- Literals are numbers or strings or characters that appear directly in a program. Python support the following literals:
  - **String Literals:** "hello" , '12345'
  - **Int Literals:** 0, 1, 2, -1, -2
  - **Long Literals:** 89675L
  - **Float Literals:** 3.14
  - **Complex Literals:** 12j
  - **Boolean Literals:** True or False
  - **Special Literals:** None
  - **Unicode Literals:** u"hello"
  - **List Literals:** [], [5, 6, 7]
  - **Tuple Literals:** (), (9), (8, 9, 0)
  - **Dict Literals:** {}, {'x':1}
  - **Set Literals:** {8, 9, 10}

**1. String Literals:**

- String literals can be formed by enclosing a text in the quotes. We can use both single quote ( ' ... ' ) as well as double quotes for ( " ... " ) a string.
- In simple words, a string literal is a collection of consecutive characters enclosed within a pair of single or double quotes.

---

**Example:** For string literal.

```
Fname= 'Hello'
Lname="Python"
print(Fname)
print(Lname)
```

**Output:**

```
Hello
Python
```

---

**2. Numeric Literals:**

- Numeric literals are immutable. Numeric literals comprise number or digits form 0 to 9.
  - Numeric literals can belong to following four different numerical types.
-

| int (Signed Integers)                                                                        | long (Long Integers)                                                                    | float (Floating Point)                                               | complex (Complex)                                                                                                   |
|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|----------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| Numbers (can be both positive (+) and negative (-)) with no fractional part.<br>Example: 100 | Integers of unlimited size followed by lower-case or uppercase L.<br>Example: 87032845L | Real numbers with both integer and fractional part.<br>Example: 26.2 | In the form of a+bj where a forms the real part and b forms the imaginary part of complex number.<br>Example: 3.14j |

### 3. Boolean Literals:

- A Boolean literal can have any of the two values namely, True or False.

**Example:** For Boolean literal.

```
>>> 5<=2
False
>>> 3<9
True
>>>
```

### 4. Special Literals:

- Python contains one special literal i.e., None. It is special constant in Python programming that represent the absence of a value or NULL value.
- None is used to specify to that field that is not created. It is also used for end of lists in Python.

**Example:** For special literal.

```
>>> val1=10
>>> val2=None # N is in uppercase here
>>> val1
10
>>> val2
>>> print (val2)
None
>>>
```

### 5. Literal Collections:

- Collections such as tuples, lists and dictionary are used in Python.

#### (i) List:

- List contains items of different data types. Lists are mutable i.e., modifiable. The values stored in list are separated by commas (,) and enclosed within a square brackets ([]). We can store different type of data in a list.
- Value stored in a list can be retrieved using the slice operator ([] and [:]). The plus sign (+) is the list concatenation and asterisk (\*) is the repetition operator.

**(ii) Tuple:**

- Tuple is used to store the sequence of immutable python objects.
- A tuple can be created by using () brackets and separated by commas (,).

**(iii) Dictionary:**

- The dictionary in Python is a collection of key value pairs created using { }.
  - The key and value are separated by a colon (:) and the elements/items are separated by commas (,).
- 

**Example:** For literal collections.

```
create list
>>> numbers=[1,2,3,4,5,6,7]
>>> print(numbers)
create tuples
>>> list=('a','b','c')
>>> print(list)
create dictionary
>>> list2={'fname':'vijay', 'lname':'patil'}
>>> print(list2)
```

**Output:**

```
[1,2,3,4,5,6,7]
('a','b','c')
{'fname':'vijay', 'lname':'patil'}
```

---

**6. Value and Type of Literals:**

- Programming languages contain data in terms of input and output and any kind of data can be presented in terms of value.
- Value can be of numbers, strings or characters. To know the exact type of any value, python provides in-built method called type.

**Syntax:** type(value)

---

**Example:** For value and type literals.

```
>>> type('hello python')
<class 'str'>
>>> type('a')
<class 'str'>
>>> type(123)
<class 'int'>
>>> type(11.22)
<class 'float'>
```

---

---

### 1.2.4 Identifiers

---

- A Python identifier is a name given to a function, class, variable, module or other objects that is used in Python program.
- All identifiers must obey the following rules:
  1. An identifier can be a combination of uppercase letters, lowercase letters, underscores, and digits (0-9). Examples include, Name, myClass, Emp\_Salary, var\_1, \_Address and print\_hello\_world.
  2. We can use underscores to separate multiple words in the identifier. For example, Emp\_Salary.
  3. An identifier starts with a letter which can be alphabet (either lowercase or uppercase), underscore (\_).
  4. Identifiers can be of any length.
  5. Identifiers cannot start with digit and must not contain any space or tabs. Example include, 2variable, 10ID.
  6. We cannot use Python keywords as identifiers.
  7. Special characters such as %, @, and \$ are not allowed within identifiers. Example include, \$Money, @salary.
  8. Python is a case-sensitive language and this behavior extends to identifiers. Thus, identifier Age and age are two distinct identifiers in Python.
- Example of valid identifiers includes: Circle\_Area, EmpName, Student, Sum, Salary2018, \_PhoneNo .
- Example of invalid identifiers includes: !count, 4marks, %Loan, 2018Salary.

---

### 1.2.5 Reserved Words

---

- Python keywords are reserved words with that have special meaning and functions. The keywords are predefined words with specific meaning in the Python programs.
- Keywords should not be used as variable name, constant, function name, or identifier in the program code.
- In Python keywords are case sensitive. Keywords are used to define the syntax and structure of the programming language.
- Following table lists keywords in Python programming:

|        |         |        |        |        |          |
|--------|---------|--------|--------|--------|----------|
| and    | as      | assert | break  | class  | continue |
| def    | del     | else   | elif   | except | exec     |
| false  | finally | for    | from   | global | if       |
| import | in      | is     | lambda | none   | not      |
| or     | pass    | print  | raise  | return | true     |
| try    | while   | with   | yield  |        |          |

## 1.2.6 Lines, Indentation and Multi-Line Statements

- Python statements are the code instructions that are executed by the Python interpreter. Python statements are usually written in a single line.
- Instructions that a Python interpreter can execute are called statements. For example, `a = 1` is an assignment statement.
- In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (`\`).
- Most of the programming languages like C, C++, Java use braces `{ }` to define a block of code. Python uses indentation.
- Indentation helps to convey a better structure of a program to the readers. It is used to clarify the link between control flow constructs such as conditions or loops, and code contained within and outside of them.
- A code block (body of a function, loop etc.) starts with indentation and ends with the first un-indented line. The amount of indentation is up to us, but it must be consistent throughout that block.
- Generally, four whitespaces are used for indentation and is preferred over tabs, (See Fig. 1.25).

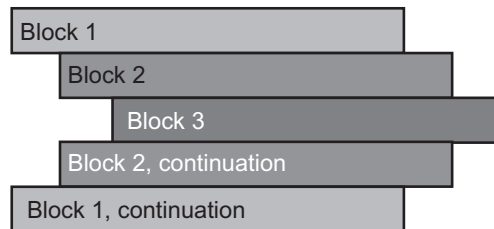


Fig. 1.25: Indentation in Python

**Example:** For indentation in python.

```
>>> for i in range(1,11):
 print(i)
 if i==5:
 break
```

**Output:**

```
1
2
3
4
5
```

- The enforcement of indentation in Python makes the code look neat and clean. In Python all the continuous lines indented with same number of spaces would form a block.

**Multi-Line Statements:**

- **Explicit Line Continuation:** Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue.

**Example:**

```
stat = line_one + \
line_two + \
line_three
```

- **Implicit Line Continuation:** Statements contained within the [], {}, or () brackets do not need to use the line continuation character.

**Example:**

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

### 1.2.7 Comments

- Comment writing is a good programming practice. Comments are non-executable chunks of line in a program.
- Comments are meant for computer programmers for better understanding a program. Python interpreter ignores the comment in the program.
- In Python, there are two types of comments namely, single line comment and multi-line comment.

**1. Single Line Comment (#):**

- Single line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

**Example 1:** For single line comment.

```
print is a statement
print('Hello Python')
```

**Example 2:** For single line comment.

```
print('Hello Python') # print is a statement
```

- When the python interpreter sees #, it ignores all the text after # on the same line.

**2. Multiple Line Comments ("""):** 

- In some situations, multiline documentation is required for a program. If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. Another way of doing this is to use quotation marks, either ''' or """".
- Similarly, when it sees the triple quotation marks ''' it scans for the next ''' and ignores any text in between the triple quotation marks.

**Example 1:** For multi-line comment.

```
'''This is first python program
Print is a statement'''
```



**Example 2:** For multi-line comment.

```
"""This is also a perfect example of
multi-line comments"""
```

---

### 1.2.8 Input/Output Functions

---

- A python program needs to interact with the user to accomplish the desired task or result this can be achieved using Input-Output functions.
- The input() function helps to enter data at run time by the user and the output function print() is used to display the result of the program on the screen after execution.
- Input means the data entered by the user/programmer in the form of program. In python, the input() function is used to accept an input from a user.

**Syntax:** variable\_name=input()                      # without any argument  
             variable\_name=input('String')        # with argument

---

**Example:** For input in Python.

```
>>> input()
Hello python
Hello python
>>> x= input ("Enter data:")
Enter data: 11.22
>>> print(x)
11.22
```

---

- Output means the data comes from computer after processing. In Python programming the print() function display the input value on screen.

**Syntax:** print(expression/constant/variable)

---

**Example:** For output in python.

```
>>> print ("Hello")
Hello
>>> a="Hello"
>>> b="Python"
>>> print(a+b)
```

**Output:**

```
HelloPython
```

---

**Formatting Output:**

- Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method. This method is visible to any string object.

```
x = 10; y = 20
print('The value of x is {} and y is {}'.format(x,y))
#Output: The value of x is 10 and y is 20
```

- Here the curly braces `{}` are used as placeholders. The output is printed according to the order given as follows:

```
print('I love {} and {}'.format('apple','milk'))
Output: I love apple and milk
print('I love {} and {}'.format('apple','milk'))
Output: I love milk and apple
```

### 1.2.9 Operations on Data using Operators

---

- Operators are used to perform operations on variables and values. Operators are the constructs which can manipulate the value of operands.
- Consider the expression  $4 + 5 = 9$ . Here, 4 and 5 are called operands and  $+$  is called operator.
- In Python, the operators can be unary operators or binary operator.

**1. Unary Operators:**

- Unary operators are operators with only one operand. These operators are basically used to provide sign to the operand.  $+$ ,  $-$ ,  $\sim$  are called unary operators.

**Syntax:** operator operand

---

**Example:**

```
>>> x=10
>>> +x
10
>>> -x
-10
>>> ~x
-11
```

---

- The invert ( $\sim$ ) operator returns the bitwise inversion of long integer arguments. Inversion of  $x$  can be computed as  $\sim (x + 1)$ .

**2. Binary Operators:**

- Binary operators are operators with two operands that are manipulated to get the result. They are also used to compare numeric values and string values.

**Syntax:** operand1 operator operand2

---

- Binary operators are: \*\*, \*, /, %, +, -, <<, >>, &, |, ^, <, >, <=, >=, ==, !=, <>.

**Example:**

```
>>> x=10
>>> y=20
>>> x+y
>>> -x
30
>>> 2+3
5
```

**Expression in Python:**

- An expression is nothing but a combination of operators, variables, constants and function calls that results in a value.
- In other words, an expression is a combination of literals, variables and operators that Python evaluates to produce a value.

**For examples:** 1 + 8

(3 \* 9) / 5

a \* b + c \* 3

- Python operators allow programmers to manipulate data or operands. The Python language provides a rich set of operators.
- The types of operators supported by Python includes Arithmetic operators, Assignment operators, Relational or Comparison operators, Logical operators, Bitwise operators, Identity operators and Membership operators.

**Arithmetic Operators:**

- The arithmetic operators perform basic arithmetic operations like addition, subtraction, multiplication and division. All arithmetic operators are binary operators because they can perform operations on two operands.
- There are seven arithmetic operators provided in Python programming such as addition, subtraction, multiplication, division, modulus, floor division, and exponential operators.
- Assume variable a holds the value 10 and variable b holds the value 20.

| Sr. No. | Operator Symbol | Operator Name  | Description                                                                  | Example        |
|---------|-----------------|----------------|------------------------------------------------------------------------------|----------------|
| 1.      | +               | Addition       | Adds the value of the left and right operands.                               | >>> a+b<br>30  |
| 2.      | -               | Subtraction    | Subtracts the value of the right operand from the value of the left operand. | >>> b-a<br>10  |
| 3.      | *               | Multiplication | Multiplies the value of the left and right operand.                          | >>> a*b<br>200 |
| 4.      | /               | Division       | Divides the value of the left operand by the right operand.                  | >>> b/a<br>2.0 |

*contd. ...*

|    |    |                |                                                                                            |                 |
|----|----|----------------|--------------------------------------------------------------------------------------------|-----------------|
| 5. | ** | Exponent       | Performs exponential calculation.                                                          | >>> a**2<br>100 |
| 6. | %  | Modulus        | Returns the remainder after dividing the left operand with the right operand.              | >>> a%b<br>10   |
| 7. | // | Floor Division | Division of operands where the solution is a quotient left after removing decimal numbers. | >>> b//a<br>2   |

#### Assignment Operators (Augmented Assignment Operators):

- Assignment operators are used in Python programming to assign values to variables. The assignment operator is used to store the value on the right-hand side of the expression on the left-hand side variable in the expression.
- For example, a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.
- There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.
- Following table shows assignment operators in Python programming:

| Sr. No. | Operator | Description                                                                                 | Example                                 |
|---------|----------|---------------------------------------------------------------------------------------------|-----------------------------------------|
| 1.      | =        | Assigns values from right side operands to left side operand.                               | c = a + b assigns value of a + b into c |
| 2.      | +=       | It adds right operand to the left operand and assign the result to left operand.            | c += a is equivalent to c = c + a       |
| 3.      | -=       | It subtracts right operand from the left operand and assign the result to left operand.     | c -= a is equivalent to c = c - a       |
| 4.      | *=       | It multiplies right operand with the left operand and assign the result to left operand.    | c *= a is equivalent to c = c * a       |
| 5.      | /=       | It divides left operand with the right operand and assign the result to left operand.       | c /= a is equivalent to c = c / a       |
| 6.      | %=       | It takes modulus using two operands and assign the result to left operand.                  | c %= a is equivalent to c = c % a       |
| 7.      | **=      | Performs exponential (power) calculation on operators and assign value to the left operand. | c **= a is equivalent to c = c ** a     |
| 8.      | //=      | Performs exponential (power) calculation on operators and assign value to the left operand. | c //= a is equivalent to c = c // a     |

**Relational or Comparison Operators:**

- Comparison operators in Python programming are binary operators and used to compare values. Relational operators either return True or False according to the condition.
- Assume variable a holds the value 10 and variable b holds the value 20.

| Sr. No. | Operator                               | Description                                                                                                       | Example             |
|---------|----------------------------------------|-------------------------------------------------------------------------------------------------------------------|---------------------|
| 1.      | ==<br>(Equality Operator)              | If the values of two operands are equal, then the condition becomes true.                                         | >>> (a==b)<br>False |
| 2.      | !=<br>(Not Equality Operator)          | If values of two operands are not equal, then condition becomes true.                                             | >>> (a!=b)<br>True  |
| 3.      | ><br>(Greater Than Operator)           | If the value of left operand is greater than the value of right operand, then condition becomes true.             | >>> (a>b)<br>False  |
| 4.      | <<br>(Less Than Operator)              | If the value of left operand is less than the value of right operand, then condition becomes true.                | >>> (a<b)<br>True   |
| 5.      | >=<br>(Greater Than to Equal Operator) | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | >>> (a>=b)<br>False |
| 6.      | <=<br>(Less Than Equal to Operator)    | If the value of left operand is less than or equal to the value of right operand, then condition becomes true.    | >>> (a<=b)<br>True  |

**Logical Operators:**

- The logical operators in Python programming are used to combine one or more relational expressions that result in complex relational operations.
- The result of the logical operator is evaluated in the terms of True or False according to the result of the logical expression.
- Logical operators perform logical AND, logical OR and logical NOT operations. These operations are used to check two or more conditions. The resultant of this operator is always a Boolean value (True or False).

- Assume variable a holds True and variable b holds False then:

| Sr. No. | Operator                      | Description                                                          | Example               |
|---------|-------------------------------|----------------------------------------------------------------------|-----------------------|
| 1.      | AND<br>(Logical AND Operator) | If both the operands are true then condition becomes true.           | (a and b) is False.   |
| 2.      | OR<br>(Logical OR Operator)   | If any of the two operands are non-zero then condition becomes true. | (a or b) is True.     |
| 3.      | NOT<br>(Logical NOT Operator) | Used to reverse the logical state of its operand.                    | Not(a and b) is True. |

#### Bitwise Operators:

- Bitwise operators acts on bits and performs bit by bit operation. Python programming provides the bit manipulation operators to directly operate on the bits or binary numbers directly.
- When we use bitwise operators on the operands, the operands are firstly converted to bits and then the operation is performed on the bit directly.
- Bitwise operators in Python programming are binary operators and unary operators that can be operated on two operands or one operand.
- Following table shows bitwise operators assume a=10 (1010) and b=4 (0100).

| Sr. No. | Operator                    | Description                                                                                                                 | Example                       |
|---------|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| 1.      | &<br>(Bitwise AND Operator) | This operation performs AND operation between operands. Operator copies a bit, to the result, if it exists in both operands | a&b = 1010 & 0100 = 0000 =0   |
| 2.      | <br>(Bitwise OR Operator)   | This operation performs OR operation between operands. It copies a bit, if it exists in either operand.                     | a b = 1010   0100 = 1110 = 14 |
| 3.      | ^<br>(Bitwise XOR Operator) | This operation performs XOR operations between operands. It copies the bit, if it is set in one operand but not both.       | a^b=1010 ^ 0100 = 1110 =14    |

*contd. ...*

|    |                                                         |                                                                                               |                                                         |
|----|---------------------------------------------------------|-----------------------------------------------------------------------------------------------|---------------------------------------------------------|
| 4. | <code>~</code><br>(Bitwise Ones Complement Operator)    | It is unary operator and has the effect of 'flipping' bits i.e. opposite the bits of operand. | <code>~a = ~ 1010 = 0101</code>                         |
| 5. | <code>&lt;&lt;</code><br>(Bitwise Left Shift Operator)  | The left operand's value is moved left by the number of bits specified by the right operand.  | <code>a&lt;&lt;2 = 1010 &lt;&lt; 2 = 101000 = 40</code> |
| 6. | <code>&gt;&gt;</code><br>(Bitwise Right Shift Operator) | The left operand's value is moved right by the number of bits specified by the right operand. | <code>a&gt;&gt;2 = 1010 &gt;&gt; 2 = 0010 = 2</code>    |

- Following table shows the outcome of each operations:

| A | B | A&B | A B | A^B | ~A |
|---|---|-----|-----|-----|----|
| 0 | 0 | 0   | 0   | 0   | 1  |
| 0 | 1 | 0   | 1   | 1   | 1  |
| 1 | 0 | 0   | 1   | 1   | 0  |
| 1 | 1 | 1   | 1   | 0   | 0  |

#### Identity Operators:

- Sometimes, in Python programming, need to compare the memory address of two objects; this is made possible with the help of the identity operator.
- Identity operators are used to check whether both operands are same or not. Python provides 'is' and 'is not' operators which are called identity operators and both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

| Sr. No. | Operator | Description                                                                                                | Example                                                                           |
|---------|----------|------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| 1.      | is       | Return true, if the variables on either side of the operator point to the same object and false otherwise. | <pre>&gt;&gt;&gt; a=3 &gt;&gt;&gt; b=3 &gt;&gt;&gt; print(a is b) True</pre>      |
| 2.      | is not   | Return false, if the variables on either side of the operator point to the same object and true otherwise. | <pre>&gt;&gt;&gt; a=3 &gt;&gt;&gt; b=3 &gt;&gt;&gt; print(a is not b) False</pre> |

#### Example 1:

```
>>> a=3
>>> b=3.5
>>> print(a is b)
```

```
False
>>> a=3
>>> b=4
>>> print(a is b)
False
>>> a=3
>>> b=3
>>> print(a is b)
True
>>>
```

---

**Example 2:**

```
>>> x=10
>>> print(type(x) is int)
True
>>>
```

---

**Example 3:**

```
>>> x2 = 'Hello'
>>> y2 = 'Hello'
>>> print(x2 is y2)
True
>>> x3 = [1,2,3]
>>> y3 = [1,2,3]
>>> print(x3 is y3)
False
>>> x4=(1,2,3)
>>> y4=(1,2,3)
>>> print(x4 is y4)
False
```

---

- In this example x3 and x4 are equal list but not identical. Interpreter will locate them separately in memory even though they have equal content. Similarly x4 and y4 are equal tuples but not identical.

**Membership Operators:**

- The membership operators in Python programming are used to find the existence of a particular element in the sequence and used only with sequences like string, tuple, list, dictionary etc.
  - Membership operators are used to check an item or an element that is part of a string, a list or a tuple. A membership operator reduces the effort of searching an element in the list.
-



- Python provides 'in' and 'not in' operators which are called membership operators and used to test whether a value or variable is in a sequence.

| Sr. No. | Operator | Description                                                                                     | Example                                                                            |
|---------|----------|-------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| 1.      | in       | True if value is found in list or in sequence, and false if item is not in list or in sequence. | <pre>&gt;&gt;&gt; x="Hello World" &gt;&gt;&gt; print('H' in x) True</pre>          |
| 2.      | not in   | True if value is not found in list or in sequence, and false if item is in list or in sequence. | <pre>&gt;&gt;&gt; x="Hello World" &gt;&gt;&gt; print("Hello" not in x) False</pre> |

**Example:**

```
>>> x="Hello World" # using string
>>> print("H" in x)
True
>>> print("Hello" not in x)
False
>>> y={1:"a",2:"b"} # using dictionary
>>> print(1 in y)
True
>>> print("a" in y)
False
>>> z=("one","two","three") # using tuple
>>> print ("two" in z)
True
```

**1.2.10 Dry Run**

- A dry run is the process of a programmer manually working through their code to trace the value of variables.
- Dry run is a manual way of testing for a process or algorithms for its correctness and functionality.
- In dry run method a table is prepared with different columns for each variable used in the program and the values of each variable are updated in the table as we proceed through the algorithm, one step at a time.
- Success of the dry run method depends upon the ability to step through the instructions exactly as a computer would execute them. Let us consider the following program codes in Python:

```
A = 60
B = 30
C = A + B
print(C)
```

- From the above program code if we proceed with a dry run, then the value of C will be 90. Given following is a new example with seven steps and a dry run table is made for it.

**Step 1 :** `x = 5`

**Step 2 :** `y = 10`

**Step 3 :** `d = 0`

**Step 4 :** `d = x + y + (x * y)`

**Step 5 :** `y = y + 6`

**Step 6 :** `d = d + y`

**Step 7 :** `print (x, y, d)`

- The table is as given below:

|               | <b>x</b> | <b>y</b> | <b>d</b> |
|---------------|----------|----------|----------|
| Initial value | 5        | 10       | 0        |
| After Step 4  | 5        | 10       | 65       |
| After Step 5  | 5        | 16       | 65       |
| After Step 6  | 5        | 16       | 81       |

- So, the result that will be displayed on the screen will be 5, 16 and 81.

### 1.2.11 Simple Input and Output

- Any information or data sent to the computer from the user through the keyboard is called input. The information produced by the computer to the user is called output.
- To be useful, a program usually needs to communicate with the outside world by obtaining input data from the user and displaying result data back to the user.
- Input may come directly from the user via the keyboard, or from some external source like a file or database. Output can be displayed directly to the console or IDE, to the screen via a Graphical User Interface (GUI).
- Programs often need to obtain data from the user, usually by way of input from the keyboard. The simplest way to accomplish this in Python is with `input()` function.
- The `input()` function reads a line of input from the keyboard. In python, `input()` function is used to take the input from the user.

**Syntax:** `input([<prompt>])`

where, `prompt` is the string we wish to display on the screen. It is optional.

- The `input()` pauses program execution to allow the user to type in a line of input from the keyboard. Once, the user presses the Enter key, all characters typed are read and returned as a string.

**Example:**

```
num = input('Enter a number: ')
Enter a number: 5
print(num)
Output: '5'
```

- The value num is a string not number, so to convert int() or float() functions can be used.

```
>>> int('5')
5
>>> float('5')
5.0
```

- The eval() function can evaluate expressions, provided the input is a string.  

```
>>> eval('2+3')
5
```
- In previous versions of Python the raw\_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).

**print() Function:**

- The print() function is used to output data to the standard output device (screen). Python programming language provides print() function to present the output of a program.

**Example:**

```
>>> print('This is the output to the screen')
```

**Output:**

```
This is the output to the screen
```

**Example:**

```
>>> a = 8
>>> print('The value of a is', a)
```

**Output:**

```
The value of a is 8
```

**Example:**

```
>>> a = "Hello"
>>> print(a)
```

**Output:**

```
Hello
```

**Example:**

```
>>> print(1,2,3,4)
1 2 3 4
>>> print(1,2,3,4,sep='*')
1*2*3*4
>>> print(1,2,3,4,sep='#',end='&')
1#2#3#4&
```

- To format the output str.format() method is used.

```
>>> x = 10; y = 20
>>> print('The value of x is {} and y is {}'.format(x,y))
```
- The value of x is 10 and y is 20. Here the curly braces {} are used as placeholders. The output is printed according to the order given as follows:

```
>>> print('I love {} and {}'.format('apple','milk'))
Output: I love apple and milk
>>> print('I love {} and {}'.format('apple','milk'))
Output: I love milk and apple
```

## PROGRAMS

**Program 1:** Program to find the square root of a number.

```
x=int(input("Enter an integer number:"))
ans=x**0.5
print("Square root= ", ans)
```

**Output:**

```
Enter an integer number: 144
Square root= 12.0
```

**Program 2:** Program to find the area of rectangle.

```
l=float(input("Enter length of the rectangle: "))
b=float(input("Enter breadth of the rectangle: "))
area=l*b
print("Area of Rectangle= ",area)
```

**Output:**

```
Enter length of the rectangle: 5
Enter breadth of the rectangle: 6
Area of Rectangle= 30.0
```

**Program 3:** Program to calculate area and perimeter of the square.

```
int(input("Enter side length of square: "))
area=side*side
perimeter = 4*side
print("Area of Square =", area)
print("Perimeter of Square =", perimeter)
```

**Output:**

```
Enter side length of square: 5
Area of Square = 25
Perimeter of Square = 20
```

**Program 4:** Program to calculate surface volume and area of a cylinder.

```
pi=22/7
height = float(input('Height of cylinder: '))
radian = float(input('Radius of cylinder: '))
volume = pi * radian * radian * height
sur_area = ((2*pi*radian) * height) + ((pi*radian**2)*2)
print("Volume is: ", volume)
print("Surface Area is: ", sur_area)
```

**Output:**

```
Height of cylinder: 4
Radius of cylinder: 6
Volume is: 452.57142857142856
Surface Area is: 377.1428571428571
```

---

**Program 5:** Program to swap the value of two variables.

```
num1=input("Enter first value: ")
num2=input("Enter second value: ")
print("Numbers before swapping")
print("num1= ",num1)
print("num2= ",num2)
temp=num1
num1=num2
num2=temp
print("Numbers after swapping")
print("num1= ",num1)
print("num2= ",num2)
```

**Output:**

```
Enter first value: 10
Enter second value: 20
Numbers before swapping
num1= 10
num2= 20
Numbers after swapping
num1= 20
num2= 10
```

## PRACTICE QUESTIONS

### Q.I Multiple Choice Questions:

1. Python is an open source, object-oriented, high-level powerful programming language.  
(a) C (b) C++  
(c) Python (d) Java
2. Applications of Python programming includes,  
(a) Google's App Engine (b) YouTube  
(c) Raspberry Pi (d) All of the mentioned
3. In python, which function is used to take the input from the user.  
(a) input() (b) print()  
(c) printf() (d) None of mentioned
4. Python editors includes,  
(a) IDLE (b) Spyder  
(c) PyDev (d) All of the mentioned
5. Which sign is used to assign values to variables?  
(a) # (b) =  
(c) ? (d) \$
6. Which of the following symbols are used for comments in Python?  
(a) // (b) '  
(c) # (d) /\* \*/
7. Assignment operators are used to,  
(a) to assign values to variables (b) to compare values  
(c) to sort values (d) all of the above
8. Following is a valid identifier in Python,  
(a) sum x (b) salary 2019  
(c) sum\_x (d) salary@2019
9. Who developed the Python language?  
(a) Zim Den (b) Guido van Rossum  
(c) Niene Stom (d) Wick van Rossum
10. In the Python statement  $x = a + 5 - b$ . Then  $a + 5 - b$  is,  
(a) a variable (b) a statement  
(c) an expression (d) an operator
11. Which one of the following is the correct extension of the Python file?  
(a) .py (b) .p  
(c) .phy (d) .ph
12. What is the output of the following code,  

```
x = 6
y = 2
print(x ** y)
print(x // y)
```

  
(a) 36,3 (b) 66,3  
(c) 36,0 (d) 66,0

- 
13. In Python which is used for defining null variables.
- (a) int
  - (b) bool
  - (c) void
  - (d) None
14. Which of the following is incorrect variable name in Python?
- (a) variable\_1
  - (b) variable1
  - (c) 1variable
  - (d) \_variable
15. Which statement is correct?
- (a) List is immutable & Tuple is mutable
  - (b) List is mutable & Tuple is immutable
  - (c) Both are Mutable.
  - (d) Both are Immutable
16. What type of data is: arr = [(1,1),(2,2),(3,3)]?
- (a) Array of tuples
  - (b) Tuples of lists
  - (c) List of tuples
  - (d) Invalid type
17. The 'in' operator is,
- (a) Identity operator
  - (b) Membership operator
  - (c) Arithmetic operator
  - (d) Assignment operator
18. Tuples are shown by,
- (a) { }
  - (b) [ ]
  - (c) ( )
  - (d) < >
19. Jython is a Python implementation written in,
- (a) C
  - (b) C++
  - (c) Java
  - (d) C#
20. Python features includes,
- (a) Platform Independence/Portable
  - (b) Simple and Easy-to-Learn
  - (c) Scalable and Dynamic
  - (d) All of the mentioned
21. String in Python enclosed in,
- (a) '...'
  - (b) "..."
  - (c) Both (a) and (b)
  - (d) None of the mentioned
22. Which are numbers or strings or characters that appear directly in a program?
- (a) Literals
  - (b) Variables
  - (c) Constants
  - (d) Operators
23. Which Python function is used to output data to the standard output device like screen/monitor?
- (a) print()
  - (b) input()
  - (c) printf()
  - (d) All of the above
24. Which data type in Python is an unordered collection of key-value pairs?
- (a) tuple
  - (b) dictionary
  - (c) list
  - (d) string
-

25. Which is a type of variable whose value cannot be changed during the execution of program?
- (a) constant (b) identifier  
(c) string (d) literals
26. Python programming standard data types includes,
- (a) list and tuple (b) set and dictionary  
(c) Boolean and None (d) All of the mentioned
27. Characteristics of Python includes,
- (a) provides very high-level dynamic data types and supports dynamic type checking  
(b) supports automatic garbage collection  
(c) easily integrated with C, C++, COM, ActiveX, CORBA, and Java  
(d) All of the mentioned
28. What is the output of the expression:  $3*1**3$ ?
- (a) 27 (b) 9  
(c) 3 (d) 1

### Answers

|         |         |         |         |         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1. (c)  | 2. (d)  | 3. (a)  | 4. (d)  | 5. (b)  | 6. (c)  | 7. (a)  | 8. (b)  | 9. (b)  | 10. (c) |
| 11. (a) | 12. (a) | 13. (d) | 14. (c) | 15. (b) | 16. (c) | 17. (b) | 18. (c) | 19. (c) | 20. (d) |
| 21. (c) | 22. (a) | 23. (a) | 24. (b) | 25. (a) | 26. (d) | 27. (d) | 28. (c) |         |         |

### Q.II Fill in the Blanks:

- \_\_\_\_\_ is a widely used high-level, general-purpose, interpreted, dynamic programming language.
- Python scripts normally have the file extension \_\_\_\_\_.
- A \_\_\_\_\_ refers to the fixed value that directly appears in the program.
- \_\_\_\_\_ is an Integrated Development Environment (IDE) for Python.
- Python's \_\_\_\_\_ is portable and cross platform compatible on UNIX, Linux, Windows and Macintosh.
- A Python \_\_\_\_\_ is a complete instruction executed by the Python interpreter.
- The \_\_\_\_\_ function helps to enter data at run time by the user and the output function \_\_\_\_\_ is used to display the result of the program on the screen after execution.
- The \_\_\_\_\_ indicates that the shell is ready to accept interactive commands.
- The \_\_\_\_\_ determines how much memory is allocated to store data and what operations can be performed on it.
- In Python the declaration of variables happens automatically when we assign a \_\_\_\_\_ to a variable using = (simple assignment operator).
- Python String is a sequence of Unicode characters and the default encoding for Python source code is \_\_\_\_\_.



12. An \_\_\_\_\_ is a combination of literals, variables and operators that Python evaluates to produce a value.
13. The \_\_\_\_\_ is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets [ ].
14. \_\_\_\_\_ are the constructs which can manipulate the value of operands. Consider the expression  $6 + 5 = 11$ . Here, 6 and 5 are called operands and + is called operator.
15. A \_\_\_\_\_ is the process of a programmer manually working through their code to trace the value of variables.
16. \_\_\_\_\_ are nothing but reserved memory locations to store values.
17. \_\_\_\_\_ and \_\_\_\_\_ not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).
18. \_\_\_\_\_ are the reserved words in Python and we cannot use a keyword as a variable name, function name or any other identifier.
19. \_\_\_\_\_ is defined within parentheses () where items are separated by commas.
20. An \_\_\_\_\_ is a name given to entities like class, functions, variables, etc.
21. We can make a statement extend over multiple lines with the line continuation character \_\_\_\_\_.
22. \_\_\_\_\_ is an unordered collection of unique items and defined by values separated by comma inside braces { }.
23. To format the output \_\_\_\_\_ method is used.
24. In Python, we use the \_\_\_\_\_ symbol to start writing a comment.
25. Python Shell executes a \_\_\_\_\_ statement.

### Answers

|                |               |                     |                |
|----------------|---------------|---------------------|----------------|
| 1. Python      | 2. .py        | 3. literal          | 4. IDLE        |
| 5. library     | 6. statement  | 7. input(), print() | 8. >>>         |
| 9. data type   | 10. value     | 11. UTF-8           | 12. expression |
| 13. list       | 14. Operators | 15. dry run         | 16. Variables  |
| 17. in, not in | 18. Keywords  | 19. Tuple           | 20. identifier |
| 21. \          | 22. Set       | 23. str.format()    | 24. hash (#)   |
| 25. single     |               |                     |                |

### Q.III State True or False:

1. Python is an open source, object-oriented, high-level powerful programming language
2. In Python, keywords are not case sensitive.
3. \*\* is an exponent operator in python.
4. Single line comment in python is written with / operator.
5. Indentation in python is define with { } braces.

6. An identifier can start with a digit.
7. A tuple is a collection of objects which ordered and immutable.
8. The input() function reads a line of input from the keyboard.
9. PyCharm is one of the widely used Python IDE which was created by Jet Brains.
10. In command line mode, we type the Python programming program on the Python shell and the interpreter prints the result.
11. Literal is a collection of group of characters.
12. The directory in Python is a collection of key value pairs created using { }. The key and value are separated by a colon (:) and the elements/items are separated by commas (,).
13. Comparison operators are used to compare values. It returns either True or False according to the condition.
14. We cannot use Python keywords as identifiers.
15. A list can be created by using [] brackets and separated by commas (,).
16. Python statements are usually written in a single line.
17. The is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory.
18. A constant is a type of variable whose value cannot be changed during execution of the program.
19. A constant is a memory location where we can store a value and its value can be vary during the program execution.
20. Python relies on indentation (whitespace at the beginning of a line) to define scope in the code.
21. Python provides a Python Shell, which is used to execute a single Python command and display the result.
22. The Python installer for Windows contains the IDLE module by default.

### Answers

|         |         |         |         |         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1. (T)  | 2. (F)  | 3. (T)  | 4. (F)  | 5. (F)  | 6. (F)  | 7. (T)  | 8. (T)  | 9. (T)  | 10. (T) |
| 11. (F) | 12. (T) | 13. (T) | 14. (T) | 15. (F) | 16. (T) | 17. (T) | 18. (T) | 19. (F) | 20. (T) |
| 21. (T) | 22. (T) |         |         |         |         |         |         |         |         |

### Q.IV Answer the following Questions:

#### (A) Short Answer Questions:

1. What is Python?
2. Define variable.
3. What is constant?
4. Which functions is used to perform I/O task in Python?
5. Define data type.
6. List comments in Python.
7. Give purpose of identifiers.

8. What is indentation?
9. What is dry run in Python?
10. List features of Python.
11. What is keyword?
12. Python is a scripting language. Comment this statement.
13. List editors used for Python programming.
14. Give the purpose of an operator.
15. What is statement and expression?

**(B) Long Answer Questions:**

1. Explain Python programming language with its applications.
2. How to declare a variable? Lists rules for declaring a variables.
3. Give short history for Python.
4. Write short note on: Comments in Python.
5. How to write and run python scripts? Explain in detail.
6. What is interpreter? How it works?
7. Explain the following features of Python programming:  
(i) Simple, (ii) Platform independent, (iii) Interactive, (iv) Object Oriented.
8. Explain about the need for learning Python programming and its importance.
9. Write the steps to install Python and to run Python code.
10. What is the role of indentation in Python?
11. What is variable? How to crate it? Explain with example
12. What are the various data types available in Python programming.
13. What is the difference between interactive mode and script mode of Python.
14. What is literal? Explain in detail.
15. Explain the following terms:  
(i) Tuple, (ii) List, (iii) Set, (iv) String, (v) Dictionary
16. How to performs input and output operations in Python? Explain in detail.
17. Write applications of Python in detail.
18. Give four examples of Python implementation.
19. Write short note on: Multi-line statements.
20. What is operator? Explain with example. List types of operators.
21. Explain identity operators with the help of example.
22. Describe program structure of Python diagrammatically.
23. Write program to calculate area of circle.
24. Write Python program to calculate area of triangle.



# Control Statements

---

## Objectives...

- To learn Control Statements in Python
- To study Strings in Python

---

### 2.0 INTRODUCTION

- In any programming language, a program is written as a set of instructions. The instructions written in programs are termed as statements.
- In Python, statements in a program are executed one after another in the order in which they are written. This is called sequential execution of the program.
- But in some situations, the programmer may need to alter the normal flow of execution of a program or to perform the same operations a number of times.
- For this purpose, Python provides a control structure which transfers the control from one part of the program to some other part of the program.
- A control structure is a statement that determines the control flow of the set of instructions i.e., a program.
- Control statements are the set of statements that are responsible to change the flow of execution of the program.
- There are different types of control statements supported by Python programming like decision/conditional control statements like if, if else, etc., loop/iteration control statements like for, while etc., and jump or loop control statements like break, continue etc.
- A string is a sequence of characters. In Python, a string is a sequence of Unicode characters.
- Strings in Python are surrounded by either single quotation marks, or double quotation marks. Even triple quotes can be used in Python. Example, 'hello' is the same as "hello" is the same as ""hello"" is the same as """"hello"""".

### 2.1 TYPES OF CONTROL STATEMENTS

- A program's control flow is the order in which the program's code executes. The order in which the specified set of statements need to be executed can be effectively governed using these control statements.

- Python programming provides a control structure which transfers the control from one part of the program to some other part of program.
- A control structure is a statement that determines the control flow of the set of instructions. The control flow refers to statement sequencing in a program to get desired result.
- Flow of control through any given program is implemented with three basic types of control structures such as Sequential, Selection and Repetition (See Fig. 2.1).

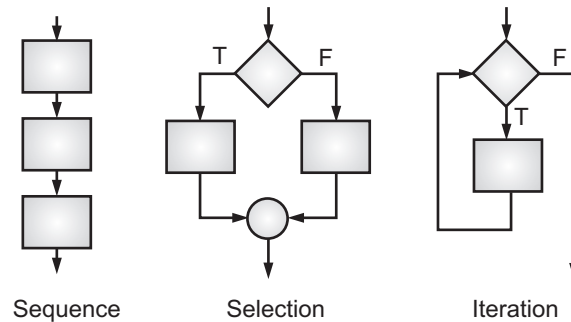


Fig. 2.1

- Sequential execution is when statements are executed one after another in order in sequential manner.
- A sequence control structure executes a sequence of statements in the order in which they are written.
- Sequential statements are a set of statements where the execution process will happen in sequence manner as shown in Fig. 2.2.
- Depending on whether a condition is true or false, the selection/decision control structure may skip the execution of an entire block of statements or even execute one block of statements instead of another.
- In Python, the selection statements are also known as decision control statements or branching statements.

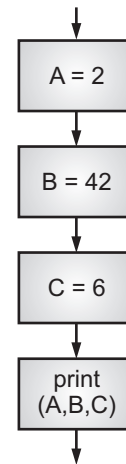


Fig. 2.2

- The selection statement allows a program to test several conditions and execute instructions based on which condition is true.
- The loop or iterative control structure allows the execution of a block of statements multiple times until a specified condition is met.
- An iteration or loop control structure is one which will execute a sequence of statements repeatedly if a condition holds true or satisfied.
- A repetition statement is used to repeat a group (block) of programming instructions.

---

### 2.1.1 Precedence and Associativity of Operators

---

- An expression may include some complex operations and may contain several operators.
- In such a scenario, the interpreter should know the order in which the operations should be solved. Operator precedence specifies the order in which the operators would be applied to the operands.
- Moreover, there may be expressions in which the operators belong to the same group, and then to resolve the operations, the associativity of the operators would be considered.
- The associativity specifies the order in which the operators of the same group will be resolved, i.e., from left to right or right to left.

#### 1. Python Operator Precedence:

- When an expression has two or more operators, we need to identify the correct sequence to evaluate these operators. This is because the final answer changes depending on the sequence thus chosen.

---

##### Example 1:

$10-4*2$  answer is 2 because multiplication has higher precedence than subtraction.

But we can change this order using parentheses () as it has higher precedence.

$(10-4)*2$  answer is 12

---

##### Example 2:

$10+5/5$

When given expression is evaluated left to right answer becomes 3. And when expression is evaluated right to left, the answer becomes 11.

- 
- Therefore, in order to remove this problem, a level of precedence is associated with the operators. Precedence is the condition that specifies the importance of each operator relative to the others.

#### 2. Associativity of Python's Operators:

- When two operators have the same precedence, associativity helps to determine which the order of operations.
  - Associativity decides the order in which the operators with same precedence are executed.
  - There are two types of associativity.
    - (i) **Left-To-Right:** The operator of same precedence is executed from the left side first.
    - (ii) **Right-To-Left:** The operator of same precedence is executed from the right side first.
  - Most of the operators in Python have left-to-right associativity.
-

**Example:**

```
>>> 5*2//3
3
>>> 5*(2//3)
0
```

- The following table lists all operators from highest precedence to the lowest.

| Sr. No. | Operator                  | Name/Description                                  | Assoicativity |
|---------|---------------------------|---------------------------------------------------|---------------|
| 1.      | () , []                   | Parentheses                                       | Left to Right |
| 2.      | **                        | Exponent                                          | Right to Left |
| 3.      | +x, -x, ~x                | Unary plus, Unary minus, Bitwise NOT              | Right to Left |
| 4.      | *, /, //, %               | Multiplication, Division, Floor division, Modulus | Left to Right |
| 5.      | +, -                      | Addition, Subtraction                             | Left to Right |
| 6.      | <<, >>                    | Bitwise left and right shift operators            | Left to Right |
| 7.      | &                         | Bitwise AND                                       | Left to Right |
| 8.      | ^                         | Bitwise XOR                                       | Left to Right |
| 9.      |                           | Bitwise OR                                        | Left to Right |
| 10.     | <=, <, >, >=              | Comparison Operator                               | Left to Right |
| 11.     | <> == !=                  | Equality operators                                | Left to Right |
| 12.     | = %= /= //=-<br>+= *= **= | Assignment Operators                              | Right to Left |
| 13.     | is, is not                | Identity                                          | Left to Right |
| 14.     | in, not in                | Membership operators                              | Left to Right |
| 15.     | NOT, OR, AND              | Logical Operators NOT, AND, OR                    | Left to Right |

**Example 1:**

```
a = 20
b = 10
c = 15
d = 5
e = 0
e = (a + b) * c / d #(30 * 15) / 5
```

```
print "Value of (a + b) * c / d is ", e
e = ((a + b) * c) / d # (30 * 15) / 5
print "Value of ((a + b) * c) / d is ", e
e = (a + b) * (c / d); # (30) * (15/5)
print "Value of (a + b) * (c / d) is ", e
e = a + (b * c) / d; # 20 + (150/5)
print "Value of a + (b * c) / d is ", e
```

**Output:**

```
Value of (a + b) * c / d is 90
Value of ((a + b) * c) / d is 90
Value of (a + b) * (c / d) is 90
Value of a + (b * c) / d is 50
```

**Example 2:**

```
Precedence of or & and
meal = "fruit"
money = 0
if meal == "fruit" or meal == "sandwich" and money >= 2:
 print("Lunch being delivered")
else:
 print("Can't deliver lunch")
```

**Output:**

```
Lunch being delivered
```

## 2.1.2 Type Conversion

- The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion.
- Python has two types of type conversion namely, Implicit Type Conversion and Explicit Type Conversion.

**1. Implicit Type Conversion:**

- Implicit type conversion: In this Python automatically converts one data type to another data type. This process doesn't need any user involvement.

**Example 1:** Converting integer to float.

```
num_i = 123
num_f = 1.23
num_new = num_i + num_f
```



```
print("datatype of num_i:",type(num_i))
print("datatype of num_f:",type(num_f))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

**Output:**

```
datatype of num_i: <class 'int'>
datatype of num_f: <class 'float'>
Value of num_new: 124.23
datatype of num_new: <class 'float'>
```

- In above example, num\_new has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

**2. Explicit Type Conversion:**

- In explicit type conversion users convert the data type of an object to required data type.
- We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

**Syntax:** <data\_type> (expression)

- To convert the built-in data types, we can use the type name as a function. There are several built-in functions to perform conversion from one data type to another.
- These functions return a new object representing the converted value. Table 2.1 list of built-in functions to perform type conversion

**Table 2.1: Built-in Functions for Data Type Conversion**

| Sr. No. | Functions                                                                                         |
|---------|---------------------------------------------------------------------------------------------------|
| 1.      | <b>int(x [,base]):</b> Converts x to an integer. base specifies the base if x is a string.        |
| 2.      | <b>long(x [,base] ) :</b> Converts x to a long integer. base specifies the base if x is a string. |
| 3.      | <b>float(x) :</b> Converts x to a floating-point number.                                          |
| 4.      | <b>complex(real [,imag]) :</b> Creates a complex number.                                          |
| 5.      | <b>str(x) :</b> Converts object x to a string representation.                                     |
| 6.      | <b>repr(x) :</b> Converts object x to an expression string.                                       |
| 7.      | <b>eval(str) :</b> Evaluates a string and returns an object.                                      |
| 8.      | <b>tuple(s) :</b> Converts string to a tuple.                                                     |

*contd. ...*

|     |                                                                                    |
|-----|------------------------------------------------------------------------------------|
| 9.  | <b>list(s)</b> : Converts string to a list.                                        |
| 10. | <b>set(s)</b> : Converts string to a set.                                          |
| 11. | <b>dict(d)</b> : Creates a dictionary. d must be a sequence of (key,value) tuples. |
| 12. | <b>frozenset(s)</b> : Converts s to a frozen set.                                  |
| 13. | <b>chr(x)</b> : Converts an integer to a character.                                |
| 14. | <b>unichr(x)</b> : Converts an integer to a Unicode character.                     |
| 15. | <b>ord(x)</b> : Converts a single character to its integer value.                  |
| 16. | <b>hex(x)</b> : Converts an integer to a hexadecimal string.                       |
| 17. | <b>oct(x)</b> : Converts an integer to an octal string.                            |

**Converting Integers to Floats:**

- Python's method float() will convert integers to floats. To use this function, add an integer inside of the parentheses:  
float(56)
- In above case, 56 will be converted to 56.0.

**Example:**

```
num_i = 100
num_s = "400"
num_s = int(num_s)
print("Data type of num_s after Type Casting:",type(num_s))
num_sum = num_i + num_s
print("Sum of num_i and num_s:",num_sum)
print("Data type of the sum:",type(num_sum))
```

**Output:**

```
Data type of num_s after Type Casting: <class 'int'>
Sum of num_i and num_s: 500
Data type of the sum: <class 'int'>
```

**Note:**

- Type conversion is the conversion of object from one data type to another data type.
- Implicit type conversion is automatically performed by the Python interpreter.
- Python avoids the loss of data in implicit type conversion.
- Explicit type conversion is also called type casting, the data types of objects are converted using predefined functions by the user.
- In type casting, loss of data may occur as we enforce the object to a specific data type.

## 2.2 CONDITIONAL STATEMENTS

- Decision or selection making is required when we want to execute a code only if a certain condition is satisfied.
- In Python, conditional statements are used to determine if a specific condition is met by testing whether a condition is True or False.
- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
- Conditional statements are used to determine how a program is executed. Decision structures evaluate multiple expressions which produce True or False as outcome.
- Python decision making statements includes, if statement, if-else statement, nested-if statements, multi-way if-elif-else statement.

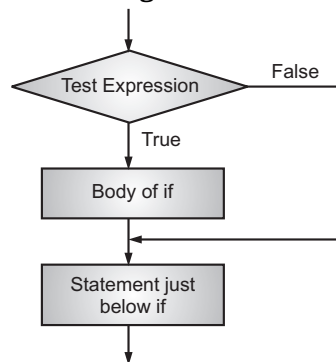
### 1. if Statement:

- The if statement executes a statement if a condition is true.

#### Syntax:

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

#### Control Flow Diagram of if Statement:



#### Example:

```
i=10
if(i<15):
 print("i is less than 15")
 print("This Statement is not in if")
```

#### Output:

```
i is less than 15
This statement is not in if
```

#### Example 1: To find out absolute value of an input number.

```
x=int(input("Enter an integer number:"))
y=x
if (x<0):
 x=-x
print('Absolute value of',y,'=',x)
```

#### Output:

```
Enter an integer number:3
Absolute value of 3 = 3
Enter an integer number:-3
Absolute value of -3 = 3
```

**Example 2:** To find whether a number is even or odd.

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

**Output:**

```
Enter any number: 10
10 is even number
Enter any number: 11
11 is odd number
```

---

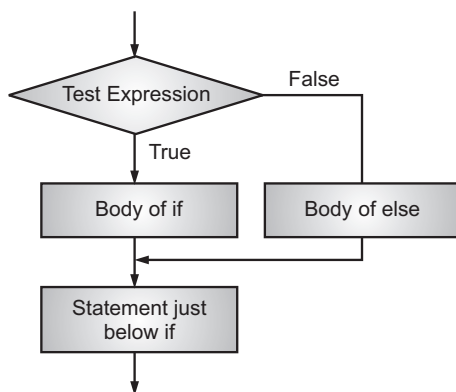
## 2. if else Statement:

- The if else statement evaluates test expression and will execute the body of if only when the test condition is True. If the condition is False, the body of else is executed.

**Syntax:**

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

**Control flow Diagram of if-else Statement:**



**Example:**

```
i=20
if(i<15):
 print("i is less than 15")
else:
 print("i is greater than 15")
```

**Output:**

```
i is greater than 15
```

---

**Example:** To check if the input year is a leap year or not.

```
year=int(input("Enter year to be checked:"))
if(year%4==0 and year%100!=0 or year%400==0):
 print(year, " is a leap year!")
else:
 print(year, " isn't a leap year!")
```

**Output:**

```

Enter year to be checked:2016
2016 is a leap year!
Enter year to be checked:2018
2018 isn't a leap year!

```

| Valid Indentation |                                                                                          | Invalid Indentation |                                                                                          |
|-------------------|------------------------------------------------------------------------------------------|---------------------|------------------------------------------------------------------------------------------|
| (a)               | <pre> if condition:     statement     statement else:     statement     statement </pre> | (b)                 | <pre> if condition:     statement     statement else:     statement     statement </pre> |
| (c)               | <pre> if condition:     statement     statement else:     statement     statement </pre> | (d)                 | <pre> if condition:     statement     statement else:     statement     statement </pre> |

Fig. 2.3: Indentation of if Statement

**3. if elif else Statement:**

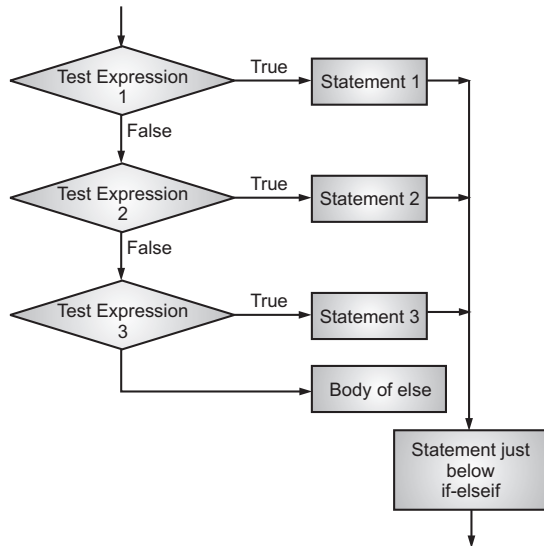
- The elif is short for else if. It allows us to check for multiple conditions. The if statements are executed from the top down.
- As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

**Syntax:**

```

if (condition 1):
 statements
elif (condition 2):
 statements
.
.
.
elif(condition-n):
 statements
else:
 statements

```

**Control Flow Diagram for if-elif-if Statement:****Example:**

```

i = 20
if (i == 10):
 print ("i is 10")
elif (i == 15):
 print ("i is 15")
elif (i == 20):
 print ("i is 20")
else:
 print ("i is not present")

```

**Output:**

```
i is 20
```

**Example:** Program to check the largest number among the three numbers.

```

x=int(input("Enter first number: "))
y=int(input("Enter second number: "))
z=int(input("Enter third number: "))
if(x>y) and (x>z):
 l=x
elif(y>x) and (y>z):
 l=y
else:
 l=z
print("largest number is: ",l)

```

**Output:**

```

Enter first number: 10
Enter second number: 20
Enter third number: 30
largest number is: 30

```

**4. Nested IF Statement:**

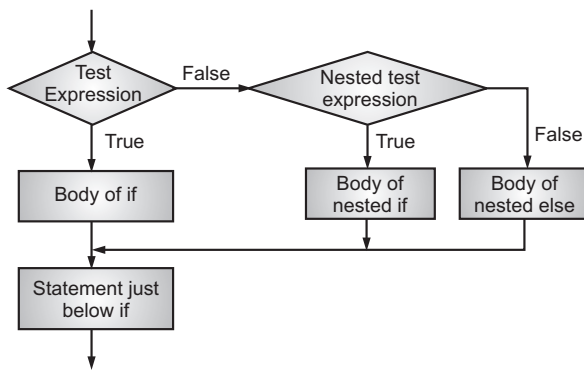
- When a programmer writes one if statement inside another if statement then it is called a nested if statement.

**Syntax:**

```

if condition1:
 if condition2:
 statement1
 else:
 statement2
else:
 statement3

```

**Control Flow diagram of Nested if Statement:****Example:**

```

a=30
b=20
c=10
if (a>b):
 if (a>c):
 print("a is greater")
 else:
 print("a is less than")
print("End of Nested if")

```

**Output:**

```

a is greater
End of Nested if

```

**Example:** For nested if statement.

```

num = float(input("Enter a number: "))
if num >= 0:
 if num == 0:
 print("Zero")
 else:
 print("Positive number")
else:
 print("Negative number")

```

**Output 1:**

```

Enter a number: 5
Positive number

```

**Output 2:**

```

Enter a number: -1
Negative number

```

## 2.3 LOOPING

- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- There may be a situation when you need to execute a block of code several number or multiple of times.
- A loop statement allows us to execute a statement or group of statements multiple times, this is called iteration.
- Looping control statements (or iterative control statements) are repeatedly executing a set of statements until a certain (stated) condition is met.
- Python programming language provides three types of loops to handle looping requirements namely, while loop, for loop and nested loops.

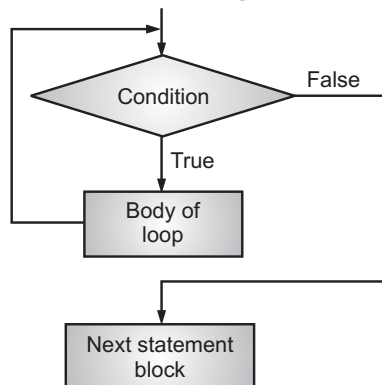
### 1. While Loop:

- A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

#### Syntax:

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

#### Control Flow Diagram for while Loop:



#### Example:

```
count=0 # initialize counter
while count<=3: # test condition
 print("count= ",count) # print value
 count=count+1 # increment counter
```

#### Output:

```
count=0
count=1
count=2
count=3
```

#### Example: To display Fibonacci series.

```
num=int(input("Enter how many number u want to display: "))
x=0
y=1
c=2
print("Fibonacci Sequence is:")
print(x)
print(y)
```



```
while(c<num):
 z=x+y
 print(z)
 x=y
 y=z
 c+=1
```

**Output:**

```
Enter how many number u want to display: 10
Fibonacci Sequence is:
0
1
1
2
3
5
8
13
21
34
```

---

## PROGRAMS

**Program 1:** Program to find out the reverse of the given number.

```
n=int(input('Enter a number: '))
rev=0
while(n>0):
 rem=n%10
 rev=rev*10+rem
 n=int(n/10)
print('Reverse of number=',rev)
```

**Output:**

```
Enter a number: 123
Reverse of number= 321
```

---

**Program 2:** Program to find sum of digit of a given number.

```
n=int(input('Enter a number:'))
sum=0
while(n>0):
 rem=n%10
 sum=sum+rem
 n=int(n/10)
print('Sum of number=',sum)
```

**Output:**

```
Enter a number: 123
Sum of number= 6
```

**Program 3:** Program to check whether the input number is Armstrong.

```
n=int(input('Enter a number:'))
num=n
sum=0
while(n>0):
 rem=n%10
 sum=sum+rem*rem*rem
 n=int(n/10)
if num==sum:
 print(num, "is armstrong")
else:
 print(num, "is not armstrong")
```

**Output:**

```
Enter a number: 153
153 is armstrong
Enter a number: 123
123 is not armstrong
```

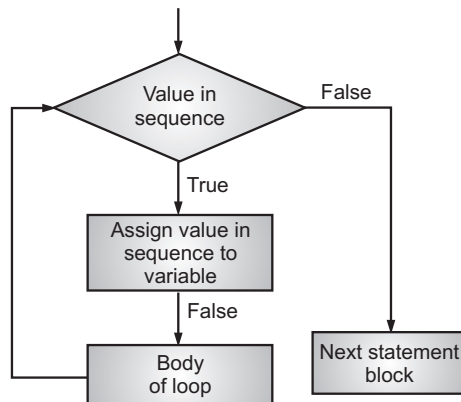
## 2. for Loop:

- The **for loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

**Syntax:**

```
for iterating_var in sequence:
 statements(s)
```

**Control Flow diagram for 'for' Loop:**



**Example:**

```
list=[10,20,30,40,50]
for x in list:
 print(x)
```

**Output:**

```
10
20
30
40
50
```

**Example 1:**

```
for letter in 'Python':
 print ('Current Letter:', letter)
fruits = ['banana', 'apple', 'mango']
for fruit in fruits:
 print ('Current fruit:', fruit)
```

**Output:**

```

Current Letter: P
Current Letter: y
Current Letter: t
Current Letter: h
Current Letter: o
Current Letter: n
Current fruit: banana
Current fruit: apple
Current fruit: mango

```

**range() Function:**

- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

**Syntax:** range(begin, end, step)

where,

- **Start:** An integer number specifying at which position to start. Default is 0.
- **End:** An integer number specifying at which position to end, which is computed as End - 1. This is mandatory argument to specify.
- **Step:** An integer number specifying the increment. Default is 1.

**Example:** For range() function.

```

>>> list(range(1,6))
[1, 2, 3, 4, 5]

```

- More examples of range() function:

| Example              | Output                       |
|----------------------|------------------------------|
| list(range(5))       | [0, 1, 2, 3, 4]              |
| list(range(1,5))     | [1, 2, 3, 4]                 |
| list(range(1,10,2))  | [1, 3, 5, 7, 9]              |
| list(range(5,0,-1))  | [5, 4, 3, 2, 1]              |
| list(range(10,0,-2)) | [10, 8, 6, 4, 2]             |
| list(range(-4,4))    | [-4, -3, -2, -1, 0, 1, 2, 3] |
| list(range(-4,4,2))  | [-4, -2, 0, 2]               |
| list(range(0,1))     | [0]                          |
| list(range(1,1))     | [] Empty                     |
| list(range(0))       | [] Empty                     |

**Example:**

```
for i in range(1,11):
 print (i, end=' ')
```

**Output:**

```
1 2 3 4 5 6 7 8 9 10
```

- The print() function has end=' ' which appends a space instead of default newline. Hence, the numbers will appear in one row.

**PROGRAMS**

**Program 1:** Program to print prime numbers in between a range.

```
start=int(input("Enter starting number: "))
end=int(input("Enter ending number: "))
for n in range(start,end + 1):
 if (n>1):
 for i in range(2,n):
 if(n%i)== 0:
 break
 else:
 print(n)
```

**Output:**

```
Enter starting number: 1
Enter ending number: 20
2
3
5
7
11
13
17
19
```

**Program 2:** Program to check whether the entered number is prime or not.

```
n=int(input("Enter a number:"))
for i in range(2,n+1):
 if n%i==0:
 break
if i==n:
 print(n," is prime number")
else:
 print(n," is not a prime number")
```

**Output:**

```
Enter a number:5
5 is prime number
Enter a number:4
4 is not a prime number
```

---

**Program 3:** Program to print and sum of all even numbers between 1 to 20.

```
sum=0
for i in range(0,21,2):
 print(i)
 sum=sum+i
print("Sum of Even numbers= ",sum)
```

**Output:**

```
0
2
4
6
8
10
12
14
16
18
20
Sum of Even numbers= 110
```

---

**Nested for and while Loops:**

- Loops within the loops or when one loop is inserted completely within another loop, then it is called nested loop. Both for and while loop statement can be nested.

**Syntax for Nested for Loop:**

```
for var in sequence:
 for var in sequence:
 statements(s)
 statements(s)
```

---

**Example:** Nested for loop.

```
for i in range(1,5):
 for j in range(1,(i+1)):
 print (j, end=' ')
 print()
```

---

**Output:**

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

---

**Syntax for Nested while Loop:**

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

---

**Example:** Nested while loop.

```
i = 1
while i < 5:
 j = 1
 while j < (i+1):
 print(j, end=' ')
 j = j + 1
 i = i + 1
 print()
```

**Output:**

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

---

- Last print() will be executed at the end of inner for while loop.

## PROGRAMS

---

**Program 1:** Program to print following pyramid:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
for i in range(1,6):
 for j in range(1,i+1):
 print(j,end=' ')
 print()
```

---

**Program 2:** Program to print following pyramid:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
for i in range(1,6):
 for j in range(1,i+1):
 print(i,end=' ')
 print()
```

---

**Program 3:** Program to print following pyramid:

```
1
2 3
4 5 6
7 8 9 1
2 3 4 5 6
count=1
for i in range(1,6):
 for j in range(1,i+1):
 print(count,end=' ')
 count=count+1
 if count>9:
 count=1
 print()
```

---

**Program 4:** Program to print following pyramid:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
count=1
for i in range(1,6):
 for j in range(1,i+1):
 print(count,end=' ')
 count=count+1
 print()
```

---

**Program 5:** Program to print pyramid:

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

---

```

for row in range(1,6):
 for sp in range(1,6-row):
 print(' ',end=' ')
 for col in range(1,row+1):
 print(col, end=' ')
 for erow in range(col-1,0,-1):
 print(erow,end=' ')
 print()

```

## 2.4 LOOP CONTROL STATEMENTS

- In Python, loop statements give us a way execute the block of code repeatedly. But sometimes, we may want to exit a loop completely or skip specific part of loop when it meets a specified condition. It can be done using loop control mechanism.
- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Loop control statements in Python programming are basically used to terminate a loop or skip the particular code in the block or it can also be used to escape the execution of the program.
- The loop control statements in Python programming includes break statement, continue statement and pass statement.

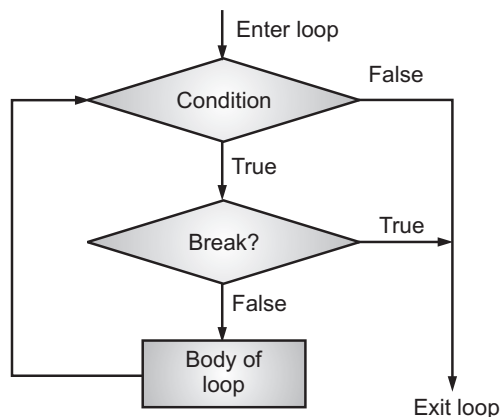
### 1. break Statement:

- The break statement is a jump statement which is used to transfer execution control. It breaks the current execution and in case of inner loop, inner loop terminates immediately and resumes execution at the next statement.
- The break statement can be used in both while and for loops.

#### Syntax:

```
break
```

#### Control Flow Diagram for break Statement:



#### Example: For break statement.

```

i=0
while i<10:
 i=i+1
 if i==5:
 break
 print("i= ",i)

```

#### Output:

```

i=1
i=2
i=3
i=4

```



**Example 1:**

```
for letter in 'tybcs': #first example
 if letter == 'b':
 break
 print('Current Letter:', letter)
```

**Output:**

```
Current Letter: t
Current Letter: y
```

**Example 2:**

```
var = 10
while var > 0:
 print('Current variable value:', var)
 var = var -1
 if var == 7:
 break
 print "Out loop"
```

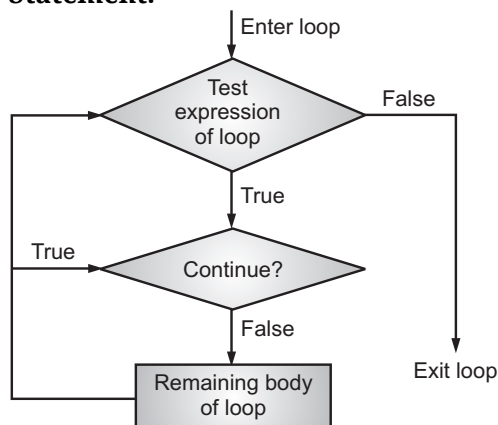
**Output:**

```
Current variable value: 10
Current variable value: 9
Current variable value: 8
Out loop
```

**2. continue Statement:**

- The continue statement in Python returns the control to the beginning of the while loop.
- The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

**Syntax:** continue

**Control Flow Diagram for continue Statement:**

**Example:** For continue statement.

```
i=0
while i<10:
 i=i+1
 if i==5:
 continue
 print("i= ",i)
```

**Output:**

```
i=1
i=2
i=3
i=4
i=6
i=7
i=8
i=9
i=10
```

**Example 1:**

```
for letter in 'Tybsc':
 if letter == 'b':
 continue
 print ('Current Letter:', letter)
```

**Output:**

```
Current Letter: T
Current Letter: y
Current Letter: s
Current Letter: c
```

---

**Example 2:**

```
var = 5
while var > 0:
 var = var -1
 if var == 3:
 continue
 print('Current variable value:', var)
 print "In loop"
```

**Output:**

```
Current variable value: 4
Current variable value: 2
Current variable value: 1
Current variable value: 0
In loop
```

---

**3. pass Statement:**

- It is used when a statement is required syntactically but we do not want any command or code to execute.
- The pass statement is a null operation; nothing happens when it executes.

**Syntax:** pass

---

**Example 1:**

```
for letter in 'Tybsc':
 if letter == 'b':
 pass
 print('This is pass block')
 print('Current Letter:', letter)
```

**Output:**

```
Current Letter: T
Current Letter: y
```

---

```
This is pass block
Current Letter: b
Current Letter: s
Current Letter: c
```

---

**Example 2:** For pass statement.

```
for i in range(1,11):
 if i%2==0: # check if the number is even
 pass # (No operation)
 else:
 print("Odd Numbers: ",i)
```

**Output:**

```
Odd Numbers: 1
Odd Numbers: 3
Odd Numbers: 5
Odd Numbers: 9
Odd Numbers: 7
```

---

## PROGRAMS

**Program 1:** Program to find factorial of a given number.

```
num=int(input("Enter Number:"))
fact=1
if num< 0:
 print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
 print("The factorial of 0 is 1")
else:
 for i in range(1,num + 1):
 fact=fact*i
 print("The factorial of ",num," is ",fact)
```

**Output:**

```
Enter Number: 5
The factorial of 5 is 120
```

---

**Program 2:** Program to print multiplication table of the given number.

```
n=int(input('Enter a number: '))
for i in range (1,11):
 print(n,' * ',i,' = ',n*i)
```

**Output:**

```
Enter a number: 3
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```

---

**Program 3:** Program to find out whether the input number is perfect number or not.

```
n=int(input("Enter number"))
sum=0
for i in range(1,n):
 if n%i==0:
 sum=sum+i
if sum==n:
 print(n,' is perfect number')
else:
 print(n,' is not perfect number')
```

**Output:**

```
Enter number 20
20 is not perfect number
Enter number 28
28 is perfect number
```

---

**Program 4:** Program to generate Student Result. Accept marks of five subject and display result according to following conditions:

| Percentage   | Division                     |
|--------------|------------------------------|
| >=75         | First class with Distinction |
| >=60 and <75 | First Class                  |
| >=45 and <60 | Second Class                 |
| >=40 and <45 | Pass                         |
| <40          | Fail                         |

```
m1=int(input("Enter marks of Subject-1:"))
m2=int(input("Enter marks of Subject-2:"))
m3=int(input("Enter marks of Subject-3:"))
total=m1+m2+m3
per=total/3
 print("Total Marks=",total)
 print("Percentage=",per)
if per >= 75:
 print("Distinction")
elif per >=60 and per<75:
 print("First class")
elif per >=45 and per<60:
 print("Second class")
elif per >=40 and per<45:
 print("Pass")
else:
 print("Fail")
```

**Output:**

```
Enter marks of Subject-1:60
Enter marks of Subject-1:70
Enter marks of Subject-1:80
Total Marks= 210
Percentage= 70.0
First class
```

---

**Program 5:** Python program to perform Addition Subtraction Multiplication and Division of two numbers.

```
num1 = int(input("Enter First Number: "))
num2 = int(input("Enter Second Number: "))
 print("Enter which operation would you like to perform?")
ch = input("Enter any of these char for specific operation +,-,*,/: ")
result = 0
if ch == '+':
 result = num1 + num2
elif ch == '-':
 result = num1 - num2
```

```
elif ch == '*':
 result = num1 * num2
elif ch == '/':
 result = num1 / num2
else:
 print("Input character is not recognized!")
print(num1, ch, num2, ":", result)
```

**Output:**

```
Enter First Number: 20
Enter Second Number: 10
Enter which operation would you like to perform?
Enter any of these char for specific operation +,-,*,/: *
20 * 10 : 200
```

---

**Program 6:** Program to check whether a string is a palindrome or not.

```
string=input("Enter string:")
if(string==string[::-1]):
 print("The string is a palindrome")
else:
 print("The string isn't a palindrome")
```

**Output:**

```
Enter string: abc
The string isn't a palindrome
Enter string: madam
The string is a palindrome
```

---

**Program 7:** Program to check whether a number is a palindrome or not.

```
num = int(input("enter a number: "))
temp = num
rev = 0
while temp != 0:
 rev = (rev * 10) + (temp % 10)
 temp = temp // 10
if num == rev:
 print("number is palindrome")
else:
 print("number is not palindrome")
```

---

**Output:**

```
enter a number: 121
number is palindrome
enter a number: 123
number is not palindrome
```

---

**Program 8:** Program to return prime numbers from a list.

```
list=[3,2,9,10,43,7,20,23]
print("list=",list)
l=[]
print("Prime numbers from the list are:")
for ain list:
 prime=True
 for i in range(2,a):
 if (a%i==0):
 prime=False
 break
 if prime:
 l.append(a)
print(l)
```

**Output:**

```
list= [3, 2, 9, 10, 43, 7, 20, 23]
Prime numbers from the list are:
[3, 2, 43, 7, 23]
```

---

**Program 9:** Program to add, subtract, multiply and division of two complex numbers.

```
print("Addition of two complex numbers : ",(4+3j)+(3-7j))
print("Subtraction of two complex numbers : ",(4+3j)-(3-7j))
print("Multiplication of two complex numbers : ",(4+3j)*(3-7j))
print("Division of two complex numbers : ",(4+3j)/(3-7j))
```

**Output:**

```
Addition of two complex numbers : (7-4j)
Subtraction of two complex numbers : (1+10j)
Multiplication of two complex numbers : (33-19j)
Division of two complex numbers : (-0.15517241379310348+0.6379310344827587j)
```

**Program 10:** Program to find the best of two test average marks out of three test's marks accepted from the user.

```
n1=int(input('enter a number'))
n2=int(input('enter 2nd number'))
n3=int(input('enter the 3rd number'))
avg1=(n1+n2)/2
avg2=(n2+n3)/2
avg3=(n3+n1)/2
maxm=max(avg1, avg2, avg3)
print(maxm)
```

---

**Program 11:** Print patterns of (\*) using loop.

(i) for i in range(0, 5):  
    for j in range(0, i+1):  
        print("\* ",end="")  
    print("\n")

**Output:**

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

(ii) for i in range(0, 5):  
    num = 1  
    for j in range(0, i+1):  
        print(num, end=" ")  
    num = num + 1  
    print("\n")

**Output:**

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

---



**Program 12:** Python program to add two numbers.

**Method 1:**

```
This program adds two numbers
num1 = 1.5
num2 = 6.3
Add two numbers
sum = float(num1) + float(num2)
Display the sum
print('The sum is, sum')
```

**Output:**

The sum is 7.8

**Method 2:**

```
Program To add two numbers that accepted from user
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')
sum = float(num1) + float(num2)
print('The sum is sum')
```

---

**Program 13:** Program to calculate the square root.

```
num = 8
#num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of %.3f is %.3f'%(num ,num_sqrt))
```

---

**Program 14:** Program to find area of triangle.

```
a = float(input('Enter first side: '))
b = float(input('Enter second side: '))
c = float(input('Enter third side: '))
calculate the semi-perimeter
s = (a + b + c) / 2
calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The Area of the Triangle is %.2f' %area)
```

---

**Program 15:** Program to swap two variables.

```
x = input('Enter value of x: ')
y = input('Enter value of y: ')
create a temporary variable and swap the values
temp = x
x = y
y = temp
print('The value of x after swapping:',x)
print('The value of y after swapping:',y)
```

**Program 16:** Program to generate a random number between 0 and 9.

```
import the random module
import random
print(random.randint(0,9))
```

---

**Program 17:** Program which accepts 5 integer values and prints “DUPLICATES” if any of the values entered are duplicates otherwise it prints “ALL UNIQUE”. Let 5 integers are (32, 45, 90, 45, 6) then output “DUPLICATES” to be printed.

```
#find duplicate
n=input("Entered any 5 integers").split()
s1=set(n)
for i in s1:
 c=n.count(i)
 if c>1:
 print("DUPLICATE"+i)
 else:
 print("UNIQUE "+i)
```

**Output:**

```
Entered any 5 integers 4 5 6 2 4
DUPLICATE 4
UNIQUE 5
UNIQUE 6
UNIQUE 2
```

---

**Program 18:** Program which accepts an integer value as command line and print “Ok” if value is between 1 to 50 (both inclusive) otherwise it prints ”Out of range”.

```
#check number is in range of 1-50
n=int(input("enter any number:"))
if n in range(1,51):
 print("OK")
else:
 print("Out of range")
```

**Output:**

```
enter any number: 55
Out of range
enter any number:50
OK
```

---

**Program 19:** Program which finds sum of digits of a number.

```
n=int(input("Enter a number:"))
tot=0
while(n>0):
 dig=n%10
 tot=tot+dig
 n=n//10
print("The total sum of digits is:",tot)
```

**Output:**

```
Enter a number:156
The total sum of digits is: 12
```

---

**Program 20:** Program which prints Fibonacci series of a number.

```
nterms = 10
uncomment to take input from the user
#nterms = int(input("How many terms? "))
first two terms
n1 = 0
n2 = 1
count = 0
check if the number of terms is valid
if nterms <= 0:
 print("Please enter a positive integer")
elif nterms == 1:
 print("Fibonacci sequence upto",nterms,":")
 print(n1)
else:
 print("Fibonacci sequence upto",nterms,":")
 while count < nterms:
 print(n1,end=' , ')
 nth = n1 + n2
 # update values
 n1 = n2
 n2 = nth
 count += 1
```

**Output:**

```
Fibonacci sequence upto 10 :
0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 ,
```

---

**Program 21:** Program to ask the user for a range and display all the prime numbers in that interval.

```
take input from the user
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))
for num in range(lower,upper + 1):
 # prime numbers are greater than 1
 if num > 1:
 for i in range(2,num):
 if (num % i) == 0:
 break
 else:
 print(num)
```

---

**Program 22:** Program to display following pattern.

```


**
*

n=int(input("Enter number of rows: "))
for i in range (n,0,-1):
 print((n-i) * ' ' + i * '*')
```

**Output:**

```
Enter number of rows: 5

**
*
```

---

**Program 23:** Program that finds distance between 2 points (x1, y1) and (x2, y2) using the equation.

```
import math
x1=int(input("Enter x1: "))
y1=int(input("Enter y1:"))
x2=int(input("Enter x2:"))
y2=int(input("Enter y2:"))
```

---

```
z1=[x1,y1]
z2=[x2,y2]
distance=math.sqrt(((z1[0]-z2[0])**2)+((z1[1]-z2[1])**2))
print("Distance is",distance)
```

**Output:**

```
Enter x1: 4
Enter y1:0
Enter x2:6
Enter y2:6
Distance is 6.324555320336759
```

---

**Program 24:** Program to check prime number.

```
num = int(input("Enter a number: "))
prime numbers are greater than 1
if num > 1:
 # check for factors
 for i in range(2,num):
 if (num % i) == 0:
 print(num,"is not a prime number")
 break
 else:
 print(num,"is a prime number")
if input number is less than or equal to 1, it is not prime
else:
 print(num,"is not a prime number")
```

**Output:**

```
Enter a number: 56
56 is not a prime number
```

---

**Program 25:** Program to find the factorial of a number.

```
change the value for a different result
num = int(input("Enter a number: "))
factorial = 1
check if the number is negative, positive or zero
if num < 0:
 print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
 print("The factorial of 0 is 1")
```

```
else:
 for i in range(1,num + 1):
 factorial = factorial*i
 print("The factorial of",num,"is",factorial)
```

**Output:**

```
Enter a number
5
The factorial of 5 is 120
```

**Program 26:** Program to display pattern.

```
n=int(input("Enter any number"))
for i in range(n,0,-1):
 for j in range(1,i+1):
 print(j,end=" ")
 print()
```

**Output:**

```
Enter any number 5
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

## 2.5 STRINGS

- A Python string is a sequence of characters. Python Strings are immutable (unchanging) sequences of Unicode points.
- Strings in python support Unicode characters. The default encoding for Python source code is UTF-8. So, we can also say that String is a sequence of Unicode characters.
- To create/declare a string, put the sequence of characters inside either single quotes, double quotes, or triple quotes and then assign it to a variable.

# all of the following are equivalent

```
my_string = 'Hello'
print(my_string)
```

```
my_string = "Hello"
print(my_string)
```

```

my_string = '''Hello'''
print(my_string)

triple quotes string can extend multiple lines
my_string = """Hello, welcome to
 the world of Python"""
print(my_string)

```

**Output:**

```

Hello
Hello
Hello
Hello, welcome to the world of Python

```

- Assigning a string to a variable is done with the variable name followed by an equal sign and the string.

**Example:**

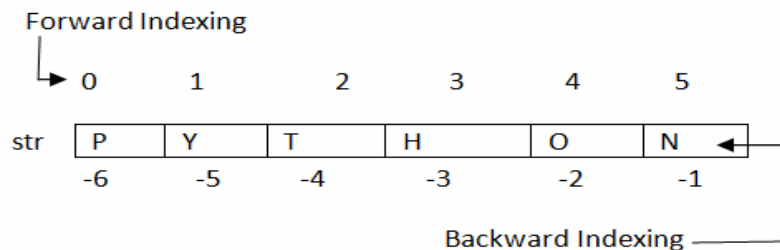
```

a = "Hello"
print(a)

```

**String Representation:**

- In Python, Strings are stored as individual characters in a contiguous memory location.
- We can access the individual characters in string by using the string variable and index.
- As per above schematic we need to generalize that forward index will start from “0” th index and end with “n-1” index. To use the index we can use square brackets “[ ]”.
- The benefit of using String is that it can be accessed from both the directions (forward and backward).
- Both forward as well as backward indexing are provided using Strings in Python.
  - Forward indexing starts with 0,1,2,3,....
  - Backward indexing starts with -1,-2,-3,-4,....

**Example:**

```

str[0]='P'=str[-6], str[1]='Y'=str[-5], str[2]='T'= str[-4], str[3]='H'=str[-3] str[4]='O'=str[-2], str[5]='N'=str[-1]

```

**Fig. 2.4**

**Example:**

```
name="Omkar"
length=len(name)
i=0
for n in range(-1,(-length-1),-1):
 print (name[i],"\t", name[n])
 i+=1
```

**Output:**

```
0 r
m a
k k
a m
r o
```

---

## 2.5.1 Accessing Values in Strings

---

- Individual characters in a string can be accessed using an index. Subsets of strings can be taken using the slice operation ([] and [i]) with index starting at 0 in the beginning of the string and -1 at the end.

**Example:** For accessing string.

```
>>> s="Hello Python"
>>> s[0] # get element at index 0
'H'
>>> s[-1] # get element at last index
'n'
>>> s[1:4] # get element from m index to n-1 index.
'ell'
>>> s[6:] # get element from m index to last index.
' Python'
>>> s[:5] # get element from 0th index to n-1 index.
'Hello'
>>> s[1:12:2] # get element from m index to n-1 index with
i increments
'el Pto'
>>> s+" Programming" # It will concatenate string
```



```
'Hello Python Programming'
>>> type(s)
<class 'str'>
>>> s*3 #It will repeat the string
'Hello PythonHello PythonHello Python'
>>>
```

## 2.5.2 Updating/Manipulating Strings

- The string can be update by (re)assigning a variable to another string.
- The new value can be related to its previous value or to a completely different string altogether.

### Example:

```
msg1 = 'Hello World!'
print("Updated String: ", msg1[:6] + 'Python')
```

### Output:

```
Updated String: Hello Python
```

## 2.5.3 Escape Characters

- An escape character gets interpreted; in a single quoted as well as double quoted strings.
- Following Table 2.2 is a list of escape or non-printable characters that can be represented with backslash notation.

**Table 2.2: List of Escape or Non-printable characters.**

| Escape Character | Meaning                    |
|------------------|----------------------------|
| \newline         | Ignored                    |
| \\               | Backslash (\)              |
| \'               | Single quote (')           |
| \"               | Double quote (")           |
| \a               | ASCII Bell (BEL)           |
| \b               | ASCII Backspace (BS)       |
| \f               | ASCII Formfeed (FF)        |
| \n               | ASCII Linefeed (LF)        |
| \r               | ASCII Carriage Return (CR) |

*contd. ...*

|             |                                             |
|-------------|---------------------------------------------|
| \t          | ASCII Horizontal Tab (TAB)                  |
| \v          | ASCII Vertical Tab (VT)                     |
| \ooo        | ASCII character with octal value ooo        |
| \xhh...     | ASCII Character with hex value hh           |
| \cx or \C-x | Control-x                                   |
| \e          | Escape                                      |
| \M-\C-x     | Meta-Control-x                              |
| \nnn        | Octal notation, where n is in the range 0-7 |
| \s          | Space                                       |

**Example:**

```
>>> print("This is printed\n in two lines")
This is printed
in two lines
>>> print("This is \x48\x45\x58 representation")
This is HEX representation
```

**2.5.4 String Special Operations**

- There are many operations that can be performed with string which makes it one of the most used data types in Python.
- Following is the list of special operators that operates on string. Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then:

| Operator | Description                                                                        | Example                     |
|----------|------------------------------------------------------------------------------------|-----------------------------|
| +        | Concatenation - Adds values on either side of the operator                         | a + b will give HelloPython |
| *        | Repetition - Creates new strings, concatenating multiple copies of the same string | a*2 will give -HelloHello   |
| []       | Slice - Gives the character from the given Index                                   | a[1] will give e            |
| [:]      | Range of Slice - Gives the characters from the given range                         | a[1:4] will give ell        |
| In       | Membership - Returns true if a character exists in the given string                | H in a will give 1          |
| not in   | Membership - Returns true if a character does not exist in the given string        | M not in a will give 1      |
| %        | Format - Performs String formatting                                                |                             |

**Example:**

```

str1 = 'Hello'
str2 = 'World!'
using + Operator
print('str1 + str2 = ', str1 + str2)
using * Operator
print('str1 * 3 = ', str1 * 3)

```

**Output:**

```

str1 + str2 = HelloWorld!
str1 * 3 = HelloHelloHello

```

### 2.5.5 String Formatting Operator

- Python's uses % as a string format operator. This operator is unique to strings and makes up for the pack of having functions from C language's printf() family.
- Following is a simple example:

```
print ("My name is %s and weight is %d kg" % (Omkar ', 62))
```

**Output:**

```
My name is Omkar and weight is 62 kg
```

- Table 2.3 lists complete set of symbols used along with %.

**Table 2.3: List of symbols used along with %**

| Format Symbol | Conversion                                         |
|---------------|----------------------------------------------------|
| %c            | Character                                          |
| %s            | string conversion via str prior to formatting      |
| %i            | signed decimal integer                             |
| %d            | signed decimal integer                             |
| %u            | unsigned decimal integer                           |
| %o            | octal integer                                      |
| %x            | hexadecimal integer ( <i>lowercase letters</i> )   |
| %X            | hexadecimal integer ( <i>Uppercase letters</i> )   |
| %e            | exponential notation( <i>with lowercase 'e'</i> )  |
| %E            | exponential notation ( <i>with uppercase 'E'</i> ) |
| %f            | floating point real number                         |
| %g            | the shorter of %f and %e                           |
| %G            | the shorter of %f and %E                           |

- Other supported symbols and functionality are listed in the following table:

| Symbol | Functionality                                                                                                  |
|--------|----------------------------------------------------------------------------------------------------------------|
| *      | argument specifies width or precision                                                                          |
| -      | left justification                                                                                             |
| +      | display the sign                                                                                               |
| <sp>   | leave a blank space before a positive number                                                                   |
| #      | add the octal leading zero '0' or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used. |
| 0      | pad from left with zeros ( <i>instead of spaces</i> )                                                          |
| %      | '%%' leaves you with a single literal '%'                                                                      |
| Var    | mapping variable ( <i>dictionary arguments</i> )                                                               |
| m.n.   | m is the minimum total width and n is the number of digits to display after the decimal point.                 |

- Python String formatting operators allow you to embed variables inside a string.

#### Embedding character in String using %c:

- %c is used when you want to embed a single character. You cannot add more than one character with %c.

```
val = 'R'
print("The name Raman starts with letter - %c" %val)
```

#### Output:

```
The name Raman starts with letter - R
```

#### Embedding string inside a string:

- To embed a string within a string with the help of %s. In the following code, a variable val has the value of 'Cat'. The variable val is inserted inside the string using %s.

```
val = 'Cat'
print("C For - %s" %val)
```

#### Output:

```
C For - cat
```

#### Embedding signed decimal integer – %i or %d:

- %i and %d can be used to insert a signed decimal integer in the string. In the following code, there are two variables val1(25) and val2(10). In the first string, val1 is inserted using %i and in the second string value is inserted using %d.

```
val1 = 25
val2 = 10
print("Val1 is %i" %val1)
print("Val2 is %d" %val2)
```

**Output:**

```
val1 is 25
val2 is 10
```

- Similarly, if you want to embed an unsigned integer use %u.

**Embedding octal integer %o:**

- In the following code, octal integer value inserted using %o. variable val1 has the value of 127. The value inserted is 177.

```
val1 = 127
print("Val1 is %o" %val1)
```

**Output:**

```
val1 is 177
```

**Embedding hexadecimal integer %x, %X:**

- Both %x and %X are used to insert hexadecimal integer value. The only difference is that one inserts value in lowercase letters and the other inserts values in uppercase letters.

```
val1 = 127
print("Val1 in lower case is %x" %val1)
print("Val1 in upper case is %X" %val1)
```

**Output:**

```
val1 in lower case is 7f
val1 in upper case is 7F
```

**Embedding Exponential Notation:**

- Both %e and %E are used to insert exponential value. The only difference is that one inserts value in lower case letters and the other inserts values in uppercase letters.

```
val1 = 127
print("Val1 exponential value in lower case is %e" %val1)
print("Val1 in upper case is %E" %val1)
```

**Output:**

```
val1 exponential value in lower case is 1.270000e+02
val1 in upper case is 1.270000e+02
```

**Embedding floating point real number:**

- If you want to insert a floating point real number in a string use %f.

```
val1 = 127
print("Val1 floating point value is %f" %val1)
```

**Output:**

```
val1 floating point value is 1.270000e+02
```

**2.5.6 Raw String and Unicode String**

- Python raw string is created by prefixing a string literal with 'r'.
- Raw strings do not treat the backslash as a special character at all. Every character we put into a raw string is written as it is.

**Example:** print (r'python:\\programming')

**Output:** python:\\programming

- Python strings are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode.
- Unicode strings use the prefix u.

**Example:** print (u'Hello, world!')

**Output:** Hello, world!

**2.5.7 Built-in String Methods**

- Python has several built-in methods associated with the string data type.
- In Python, the following built-in methods used to manipulate and modify strings:

| Sr. No. | Method                  | Description                                                                                                                        | Example                                                                                                            |
|---------|-------------------------|------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| 1.      | capitalize()            | Makes the first letter of the string capital.                                                                                      | <pre>&gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.capitalize() 'Python programming'</pre>                  |
| 2.      | center(width, fillchar) | Returns a space padded string with the original string centered to a total width columns.                                          | <pre>&gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; print(s1.center(30,'*')) *****python programming*****</pre> |
| 3.      | count(str, beg, end)    | Counts the number of times str occurs in the string or in a substring provided that starting index is beg and ending index is end. | <pre>&gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.count('o') 2 &gt;&gt;&gt; s1.count('o',5,18) 1</pre>     |

*contd. ...*

|    |                                       |                                                                                                                                                                |                                                                                                                                                                                                                                        |
|----|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4. | <code>decode()</code>                 | Decodes the string using the codec registered for encoding.                                                                                                    | <pre>&gt;&gt;&gt; s1 'python Programming' &gt;&gt;&gt; s1=s1.encode() &gt;&gt;&gt; s1 b'python Programming' &gt;&gt;&gt; s1=s1.decode() &gt;&gt;&gt; s1 'python Programming'</pre>                                                     |
| 5. | <code>encode()</code>                 | Returns encoded string version of string; on error, default is to raise a <code>ValueError</code> unless errors is given with 'ignore' or 'replace'.           | <pre>s1="python Programming" &gt;&gt;&gt; s1.encode() b'python Programming'</pre>                                                                                                                                                      |
| 6. | <code>endswith(sufi, beg, end)</code> | Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise. | <pre>&gt;&gt;&gt; s1="python programming is easy to learn." &gt;&gt;&gt; s1.endswith('learn') False &gt;&gt;&gt; s1.endswith('learn.') True &gt;&gt;&gt; s1.endswith('is',7,26) False &gt;&gt;&gt; s1.endswith('easy',7,26) True</pre> |
| 7. | <code>expandtab (tabsize)</code>      | It returns a copy of the string in which tab characters ie. '\t' are expanded using spaces, optionally using the given tabsize (default 8).                    | <pre>&gt;&gt;&gt; s1="python\tprogramming" &gt;&gt;&gt; s1 'python\tprogramming' &gt;&gt;&gt; s1.expandtabs() 'python programming' &gt;&gt;&gt; s1.expandtabs(16) 'python programming'</pre>                                           |
| 8. | <code>enumerate()</code>              | The <code>enumerate()</code> method adds counter to an iterable and returns it (the <code>enumerate</code> object).                                            | <pre>&gt;&gt;&gt; s1=['Orange','Mango','Banana',     'Pineapple'] &gt;&gt;&gt; s2=enumerate(s1) &gt;&gt;&gt; print(list(s2)) [(0, 'Orange'), (1, 'Mango'), (2, 'Banana'), (3, 'Pineapple')]</pre>                                      |

*contd. ...*

|     |                                   |                                                                                                                                                             |                                                                                                                                                                                                                                                                                                    |
|-----|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 9.  | <code>find(str, beg, end)</code>  | Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise. | <pre>&gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.find("prog") 7 &gt;&gt;&gt; s1.find("prog",5) 7 &gt;&gt;&gt; s1.find("prog",10) -1</pre>                                                                                                                                                 |
| 10. | <code>index(str, beg, end)</code> | Same as find(), but raises an exception if str not found.                                                                                                   | <pre>&gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.index("prog") 7 &gt;&gt;&gt; s1.index("prog",10) Traceback (most recent call last):   File "&lt;pyshell#64&gt;", line 1, in                                 &lt;module&gt;     s1.index("prog",10) ValueError: substring not found</pre> |
| 11. | <code>isalnum()</code>            | Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.                                                    | <pre>&gt;&gt;&gt; s1="python3" &gt;&gt;&gt; s1.isalnum() True &gt;&gt;&gt; s1="1234" &gt;&gt;&gt; s1.isalnum() True &gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.isalnum() False</pre>                                                                                                     |
| 12. | <code>isalpha()</code>            | Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.                                                      | <pre>&gt;&gt;&gt; s1="python" #no space and digit                         in this string &gt;&gt;&gt; s1.isalpha() True &gt;&gt;&gt; s1="python3" &gt;&gt;&gt; s1.isalpha() False &gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.isalpha() False</pre>                                       |

*contd. ...*



|     |                          |                                                                                                                      |                                                                                                                                                                                                                        |
|-----|--------------------------|----------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 13. | <code>isdigit()</code>   | Returns true if string contains only digits and false otherwise.                                                     | <pre>&gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.isdigit() False &gt;&gt;&gt; s1="python3" &gt;&gt;&gt; s1.isdigit() False &gt;&gt;&gt; s1="12345" #only digit in string &gt;&gt;&gt; s1.isdigit() True</pre> |
| 14. | <code>islower()</code>   | Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise. | <pre>&gt;&gt;&gt; s1="python" &gt;&gt;&gt; s1.islower() True &gt;&gt;&gt; s1="Python" &gt;&gt;&gt; s1.islower() False</pre>                                                                                            |
| 15. | <code>isnumeric()</code> | Returns true if a unicode string contains only numeric characters and false otherwise.                               | <pre>&gt;&gt;&gt; s1="12345" &gt;&gt;&gt; s1.isnumeric() True &gt;&gt;&gt; s1="\u00B2345" &gt;&gt;&gt; s1.isnumeric() True &gt;&gt;&gt; s1="python3" &gt;&gt;&gt; s1.isnumeric() False</pre>                           |
| 16. | <code>isspace()</code>   | Returns true if string contains only whitespace characters and false otherwise.                                      | <pre>&gt;&gt;&gt; s1=' ' &gt;&gt;&gt; s1.isspace() True &gt;&gt;&gt; s1="python programming" &gt;&gt;&gt; s1.isspace() False &gt;&gt;&gt; s1=' \t' &gt;&gt;&gt; s1.isspace() True</pre>                                |

*contd. ...*

|     |           |                                                                                                                        |                                                                                                                                                                                                                                                                               |
|-----|-----------|------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 17. | istitle() | Returns true if string is properly "titlecased" and false otherwise.                                                   | <pre>&gt;&gt;&gt; s1="Python Programming" &gt;&gt;&gt; s1.istitle() True &gt;&gt;&gt; s1="Python programming" &gt;&gt;&gt; s1.istitle() False &gt;&gt;&gt; s1="PYTHON" &gt;&gt;&gt; s1.istitle() False &gt;&gt;&gt; s1="123 Is A Number" &gt;&gt;&gt; s1.istitle() True</pre> |
| 18. | isupper() | Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise. | <pre>&gt;&gt;&gt; s1="PYTHON PROGRAMMING" &gt;&gt;&gt; s1.isupper() True &gt;&gt;&gt; s1="Python Programming" &gt;&gt;&gt; s1.isupper() False</pre>                                                                                                                           |

**Example for count method.**

```
str = "this is string example....wow!!!";
sub = "i";
print "str.count(sub, 4, 40) : ", str.count(sub, 4, 40)
sub = "wow";
print "str.count(sub) : ", str.count(sub)
```

**Output:**

```
str.count(sub, 4, 40) : 2
str.count(sub) : 1
```

**Example shows the usage of isdigit().**

```
str = "123456"; # Only digit in this string
print str.isdigit()
str = "this is string example....wow!!!";
print str.isdigit()
```

**Output:**

```
True
False
```

**Example shows the usage of isupper().**

```
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.isupper()
str = "THIS is string example....wow!!!";
print str.isupper()
```

**Output:**

```
True
False
```

---

## PROGRAMS

**Program 1:** Program to get a string made of the first 2 and the last 2 chars from a given a string. If the string length is less than 2, return instead of the empty string.

```
string = input("Enter string:")
count=0
for i in string:
 count=count+1
new=string[0:2]+string[count-2:count]
print("Newly formed string is:")
print(new)
```

**Output:**

```
Enter string: Hello world
Newly formed string is:
Held
```

**Program 2:** Program to get a string from a given string where all occurrences of its first char have been changed to '\$'.

```
str1 = input ("Enter a String:")
print("Original String:",str1)
char = str1[0]
str1 = str1.replace(char, '$')
str1 = char + str1[1:]
print("Replaced String:",str1)
```

**Output:**

```
Enter a String: onion
Original String: onion
Replaced String: oni$n
```

**Program 3:** Program to replace all occurrences of 'a' with \$ in a String.

```
string = input ("Enter string")
string=string.replace('a','$')
string=string.replace('A','$')
print("Modified string:")
print(string)
```

**Output:**

```
Enter string:Asia
Modified string: si
```

---

**Program 4:** Program to get a single string from two given strings, separated by a space and swap the first two characters of each string.

**Sample String:** 'abc', 'xyz'

**Expected Output:** 'xycabz'

```
str1=input("Enter first string:")
str2=input("Enter second string:")
new_a = str2[:2] + str1[2:]
new_b = str1[:2] + str2[2:]
print("The new string after swapping first two characters of both
string:", (new_a+' '+new_b))
```

**Output:**

```
Enter first string:python
Enter second string:Programming
The new string after swapping first two characters of both string: Prthon
programming
```

---

**Program 5:** Program to change a given string to a new string where the first and last chars have been exchanged.

```
str1=input("Enter a String:")
print("String after Swapping first and last character:",(str1[-1:]
+ str1[1:-1] + str1[:1]))
```

**Output:**

```
Enter a String:python
String after Swapping first and last character: nythop
```

**Program 6:** Program to remove the characters which have odd index values of a given string.

```
str1=input("Enter a String:")
result = " "
for i in range(0, len(str1), 2):
 if (i % 2) == 0:
 result =result + str1[i]
print("String after removing characters in odd positions:",result)
```

**Output:**

```
Enter a String: Python Programming
String after removing characters in odd positions
```

---

**Program 7:** Program to count the occurrence of given number.

```
a=[]
n=int(input("Enter number of elements:"))
for i in range(1,n+1):
 b=int(input("Enter element:"))
 a.append(b)
k=0
num=int(input("Enter the number to be counted:"))
for j in a:
 if(j==num):
 k=k+1
print("Number of times",num,"appears is",k)
```

**Output:**

```
Enter number of elements:4
Enter element:23
Enter element:45
Enter element:23
Enter element:67
Enter the number to be counted:23
Number of times 23 appears is 2
```

---

**PRACTICE QUESTIONS****Q.I Multiple Choice Questions:**

1. Which are the set of statements that are responsible to change the flow of execution of the program?  
(a) control statements (b) control loops  
(c) control loop statements (d) None of the mentioned
2. Python string is the collection of the characters surrounded by, , or.  
(a) single quotes ('...') (b) double quotes ("...")  
(c) triple quotes ("..." or ("..."")) (d) All of the mentioned
3. Which specifies the order in which the operators would be applied to the operands?  
(a) associativity (b) precedence  
(c) Both (a) and (b) (d) None of the mentioned
4. The process of converting the value of one data type (integer, string, float, etc.) to another data type is called as,  
(a) type conversion (b) type changing  
(c) type modification (d) None of the mentioned
5. A program's which flow is the order in which the program's code executes.  
(a) decision (b) statement  
(c) control (d) expression
6. Flow of control through any given program is implemented with following,  
(a) selection structure (b) sequential structure  
(c) Repetition/Iteration structure (d) All of the mentioned
7. Which control structure executes a sequence of statements in the order in which they are written?  
(a) selection (b) Sequential  
(c) Repetition/Iteration (d) All of the mentioned
8. In which the type conversion data types of objects are converted using predefined functions by the user.  
(a) explicit (b) type casting  
(c) Both (a) and (b) (d) implicit
9. Which control structure allows the execution of a block of statements multiple times until a specified condition is met?  
(a) selection (b) Sequential  
(c) decision (d) looping

10. In Python which statement is used to test a specific condition and if the condition is true, a block of code will be executed.
- (a) if (b) if else  
(c) if else if (d) None of the mentioned
11. Which statement allows us to check for multiple expressions?
- (a) if (b) if else  
(c) elif (d) None of the mentioned
12. Which control structure will execute a sequence of statements repeatedly if a condition holds true or satisfied?
- (a) selection (b) iterative/loop  
(c) sequential (d) All of the mentioned
13. Which loop can execute a set of statements as long as a condition is true?
- (a) for (b) while  
(c) do while (d) All of the mentioned
14. Which statement in Python is used when a statement is required syntactically but we do not want any command or code to execute?
- (a) pass (b) break  
(c) continue (d) All of the mentioned
15. A if statement inside another if statement called as,
- (a) nesting of if statement (b) looping of if statement  
(c) passing of if statement (d) crossing of if statement
16. Which function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.
- (a) remove() (b) index()  
(c) range() (d) All of the mentioned
17. Which statement terminates the loop statement and transfers execution to the statement immediately following the loop?
- (a) pass (b) break  
(c) continue (d) All of the mentioned
18. Which brackets can be used to access elements of the string?
- (a) curly (b) square  
(c) Both (a) and (b) (d) None of the mentioned
19. In Python, which is a sequence of Unicode (UTF-8) characters.
- (a) string (b) tuple  
(c) array (d) list
20. Which operator is used joining/combining of two or more strings into a single one (concatenation)?
- (a) \* (b) =  
(c) + (d) %

21. Python raw string is created by prefixing a string literal with,  
(a) u (b) r  
(c) t (d) n
22. What is the output of the following code?  

```
example = "snow world"
example[3] = 's'
print example
```

  
(a) snow (b) snow world  
(c) error (d) snos world
23. What is the output of the following code?  

```
a,b=6,7
a,b=b,a
print a,b
```

  
(a) (6,7) (b) invalid syntax  
(c) (7,6) (d) (6,6)
24. What is the output of the following loop?  

```
for l in 'Jhon':
 if l == 'o':
 pass
 print(l, end=", ")
```

  
(a) J, h, n, (b) J, h, o, n,  
(c) J,h,o (d) n,o,h,J
25. In the following code how many times will the loop run?  

```
i=2
while(i>0):
 i=i-1
```

  
(a) 2 (b) 1  
(c) 3 (d) 4
26. What is the value of x after the following nested for loop completes its execution?  

```
x = 0
for i in range(10):
 for j in range(-1, -10, -1):
 x += 1
 print(x)
```

  
(a) 99 (b) 90  
(c) 100 (d) 85
27. What is the output of the following range() function?  

```
for num in range(2,-5,-1):
 print(num, end=", ")
```

  
(a) 2, 1, 0 (b) 2, 1, 0, -1, -2, -3, -4  
(c) 2, 1, 0, -1, -2, -3, -4, -5 (d) 2, 1, 0, -1, -2, -3



28. In a string range of slice shown by character,  
 (a) [...] (b) [:]  
 (c) [--] (d) \*\*
29. Unicode string uses the prefix,  
 (a) U/u (b) R/r  
 (c) Q/q (d) S/s
30. What is the output of the following code:
- ```
a = 67
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```
- (a) a and b are equal (b) b is greater than a
 (c) a is greater than b (d) Error

Answers

1. (a)	2. (d)	3. (b)	4. (a)	5. (c)	6. (d)	7. (b)	8. (c)	9. (d)	10. (a)
11. (c)	12. (b)	13. (b)	14. (a)	15. (a)	16. (c)	17. (b)	18. (b)	19. (a)	20. (a)
21. (b)	22. (c)	23. (c)	24. (b)	25. (a)	26. (b)	27. (c)	28. (b)	29. (a)	30. (c)

Q.II Fill in the Blanks:

- _____ like +, * etc. are used to perform operations on variables and values.
- Operator _____ is the condition that specifies the importance of each operator relative to the others.
- In Python, a string is a sequence of _____ characters.
- _____ control in Python is the order in which statements or blocks of code are executed at runtime based on a condition.
- The _____ statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.
- In _____ casting, we convert variables declared in specific data types to the different data types.
- To use one loop inside another loop called as _____ loop.
- In Python, _____ statements act depending on whether a given condition is true or false.

9. A loop statement allows us to execute a statement or group of statements _____ times.
10. _____ in python are surrounded by either single quotation marks 'Welcome', or double quotation marks "Welcome".
11. _____ decides the order in which the operators with same precedence are executed.
12. The _____ statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.
13. An _____ is the combination of variables and operators that evaluate based on operator precedence.
14. The _____ statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.
15. In Python programming, the _____ statement is a null statement means does nothing.
16. Loop _____ statements change execution from its normal sequence.
17. We can generate a sequence of numbers using _____ function.
18. To concatenate or combine, two strings we can use the _____ operator.
19. An escape character is a backslash _____ followed by the character we want to insert.
20. The _____ function formats specified values in a string.
21. Unicode strings are stored as _____ Unicode.
22. In _____ type conversion, Python automatically converts one data type to another data type.
23. The range slice [:] gives the _____ from the given range.
24. The _____ function converts a single character to its integer value.
25. _____ execution is when statements are executed one after another in order n sequential manner.
26. The string format operator _____ is unique to strings and makes up for the pack of having functions from C's printf() family.
27. The _____ function merges (concatenates/combines) the string representations of elements in sequence into a string, with separator string.

Answers

1. Operators	2. precedence	3. Unicode	4. Flow
5. break	6. type	7. nested	8. selection
9. multiple	10. Strings	11. Associativity	12. continue
13. expression	14. if-else	15. pass	16. control
17. range()	18. +	19. \	20. format()
21. 16-bit	22. implicit	23. characters	24. ord(x)
25. Sequential	26. %	27. join()	

Q.III State True or False:

1. The break statement can be used in both while and for loops.
2. The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
3. The pass statement is useful when we want to write the pseudo code that you want to implement it in the future.
4. We use a break statement to terminates the loop and transfers execution to the statement immediately following the loop.
5. While loop is used when multiple statements are to executed repeatedly until the given condition becomes False.
6. The for loop can be used to iterate through the elements of lists.
7. The flow of the programs written in any programming language is sequential by default.
8. Loops should be ended with keyword "end".
9. The keyword "continue" is used to continue with the remaining statements inside the loop.
10. The len(string) function returns the length of the string.
11. The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first.
12. The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block.
13. The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion.
14. No loop can be used to iterate through the elements of strings.
15. We can assign a multiline string to a variable by using three quotes [("""...""") or ("""...""")].
16. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

17. The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied.
18. The count() function counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
19. With the break statement we can stop the current iteration of the loop, and continue with the next.
20. The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.
21. A normal string, when prefixed with 'r' or 'R' becomes a raw string.
22. The upper() function capitalizes first letter of string.
23. The str(x) function converts object x to a string representation.
24. An iteration or loop control structure is one which will execute a sequence of statements repeatedly if a condition holds true or satisfied.
25. The elif is short for else if which is used to check for multiple conditions.
26. The range() function returns a sequence of numbers, starting from 1 by default.
27. Individual characters in a string can be accessed using an index with [].
28. Unicode strings in Python use the prefix u.

Answers

1. (T)	2. (T)	3. (T)	4. (T)	5. (T)	6. (T)	7. (T)	8. (F)	9. (F)	10. (T)
11. (T)	12. (T)	13. (T)	14. (F)	15. (T)	16. (T)	17. (T)	18. (T)	19. (F)	20. (T)
21. (T)	22. (F)	23. (T)	24. (T)	25. (T)	26. (F)	27. (T)	28. (T)		

Q.IV Answer the following Questions:

(A) Short Answer Questions:

1. Define operator precedence.
2. What is type casting?
3. Define control flow in a program.
4. Give purpose of selection statements.
5. What is the role of % operator?
6. Define string.
7. Which character prefixed before a string as raw string?
8. What is the purpose of range() function?
9. Give syntax for if else statement.
10. What is looping statements?
11. List types of tyte conversions.
12. Give the use of index() in string.
13. What are break and continue statements?
14. Compare for and while loop (any two points).

(B) Long Answer Questions:

1. Explain operator precedence and associativity in detail.
2. How to convert one data types to another? Describe with example.
3. What is string? How to declare it? Explain with example.
4. Explain the following statements:
(i) if (ii) if else (iii) break (iv) continue.
5. Describe manipulation of string with example.
6. Write short note on: Unicode strings.
7. Write program to find factorial of a number.
8. What is nested loop? How to use it in program?
9. Describe the term escape character in detail.
10. List built-in string methods with example.
11. With the help of program describe pass statement.
12. Describe sequential flow control with example.
13. Write program for greatest number form three numbers using if else statement.
14. With te help of example describe following loops:
(i) while (ii) for.

■■■

Lists, Functions, Tuples, Dictionaries and Sets

Objectives...

- To learn Concepts Lists
- To study Functions in Python
- To understand Tuples in Python
- To learn Dictionary and Sets in Python

3.0 INTRODUCTION

- List is one of the most frequently used and very versatile data types used in Python. A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists.
- A function is a block of organized, reusable code that is used to perform a single, related action. Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.
- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it. Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.
- In this chapter we will study basic concepts of lists, tuples, functions, dictionaries, sets in Python programming language.

3.1 LISTS

- A Python list is a mutable sequence of data values called items or elements. An item can be of any type.
- List is one of the most frequently used and very versatile data type used in Python. The elements in the list are stored in a linear order one after other.
- A list is a collection of items or elements; the sequence of data in a list is ordered.
- The elements or items in a list can be accessed by their positions i.e. indices. The index in lists always starts with 0 and ends with n-1, if the list contains n elements.

Concept of List:

- The list is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets.
- In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.
- It can have any number of items and they may be of different types (integer, float, string etc.).

Example:

```
list1 = []; #Empty list
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

3.1.1 Creating Elements in a List

- A list in Python is created by placing all the items (elements) inside a square bracket [], separated by commas.
- Lists can have any number of items and they may be of different types (integer, float, string etc.).

Defining List:

- In Python lists are written with square brackets. A list is created by placing all the items (elements) inside a square brackets [], separated by commas.

Syntax for defining a list in Python: <list_name> = [value1, value2, ... valueN]

Here, list_name is the name of the list and value1, value2, valueN are list of values assigned to the list.

Example: Emp = [20, "Amar", 'M', 50]

- Lists are mutable or changeable. The value of any element inside the list can be changed at any point of time. Each element or value that is inside of a list is called an item.
- A list data type is a flexible data type that can be modified according to the requirements, which makes it a mutable data type.
- The empty Python list is written as two square brackets containing nothing. We can create a list by placing a comma separated sequence of values in square brackets [].
- The simplest method to create a list by simply assigning a set of values to the list using the assignment operator (=).

Example 1: For creating an empty list.

```
>>> l1=[]          # Empty list
>>> l1             # display l1
[]
>>> l1=list()      # Using list() constructor
>>> l1
[]
```

Example 2: For creating a list with any integer elements.

```
>>> l2=[10,20,30] # List of Integers
>>> l2
[10, 20, 30]
>>> l2=list([10,20,30])
>>> l2
[10, 20, 30]
```

Example 3: For creating a list with string elements.

```
>>> l3=["Mango","Orange","Banana"] # List of Strings
>>> l3
['Mango', 'Orange', 'Banana']
>>> l3=list(["red","yellow","green"] )
>>> l3
['red', 'yellow', 'green']
```

Example 4: For creating a list with mixed data.

```
>>> l4=[1,"Two",11.22,'X'] # List of mixed data types
>>> l4
[1, 'Two', 11.22, 'X']
```

- We can convert other data types to lists using Python's list() constructor.

Example 1: For creating a list using inbuilt range() function.

```
>>> l5=list(range(0,5))
>>> l5
[0, 1, 2, 3, 4]
```

Example 2: For creating a list with in-built characters A, B and C.

```
>>> l6=list("ABC")
>>> l6
['A', 'B', 'C']
```

3.1.2 Accessing Elements of a List

- There are various ways in which we can access the elements of a list.
 1. **List Index:** We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4. Trying to access an element other than this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.
 2. **Negative Indexing:** Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

Slicing List:

- We can access a range of items in a list by using the slicing operator : (colon). Slicing can be best visualized by considering the index to be between the elements. So if we want to access a range, we need two indexes that will slice that portion from the list.
- Accessing elements/items from a list in Python programming is a method to get values that are stored in the list at a particular location or index.
- To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.
- **Syntax** to access Python list is: <list_name>[index]

Example: For accessing list values.

```
>>> list1 = ["one", "two", 3, 10, "six", 20]
>>> list1[0]           # positive indexing
'one'
>>> list1[-2]          # negative indexing
'six'
>>> list1[1:3]          # get element from mth index to n-1 index
['two', 3]
>>> list1[3:]           # get element from mth index to last index
[10, 'six', 20]
>>> list1[:4]           # get element from 0th index to n-1 index
['one', 'two', 3, 10]
>>>
```

-
- Accessing elements from a particular list in Python at once allows the user to access all the values from the lists. This is possible by writing the list name in the print() statement.
 - For example, print[list1]

3.1.3 Updating List

-
- Lists are mutable, meaning their elements can be changed or updated unlike string or tuple.
 - Mutability is the ability for certain types of data to be changed without entirely recreating it. Using mutable data types can allow programs to operate quickly and efficiently.
 - Multiple values can be added into list. We can use assignment operator (=) to change an item or a range of items.
 - We can update items of the list by simply assigning the value at the particular index position. We can also remove the items from the list using remove() or pop() or del statement.

Example: For updating lists.

```
>>> list1= [10, 20, 30, 40, 50]
>>> list1
[10, 20, 30, 40, 50]
>>> list1[0]=0                # change 0th index element
>>> list1
[0, 20, 30, 40, 50]
>>> list1[-1]=60              # change last index element
>>> list1
[0, 20, 30, 40, 60]
>>> list1[1]=[5,10]           # change 1st index element as sublist
>>> list1
[0, [5, 10], 30, 40, 60]
>>> list1[1:1]=[3,4]           # add elements to a list at the desired
location
>>> list1
[0, 3, 4, [5, 10], 30, 40, 60]
```

- Following table shows the list methods used for updating list.

Sr. No.	Method	Syntax	Argument Description	Return Type
1.	append()	list.append(item)	The item can be numbers, strings, another list, dictionary etc.	Only modifies the original list. It does not return any value.
2.	extend()	list1.extend(list2)	This extend() method takes a list and adds it to the end.	Only modifies the original list. It doesn't return any value.
3.	insert()	list.insert(index, element)	This index is position where an element needs to be inserted element - this the element to be inserted in the list.	It does not return anything; returns None.

- Let us see above methods in detail:

1. append() Method:

- The append() method adds an element to the end of a list. We can insert a single item in the list data time with the append().

Example: For append() method.

```
>>> list1=[10,20,30]
>>> list1
[10, 20, 30]
>>> list1.append(40)           # add element at the end of list
>>> list1
[10, 20, 30, 40]
```

2. extend() Method:

- The extend() method extends a list by appending items. We can add several items using extend() method.

Example: Program for extend() method.

```
>>> list1=[10, 20, 30, 40]
>>> list1
[10, 20, 30, 40]
>>> list1.extend([60,70])      # add elements at the end of list
>>> list1
[10, 20, 30, 40, 60, 70]
```

3. insert() Method:

- We can insert one single item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

Example: Program for insert() method.

```
>>> list1=[10, 20]
>>> list1
[10,20]
>>> list1.insert(1,30)
>>> list1
[10, 30, 20]
>>> list1.insert(1,[15,25])
>>> list1
[10,[15, 25], 30, 20]
```

4. + Operator:

- The concatenation operation in Python programming is used to combine the elements/items or two lists. We use + operator to combine two lists.

Example: Using + operator to combine with list.

```
>>> list1=[10,20,30]
>>> list1
[10, 20, 30]
>>> list1 + [40,50,60]          # using + to combine two lists
[10, 20, 30, 40, 50, 60]
>>> list2=["A","B"]
>>> list1 + list2
[10, 20, 30, 'A', 'B']
```

5. * Operator:

- Sometimes, there is a need or requirement to repeat all the items of the lists as specific number of times. The * operator repeats a list for the given number of times.

Example: Using * operator with list.

```
>>> list2=['A', 'B']
>>> list2
['A', 'B']
>>> list2 *2                      # using * to repeat a list
['A', 'B', 'A', 'B']
```

6. sort() Method:

- It arranges the list items in ascending order or descending order. The sort() is called by a list items and sort the list by default in ascending order.

Example: For sort() method.

```
>>> list=[4,2,5,1,7,3]
>>> list.sort()
>>> list
[1, 2, 3, 4, 5, 7]
>>> list.sort(reverse=True)
>>> list
[7, 5, 4, 3, 2, 1]
```

3.1.4 Deleting List

- Python provides many ways in which the elements in a list can be deleted. Some of them are explained below:

1. pop() Method:

- The pop() method in Python is used to remove a particular item/element from the given index in the list.
 - The pop() method removes and returns the last item if index is not provided. This helps us implement lists as stack data structure.
-

Example: For pop() method.

```
>>> list = [10, 20, 30, 40]
>>> list
[10, 20, 30, 40]
>>> list.pop(2)          # pop with index
30
>>> list
[10, 20, 40]
>>> list.pop()           # pop without index
40
>>> list
[10, 30]
```

2. Using del Keyword:

- We can delete one or more items from a list using the keyword 'del'. It can even delete the list entirely. But it does not store the value for further use.

Example: Using del keyword.

```
>>> list = [10, 20, 30, 40]
>>> list
[10, 20, 30, 40]
>>> del (list[1])        # del() with index
>>> list
[10, 30, 40]
>>> del list[2]          # del with index
>>> list
[10, 30]
>>> del list              # del without index
>>> list
<class 'list'>
```

3. remove() Method:

- The remove() method in Python is used to remove a particular element from the list.
- We use the remove() method if we know the item that we want to remove or delete from the list (but not the index).

Example: For remove() method.

```
>>> list=[10,"one",20,"two"]    # heterogeneous list
>>> list.remove(20)             # remove element 20
>>> list
[10, 'one', 'two']
>>> list.remove("one")          # remove element one
>>> list
[10, 'two']
>>>
```

4. clear() Method:

- The clear() method is used to clear the complete list.

Example:

```
or_list = [1,2,3,4]
print (or_list)
or_list.clear()
print (or_list)
```

Output:

```
[1, 2, 3, 4]
[]
```

3.1.5 Basic List Operations

- The operations on list include indexing and slicing explained in this section.

Indexing:

- An individual item in the list can be referenced by using an index, which is an integer number that indicates the relative position of the item in the list.
- There are various ways in which we can access the elements of a list, some as they are given below:

1. **List Index:** We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

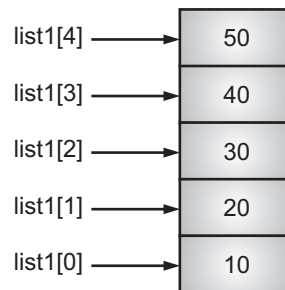


Fig. 3.1: List has Five Elements with Index 0 to 4

Example: For list index in list.

```
>>> list1=[10,20,30,40,50]
>>> list1[0]
10
>>> list1[3:] # list[m:] will return elements indexed
               from mth index to last index
[40, 50]
```

```
>>> list1[:4] # list[:n] will return elements indexed from
                                     first index to n-1th index
[10, 20, 30, 40]
>>> list1[1:3] # list[m:n] will return elements indexed from m to n-1.
[20, 30]
>>> list1[5]
Traceback (most recent call last):
File "<pyshell#71>", line 1, in <module>
list1[5]
IndexError: list index out of range
```

2. **Negative Indexing:** Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

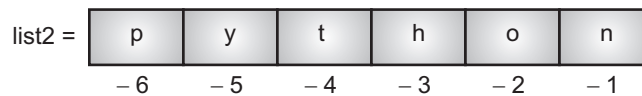


Fig. 3.2: List with Negative Index

Example: For negative indexing in list.

```
>>> list2=['p','y','t','h','o','n']
>>> list2[-1]
'n'
>>> list2[-6]
'p'
>>> list2[-3:]
['h', 'o', 'n']
>>> list2[-7]
Traceback (most recent call last):
File "<pyshell#76>", line 1, in <module>
list2[-7]
IndexError: list index out of range
```

Slicing:

- Slicing is an operation that allows us to extract elements from units. The slicing feature used by Python to obtain a specific subset or element of the data structure using the colon (:) operator.
- The slicing operator returns a subset of a list called slice by specifying two indices, i.e. start and end.

Syntax: list_variable[start_index:end_index]

- This will return the subset of the list starting from start_index to one index less than that of the end index.

Example: For slicing list.

```
>>> l1=[10,20,30,40,50])
>>> l1[1:4]
[20, 30, 40]
>>> l1[2:5]
[30,40,50]
```

List Slicing with Step Size:

- Step is the integer value which determines the increment between each index for slicing.

Syntax: list_name[start_index:end_index:step_size]

Example 1: For slicing operation in list.

```
>>> l1=['Red', 1, 'yellow', 2, 'Green', 3, 'Blue', 4]
>>>l1
['Red', 1, 'yellow', 2, 'Green', 3, 'Blue', 4]
>>> l2=l1[0:6:2]
>>> l2
['Red', 'yellow', 'Green']
```

Example 2: Complex example of list slicing.

```
>>> l1                # list of five elements
[10, 20, 30, 40, 50]
>>> l1[::-1]          # display list in reverse order
[50, 40, 30, 20, 10]
>>> l1[-1:0:-1]        # start index with -1 and end index with 0 and step
size with -1
[50, 40, 30, 20]
```

3.1.6 Traversing a List

- Traversing a list means accessing all the elements or items of the list.
- Traversing can be done by using any conditional statement of Python, but it is preferable to use for loop.

Example 1:

```
list=[10,20,30,40]
for x in list:
    print(x)
```


Output:

```
10
20
30
40
```

Example 2:

```
list1=[[1,2,3,4],['A','B','C','D'],['@','#','$','%']]
for i in list1:
    for j in i:
        print(j,end=' ')
    print('\n')
```

Output:

```
1 2 3 4
A B C D
@ # $ %
```

3.1.7 Reverse a List

- Reverses the order of the elements in the list, this places the final elements at the beginning, and the initial elements at the end.
- Python provides us with various ways of reversing a list. We will go through few of the many techniques on how a list in python can be reversed.

Example:

Input: list = [10, 11, 12, 13, 14, 15]

Output: [15, 14, 13, 12, 11, 10]

Method 1 (Using the reversed() Built-in Function):

- In this method, we neither reverse a list in-place(modify the original list), nor we create any copy of the list. Instead, we get a reverse iterator which we use to cycle through the list.

```
# Reversing a list using reversed()
mylist = [1, 2, 3, 4, 5]
for item in reversed(mylist):
    print(item)
```

Output:

```
5
4
3
2
1
>>> mylist
>>> [1, 2, 3, 4, 5]
```

Method 2 (Using the reverse() Built-in Function):

- Using the reverse() method we can reverse the contents of the list object in-place i.e., we don't need to create a new list instead we just copy the existing elements to the original list in reverse order.
- The remove() method directly modifies the original list. It is a fast process which doesn't take extra memory.

```
# Reversing a list using reverse()

>>> mylist = [1, 2, 3, 4, 5]

>>> mylist

[1, 2, 3, 4, 5]

>>> mylist.reverse()

None

>>> mylist

[5, 4, 3, 2, 1]
```

Method 3 (Using the Slicing Technique):

- In this technique, a copy of the list is made and the list is not sorted in-place.
- Creating a copy requires more space to hold all of the existing elements. This exhausts more memory.

```
# Reversing a list using the slicing technique

>>> mylist

[1, 2, 3, 4, 5]

>>> mylist[::-1]

[5, 4, 3, 2, 1]
```

3.1.8 Basic List Operations

- List operations are the operations that can be performed on the data in the list.
- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.
- In fact, lists respond to all of the general sequence operations we used on strings in the previous section.

Table 3.1: List of general String operations

Python Expression	Results	Description
<code>len([4, 5, 6,7])</code>	4	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 3</code>	<code>['Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for i in [1, 2, 3]: print i,</code>	1 2 3	Iteration
<code>my_list = [5, 1, 8, 3, 7] print(8 in my_list) print(0 in my_list)</code>	True False	In operator (lets us loop through all the members of a collection(such as a list or a tuple) and check if there's a member in the list that's equal to the given item.)

3.1.9 Built-in List Functions and Methods

- Python provide certain built-in methods and functions work with list.
- The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. A number of built-in functions create or deal with lists.
- Python includes the following built-in list functions:

Sr. No.	Built-in Function	Description	Example
1.	<code>len(list)</code>	It returns the length of the list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> len(list1) 5</pre>
2.	<code>max(list)</code>	It returns the item that has the maximum value in a list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> max(list1) 5</pre>
3.	<code>sum(list)</code>	Calculates sum of all the elements of list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> sum(list1) 15</pre>

contd. ...

4.	<code>min(list)</code>	It returns the item that has the minimum value in a list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> min(list1) 1</pre>
5.	<code>list(seq)</code>	It converts a tuple into a list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> list(list1) [1, 2, 3, 4, 5]</pre>

Methods of List:

Sr. No.	Methods	Description	Example
1.	<code>list.append(item)</code>	It adds the item to the end of the list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> list1.append(6) >>> list1 [1, 2, 3, 4, 5, 6]</pre>
2.	<code>list.count(item)</code>	It returns number of times the item occurs in the list.	<pre>>>> list1 [1, 2, 3, 4, 5, 6, 3] >>> list1.count(3) 2</pre>
3.	<code>list.extend(seq)</code>	It adds the elements of the sequence at the end of the list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> list2 ['A', 'B', 'C'] >>> list1.extend(list2) >>> list1 [1, 2, 3, 4, 5, 'A', 'B', 'C']</pre>
4.	<code>list.index(item)</code>	It returns the index number of the item. If item appears more than one time, it returns the lowest index number.	<pre>>>> list1=[1,2,3,4,5,3] >>> list1 [1, 2, 3, 4, 5, 3] >>> list1.index(3) 2</pre>

contd. ...

5.	<code>list.insert(index,item)</code>	It inserts the given item onto the given index number while the elements in the list take one right shift.	<pre>>>> list1 [1, 2, 3, 4, 5, 3] >>> list1.insert(2,7) >>> list1 [1, 2, 7, 3, 4, 5, 3]</pre>
6.	<code>list.pop(item=list[-1])</code>	It deletes and returns the last element of the list.	<pre>>>> list1 [1, 2, 7, 3, 4, 5, 3] >>> list1.pop() 3 >>> list1.pop(2) 7</pre>
7.	<code>list.remove(item)</code>	It deletes the given item from the list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> list1.remove(3) >>> list1 [1, 2, 4, 5]</pre>
8.	<code>list.reverse()</code>	It reverses the position (index number) of the items in the list.	<pre>>>> list1 [1, 2, 3, 4, 5] >>> list1.reverse() >>> list1 [5, 4, 3, 2, 1]</pre>
9.	<code>list.sort()</code>	Sorts items in the list.	<pre>>>> list1 [1, 3, 2, 4, 5] >>> list1.sort() >>> [1, 3, 2, 4, 5]</pre>
10.	<code>list.sort([func])</code>	It sorts the elements inside the list and uses compare function if provided.	<pre>>>> list1 [1, 3, 2, 5, 4] >>> list1.sort() >>> list1 [1, 2, 3, 4, 5]</pre>

3.2 FUNCTIONS

- A function is a block of organized, reusable code that is used to perform a single, related action/operation. Python has excellent support for functions.
- A function can be defined as the organized block of reusable code which can be called whenever required. A function is a piece of code that performs a particular task.
- A function is a block of code which only runs when it is called. Python gives us many built-in functions like `print()` but we can also create our own functions called as user-defined functions.

3.2.1 Definitions and Uses

- A function is defined as a block of organized, reusable code used to perform a single, related action.
- To provide a high degree of modularity functions are used. Functions are used to group together a certain number of related instructions. These are reusable blocks of codes written to carry out a specific task.
- Functions are only executed when they are specifically called. Depending on the task a function is supposed to carry out, it might or might not return a value.
- We can use functions in Python programming to bundle a set of instructions that we want to use repeatedly in the program.
- A function is a self-contained block of statements written to carry out a specified task. There are following two types of functions:
 1. **Built-in Functions:** Functions that are predefined or built-in and organized into a library.
 2. **User-Defined Functions:** Functions that are created by the programmer to meet his/her requirements.

Advantages of Functions:

1. To makes the code easier to manage, debug, and scale, functions are used.
2. Every time you need to execute a sequence of statements, all we need to do is to call the function, so that we can reuse the code.
3. Functions allow us to change functionality easily, and different programmers can work on different functions.

3.2.2 Defining Functions

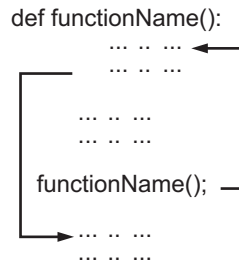
- Function definition is a block where the statements inside the function body are written. Functions allow us to define a reusable block of code that can be used repeatedly in a program.
- Function blocks begin with the keyword `def` followed by the function name and parentheses `()`.

- Any input parameters or arguments should be placed within these parentheses. We can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon: and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Syntax:

```
def function_name( param1, param2,... ):
    """function_docstring"""
    function_body...
    return [expression]
```

- By default, parameters have a positional behavior and you need to inform them in the same order that they were defined. Every function must be defined before it is called.

**Fig. 3.3: How function works in Python?****Example:**

```
def printme(str):
    "This prints a passed string into the function"
    print str
    return
```

3.2.3 Function Calling

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, we can execute it by calling it from another function or directly from the Python prompt.
- To call a function means that we are telling the program to execute the function. If there is a return value defined, the function would return the value, else the function would return `None`.

- To call a function, use the function name followed by parenthesis, if we need to pass parameters/arguments to the function, write them inside the parentheses.

Syntax: function_name(arg1, arg2)

Example:

1. Call a function that performs a task and has no return value.

```
def my_function():  
    print("Hello from a function")  
my_function()
```

Output: Hello from a function

2. Call a function with a return value.

```
def my_function(name):  
    print("Hello" + name)  
my_function("Omkar")  
my_function("Shanti")  
my_function("Ramesh")
```

Output:

```
Hello Omkar  
Hello Shanti  
Hello Ramesh
```

3. To call a function with arguments.

```
def avg_number(x, y):  
    print("Average of ",x," and ",y, " is ",(x+y)/2)  
avg_number(3, 4)
```

Output:

```
Average of 3 and 4 is 3.5
```

4. Example for calling a function.

```
>>> def square(x): # function definition  
    return x*x  
>>> square(4)      # function call  
16  
>>>
```

Program 1: Program to find HCF or GCD.

```
# define a function  
def calhcf(x, y):  
    # choose the smaller number  
    if x > y:  
        smaller = y
```

```
    else:
        smaller = x
    for i in range(1, smaller+1):
        if((x % i == 0) and (y % i == 0)):
            hcf = i
    return hcf

num1 = 54
num2 = 24
# take input from the user
# num1 = int(input("Enter first number: "))
# num2 = int(input("Enter second number: "))
print("The H.C.F. of", num1,"and", num2,"is", calhcf(num1, num2))
```

Output:

The H.C.F. of 54 and 24 is 6

Program 2: Program to find the largest among three numbers.

```
x = float(input("Enter first number: "))
y = float(input("Enter second number: "))
z = float(input("Enter third number: "))
def max_of_three(x, y, z):
    if (x >= y) and (x >= z):
        largest = x
    elif (y >= x) and (y >= z):
        largest = y
    else:
        largest = z
    return(largest)
print("The largest number is" max_of_three(x, y, z))
```

Output:

```
Enter first number: 12
Enter second number: 15
Enter third number: 80
The largest number is : 80
```

3.2.4 Type Conversion Functions

- Python defines type conversion functions to directly convert one data type to another. The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion.
- Python has two types of type conversion.

1. **Implicit Type Conversion:** In this type conversion, Python automatically converts one data type to another data type. In the following example above, we have taken and added two variables of integer and float data. In addition, we have declared and stored the result of the added variable called 'sum'. By checking the sum variable data type, the Python compiler transforms the sum variable data type automatically into the float data type. This is referred to as conversion of implicit type.

```
value1 = 10
value2 = 10.5
sum = value1 + value2
print (sum)
print (type (sum))
```

Output:

20.5

2. **Explicit Type Conversion:** In type conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc. to perform explicit type conversion. This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax: (required_data_type)(expression)

Example:

```
# initializing string
s = "10010"
# printing string converting to int base 2
c = int(s,2)
print ("After converting to integer base 2 : ", end="")
print (c)
# printing string converting to float
e = float(s)
print ("After converting to float : ", end="")
print (e)
```

Output:

```
After converting to integer base 2 : 18
After converting to float : 10010.0
```

- Sometimes, we may need to perform conversions between the built-in types. To convert between types, we simply use the type name as a function.
- There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr. No.	Function	Description	Example
1.	<code>int(x [,base])</code>	Converts x to an integer. base specifies the base if x is a string.	<code>x=int('1100',base=2)=12</code> <code>x=int('1234',base=8)=668</code>
2.	<code>long(x [,base])</code>	Converts x to a long integer. base specifies the base if x is a string.	<code>x=long('123',base=8)=83L</code> <code>x=long('11',base=16)=17L</code>
3.	<code>float(x)</code>	Converts x to a floating-point number.	<code>x=float('123.45')=123.45</code>
4.	<code>complex(real[, imag])</code>	Creates a complex number.	<code>x=complex(1,2) = (1+2j)</code>
5.	<code>str(x)</code>	Converts object x to a string representation.	<code>x=str(10) = '10'</code>
6.	<code>repr(x)</code>	Converts object x to an expression string.	<code>x=repr(3) = 3</code>
7.	<code>eval(str)</code>	Evaluates a string and returns an object.	<code>x=eval('1+2') = 3</code>
8.	<code>tuple(s)</code>	Converts s to a tuple.	<code>x=tuple('123') = ('1', '2', '3')</code> <code>x=tuple([123]) = (123,)</code>
9.	<code>list(s)</code>	Converts s to a list.	<code>x=list('123') = ['1', '2', '3']</code> <code>x=list(['12']) = ['12']</code>
10.	<code>set(s)</code>	Converts s to a set.	<code>x=set('Python') = {'y', 't', 'o', 'P', 'n', 'h'}</code>
11.	<code>dict(d)</code>	Creates a dictionary. d must be a sequence of (key, value) tuples.	<code>dict={'id':'11','name':'vijay'}</code> <code>print(dict)</code> <code>x={'id': '11', 'name': 'vijay'}</code>

contd. ...

12.	<code>chr(x)</code>	Converts an integer to a character.	<code>x=chr(65) = 'A'</code>
13.	<code>unichr(x)</code>	Converts an integer to a Unicode character.	<code>x=unichr(65) =u'A'</code>
14.	<code>ord(x)</code>	Converts a single character to its integer value.	<code>x=ord('A')= 65</code>
15.	<code>hex(x)</code>	Converts an integer to a hexadecimal string.	<code>x=hex(12) = 0xc</code>
16.	<code>oct(x)</code>	Converts an integer to an octal string.	<code>x=oct(8) = 0o10</code>

3.2.5 Math Functions

- Function can be described as a piece of code that may or may not take some value(s) as input, process it, and then finally may or may not return any value as output.
- Python's math module is used to solve problems related to mathematical calculations. Some functions are directly executed for maths functions we need to import math module first.
- In python, there are two types of pre-defined functions.

1. Built-In Functions (Mathematical):

- These are the functions which doesn't require any external code file/Modules/ Library Files. These are a part of the python core and are just built within the Python compiler hence there is no need of importing these modules/libraries in our code.
- Following table shows some of in built mathematical functions:

Sr. No.	Functions	Description	Example
1.	<code>min()</code>	Returns smallest value among supplied arguments.	<pre>>>> min(20, 10, 30) 10</pre>
2.	<code>max()</code>	Returns largest value among supplied arguments.	<pre>>>> max(20, 10, 30) 30</pre>
3.	<code>pow()</code>	The <code>pow()</code> function returns the value of x to the power of y (xy). If a third parameter is present, it returns x to the power of y, modulus z.	<pre>>>> pow(2,3) 8 >>> pow(2,3,2) 0</pre>
4.	<code>round()</code>	The <code>round()</code> function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals. The default number of decimals is 0, meaning that the function will return the nearest integer.	<pre>>>> round(10.2345) 10 >>> round(5.76543) 6 >>> round(5.76543,2) 5.77</pre>
5.	<code>abs()</code>	Absolute function, also known as Modulus (not to be confused with Modulo), returns the non-negative value of the argument value.	<pre>>>> abs(-5) 5 >>> abs(5) 5</pre>

2. Built-In Functions (Math Module):

- The second type of functions require some external files (modules) in order to be used. The process of using these external files in our code is called importing.
- So all we have to do is import the file into our code and use the functions which are already written in that file.
- Following table shows some of in built mathematical functions of Math module:

Sr. No.	Functions	Description	Example
1.	ceil()	This function returns the smallest integral value greater than the number. If number is already integer, same number is returned.	>>> math.ceil(2.3) 3
2.	floor():	This function returns the greatest integral value smaller than the number. If number is already integer, same number is returned.	>>> math.floor(2.3) 2
3.	cos()	This function returns the cosine of value passed as argument. The value passed in this function should be in radians.	>>> math.cos(3) -0.9899924966004454 >>> math.cos(-3) -0.9899924966004454 >>> math.cos(0) 1.0
4.	cosh()	Returns the hyperbolic cosine of x.	>>> print(math.cosh(3)) 10.06766199577765
5.	copysign()	Return x with the sign of y. On a platform that supports signed zeros, copysign(1.0, -0.0) returns -1.0.	>>> math.copysign(10, -12) -10.0
6.	exp()	The method exp() returns returns exponential of x.	>>> math.exp(1) 2.718281828459045 >>>
7.	fabs()	This function will return an absolute or positive value.	>>> math.fabs(10) 10.0 >>> math.fabs(-20) 20.0
8.	factorial()	Returns the factorial of x.	>>> math.factorial(5) 120
9.	fmod()	This function returns $x \% y$.	>>> math.fmod(50,10) 0.0 >>> math.fmod(50,20) 10.0

contd. ...

10.	<code>log(a, (Base))</code>	This function is used to compute the natural logarithm (Base e) of a.	<pre>>>> print(math.log(14)) 2.6390573296152584</pre>
11.	<code>log2(a)</code>	This function is used to compute the logarithm base 2 of a. Displays more accurate result than <code>log(a,2)</code> .	<pre>>>> rint(math.log2(14)) 3.807354922057604</pre>
12.	<code>log10(a)</code>	This function is used to compute the logarithm base 10 of a. Displays more accurate result than <code>log(a,10)</code> .	<pre>>>> print(math.log10(14)) 1.146128035678238</pre>
13.	<code>sqrt()</code>	The method <code>sqrt()</code> returns the square root of x for x > 0.	<pre>>>> math.sqrt(100) 10.0 >>> math.sqrt(5) 2.23606797749979</pre>
14.	<code>trunc()</code>	This function returns the truncated integer of x.	<pre>>>> math.trunc(3.354) 3</pre>

3.2.6 Composition

- We can call one function from within another. This ability is called composition.
- As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.
- Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points.
- We just wrote a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```
- The next step is to find the area of a circle with that radius; we just wrote that, too:

```
result = area(radius)
```
- Encapsulating these steps in a function, we get:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```
- The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

3.2.7 Adding New Functions

- In Python it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.
- A Python function is used to accomplish a specific task. Once we define a function, we can reuse the function over and over throughout our program.

Example:

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all day.')
```

- The `def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. We can “call” the function by name.

3.2.8 Flow of Execution

- When we are working with functions it is really important to know the order in which statements are executed called as the flow of execution.
- The flow of execution refers to the order in which statements are executed during a program run.
- In other words, the order in which statements are executed during a program run called flow of execution.
- Execution always begins at the first statement of the program. Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Function calls are like a detour in the flow of execution.
- Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

Example:

```
def my_function():  
    print("Hello from a function")  
my_function()
```

Output:

```
Hello from a function
```

3.2.9 Parameters and Arguments

- The terms parameter and argument can be used for the same thing ‘information that are passed into a function’.
- A parameter is the variable listed inside the parentheses in the function definition. An argument is the value that is sent to the function when it is called.

Concept of Actual and Formal Parameters:

- Function arguments are the real values passed to (and received by) the function.
- The arguments in the function definition bind the arguments passed at function invocation (i.e. when the function is called), which are called actual parameters, to the names given when the function is defined, which are called formal parameters.

1. Actual Parameters:

- The parameters used in the function call are called actual parameters. These are the actual values that are passed to the function. The actual parameters may be in the form of constant values or variables.
- The data types of actual parameters must match with the corresponding data types of formal parameters (variables) in the function definition.
 - (i) They are used in the function call
 - (ii) They are actual values that are passed to the function definition through the function call.
 - (iii) They can be constant values or variable names (such as local or global).

2. Formal Parameters:

- The parameters used in the header of function definition are called formal parameters of the function. These parameters are used to receive values from the calling function.
 - (i) They are used in the function header.
 - (ii) They are used to receive the values that are passed to the function through function call.
 - (iii) They are treated as local variables of a function in which they are used in the function header.

Example: For actual and formal parameters.

```
>>> def cube(x):      # formal parameters
    return x*x*x

>>> result = cube(7) # actual parameters
>>> print(result)
```

Function Arguments:

- Many build in functions need arguments to be passed with them. Many build in functions require two or more arguments. The value of the argument is always assigned to a variable known as parameter.
 - There are four types of arguments using which can be called are Required arguments, Keyword arguments, Default arguments, and Variable-length arguments.
-

1. Keyword Arguments:

- Keyword arguments are related to the function calls. When you use keyword 'arguments' in a function call, the caller identifies the arguments by the parameter name.
 - The Python interpreter is able to use the keywords provided to match the values with parameters. The order of parameters does not matter.
-

Example 1:

```
# Function definition is here
def func(str):
    #This prints a string passed into this function
    print(str)
    return
# calling func function
func(str = "TYBSc")
```

Output:

```
TYBSc
```

Example 2:

```
# Function Defination
def mydata( name, age ):
    #This prints a passed info into this function
    print ("Name: ", name)
    print ("Age: ", age)
    return
# calling the function
mydata( age=50, name="manisha" )
```

Output:

```
Name:  manisha
Age:  50
```

2. Default arguments:

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
-

Example 1:

```
# Function Definition
def mydata( name, age = 40 ):
    This prints a passed info into this function
    print ("Name: ", name)
    print ("Age: ", age)
    return
# calling mydata function
mydata( age=50, name="manisha" )
mydata( name="manisha" )
```

Output:

```
Name: manisha
Age: 50
Name: manisha
Age: 40
```

Example 2:

```
# function to calculate the square of the sum of two numbers
def nsquare(x, y = 2):
    return (x*x + 2*x*y + y*y)
print("The square of the sum of 2 and 2 is: ", nsquare(2))
print("The square of the sum of 2 and 3 is: ", nsquare(2,4))
```

Output:

```
The square of the sum of 2 and 2 is: 16
The square of the sum of 2 and 4 is: 36
```

3. Required Arguments:

- Required arguments are the arguments passed to a function in correct positional order.
- The number of arguments in the function call should match exactly with the function definition.

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return
# calling function
printme()
```

Output:

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme()
TypeError: printme() takes exactly 1 argument (0 given)
```

4. Variable-length Arguments:

- The function can have more arguments than the arguments defining the function.
 - These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.
-

Syntax:

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

- An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example 1:

```
# Function definition
def printinfo( arg1, *vartuple ):
    #This prints a variable passed arguments"
    print("Output is: ")
    print(arg1)
    for var in vartuple:
        print (var)
    return
#call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:

```
Output is:
10
Output is:
70
60
50
```

3.2.10 Variables and Parameters

- The two parameter passing mechanisms are pass-by-reference and pass-by-value.
 1. **Pass by reference** is the term used in some programming languages, where values to the argument of the function are passed by reference, that is the address of the variable is passed and then the operation is done on the value stored at these addresses.

2. **Pass by value** means that the value is directly passed as the value to the argument of the function. In this case, the operation is done on the value and then the value is stored at the address.
 - All parameters (arguments) in the Python language are passed by reference. It means if we change what a parameter refers to within a function, the change also reflects back in the calling function.
-

Example 1:

```
# Function definition
def foo( lst ):
    "This changes a list pass as a parameter into this function"
    lst.append([1,2,3,4])
    print("Values inside the function: ", lst)
    return

# calling foo function
lst = [10,20,30]
foo( lst )
print("Values outside the function: ", lst)
```

- We use reference of the passed object and appending values in the same object. The output is:
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
-

Example 2:

```
# the reference is being overwritten inside the called function.
# Function definition
def foo( lst ):
    "This changes a list into this function"
    lst = [1,2,3,4]; # This would assign new reference in lst
    print("Values inside the function: ", lst)
    return

# calling the function
lst = [10,20,30]
foo( lst )
print("Values outside the function: ", lst)
```

- Here, changing list within the function does not affect lst and the output is:
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
-

- A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data that can be changed later in the program.
- All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular variable/identifier.
- The availability/accessibility of a variable in different parts of a program is referred to as its scope.
- There are two basic scopes of variables in Python:
 1. **Global Variables:** Global variables can be accessed throughout (outside) the program body by all functions.
 2. **Local Variables:** Local variables can be accessed only inside the function in which they are declared.
- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

Example: For scope of variables.

```
>>> g=10          # global variable g
>>> def show():
l=20              # local variable l
print("local variable=",l)
print("Global variable=",g)
>>> show()
local variable= 20
Global variable= 10
>>>
```

Difference between Local Variables and Global Variables:

Sr. No.	Local Variables	Global Variables
1.	Local variables are declared inside a function.	Global variables are declared outside any function.
2.	Accessed only by the statements, inside a function in which they are declared.	Accessed by any statement in the entire program.
3.	Local variables are alive only for a function	Global variables are alive till the end of the program
4.	A local variable is destroyed when the control of the program exit out of the block in which local variable is declared.	Global variable is destroyed when the entire program is terminated.

Nonlocal Variables:

- Nonlocal variables are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.
- In the following code , we use nonlocal keyword to create nonlocal variable.

```
def outer():
    x = "local"
    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
```

- When we run the code, the will output be:
inner: nonlocal
outer: nonlocal
- In the above code, there is a nested function inner(). We use nonlocal keyword to create nonlocal variable. The inner() function is defined in the scope of another function outer().

Program 1: Program to find factors of a number.

```
# define a function
def factors(x):
    print("The factors of",x,"are:")
    for i in range(1, x + 1):
        if x % i == 0:
            print(i)
num = int(input("Enter a number: "))
print_factors(num)
```

Output:

```
Enter a number: 50
The factors of 50 are:
1
2
5
25
50
```

Program 2: Program to find the sum of natural numbers.

```
add_num(num):
    if num < 0:
        print("Enter a positive number")
    else:
        sum = 0
        # use while loop to iterate un till zero
        while(num > 0):
            sum += num
            num -= 1
        return(sum)
num1 = int(input("Enter a number: "))
print("Sum of Natural numbers is",add_num(num1))
```

Output:

The sum of Natural Numbers is 55

3.2.11 Return Statement

- The return statement is used to exit a function. The return statement is used to return a value from the function. A function may or may not return a value.
- If a function returns a value, it is passed back by the return statement to the caller. If it does not return a value, we simply write return with no arguments.

Syntax: return(expression)

Example: For return statement.

```
>>> def sum( arg1, arg2 ):
    "Add both the parameters and return them."
    total = arg1 + arg2
    print ("Sum: ", total)
    return total
>>> result=sum(30,35)
Sum:  65
>>>
```

3.2.12 Stack Diagrams

- We used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.
- Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.
- To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

- Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it.
- The stack diagram is shown in Fig. 3.4.

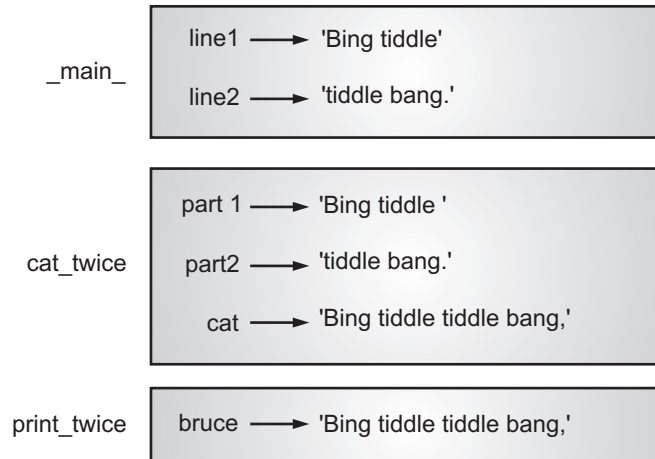


Fig. 3.4: Stack Diagram

- The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When we create a variable outside of any function, it belongs to `__main__`.
- Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.
- If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called that, all the way back to `__main__`.
- For example, if we try to access `cat` from within `print_twice`, we get a `NameError`:

```

Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print cat
NameError: name 'cat' is not defined
  
```

- This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.
- The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

3.2.13 void Functions

- In Python, it is possible to compose a function without a return statement. Functions like this are called void, and they return None,
- The void functions are those functions which do not return any value.

Example: For void function.

```
>>> def show():
        str="hello"
        print(str)
>>> show()
hello
>>>
```

3.2.14 Anonymous Functions

- Anonymous functions are the functions that are not bond to name. It means anonymous function does not has a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword. Hence, anonymous functions are also called Lambda Functions.
- A lambda function can take any number of arguments, but can only have one expression. Lambda is created without using the def keyword.

Syntax of a lambda expression in Python:

```
lambda [arg1,arg2,...]:[expression]
```

where, we place lambda expression in Python, it returns the value of the expression.

Example 1:

```
#Function Definiton
square=lambda x1: x1*x1
#Calling square as a function
print("Square of number is",square(10))
```

Output:

```
Square of number is 100
```

Example 2: Program To Display Powers of 2 Using Anonymous Function.

```
terms = int(input("How many terms? "))
# use anonymous function
result = list(map(lambda x: 2 ** x, range(terms)))
# display the result
print("The total terms is:",terms)
for i in range(terms):
    print("2 raised to power",i,"is",result[i])
```

Output:

```
The total terms is: 10
2 raised to power 0 is 1
2 raised to power 1 is 2
2 raised to power 2 is 4
2 raised to power 3 is 8
2 raised to power 4 is 16
2 raised to power 5 is 32
2 raised to power 6 is 64
2 raised to power 7 is 128
2 raised to power 8 is 256
2 raised to power 9 is 512
```

3.2.15 Return Values

- A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller.
- Functions that return values are sometimes called fruitful functions. Fruitful functions are those functions which return values. The built-in functions we have used, such as abs, pow, and max, have produced results.
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.
- Following function calculates area of circle and return the value:

```
>>> def area(radius):
    temp = 3.14159 * radius**2
    return temp
>>> area(5)
78.53975
```

3.2.16 Boolean Functions

- Functions can return Booleans, which is often convenient for hiding complicated tests inside functions.

Example:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

- It is common to give boolean functions names that sound like yes/no questions; is_divisible returns either True or False to indicate whether x is divisible by y.

- Here, is an example:


```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```
- The result of the == operator is a boolean, so we can write the function more concisely by returning it directly:


```
def is_divisible(x, y):
    return x % y == 0
```

3.2.17 Recursion

- Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body.
- Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body.
- Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.
- A function is said to be a recursive if it calls itself. For example, let's say we have a function abc() and in the body of abc() there is a call to the abc().

Example: For recursive function.

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
print(fact(0))
print(fact(4))
print(fact(6))
```

Output:

```
1
24
720
```

- The factorial of 4 (denoted as 4!) is $1*2*3*4 = 24$.
- Each function call multiplies the number with the factorial of number 1 until the number is equal to one.

```
fact(4)           # 1st call with 4
4 * fact(3)       # 2nd call with 3
4 * 3 * fact(2)   # 3rd call with 2
4 * 3 * 2 * fact(1) # 4th call with 1
4 * 3 * 2 * 1     # return from 4th call as number=1
4 * 3 * 2         # return from 3rd call
4 * 6             # return from 2nd call
24               # return from 1st call
```

- Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of Recursion:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.
4. It consumes more storage space because the recursive calls along with variables are stored on the stack.
5. It is not more efficient in terms of speed and execution time.

Example: Programs to convert U.S. dollars to Indian rupees.

```
def dol_rup():
    dollars = float(input("Please enter dollars:"))
    rupees = dollars * 70
    print("Dollars: ",dollars)
    print("Rupees: ",rupees)
def euro_rup():
    euro= float(input("Please enter euro:"))
    rupees = euro * 79.30
    print("Euro: ",euro)
    print("Rupees: ",rupees)
def menu():
    print("1: Doller to Rupees")
    print("2: Euro to Rupees")
    print("3: Exit")
    choice=int(input("Enter your choice: "))
    if choice==1:
        dol_rup()
    if choice==2:
        euro_rup()
    if choice==3:
        print("Good bye!")
menu()
```

Output:

```
1: Doller to Rupees
2: Euro to Rupees
3: Exit
Enter your choice: 1
Please enter dollars:75
Dollars: 75.0
Rupees: 5250.0
```

Program 1: Program to find the factorial of an integer using recursion.

```
def calcfact(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * calcfact(x-1))
num = 4
print("The factorial of", num, "is", calcfact(num))
```

Output:

```
The factorial of 4 is 24
```

Program 2: Program to determine whether a given number is even or odd.

```
def check(n):
    if (n < 2):
        return (n % 2 == 0)
    return (check(n - 2))
n=int(input("Enter number:"))
if(check(n)==True):
    print("Number is even!")
else:
    print("Number is odd!")
```

3.2.18 More Recursion

- Recursion is the process of defining something in terms of itself. We will evaluate a few recursively defined mathematical functions.
 - A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined.
-

- If we looked up the definition of the factorial function, denoted with the symbol $!$, we might get something like this:
 $0! = 1$
 $n! = n(n-1)!$
- Above definition says that the factorial of 0 is 1, and the factorial of any other value, n , is n multiplied by the factorial of $n-1$.
- So $3!$ is 3 times $2!$, which is 2 times $1!$, which is 1 times $0!$. Putting it all together, $3!$ equals 3 times 2 times 1 times 1, which is 6.
- If we can write a recursive definition of something, you can usually write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that factorial takes an integer:

```
def factorial(n):
```

- If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
```

```
    if n == 0:
```

```
        return 1
```

- Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$ and then multiply it by n :

```
def factorial(n):
```

```
    if n == 0:
```

```
        return 1
```

```
    else:
```

```
        recurse = factorial(n-1)
```

```
        result = n * recurse
```

```
        return result
```

- The flow of execution for this program is similar to the flow of countdown. if we call factorial with the value 3:
 - Since, 3 is not 0, we take the second branch and calculate the factorial of $n-1$...
 - Since, 2 is not 0, we take the second branch and calculate the factorial of $n-1$...
 - Since, 1 is not 0, we take the second branch and calculate the factorial of $n-1$...
 - Since, 0 is 0, we take the first branch and return 1 without making any more recursive calls.
 - The return value (1) is multiplied by n , which is 1, and the result is returned.
 - The return value (1) is multiplied by n , which is 2, and the result is returned.
 - The return value (2) is multiplied by n , which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

- Here, is what the stack diagram looks like for this sequence of function calls:

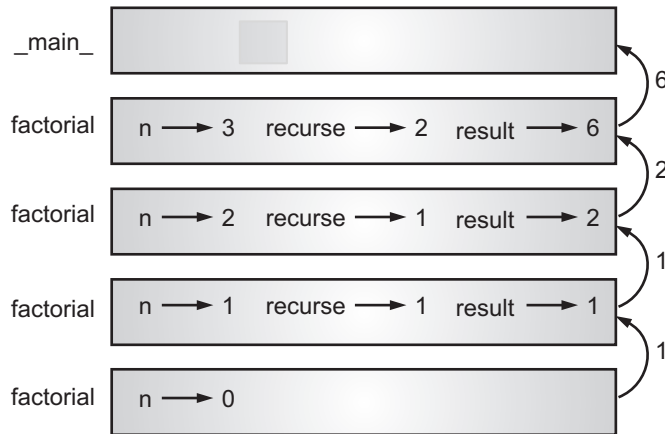


Fig. 3.5

- The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.
- In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not execute.

3.2.19 Lambda Forms

- A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.
- Python offers the lambda form as a way to simplify using higher-order functions. A lambda form allows us to define a small, anonymous function. The function's body is limited to a single expression.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to `print` because lambda requires an expression.
- The syntax of lambda functions contains only a single statement, which is as follows:


```
lambda [arg1 [,arg2,...,argn]]:expression
```
- The following is an example of using a simple lambda expression as the key:


```
long = max(trip, key=lambda leg: leg[2])
short = min(trip, key=lambda leg: leg[2])
```
- The lambda we've used will be given an item from the sequence; in this case, each argument variable, `leg` three tuple will be given to the lambda.
- The lambda argument variable, `leg`, is assigned and the expression, `leg (2)`, is evaluated, plucking the distance from the three tuple.
- In cases where a lambda is used exactly once, this form is ideal. When reusing a lambda, it's important to avoid copy and paste. What's the alternative?

- We can assign lambdas to variables, by doing something like this:

```
start - lambda x: x[0]
end - lambda x: x[1]
dist - lambda x: x[2]
```
- Each of these lambda forms is a callable object, similar to a defined function. They can be used like a function.

Example:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2
# Now you can call sum as a function
print ("Value of total: ", sum( 10, 20 ))
print ("Value of total: ", sum( 20, 20 ))
```

Output:

```
Value of total:  30
Value of total:  40
```

3.2.20 Functional Programming Tools: filter(), map() and reduce()

- Functional programming is all about expressions. We may say that the Functional programming is an expression oriented programming.
- In Python, anonymous function means that a function is without a name. The def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- The **syntax** of lambda functions contains only a single statement as follows:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Example 1:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
print ("Value of total: ", sum( 10, 20 ))
print ("Value of total: ", sum( 20, 20 ))
```

Output:

```
Value of total:  30
Value of total:  40
```

Example 2:

```
x = lambda a: a + 10
print(x(5))
```

Output:

```
15
```

-
- Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Use of lambda() with filter():

- The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.

Syntax: filter(function, list)

Example:

```
# Python code to illustrate filter() with lambda()
lst = [2, 7, 20, 87, 44, 52, 77, 67, 79, 81]
lst_result = list(filter(lambda x: (x%2 != 0) , lst))
print(lst_result)
```

Output:

```
[7, 87, 77, 67, 79, 81]
```

Use of lambda() with map():

- The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

Syntax: map(function, list)

Example:

```
# Python code to illustrate map() with lambda() to get double of a list.
lst = [2, 7, 20, 87, 44, 52, 77, 67, 79, 81]
lst_result = list(map(lambda x: x*2 , lst))
print(lst_result)
```

Output:

```
[4, 14, 40, 174, 88, 104, 154, 134, 158, 162]
```

Use of lambda() with reduce():

- The reduce() function in Python takes in a function and a list as argument.
- The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list.

Example:

```
# Python code to illustrate reduce() with lambda() to get sum of a list
# from functools import reduce
lst = [5, 10, 15, 20, 50, 60]
sum = reduce((lambda x, y: x + y), lst)
print (sum)
```

Output:

160

3.3 TUPLES

- A tuple is a collection of objects which ordered and immutable.
- A Python tuple is a sequence of data values called as items or elements.
- A tuple is a collection of items which is ordered and unchangeable.
- A tuple is a data structure that is an immutable or unchangeable, ordered sequence of elements/items. Because tuples are immutable, their values cannot be modified.
- A tuple is a heterogeneous data structure and used for grouping data. Each element or value that is inside of a tuple is called an item.
- A tuple is an immutable data type. An immutable data type means that we cannot add or remove items from the tuple data structure.
- In Python tuples are written with round brackets () and values in tuples can also be accessed by their index values, which are integers starting from 0.
- Tuples are the sequence or series values of different types separated by commas (.). Tuples are just like lists, but we can not change their values.

Difference between Tuples and Lists:

1. The syntax of tuples is shown by parenthesis() whereas the syntax of lists is shown by square brackets[].
2. List has variable length, tuple has fixed length.
3. List has mutable nature, tuple has immutable nature.
4. List has more functionality than the tuple.
5. Tuples are heterogeneous while lists are homogeneous.
6. Tuples show structure whereas lists show order.
7. Unexpected errors and changes can easily occur in lists while in tuples unexpected errors and changes rarely occur in tuples.

- Following table shows difference between strings, tuples and lists.

Sr. No.	Immutable (Value cannot be modified)		Mutable (values can be modified)
	Strings str="hi"	Tuples tuples=(5,4.0,'a')	List list=[5,4.0,'a']
1.	Sequence Unicode character.	Ordered sequence.	Order sequence.
2.	Values cannot be modified.	Same as list but it is faster than list because it is immutable.	Value can be changed dynamically.
3.	It is a sequence of character.	Values stored in alpha numeric.	Values stored in alpha numeric.
4.	Access values from string.	Access values from tuples.	Access values from list.
5.	Adding values in not possible.	Adding values is not possible.	Adding values is possible.
6.	Removing values is not possible.	Removing values is not possible.	Removing values is possible.

3.3.1 Creating Tuples

- To create tuple, all the items or elements are placed inside parentheses () separated by commas and assigned to a variable.
- The **syntax** for defining a tuple in Python is given below:

`<tuple_name>=(value1, value2, ... valueN).`

Here, tuple name indicate name as the tuple and value1, value2,...valueN are the values assigned to the tuple.

Example: Emp (20, "Amar", 'M', 50)

- The empty tuple is written as two parentheses containing nothing.
`tup1 = ()`
- To write a tuple containing a single value you have to include a comma, even though there is only one value:
`tup1 = (50)`
- A tuple in Python is an immutable data type which means a tuple once created cannot be altered/modified. Tuples can have any number of different data items (integer, float, string, list etc.).
- The simplest method to create a tuple in Python is simply assigning a set of values to the tuple using assignment operator (=). For example: `t()` # creates an empty tuple with name 't'.

Example: For creating tuples.

```
>>> tuple1=(10,20,30)           # A tuple with integer values
>>> tuple1
(10, 20, 30)
>>> tuple2=(10,"abc",11.22,'X') # A tuple with different data types
>>> tuple2
(10, 'abc', 11.22, 'X')
>>> tuple3=("python",[10,20,30],[11,"abc",22.33]) # Nested tuple
>>> tuple3
('python', [10, 20, 30], [11, 'abc', 22.33])
>>> tuple4=10,20,30,40,50      # Tuple can be created without parenthesis
>>> tuple4
(10, 20, 30, 40, 50)
>>> type(tuple4)
<class 'tuple'>
```

Note: In case generating a tuple with a single element, make sure to add a comma after the element otherwise it would not be considered as tuple.

```
>>> tuple=10
>>> type(tuple)
<class 'int'>
>>> tuple="hello"
>>> type(tuple)
<class 'str'>
>>> tuple=("hello",)
>>> type(tuple)
<class 'tuple'>
```

3.3.2 Accessing Values in Tuple

- Accessing items/elements from the tuple is a method to get values stored in the tuple from a particular location or index.
- To access values in tuple, use the square brackets [] for slicing along with the index or indices to obtain value available at that index.

Example: For accessing values in tuples.

```
>>> tuple=(10,20,30,40,50)
>>> tuple[1] # access value at specific index
20
>>> tuple[1:4] # tuple[m:n] will return elements from
                                     mth index to n-1th index.
(20, 30, 40)
```

```
>>> tuple[:2] # tuple[:n] will return elements from
                                0th index to n-1th index.
(10, 20)
>>> tuple[2:] # tuple[m:] will return elements from
                                mth index to last index.
(30, 40, 50)
>>> tuple[-1] # access value at last index
50
```

- Accessing elements/items from a particular tuple at once allows the user to access all the values from the tuple only by writing a single statement. This is possible by writing the tuple name in the print() statement.

3.3.3 Tuple Assignment

- It allows the assignment of values to a tuple of variables on the left side of the assignment from the tuple of values on the right side of the assignment.
- The number of variables in the tuple on the left of the assignment must match the number of elements/items in the tuple on the right of the assignment.

Example 1: For tuple assignment.

```
>>> vijay=(11,"Vijay","Thane")
>>> (id,name,city)=vijay
>>> print(id)
11
>>> print(name)
Vijay
```

Example 2:

```
>>> language=('python','java')
>>> language
('python', 'java')
>>> (a,b)=language
>>> a
'python'
>>> b
'java'
```

- Similarly swapping of two values of variables can be solve by using tuple assignment:
Swapping the values of two variables using traditional method:

```
>>> a=10
>>> b=20
>>> print(a,b)
10 20
>>> temp=a
>>> a=b
>>> b=temp
>>> print(a,b)
20 10
```

Swapping the values of two variables using tuple assignment:

```
>>> x=10
>>> y=20
>>> print(x,y)
10 20
>>> x,y=y,x
>>> print(x,y)
20 10
```

3.3.4 Tuples as Return Values

- Functions can return tuples as return values. Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.
- For example, if we want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then $x\%y$. It is better to compute them both at the same time.
- The built-in function `divmod()` takes two arguments and returns a tuple of two values namely, the quotient and remainder. We can store the result as a tuple:

```
>>> T = divmod(7, 3)
>>> t
(2, 1)
```
- Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7,3)
>>> quot
2
>>> rem
1
```
- Following is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```
- The `max` and `min` are built-in functions that find the largest and smallest elements of a sequence. The `min_max` computes both and returns a tuple of two values.

3.3.5 Variable-length Argument Tuples

- Functions can take a variable number of arguments. A parameter name that begins with * gathers arguments into a tuple.
- For example, `printall()` takes any number of arguments and prints them:

```
def printall(*args):  
    print(args)
```

- The gather parameter can have any name we like, but `args` is conventional. Here's how the function works:

```
>>> printall(1, 2.0, '3')  
(1, 2.0, '3')
```

- The complement of gather is scatter. If we have a sequence of values and we want to pass it to a function as multiple arguments, we can use the * operator.
- For example, `divmod()` takes exactly two arguments; it does not work with a tuple:

```
>>> t = (7, 3)  
>>> divmod(t)  
TypeError: divmod expected 2 arguments, got 1
```

- But if we scatter the tuple, it works:

```
>>> divmod(*t)  
(2, 1)
```

- Many of the built-in functions use variable-length argument tuples. For example, `max` and `min` can take any number of arguments:

```
>>> max(1, 2, 3)  
3
```

3.3.6 Deleting Tuples

- Tuples are unchangeable, so we cannot remove items from it, but we can delete the tuple completely. To explicitly remove an entire tuple, just use the `del` statement.

Example: For delete entire tuple.

```
>>> t1  
(10, 20)  
>>> del t1  
>>> t1  
Traceback (most recent call last):  
File "<pyshell#65>", line 1, in <module>  
t1  
NameError: name 't1' is not defined
```

- If we want to remove specific elements from a tuple then Python does not provide any explicit statement but we can use index slicing to leave out a particular index.

Example: For leave out elements from tuple.

```
a = (1, 2, 3, 4, 5)
b = a[:2] + a[3:]    # element 3 is leave out
print(b)
```

Output:

```
(1, 2, 4, 5)
```

Deleting elements of tuple:

- One way to delete the elements from the tuple, we have to convert into list and then perform deletion.
- We can convert tuple into a list, remove the item and convert back to a tuple.

Example:

```
tuple1 = (1, 2, 3, 4, 5)
list1 = list(tuple1)    #convert tuple to list
del list1[2]
b = tuple(list1)        #convert list to tuple
print(b)
```

Output:

```
(1, 2, 4, 5)
```

3.3.7 Updating Tuples

- Tuples are immutable which means we cannot update or change the values of tuple elements. We are able to take portions of existing tuples to create new tuples.

Example: For updating tuple.

```
>>> tuple1
(10, 20, 30)
>>> tuple2
('A', 'B', 'C')
>>> tuple1[1]=40          # get error as tuples are immutable
Traceback (most recent call last):
File "<pyshell#39>", line 1, in <module>
tuple1[1]=40
TypeError: 'tuple' object does not support item assignment
>>> tuple3=tuple1+tuple2    # concatenating two tuples
>>> tuple3
(10, 20, 30, 'A', 'B', 'C')
```

- If the element of tuple is itself a mutable data type like list, its nested items can be changed as shown in following example:

```
>>> tuple1 = (1,2,3,[4,5])
>>> tuple1[3][0]= 14
>>> tuple1[3][1]=15
>>> tuple1
(1, 2, 3, [14, 15])
```

- In order to change a value, we can convert tuple into list and then change the values and revert back list into tuple can solve the purpose of update as shown in following example.

Example:

```
tuple1 = (1, 2, 3, 4, 5)
list1 = list (tuple1)
list1[2]=20
b = tuple (list1)
print(b)
```

Output:

```
(1, 2, 20, 4, 5)
```

3.3.8 Basic Tuple Operations

- A set of operations can be applied to Python programming tuple. The operations described below are supported by most sequence types, both mutable and immutable. Here we will consider different operations in context of immutable sequence.

1. Concatenation and Repetition:

- We can use + operator to combine two tuples. This is also called concatenation operation.
- We can also repeat the elements in a tuple for a given number of times using the * operator.

Example: For tuple operations using + and * operators.

```
>>> t1=(10,20)
>>> t2=(30,40)
>>> t1+t2
(10, 20, 30, 40)
>>> t1*2
(10, 20, 10, 20)
```

2. Membership Function:

- We can test if an item exists in a tuple or not, using the keyword in. The in operator evaluates to true if it finds a variable in the specified sequence and false otherwise.
-

Example: For membership function in tuple.

```
>>> tuple
(10, 20, 30, 40, 50)
>>> 30 in tuple
True
>>> 25 in tuple
False
```

3. Iterating through a Tuple:

- Iteration over a tuple specifies the way which the loop can be applied to the tuple.
- Using a for loop we can iterate through each item in a tuple. Following example uses a for loop to simply iterates over a tuple.

Example: For iterating items in tuple using for loop.

```
>>> tuple=(10,20,30)
>>> for i in tuple:
print(i)          # use two enter to get the output
```

Output:

```
10
20
30
```

3.3.9 Indexing, Slicing and Matrices

- Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings.
- There are various ways in which we can access the elements of a tuple.

Indexing:

- We can use the index operator [] to access an item in a tuple where the index starts from 0.
- Let a tuple having 6 elements will have index from 0 to 5. Trying to access an element other than (6, 7,...) will raise an IndexError. The index must be an integer, so we cannot use float or other types. This will result into TypeError. The nested tuple are accessed using nested indexing, as shown in the example below.

```
tupl = ('b','l','o','c','k','s')
print(tupl[0])          # Output: 'b'
print(tupl[5])          # Output: 's'
# index must be in range
# IndexError: list index out of range(following line)
```

```
#print(tupl[6])
# index must be an integer
# TypeError: list indices must be integers, not float
#tupl[2.0]
# nested tuple
n_tuple = ("hello", [8, 4, 6], (1, 2, 3))
# nested index
print(n_tuple[0][3])          # Output: 'l'
# nested index
print(n_tuple[1][1])          # Output: 4
```

Negative Indexing:

- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on.

```
tupl = ('b','l','o','c','k','s')
print(tupl[-1])
print(tupl[-6])
Output:  s
          b
```

Slicing:

- We can access a range of items in a tuple by using the slicing operator - colon ":".
- The syntax for tuple slicing is

```
tuple_name[start:stop[:step]]
```

where, both 'start' and 'stop' are integer values (+ve or -ve) values).

- Tuple slicing returns a part of the tuple from the start index value to stop index value. This includes the start index value but excludes the stop index value. The stop specifies the increment value to slice by and it is optional.

Example:

```
tupl = ('p','r','o','g','r','a','m','i','n','g')
# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(tupl[1:4])
# elements beginning to 2nd
# Output: ('p', 'r')
print(tupl[:2])
# elements 8th to end
# Output: ('i', 'n', 'g')
```

```

print(tup1[7:])
# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'n', 'g')
print(tup1[:])
print(tup1[::2]) # print using step sizes of 2
# Output: ('p', 'o', 'r', 'm', 'n')

```

- Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

Nested Tuples:

- A tuple can also be nested, i.e. just like a list inside a tuple, we can have tuples inside a tuple.

```

a = (1, 2)
b = (4, 5)
t = (a, b)
print(t)
Output: ((1, 2), (4, 5))

```

Matrices:

- A matrix is a two-dimensional data structure where numbers are arranged into rows and columns.
- Use tuple as array indices to find element in matrix:

Example:

```

matrix = [[1,2,3], [4,5,6]]
p=(1,2)
print (matrix[p[0]][p[1]])

```

Output: 6

3.3.10 Built-in Tuple Functions

- Following table built-in tuple functions in Python programming:

Sr. No.	Function	Description	Example
1.	len(tuple)	Gives the total length of the tuple.	<pre>>>> tup1 (1, 2, 3) >>> len(tup1) 3</pre>

contd. ...

2.	<code>max(tuple)</code>	Returns item from the tuple with max value.	<pre>>>> tup1 (1, 2, 3) >>> max(tup1) 3</pre>
3.	<code>min(tuple)</code>	Returns item from the tuple with min value.	<pre>>>> tup1 (1, 2, 3) >>> min(tup1) 1</pre>
4.	<code>zip(tuple1,tuple2)</code>	It zips elements from two tuples into a list of tuples.	<pre>>>> tup1=(1,2,3) >>> tup2=('A','B','C') >>> tup3=zip(tup1,tup2) >>> list(tup3) [(1, 'A'), (2, 'B'), (3, 'C')]</pre>
5.	<code>tuple(seq)</code>	Converts a list into tuple.	<pre>>>> tuple1 = (1, 2, 3, 4, 5) >>> list1 = list (tuple1) >>> list1 [1,2,3,4,5]</pre>

Methods of Tuple:

Sr. No.	Function	Description	Example
1.	<code>count()</code>	Returns the number of times a specified value occurs in a tuple	<pre>>>> tup1 (1, 2, 3, 2, 4) >>> tup1.count(2) 2</pre>
2.	<code>index()</code>	Searches the tuple for a specified value and returns the position of where it was found.	<pre>>>> tup1 (1, 2, 3) >>> tup1.index(3) 2</pre>

3.4 DICTIONARIES

- The dictionary data structure is used to store key value pairs indexed by keys. A dictionary is an associative data structure, means that elements/items are stored in a non linear fashion.

- Python dictionary is an unordered collection of items or elements. Items stored in a dictionary are not kept in any particular order. The Python dictionary is a sequence of data values called as items or elements.
- While other compound data types have only value as an element, a dictionary has a key:value pair. Each value is associated with a key.
- Dictionaries are optimized to retrieve values when the key is known. A key and its value are separated by a colon (:) between them.
- The items or elements in a dictionary are separated by commas and all the elements must be enclosed in curly braces.
- **Syntax** for define a dictionary in Python programming is as follows:
`<dictionary_name> = {key1:value1, key2:value2, ...keyN:valueN}`
Example: `Emp = {"ID":20, "Name":"Amar", "Gender":Male, "SalaryPerHr":50}`
- A pair of curly braces with no values in between is known as an empty dictionary. Dictionary items are accessed by keys, not by their position (index).
- The values in a dictionary can be duplicated, but the keys in the dictionary are unique. Dictionaries are changeable (mutable). We can change or update the items in dictionary as and when required.
- The key can be looked up in much the same way that we can look up a word in a paper-based dictionary to access its definition i.e., the word is the 'key' and the definition is its corresponding 'value'.
- Dictionaries can be nested i.e. a dictionary can contain another dictionary.

3.4.1 Creating a Dictionary

- A dictionary can be used to store a collection of data values in a way that allows them to be individually referenced. However, rather than using an index to identify a data value, each item in a dictionary is stored as a key value pair.
- The simplest method to create dictionary is to simply assign the pair of key:values to the dictionary using operator (=).
- There are two ways for creation of dictionary in python.
 1. We can create a dictionary by placing a comma-separated list of key:value pairs in curly braces {}. Each key is separated from its associated value by a colon(:).

Example: For creating a dictionary using { }.

```
>>> dict1={} # Empty dictionary
>>> dict1
{}
>>> dict2={1:"Orange", 2:"Mango", 3:"Banana"} # Dictionary
# with integer keys
>>> dict2
{1: 'Orange', 2: 'Mango', 3: 'Banana'}
>>> dict3={"name":"vijay", 1:[10,20]} # Dictionary with mixed keys
>>> dict3
{'name': 'vijay', 1: [10, 20]}
```

2. Python provides a built-in function `dict()` for creating a dictionary.

Example: Creating directory using `dict()`.

```
>>> d1=dict({1:"Orange",2:"Mango",3:"Banana"})
>>> d2=dict([(1,"Red"),(2,"Yellow"),(3,"Green")])
>>> d3=dict(one=1, two=2, three=3)
>>> d1
{1: 'Orange', 2: 'Mango', 3: 'Banana'}
>>> d2
{1: 'Red', 2: 'Yellow', 3: 'Green'}
>>> d3
{'one': 1, 'two': 2, 'three': 3}
```

3.4.2 Accessing Values in a Dictionary

- We can access the items of a dictionary by following ways:

1. Referring to its key name, inside square brackets (`[]`).

Example: For accessing dictionary items `[]` using.

```
>>> dict1={'name':'vijay','age':40}
>>> dict1['name']
'vijay'
>>> dict1['adr']
Traceback (most recent call last):
File "<pyshell#79>", line 1, in <module>
dict1['adr']
KeyError: 'adr'
>>>
```

Here, if we refer to a key that is not in the dictionary, you'll get an exception. This error can be avoided by using `get()` method.

2. Using `get()` method returns the value for key if key is in the dictionary, else `None`, so that this method never raises a `KeyError`.

Example: For accessing dictionary elements by `get()`.

```
>>> dict1={'name':'vijay','age':40}
>>> dict1.get('name')
'vijay'
>>> dict1.get('adr')
```

3.4.3 Deleting Elements from a Dictionary

- We can remove a particular item in a dictionary by using the method `pop()`. This method removes an item with the provided key and returns the value.
 - The method, `popitem()` can be used to remove and return an arbitrary item (key, value) from the dictionary.
-

- All the items can be removed at once using the `clear()` method. We can also use the `del` keyword to remove individual items or the entire dictionary itself.

Example: For deleting dictionary items/elements.

```
>>> squares={1:1,2:4,3:9,4:16,5:25}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> print(squares.popitem()) # remove an arbitrary item
(5, 25)
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16}
>>> squares.pop(2)          # remove a particular item with given key
4
>>> squares
{1: 1, 3: 9, 4: 16}
>>> del squares[3]          # delete a particular item
>>> squares
{1: 1, 4: 16}
>>> squares.clear()         # removes all items
>>> squares
{}
>>> del squares             # delete a dictionary itself
>>> squares
Traceback (most recent call last):
File "<pyshell#29>", line 1, in <module>
dquares
NameError: name 'dquares' is not defined
```

3.4.4 Updating Dictionary

- The dictionary data type is flexible data type that can be modified according to the requirements which makes it a mutable data type.
- The dictionary is mutable means changeable. We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key:value pair is added to the dictionary.

Example: For updating dictionary items/elements.

```
>>> dict1
{'name': 'vijay', 'age': 40}
>>> dict1['age']=35          # updating values in dictionary
>>> dict1
{'name': 'vijay', 'age': 35}
>>> dict1['address']='thane' # add new item in dictionary
>>> dict1
{'name': 'vijay', 'age': 35, 'address': 'thane'}
```

3.4.5 Basic Operations on a Dictionary

- In previous sections we study basic operations of directory such as create, delete, update etc., other operations that can be applied to the element/items of the directory are explained below.

1. Dictionary Membership Test:

- We can test if a key is in a dictionary or not using the keyword in. Notice that membership test is for keys only, not for values.

Example: For directory membership test.

```
>>> squares={1:1,2:4,3:9,4:16,5:25}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> print(1 in squares)
True
>>> print(6 in squares)
False
>>> squares={1:1,2:4,3:9,4:16,5:25}
>>> print(9 in squares)
False
```

2. Traversing Dictionary:

- Using for loop we can iterate though each key in a dictionary.

Example: For traversing dictionary.

```
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> for i in squares:
    print(i,squares[i])
```

Output:

```
1 1
2 4
3 9
4 16
5 25
```

3.4.6 Properties of Dictionary Keys

- Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

- There are two important points to remember about dictionary keys:
 - More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

Example:

```
>>> dict={1:'Vijay',2:'Amar',3:'Santosh',2:'Umesh'}
>>> dict
{1: 'Vijay', 2: 'Umesh', 3: 'Santosh'}
```

- Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Example:

```
>>> dict={ [1]:'Vijay',2:'Amar',3:'Santosh'}
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
dict={ [1]:'Vijay',2:'Amar',3:'Santosh'}
TypeError: unhashable type: 'list'
```

3.4.7 Built-in List Functions and Methods in Dictionary

- Python has a set of built-in functions that we can use on dictionaries. Some built-in functions of dictionary given in following table:

Sr. No.	Function	Description	Example
1.	all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).	<pre>>>> dict {1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> all(dict) True</pre>
2.	any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.	<pre>>>> dict={} >>> any(dict) False</pre>
3.	len()	Return the length (the number of items) in the dictionary.	<pre>>>> dict {1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> len(dict) 3</pre>
4.	sorted()	Return a new sorted list of keys in the dictionary.	<pre>>>> dict1 {2: 'Amar', 1: 'Vijay', 4: 'Umesh', 3: 'Amar'} >>> sorted(dict1) [1, 2, 3, 4]</pre>
5.	type()	Returns the type of the passed variable.	<pre>dict = {'Name': 'Zara', 'Age': 7}; print "Variable Type: %s" % type (dict)</pre>

- Python has a set of dictionary methods that we can use on dictionaries. Some of them are given in following table.

Sr. No.	Method	Description	Example
1.	<code>clear()</code>	Removes all the elements from the dictionary.	<pre>dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> dict {1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> dict.clear() >>> dict {}</pre>
2.	<code>copy()</code>	Returns a copy of the dictionary.	<pre>>>> dict={1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> X=dict.copy() >>> X {1: 'Vijay', 2: 'Amar', 3: 'Santosh'}</pre>
3.	<code>fromkeys()</code>	The <code>fromkeys()</code> method creates a new dictionary with default value for all specified keys. If default value is not specified, all keys are set to None.	<pre>>>> dict=dict.fromkeys(['Vijay', 'Manisha'], 'Author') >>> dict {'Vijay': 'Author', 'Manisha': 'Author'} >>></pre>
4.	<code>get()</code>	Returns the value of the specified key.	<pre>>>> dict1={'name':'vijay','age':40} >>> dict1.get('name') 'vijay'</pre>
5.	<code>items()</code>	Returns a list containing the a tuple for each key value pair.	<pre>dict{1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> for i in dict.items(): print(i) (1, 'Vijay') (2, 'Amar') (3, 'Santosh')</pre>

contd. ...

6.	<code>keys()</code>	Returns a list containing the dictionary's keys.	<pre>dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> dict.keys() dict_keys([1, 2, 3])</pre>
7.	<code>pop()</code>	Removes the element with the specified key.	<pre>dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> print(dict.pop(2)) Amar</pre>
8.	<code>popitem()</code>	Removes the last inserted key-value pair.	<pre>dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> dict.popitem() (3, 'Santosh')</pre>
9.	<code>setdefault()</code>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value.	<pre>>>> dict {2: 'Amar', 3: 'Santosh'} >>> dict.setdefault(1,'Vijay') >>> dict {2: 'Amar', 3: 'Santosh', 1: 'Vijay'}</pre>
10.	<code>update()</code>	Updates the dictionary with the specified key-value pairs.	<pre>>>> dict1 {2: 'Amar', 4: 'Umesh'} >>> dict2={1:'Vijay',3:'Santosh'} >>> dict2 {1: 'Vijay', 3: 'Santosh'} >>> dict1.update(dict2) >>> dict1 {2: 'Amar', 4: 'Umesh', 1: 'Vijay', 3: 'Santosh'}</pre>
11.	<code>values()</code>	Returns a list of all the values in the dictionary.	<pre>>>> dict {1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> dict.values() dict_values(['Vijay', 'Amar', 'Santosh'])</pre>

3.5 SETS

- A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).
- However, the set itself is mutable. We can add or remove items from it. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- A set data structure in python programming includes an unordered collection of items without duplicates. Sets are unindexed that means we cannot access set items by referring to an index.
- Sets are changeable (mutable) i.e., we can change or update a set as and when required. Type of elements in a set need not be the same, various mixed data type values can also be passed to the set.
- The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc.

3.5.1 Creating/Defining a Set

- Sets are used to store multiple items in a single variable. A set is a collection which is both unordered and unindexed.
- A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.
- A set is used to contain an unordered collection of items. There are two ways for creation of sets in python:

1. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

Syntax for defining set in Python is: <set_name>={value1, value2,...,valueN}

Example: Emp={20, "Amar", 'M', 50}

Example: For creating sets with { }.

```
>>> a={1,3,5,4,2}
>>> print("a=",a)
a = {1, 2, 3, 4, 5}
>>> print(type(a))
<class 'set'>
>>> colors = {'red', 'green', 'blue','red'}
>>> colors
{'blue', 'red', 'green'}
```

2. A set does not contain any duplicate values or elements. We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

Example:

```
>>> a={1,2,2,3,3,3}
>>> a
{1, 2, 3}
>>>
```

-
3. Set can be created by calling a type constructor called set().

Example: Creating sets with set().

```
>>> s=set('abc')
>>> s           # elements will be unordered
{'c', 'b', 'a'}
>>> s=set(range(1,3))
>>> s
{1, 2}
>>>
```

3.5.2 Accessing Values in a Set

- We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
for d in Days:
    print(d)
```

Output:

```
Mon
Thu
Fri
Sat
Tue
Sun
Wed
```

Frozen Set:

- Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

- Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.
- Frozen sets can be created using the function frozenset().
- This data type supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have methods that add or remove elements.

3.5.3 Deleting Values in a Set

- There are different ways for deletion of sets in Python programming. Some of them are given below:
 1. Remove elements from a set by using discard() method.
 2. Remove elements from a set by using remove() method. The only difference between remove and discard method is that If specified item is not present in a set then remove() method raises KeyError.
 3. The pop() method removes random item from a set and returns it.
 4. The clear() method remove all items from the set.

Example: For deleting values in sets.

```
>>> b={"a","b","c"}
>>> b
{'c', 'b', 'a'}
>>> b.discard("b")
>>> b
{'c', 'a'}
>>> b.remove("a")
>>> b
{'c'}
>>> b={"a","b","c"}
>>> b.pop()
'c'
>>> b.clear()
>>> b
set()
```

3.5.4 Updating Sets

- We can update a set by using add() method or update() method.
 1. **Using add() Method:**
 - We can add elements to a set by using add() method. The add() method takes one argument which is the element we want to add in set.
-

- At a time only one element can be added to the set by using add() method. Loops can be used to add multiple elements at a time using add() method.

Syntax: A.add(element)

here, A is set which will be updated by given element.

Example: For add() method which updates sets.

```
>>> a={1,2,3,4}
>>> a.add(6)
>>> print(a)
{1, 2, 3, 4, 6}
>>> a={2,4,6,8}
>>> a.add(3)
>>> print(a)
{2, 3, 4, 6, 8}
>>>
```

2. Using update() Method:

- The update() adds elements from lists/strings/tuples/sets to the set.
- The update() method can accept lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.

Syntax: A.update(B)

Here, B can be lists/strings/tuples/sets and A is set which will be updated without duplicate entries.

Example: For update() method which updates sets.

```
>>> a={1,2,3}
>>> b={'a','b','c'}
>>> a
{1, 2, 3}
>>> b
{'c', 'b', 'a'}
>>> a.update(b)
>>> a
{1, 2, 3, 'b', 'a', 'c'}
>>> a.update({3,4})
>>> a
{1, 2, 3, 'b', 4, 'a', 'c'}
```

3.5.5 Basic Set Operations

- In this section we will study basic operations which are performed on sets.

1. Union of Sets:

- The union operation on two sets produces a new set containing all the distinct elements from both the sets.
- Union operation performed on two sets returns all the elements from both the sets.
- The union of A and B are defined as the set that consists of all elements belonging to either set A or set B (or both). It is performed by using | operator.

Example: For union operation in sets.

```
>>> A={1,2,4,6,8}
>>> B={1,2,3,4,5}
>>> C=A | B
>>> C
{1, 2, 3, 4, 5, 6, 8}
```

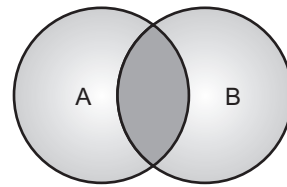


Fig. 3.6: Union Operation

- In the below example the element “Wed” is present in both the sets.

Example:

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

Output:

```
{'Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'}
```

2. Intersection of Sets:

- The intersection operation on two sets produces a new set containing only the common elements from both the sets.
- Intersection operation performed on two sets returns all the elements which are common or in both the sets.
- The intersection of A and B is defined as the set composed of all elements that belong to both A and B. It is performed by using & operator.

Example: For intersection operation in sets.

```
>>> A={1,2,4,6,8}
>>> B={1,2,3,4,5}
>>> C=A & B
>>> C
{1, 2, 4}
```

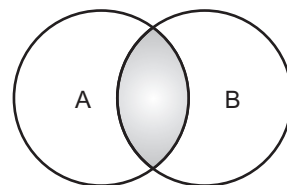


Fig. 3.7: Intersection Operation

- In the below example the element “Wed” is present in both the sets.

Example:

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

Output:

```
{'Wed'}
```

3. Difference of Sets:

- The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set.
- Difference operation on two sets set1 and set2 returns all the elements which are present on set1 but not in set2. It is performed by using – operator.

Example: For difference operation in sets.

```
>>> A={1,2,4,6,8}
>>> B={1,2,3,4,5}
>>> C=A - B
>>> C
{8, 6}
```

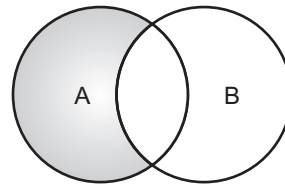


Fig. 3.8: Difference Operation

- In the following example, the element “Wed” is present in both the sets so it will not be found in the result set.

Example:

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

Output:

```
{'Mon', 'Tue'}
```

4. Compare Sets:

- We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

Example:

```

DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes)
print(SupersetRes)

```

Output:

```

True
True

```

5. Symmetric Difference:

- Symmetric Difference operation performed by using \wedge operation. The symmetric difference of Two Sets A and B is the set $(A - B) \cup (B - A)$. It represent set of all elements which belongs either to A or B but not both.

Example: For symmetric difference operation in sets.

```

>>> A={1,2,4,6,8}
>>> B={1,2,3,4,5}
>>> C=A ^ B
>>> C
{3, 5, 6, 8}

```

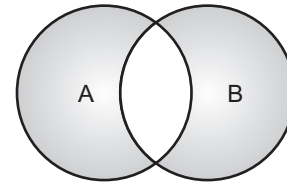


Fig. 3.9: Symmetric Difference Operation

3.5.6 Working with Sets

- A set is an unordered and mutable collection of unique elements. Sets are written with curly brackets ($\{\}$), being the elements separated by commas.
- Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc. are commonly used with set to perform different tasks. Consider set $A = \{3, 1, 6, 7\}$.
- Following table lists built-in functions for set:

Sr. No.	Function	Description	Example
1.	<code>all()</code>	Return True if all elements of the set are true (or if the set is empty).	<pre>>>> all (A) True</pre>
2.	<code>any()</code>	Return True if any element of the set is true. If the set is empty, return False.	<pre>>>> any (A) True</pre>

contd. ...

3.	<code>len()</code>	Return the length (the number of items) in the set.	<code>>>> len(A)</code> 4
4.	<code>max()</code>	Return the largest item in the set.	<code>>>> max(A)</code> 7
5.	<code>min()</code>	Return the smallest item in the set.	<code>>>> min(A)</code> 1
6.	<code>sorted()</code>	Return a new sorted list from elements in the set(does not sort the set itself).	<code>>>> sorted(A)</code> [1, 3, 6, 7]
7.	<code>sum()</code>	Return the sum of all elements in the set.	<code>>>> sum(A)</code> 17

Example: For set functions.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
print("Union ", A.union(B))
print("Intersection ", A.intersection(B))
A.intersection_update(B)
print("Intersection Update ", A)
A = {'red', 'green', 'blue'}
print("Difference ", A.difference(B))
A.difference_update(B)
print("Difference Update ", A)
A = {'red', 'green', 'blue'}
print("Symmetric Difference", A.symmetric_difference(B))
A = {'red', 'green', 'blue'}
A.symmetric_difference_update(B)
print("Symmetric Difference Update", A)
A={'red'}
print(A.isdisjoint(B))
print(A.issubset(B))
print(A.issuperset(B))
```

Output:

```
Union {'yellow', 'orange', 'green', 'red', 'blue'}
Intersection {'red'}
Intersection Update {'red'}
Difference {'green', 'blue'}
Difference Update {'blue', 'green'}
Symmetric Difference {'orange', 'blue', 'yellow', 'green'}
Symmetric Difference Update {'blue', 'orange', 'yellow', 'green'}
False
True
False
```

Methods for Set:

Sr. No.	Method	Description
1.	union()	Return a new set containing the union of two or more sets
2.	intersection()	Returns a new set which is the intersection of two or more sets
3.	intersection_update()	Removes the items from this set that are not present in other sets
4.	difference()	Returns a new set containing the difference between two or more sets
5.	difference_update()	Removes the items from this set that are also included in another set
6.	symmetric_difference()	Returns a new set with the symmetric differences of two or more sets
7.	symmetric_difference_update()	Modify this set with the symmetric difference of this set and other set
8.	isdisjoint()	Determines whether or not two sets have any elements in common
9.	issubset()	Determines whether one set is a subset of the other
10.	issuperset()	Determines whether one set is a superset of the other.
11.	add(item)	It adds an item to the set. It has no effect if the item is already present in the set.
12.	discard(item)	It removes the specified item from the set.
13.	remove(item)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
14.	pop()	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
15.	update()	Updates the set with the union of itself and others.

PROGRAMS

Program 1: Program to function which print Fibonacci series.

```
def fibonacci(n):
    if(n <= 1):
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
    n = int(input("Enter number of terms:"))
    print("Fibonacci sequence:")
    for i in range(n):
        print fibonacci(i)
```

Output:

```
Enter number of terms:5
Fibonacci sequence:
0 1 1 2 3
```

Program 2: Program for anonymous function to find area of circle.

```
import math
area=lambda r : math.pi *r*r
r= float(input("Enter Radius :"))
print(area(r))
```

Output:

```
Enter Radius :4
50.26548245743669
```

Program 3: Program for function which print a dictionary where the keys are numbers between 1 and n (both included) and the values are square of keys.

```
n=int(input("Enter a number:"))
d={x:x*x for x in range(1,n+1)}
print(d)
```

Output:

```
Enter a number:5
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Program 4: Program to find GCD of number using recursion.

```
def gcd(a,b):
    if(b==0):
        return a
    else:
        return gcd(b,a%b)
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
GCD=gcd(a,b)
print("GCD is: ")
print(GCD)
```

Output:

```
Enter first number:5
Enter second number:15
GCD is:
5
```

Program 5: Program for a function that accept two strings as input and find concatenated string.

```
def concat(s1,s2)
    print ("First string:",s1)
    print ("Second string:",s2)
    print (Concatenated string:",(s1+s2))
s1=input("Enter first string: ")
s2=input("Enter second string: ")
concat(s1,s2)
```

Output:

```
Enter first string: hello
Enter second string: students
First string: hello
Second string: students
Concatenated string: hellostudents
```

Program 6: Program for user defined function bin() which convert given number into binary.

```
def bin(n):
    if n >=1:
```

```
        bin(n//2)
    print (n%2, end='')
    dec = int(input( "Enter an integer : "))
    bin(dec)
```

Output:

```
Enter an integer : 6
0110
```

Program 7: Program for searches an item in a sorted list. The function should return the index of element to be searched in the list.

```
def binary_search(alist, key):
    start = 0
    end = len(alist)
    while start < end:
        mid = (start + end)//2
        if alist[mid] > key:
            end = mid
        elif alist[mid] < key:
            start = mid + 1
        else:
            return mid
    return -1

alist = input('Enter the sorted list of numbers: ') alist = alist.split()
alist = [int(x) for x in alist]
key = int(input('The number to search for: '))
index = binary_search(alist, key)
if index < 0:
    print('{} was not found.'.format(key))
else:
    print('{} was found at index {}'.format(key, index))
```

Output:

```
Enter the sorted list of numbers: 3 5 10 12 15 20
The number to search for: 12
12 was found at index 3.
```


Program 8: Program to create a list of tuples with the first element as the number and second element as the square of the number.

```
l_range=int(input("Enter the lower range:"))
u_range=int(input("Enter the upper range:"))
a=[(x,x**2) for x in range(l_range,u_range+1)]
print(a)
```

Output:

```
Enter the lower range:1
Enter the upper range:4
[(1, 1), (2, 4), (3, 9), (4, 16)]
```

Program 9: Program to create a tuple with numbers and print one item.

```
#Create a tuple with numbers
tuplex = 5, 10, 15, 20, 25
print(tuplex)
#Create a tuple of one item
tuplex = 5,
print(tuplex)
```

Output:

```
(5, 10, 15, 20, 25)
(5,)
```

Program 10: Program to unpack a tuple in several variables.

```
#create a tuple
tuplex = 4, 8, 3
print(tuplex)
n1, n2, n3 = tuplex
#unpack a tuple in variables
print(n1 + n2 + n3)
# The number of variables must be equal to the number of items of the tuple
# n1, n2, n3, n4= tuplex
```

Output:

```
(4, 8, 3)
15
Traceback (most recent call last):
File "32fd05c0-3096-11e7-a6a0-0b37d4d0b2c6.py", line 8, in <module>
n1, n2, n3, n4 = tuplex
ValueError: not enough values to unpack (expected 4, got 3)
```

Program 11: Program to add an item in a tuple.

```
#create a tuple
tuplex = (4, 6, 2, 8, 3, 1)
print(tuplex)
#tuples are immutable, so you can not add new elements
#using merge of tuples with the + operator you can add an element and it
will create a new tuple
tuplex = tuplex + (9,)
print(tuplex)
#adding items in a specific index
tuplex = tuplex[:5] + (15, 20, 25) + tuplex[:5]
print(tuplex)
#converting the tuple to list
listx = list(tuplex)
#use different ways to add items in list
listx.append(30)
tuplex = tuple(listx)
print(tuplex)
```

Output:

```
(4, 6, 2, 8, 3, 1)
(4, 6, 2, 8, 3, 1, 9)
(4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3)
(4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3, 30)
```

Program 12: Program to convert a tuple to a string.

```
tup = ('e', 'x', 'e', 'r', 'c', 'i', 's', 'e', 's')
str = ''.join(tup)
print(str)
```

Output:

```
exercises
```

Program 13: Program to get the 4th element from front and 4th element from last of a tuple.

```
#Get an item of the tuple
tuplex = ("p", "y", "t", "h", "o", "n", "b", "o", "o", "k")
print(tuplex)
#Get item (4th element)of the tuple by index
```

```
item = tuplex[3]
print(item)
#Get item (4th element from last)by index negative
item1 = tuplex[-4]
print(item1)
```

Output:

```
('p', 'y', 't', 'h', 'o', 'n', 'b', 'o', 'o', 'k')
h
b
```

Program 14: Program to find the repeated items of a tuple.

```
#create a tuple
tuplex = 2, 4, 5, 6, 2, 3, 4, 4, 7
print(tuplex)
#return the number of times it appears in the tuple.
count = tuplex.count(4)
print(count)
```

Output:

```
(2, 4, 5, 6, 2, 3, 4, 4, 7)
3
```

Program 15: Program to do iteration over sets.

```
#Create a set
num_set = set([0, 1, 2, 3, 4, 5])
for n in num_set:
    print(n)
```

Output:

```
0
1
2
3
4
5
```

Program 16: Program to add and remove operation on set.

```
>>> #A new empty set
>>> color_set = set()
>>> #Add a single member
```

```
>>> color_set.add("Red")
>>> print(color_set)
{'Red'}
>>> #Add multiple items
>>> color_set.update(["Blue", "Green"])
>>> print(color_set)
{'Red', 'Blue', 'Green'}
>>>
>>># Remove operation on set
pop() function:
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> num_set.pop()
0
>>> print(num_set)
{1, 2, 3, 4, 5}
>>> num_set.pop()
1
>>> print(num_set)
{2, 3, 4, 5}
>>>
remove() function:
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> num_set.remove(0)
>>> print(num_set)
{1, 2, 3, 4, 5}
>>>
discard() function:
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> num_set.discard(3)
>>> print(num_set)
{0, 1, 2, 4, 5}
>>>
```

Program 17: Program to perform different set operations like in mathematics.

```
# define sets
E = {0, 2, 4, 6, 8};
N = {1, 2, 3, 4, 5};

# set union
print("Union of E and N is",E | N)

# set intersection
print("Intersection of E and N is",E & N)

# set difference
print("Difference of E and N is",E - N)

# set symmetric difference
print("Symmetric difference of E and N is",E ^ N)
```

Output:

```
Union of E and N is {0, 1, 2, 3, 4, 5, 6, 8}
Intersection of E and N is {2, 4}
Difference of E and N is {8, 0, 6}
Symmetric difference of E and N is {0, 1, 3, 5, 6, 8}
```

Program 18: Program to find the length of a set.

```
string=input("Enter string:")
count=0
for i in string:
    count=count+1
print("Length of the string is:")
print(count)
```

Output:

```
Enter string:Hello
Length of the string is:
5
```

Program 19: Program to find maximum and the minimum value in a set.

```
#Create a set
seta = set([5, 10, 3, 15, 2, 20])
#Find maximum value
print(max(seta))
#Find minimum value
print(min(seta))
```

Output:

```
20
2
```

Program 20: Program to sort (ascending and descending) a dictionary by value.

```
import operator
d= {1:2, 3:4, 4:3, 2:1, 0:0}
print ('Original Dictionary :',d)
sorted_d =sorted(d.items(), key=operator.itemgetter(1))
print('Dictionary in ascending order by value:',sorted_d)
sorted_d=sorted(d.items(), key=operator.itemgetter(1), reverse =True)
print('Dictionary in descending order by value :',sorted_d)
```

Program 21: Program to add a key to a dictionary.

Sample Dictionary: {0: 10, 1: 20}

Expected Result: {0: 10, 1: 20, 2: 30}

```
d = {0:10, 1:20}
print(d)
d.update({2:30})
print(d)
```

Program 22: Program to concatenate following dictionaries to create a new one.

Sample Dictionary:

```
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
```

Expected Result: {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

```
dic1={1:10,2:20}
dic2={3:30,4:40}
dic3={5:50,6:60}
dic4 = {}
for d in (dic1, dic2, dic3):
    dic4.update(d)
print(dic4)
```

Program 23: Program to check if a given key already exists in a dictionary.

```
d = {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
def is_key_present(x):
    if x in d:
        print('Key is present in the dictionary')
    else:
        print('Key is not present in the dictionary')
is_key_present(5)
is_key_present(9)
```

Program 24: Program to generate and print a dictionary that contains a number (between 1 and n) in the form (x, x*x).

Sample Dictionary: (n = 5)

Expected Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

```
n=int(input("Input a number "))
d = dict()
for x in range(1,n+1):
    d[x]=x*x
print(d)
```

Output:

```
10
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Program 25: Program to print a dictionary where the keys are numbers between 1 and 15 (both included) and the values are square of keys.

Sample Dictionary: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225}

```
d=dict()
for x in range(1,16):
    d[x]= x**2
print(d)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100,
11: 121, 12: 144, 13: 169, 14: 196, 15: 225}
```

Program 26: Program to combine two dictionary adding values for common keys.

```
d1 = {'a': 100, 'b': 200, 'c':300}
d2 = {'a': 300, 'b': 200, 'd':400}
d1 = {'a': 100, 'b': 200, 'c':300}
d2 = {'a': 300, 'b': 200, 'd':400}
d = Counter(d1) + Counter(d2)
print(d)
```

Output:

```
Counter({'b': 400, 'd': 400, 'a': 400, 'c': 300})
```

Program 27: Program to create a dictionary from two lists without losing duplicate values.

Sample lists: ['Class-V', 'Class-VI', 'Class-VII', 'Class-VIII'], [1, 2, 2, 3]

Expected Output:

```
defaultdict(<class 'set'>, {'Class-VII': {2}, 'Class-VI': {2}, 'Class-VIII': {3}, 'Class-V': {1}})
```

```
class_list = ['Class-V', 'Class-VI', 'Class-VII', 'Class-VIII']
id_list = [1, 2, 2, 3]
temp = defaultdict(set)
for c, i in zip(class_list, id_list):
    temp[c].add(i)
print(temp)
```

Output:

```
defaultdict(<class 'set'>, {'Class-VII': {2}, 'Class-VI': {2}, 'Class-V': {1}, 'Class-VIII': {3}})
```

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

1. Which is mutable (changeable) means the contents of a list can be changed during program execution?
(a) List (b) Tuple
(c) Dictionary (d) String
2. In Python list can be created by using
(a) [] (b) list()
(c) Both (a) and (b) (d) None of the mentioned
3. We can access a range of items in a list by using the,
(a) assignment operator (=) (b) slicing operator : (colon)
(c) + operator (d) * operator
4. Which method adds an item to the end of the list?
(a) append() (b) insert()
(c) extend() (d) All of the mentioned
5. For removing/deleting list items use,
(a) del keyword (b) remove()
(c) pop() (d) All of the mentioned
6. Which means accessing all the elements or items of the list?
(a) Inserting (b) Traversing
(c) Deleting (d) None of the mentioned
7. For revering a list use,
(a) reverse() (b) reversed()
(c) Both (a) and (b) (d) None of the mentioned
8. Which method in a list inserts an item at the defined index?
(a) insert() (b) extend()
(c) append() (d) All of the mentioned
9. To change an item or a range of items in a list we use,
(a) assignment operator (=) (b) slicing operator : (colon)
(c) + operator (d) * operator
10. Which is a group of related statements that performs a specific task?
(a) string (b) tuple
(c) function (d) list
11. A function is defined using the,
(a) def keyword (b) create keyword
(c) defn keyword (d) All of the mentioned
12. Which is the value that is sent to the function when it is called?
(a) parameter (b) values
(c) argument (d) None of the mentioned

-
13. To call a function use,
(a) the function name followed by parenthesis ()
(b) the function name followed by brackets []
(c) the function name followed by curly braces {}
(d) None of the mentioned
14. To perform explicit type conversion use,
(a) int() (b) float()
(c) string() (d) All of the mentioned
15. Mathematical built-in function includes,
(a) pow() (b) abs()
(c) max() (d) All of the mentioned
16. The math module of Python contains following which function,
(a) sqrt() (b) cos()
(c) ceil() (d) All of the mentioned
17. Which means that a function calls itself?
(a) Anonymous (b) Recursion
(c) Boolean (d) All of the mentioned
18. Which statement is used to exit a function and go back to the place from where it was called?
(a) exit (b) return
(c) null (d) None of the mentioned
19. The order in which statements are executed during a program run called as,
(a) control of execution (b) flow of selection
(c) flow of execution (d) flow of looping
20. Keyword arguments are related to the,
(a) function calls (b) function definition
(c) function parameters (d) None of the mentioned
21. The accessibility of a variable in different parts of a program is referred to as its,
(a) argument (b) scope
(c) parameters (d) None of the mentioned
22. Which diagram to represent the state of a program during a function call?
(a) stack (b) tuple
(c) string (d) list
23. Which variables are used in nested functions whose local scope is not defined?
(a) local (b) nonlocal
(c) global (d) None of the mentioned
24. Which functions which do not return any value.
(a) void (b) lambda
(c) Boolean (d) All of the mentioned
-

25. The anonymous functions are declared by using the,
(a) void (b) lambda
(c) Boolean (d) All of the mentioned
26. Which is a collection of items or elements in ordered and immutable?
(a) tuple (b) string
(c) list (d) set
27. We can use the index operator [] to access an item in a tuple, where the index starts from,
(a) 1 (b) -1
(c) 2 (d) 0
28. Which operator to combine two tuples?
(a) * (b) +
(c) % (d) /
29. Which is a two-dimensional data structure where numbers are arranged into rows and columns?
(a) set (b) list
(c) matrix (d) tuple
30. Which method converts list in to the tuple?
(a) len(tuple) (b) cmp(tuple1, tuple2)
(c) index() (d) tuple(seq)
31. Which are used to store data values in key:value pairs?
(a) dictionary (b) list
(c) tuple (d) set
32. Which method will return a list of all the keys in the dictionary?
(a) get() (b) keys()
(c) fromkeys() (d) clear()
33. Which method removes the last inserted item in directory?
(a) pop() (b) popdict()
(c) popitem() (d) All of the mentioned
34. Which method of directory updates the dictionary with the specified key-value pairs?
(a) update() (b) modify()
(c) getupdate() (d) All of the mentioned
35. Which is a collection of items not in any particular order?
(a) tuple (b) set
(c) list (d) dictionary
36. The set in Python can be created by,
(a) placing all the items (elements) inside curly braces {}, separated by comma
(b) using the built-in set() function
(c) Both (a) and (b) (d) None of the mentioned

37. To add one item to a set use,
(a) add() (b) pop()
(c) discard() (d) remove()
38. A particular item can be removed from a set using the methods,
(a) discard() (b) remove()
(c) Both (a) and (b) (d) None of the mentioned
39. Sets can be used to carry out mathematical set operations like,
(a) union (b) intersection
(c) difference (d) All of the mentioned
40. In order to check if an item exists in the Set, we can use,
(a) in operator (b) not in operator
(c) * operator (d) + operator
41. Suppose $s1 = \{1, 2, 4, 3\}$ and $s2 = \{1, 5, 4, 13\}$, what is $s1 \cup s2$?
(a) $\{1, 2, 4, 3, 1, 5, 4, 13\}$ (b) $\{1, 2, 4, 3, 5, 13\}$
(c) $\{1, 2, 4, 3\}$ (d) $\{1, 5, 4, 13\}$
42. Use of type() function is,
(a) Returns the type of the any variable
(b) Changes the type of the passed variable
(c) Declares the type of variable.
(d) Returns the type of the passed variable
43. A recursive function is,
(a) A function that calls itself
(b) A function that that calls all the functions in the program except itself
(c) A function that calls all the functions in the program
(d) None of the above
44. What is the output of following code?

```
def my_function(x):  
    return 5 * x  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```


(a) 15 25 45 (b) 25 45 15
(c) 15 45 25 (d) Error
45. What is the output of following code?

```
def myfunc(n):  
    return lambda a : a * n  
mydoubler = myfunc(2)  
print(mydoubler(11))
```


(a) 24 (b) 22
(c) 27 (d) Error

46. What is the output of following code?

```
tuple = tuple(("apple", "banana"))  
print(len(tuple))
```

- | | |
|-------|-------|
| (a) 3 | (b) 4 |
| (c) 2 | (d) 1 |

47. What is the output of following code?

```
tuple1 = ("a", "b", "c")  
tuple2 = (1, 2, 3)  
tuple3 = tuple1 + tuple2  
print(tuple3)
```

- | | |
|------------------------------|------------------------------|
| (a) ('a', 'b', 'c', 1, 2, 3) | (b) (1, 2, 3, 'a', 'b', 'c') |
| (c) ('a', 1, 'b', 2, 'c', 3) | (d) (1, 'a', 2, 'b', 3, 'c') |

48. What is the output of following code?

```
x = {"apple", "banana"}  
y = {"google", "microsoft", "apple"}  
x.update(y)  
print(x)
```

- (a) {'apple', 'google', 'banana', 'microsoft'}
(b) {'microsoft', 'google', 'apple', 'banana'}
(c) {'google', 'apple', 'banana', 'microsoft'}
(d) {'banana', 'google', 'apple', 'microsoft'}

49. What is the output of following code?

```
dict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
dict.pop("model")  
print(thisdict)
```

- | | |
|---|--|
| (a) {'brand': 'Ford', 'year': 1964} | (b) {'brand': 'Mustang', 'year': 1964} |
| (c) {'brand': 'Ford', 'model': 'Mustang'} | (d) Error |

50. What is the output of following code?

```
fruits = ['apple', 'banana', 'cherry']  
fruits.reverse()  
print(fruits)
```

- | | |
|-----------------------------------|-----------------------------------|
| (a) ['banana', 'cherry', 'apple'] | (b) ['cherry', 'banana', 'apple'] |
| (c) ['apple', 'cherry', 'banana'] | (d) Error |

51. What is the output of following code?

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

- (a) {'cherry', 'apple'} (b) {'apple', 'cherry'}
(c) {'apple'} (d) Error

52. What is the output of following code?

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

- (a) ("apple", "kiwi", "cherry") (b) ("kiwi", "apple", "cherry")
(c) ("cherry", "apple", "kiwi") (d) Error

Answers

1. (a)	2. (c)	3. (b)	4. (a)	5. (d)	6. (b)	7. (c)	8. (a)	9. (a)	10. (c)
11. (a)	12. (c)	13. (a)	14. (d)	15. (d)	16. (d)	17. (a)	18. (b)	19. (c)	20. (a)
21. (b)	22. (a)	23. (b)	24. (a)	25. (b)	26. (a)	27. (d)	28. (b)	29. (c)	30. (d)
31. (a)	32. (b)	33. (c)	34. (a)	35. (b)	36. (c)	37. (a)	38. (c)	39. (d)	40. (a)
41. (b)	42. (d)	43. (a)	44. (a)	45. (a)	46. (c)	47. (a)	48. (c)	49. (a)	50. (b)
51. (b)	52. (a)								

Q.II Fill in the Blanks:

- _____ are used to store multiple items in a single variable.
- _____ are written with curly brackets, and have keys and values
- A _____ is a block of organized, reusable code that is used to perform a single, related operation/task.
- A _____ is a collection of items which is ordered and unchangeable.
- A _____ itself is mutable i.e., we can add or remove items from it.
- The items in the list are separated with the _____ and enclosed with the _____.
- The _____ diagrams show the value of each variable, but they also show the function each variable belongs to.
- To determine how many items a tuple has, use the _____ function.
- Lists are _____, meaning their elements can be changed unlike string or tuple.
- The _____ statement can contain an expression that gets evaluated and the value is returned.
- In _____ type conversion, Python automatically converts one data type to another data type.

12. A set is created by using the _____ function.
13. We can access tuple items by referring to the _____ number, inside square brackets ().
14. _____ of a variable is the portion of a program where the variable is accessible.
15. To add items from another set into the current set, use the _____ method.
16. The element of the list can access by _____.
17. If we want to multiply the content of a tuple a given number of times, we can use the _____ operator.
18. Python provides the _____ keyword to define the function.
19. The _____ keyword will delete the set completely.
20. The _____ method searches the tuple for a specified value and returns the position of where it was found
21. A _____ function is a small anonymous function.
22. The _____ method return a set containing the union of sets.
23. The _____ method to remove an item at the given index.
24. We can access an item of a tuple by using its index number inside the index operator [] and this process is called as, _____.
25. To call the function, use the function _____ followed by the parentheses ().
26. The _____ method Add all elements of a list to the another list.
27. In Python, _____ means passing the actual value as an argument in the function.
28. A _____ argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
29. The _____ are types of information which can be passed into the function.
30. The * operator _____ a list for the given number of times.
31. Variables that are defined _____ a function body have a local scope (means that local variables can be accessed only inside the function in which they are declared)
32. The elements of the list can be accessed by using the _____ operator [].
33. The ability to call one function from within another called as _____.
34. The dictionary can be created by using _____ key-value pairs enclosed with the curly brackets {} and each key is separated from its value by the colon (:).
35. A function is said to be a recursive function if it is called _____.
36. The variable defined outside any function is known to have a _____ scope.
37. The _____ of execution refers to the order in which statements are executed during a program run.
38. To determine if a specified key is present in a dictionary use the _____ keyword.
39. Python provides the _____ function which is used to remove the element from the list.

40. The parameters used in the function call are called _____ parameters.
41. The len() method is used to calculate the _____ of the list.
42. Functions that return values are sometimes called _____ functions.
43. The _____ method will return a new set with only the common elements in all the sets.
44. We can access the items of a dictionary by referring to its _____ name, inside square brackets.
45. Lists are _____ collections with unique indexes for each item.
46. _____ means that the value is directly passed as the value to the argument of the function.
47. The elements in the list can be accessed from right to left by using _____ indexing (value starts from -1 to -length of the list).
48. The parameters used in the header of function definition are called _____ parameters of the function.
49. The _____ method will return a list of all the values in the dictionary.
50. The _____ method sorts the elements in the list in ascending order.
51. The _____ is used to find whether a set is a subset of another set.

Answers

1. Lists	2. Dictionaries	3. function	4. tuple
5. set	6. , []	7. stack	8. len()
9. mutable	10. return	11. implicit	12. set()
13. index	14. Scope	15. update()	16. index
17. *	18. def	19. del	20. index()
21. lambda	22. union()	23. pop()	24. Indexing
25. name	26. extend()	27. call by reference	28. default
29. arguments	30. repeats	31. inside	32. slice
33. composition	34. multiple	35. itself	36. global
37. flow	38. in	39. remove()	40. actual
41. length	42. fruitful	43. intersection()	44. key
45. ordered	46. Pass by value	47. negative	48. formal
49. values()	50. sort()	51. issubset()	

Q.III State True or False:

1. The list is mutable, meaning that we can change, add, and remove items in a list after it has been created.
2. A function can be defined as the organized block of reusable code, which can be called whenever required.
3. Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

4. Tuples are ordered collections of heterogeneous (means tuple can store variables of all types) data that are unchangeable.
5. A set is ordered collection of items.
6. The `max()` returns the maximum value in the list while the `min()` returns the minimum value in the list.
7. Using the `setdefault()` method default value can be assigned to a key in the dictionary.
8. A void function can take any number of arguments, but can only have one expression.
9. All the items in a directory can be removed at once, using the `clear()` method.
10. A Frozen set is an unordered collection of immutable unique items.
11. We can iterate a tuple using a while loop.
12. The return statement cannot be used outside of the function.
13. Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
14. A list is created by placing all the items (elements) inside square brackets [], separated by commas.
15. Functions are only executed when they are called.
16. we can access any item from a tuple using its item/element number.
17. To add several items using `extend()` method.
18. The `item()` method of directory returns all the items present in the dictionary.
19. User-defined functions are created by the user to meet his/her requirements.
20. We can even specify a range of items to be accessed from a tuple using the technique called 'Slicing.' The operator used is ':'.
21. We can use the boolean keyword to create small anonymous functions.
22. In order to check whether a particular key exists in a dictionary, we can use the `keys()` method and `in` operator.
23. A list can also have another list as an item. This is called a nested list.
24. The process of converting the value of one data type to another data type is called type conversion.
25. Creating a tuple is as simple as placing items inside curly braces {} separated by commas. An item has a key and a corresponding value that is expressed as a pair (key:value).
26. The `add()` method is used to add a single element whereas the `update()` method is used to add multiple elements to the set.
27. The values in dictionary items can be of any data type.
28. To call a function means that we are telling the program to execute the function.
29. We can access a range of items in a list by using the * operator.

30. We can also use the del keyword to remove individual items or the entire dictionary itself.
31. The return terminates the function execution and transfers the result where the function is called.
32. Information can be passed into functions as arguments.
33. We can use the index operator () to access an item in a list.
34. As tuples are ordered sequences of items, the index values start from 0 to the tuple's length.
35. Fruitful functions are those functions which return values.
36. The remove() method removes the specified item.
37. List objects have a sort() method that will sort the list alphanumerically, ascending, by default.
38. We can search for a certain item in a tuple using the count() method and it will return the position of that particular item in the tuple.
39. The del keyword will delete the entire tuple.
40. A parameter is the variable listed inside the parentheses in the function definition.
41. The reverse() method reverses the current sorting order of the elements.
42. Dictionaries are immutable.
43. We use nonlocal keyword to create nonlocal variable.
44. The function can have more arguments than the arguments defining the function called as variable-length arguments.
45. To join, or concatenate, two or more lists in Python by using the + operator.
46. The repetition operator * enables the tuple elements to be repeated multiple times.
47. The scope of a variable determines the portion of the program where the variable is visible.
48. To determine how many times a specific item occurred in a tuple, we can use the count() method of a tuple element.
49. Lambda function is created without using the def keyword.
50. The count() method of set returns the number of elements with the specified value.
51. Recursion is process in which a function calls itself.
52. Nested tuples are tuples within a tuple i.e., when a tuple contains another tuple as its member then it is called a nested tuple.
53. A variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.
54. The values() method will return each item in a dictionary, as tuples in a list.
55. A tuple in Python is an immutable data type which means a tuple once created cannot be altered/modified.

Answers

1. (T)	2. (T)	3. (T)	4. (F)	5. (T)	6. (T)	7. (T)	8. (F)	9. (T)	10. (T)
11. (F)	12. (T)	13. (T)	14. (T)	15. (T)	16. (F)	17. (T)	18. (T)	19. (T)	20. (T)
21. (F)	22. (T)	23. (T)	24. (T)	25. (F)	26. (T)	27. (T)	28. (T)	29. (F)	30. (T)
31. (T)	32. (T)	33. (F)	34. (T)	35. (T)	36. (T)	37. (T)	38. (F)	39. (T)	40. (T)
41. (T)	42. (F)	43. (T)	44. (T)	45. (T)	46. (F)	47. (T)	48. (T)	49. (T)	50. (F)
51. (T)	52. (T)	53. (T)	54. (F)	55. (T)					

Q.IV Answer the following Questions:**(A) Short Answer Questions:**

1. Define list.
2. Define tuple.
3. What is function?
4. Define set.
5. Define dictionary.
6. What is the purpose of reduce()?
7. What is the use of + and * operators in list.
8. Which methods are used to add and remove elements from set?
9. What is the purpose of stack diagrams?
10. Define recursion.
11. Tuples are ordered and unchangeable. State true or false.
12. What is variable-length argument tuples?
13. What is the use of in Operator?
14. List operations on dictionary.
15. Define void function.
16. Define argument in function.
17. Which function is used for creating Frozenset?
18. List characteristic of dictionary.
19. Compare list and tuple, (any two point).
20. Define anonymous function.
21. List operations on tuple.
22. Give the purpose of a stack diagram.
23. What are the built-in functions used for type conversion?
24. What is flow of execution?
25. What is difference between del and pop() in list.
26. What is use of dict() function?
27. What is use of set() function?

(B) Long Answer Questions:

1. What is list? How to crate it? Explain with example.
2. What is function? How to crate it? Explain with example.
3. Describe any four math functions with example.
4. What is meant by parameters and arguments? How to use them in function? Explain with example.
5. What is Boolean function? Explain with the help of example.
6. What is lambda function? Explain its forms in detail.
7. How to add, update and delete elements in list?
8. What is dictionary? How to create it? Explain with example.
9. How to accessing elements in dictionary? Describe with example.
10. Describe the term composition with example.
11. What is recursion? Explain recursive function with example.
12. What is set? How to create it? Explain with example.
13. How to accessing elements in set?
14. How we can remove elements from dictionary?
15. What are the usage of dictionary copy(), get(), items() and keys() methods?
16. What is the difference between list, set and dictionary in python?
17. What are negative indexes in list and why are they used?
18. What is tuple? How to create, access and delete tuple? Explain with example.
19. With the help of example explain the use of + and * operators in tuples?
20. Which are basic tuple operations? Explain with example.
21. What are built-in dictionary functions? Explain two of them.
22. Explain how to update and delete elements in dictionary.
23. Which are properties of dictionary keys?
24. What is anonymous function? How to create it? Explain with example.
25. What is stack diagram? Explain with example.
26. Explain set union and intersection with example.
27. Describe actual and formal parameters in detail.
28. What is difference between call by value and call by reference?
29. What are void and boolean functions? Explain with example.
30. Describe the use of filter(), map() and reduce() functions.
31. What will be the output of following code?

```
tuple1=(2,4,3)
tuple3=tuple1*2
print(tuple3)
```

32. What will be the output of following code?

```
def sum():
    a = 10
    b = 20
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is:",sum())
```

33. What will be the output of following code?

```
x = lambda a: a + 10
print(x(5))
```

34. What will be the output of following code?

```
def myfun(name, msg="Good morning!"):
    print("Hello", name + ', ' + msg)
myfun("Amar")
myfun("Kiran", "How do you do?")
```

35. What will be the output of following code?

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 5
print("The factorial of", num, "is", factorial(num))
```

36. What will be the output of following code?

```
my_set = {1, 2, 3}
print(my_set)
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

37. What will be the output of following code?

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)
print('a' in my_tuple)
print('b' in my_tuple)
```

38. What will be the output of following code?

```
marks = {'Physics':67, 'Maths':87}
internal_marks = {'Practical':48}
marks.update(internal_marks)
print(marks)
```



Modules, Working with Files and Exception Handling

Objectives...

- To learn Concepts of Files
- To study Basic Concepts in Modules
- To understand Packages
- To learn Regular Expressions
- To study Exception Handling

4.0 INTRODUCTION

- Modules and packages are the constructs in Python programming that promote code modularization.
- The modularization (modular programming) refers to the process of breaking a large programming task into separate, smaller, more manageable subtasks or modules.
- A module in Python programming allows us to logically organize the python code. A module is a single source code file.
- The module in Python has the .py file extension. The name of the module will be the name of the file.
- A python module can be defined as a python program file which contains a python code including python functions, class, or variables.
- In other words, we can say that our Python code file saved with the extension (.py) is treated as the module.
- In Python, packages allow us to create a hierarchical file directory structure of modules. For example, mymodule.mod1 stands for a module mod1, in the package mymodule.
- A Python package is a collection of modules which have a common purpose. In short, modules are grouped together to forms packages.
- A file is a collection of related data that acts as a container of storage as data permanently.
- The file processing refers to a process in which a program processes and accesses data stored in files.

- A file is a computer resource used for recording data in a computer storage device. The processing on a file is performed using read/write operations performed by programs.
- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- Python programming provides modules with functions that enable us to manipulate text files and binary files. Python allows us to create files, update their contents and also delete files.
- Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).
- A text file is a file that stores information in the form of a sequence of characters (textual information), while a binary file stores data in the form of bits (0s and 1s) and used to store information in the form of text, images, audios, videos etc.
- The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information.
- We can access the stored information (non-volatile) after the program termination. Python has several functions for creating, reading, updating, and deleting files.
- When an error occurs, or exception as we call it, Python will normally stop and generate an error message. Exception in Python is nothing but errors which are encountered at the run time.
- Exception Handling is the mechanism it allows to handle errors very smartly while the program is running.

4.1 MODULES

- Modules are primarily the (.py) files which contain Python programming code defining functions, class, variables, etc. with a suffix .py appended in its file name. A file containing .py python code is called a module.
- If we want to write a longer program, we can use file where we can do editing, correction. This is known as creating a script. As the program gets longer, we may want to split it into several files for easier maintenance.
- We may also want to use a function that we have written in several programs without copying its definition into each program.
- In Python we can put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module.

4.1.1 Writing/Creating and Exploring Module

- Writing a module means simply creating a file which contains python definitions and statements. The file name is the module name with the extension .py. To include module in a file, use 'import' statement.
- Follow the following steps to create modules:
 1. Create a first file as a python program with extension as .py. This is your module file where we can write a function which performs some task.

2. Create a second file in the same directory called main file where we can import the module to the top of the file and call the function.
- Second file needs to be in the same directory so that Python knows where to find the module since it's not a built-in module.
-

Example: For creating a module. Type the following code and save it as p1.py.

```
def add(a, b):
    "This function adds two numbers and return the result"
    result = a + b
    return result
def sub(a, b):
    "This function subtract two numbers and return the result"
    result = a - b
    return result
def mul(a, b):
    "This function multiply two numbers and return the result"
    result = a * b
    return result
def div(a, b):
    "This function divide two numbers and return the result"
    result = a / b
    return result
```

Import the definitions inside a module:

```
import p1
print("Addition=" , p1.add(10,20))
print("Subtraction=" ,p1.sub(10,20))
print("Multiplication=" ,p1.mul(10,20))
print("division=" ,p1.div(10,20))
```

Output:

```
Addition= 30
Subtraction= -10
Multiplication= 200
division= 0.5
```

Exploring Modules:

- Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented.
 - To exploring modules in Python the dir() function is used. For example, we have defined a function add() in the module example that we had in the beginning.
-

- We can use `dir` in example module in the following way:

```
>>> dir(example)
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__initializing__',
 '__loader__',
 '__name__',
 '__package__',
 'add']
```
- Here, we can see a sorted list of names (along with `add`). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).
- Python gives us several different ways to view module content. The method that most developers use is to work with the `dir()` function, which tells us about the attributes that the module provides.
- Function attributes are automatically generated by Python for you. These attributes perform the following tasks or contain the following information:
 1. **`__builtins__`**: Contains a listing of all the built-in attributes that are accessible from the module. Python adds these attributes automatically for us.
 2. **`__cached__`**: Tells us the name and location of the cached file that is associated with the module. The location information (path) is relative to the current Python directory.
 3. **`__doc__`**: Outputs help information for the module, assuming that you've actually filled it in. For example, if we type `os.__doc__` and press Enter, Python will output the help information associated with the `os` library.
 4. **`__file__`**: Tells us the name and location of the module. The location information (path) is relative to the current Python directory.
 5. **`__initializing__`**: Determines whether the module is in the process of initializing itself. Normally this attribute returns a value of `False`. This attribute is useful when we need to wait until one module is done loading before we import another module that depends on it.
 6. **`__loader__`**: Outputs the loader information for this module. The loader is a piece of software that gets the module and puts it into memory so that Python can use it. This is one attribute you rarely (if ever) use.
 7. **`__name__`**: Tells us just the name of the module.
 8. **`__package__`**: This attribute is used internally by the import system to make it easier to load and manage modules. We don't need to worry about this particular attribute.

4.1.2 Importing Modules

- The import statement is used to imports a specific module by using its name.
- The import statement creates a reference to that module in the current namespace. After using import statement we can refer the things defined in that module.
- We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this.
- Create second file. Let p2.py in same directory where p1.py is created. Write following code in p2.py.

- **Import the definitions inside a module:**

```
import p1
print(p1.add(10,20))
print(p1.sub(20,10))
```

Output:

```
30
10
```

- **Import the definitions using the interactive interpreter:**

```
>>> import p1
>>> p1.add(10,20)
30
>>> p1.sub(20,10)
10
>>>
```

Importing Objects from Module:

- Import statement in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. Python provides three different ways to import modules.

1. From x import a:

- Imports the module x, and creates references in the current namespace to all public objects defined by that module. If we run this statement, we can simply use a plain name to refer to things defined in module x.
- We can access attribute/method directly without dot notation.

Example 1: Import inside a module (from x import a).

```
from p1 import add
print("Addition=" , add(10,20))
```

Output:

```
Addition= 30
```

Example 2: For import on interactive interpreter.

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import sqrt
>>> sqrt(144)
12.0
```

2. From x import a, b, c:

- Imports the module x and creates references in the current namespace to the given objects. Or we can use a, b and c function in our program.
-

Example 1: Import inside a module (from x import a, b, c).

```
from p1 import add, sub
print("Addition=" , add(10,20))
print("Subtraction=" ,sub(10,20))
```

Output:

```
Addition= 30
Subtraction= -10
```

Example 2: For import on interactive interpreter.

```
>>> from math import sqrt, ceil, floor
>>> sqrt(144)
12.0
>>> ceil(2.6)
3
>>> floor(2.6)
2
```

3. From x import *:

- We can use * (asterisk) operator to import everything from the module.
-

Example 1: Import inside a module (from x import *).

```
from p1 import *
print("Addition=" , add(10,20))
print("Subtraction=" ,sub(10,20))
print("Multiplication=" ,mul(10,20))
print("division=" ,div(10,20))
```

Output:

```
Addition= 30
Subtraction= -10
Multiplication= 200
division= 0.5
```

Example 2: For import on interactive interpreter.

```
>>> from math import *
>>> cos(60)
-0.9524129804151563
>>> sin(60)
-0.3048106211022167
>>> tan(60)
0.320040389379563
```

4.1.3 Aliasing Modules

- It is possible to modify the names of modules and their functions within Python by using the 'as' keyword.
- We can make alias because we have already used the same name for something else in the program or we may want to shorten a longer name.

Syntax: import module as another_name

Example: Create a module to define two functions. One to print Fibonacci series and other for finding whether the given number is palindrome or not.

Step 1: Create a new file p1.py and write the following code in it and save it.

```
def add(a, b):
    "This function adds two numbers and return the result"
    result = a + b
    return result

def sub(a, b):
    "This function subtract two numbers and return the result"
    result = a - b
    return result

def mul(a, b):
    "This function multiply two numbers and return the result"
    result = a * b
    return result

def div(a, b):
    "This function divide two numbers and return the result"
    result = a / b
    return result
```

Step 2: Create new file p2.py to include the module. Add the following code and save it.

```
import p1 as m
print("Addition=" , m.add(10,20))
print("Subtraction=" ,m.sub(10,20))
print("Multiplication=" ,m.mul(10,20))
print("division=" ,m.div(10,20))
```

Step 3: Execute p2.py file.

```
Addition= 30
Subtraction= -10
Multiplication= 200
division= 0.5
```

4.1.4 Predefined Modules

- A module is a collection of Python objects such as functions, classes, and so on. Python interpreter is bundled with a standard library consisting of large number of built-in modules,
- Built-in modules are generally written in C and bundled with Python interpreter in precompiled form. A built-in module may be a Python script (with .py extension) containing useful utilities.
- A module may contain one or more functions, classes, variables, constants, or any other Python resources.

(I) Numeric and Mathematical Modules:

- This module provides numeric and math-related functions and data types. Following are the modules which are classified as numeric and mathematical modules:
 - (i) numbers (Numeric abstract base classes).
 - (ii) math (Mathematical functions).
 - (iii) cmath (Mathematical functions for complex numbers).
 - (iv) decimal (Decimal fixed point and floating point arithmetic).
 - (v) fractions (Rational numbers).
 - (vi) random (Generate pseudo-random numbers).
 - (vii) statistics (Mathematical statistics functions).
 - The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. The decimal module supports exact representations of decimal numbers, using arbitrary precision arithmetic.
- ##### 1. math and cmath Modules:
- Python provides two mathematical modules namely math and cmath.
 - The math module gives us access to hyperbolic, trigonometric, and logarithmic functions for real numbers and cmath module allows us to work with mathematical functions for complex numbers.

Example 1: For math module.

```
>>> import math
>>> math.ceil(1.001)
2
>>> from math import *
>>> ceil(1.001)
2
>>> floor(1.001)
1
>>> factorial(5)
120
>>> trunc(1.115)
1
>>> sin(90)
0.8939966636005579
>>> cos(60)
-0.9524129804151563
>>> exp(5)
148.4131591025766
>>> log(16)
2.772588722239781
>>> log(16,2)
4.0
>>> log(16,10)
1.2041199826559246
>>> pow(144,0.5)
12.0
>>> sqrt(144)
12.0
>>>
```

-
- The mathematical functions for complex numbers.
-

Example 2: For cmath module.

```
>>> from cmath import *
>>> c=2+2j
>>> exp(c)
(-3.074932320639359+6.71884969742825j)
>>> log(c,2)
(1.5000000000000002+1.1330900354567985j)
>>> sqrt(c)
(1.5537739740300374+0.6435942529055826j)
```

2. Decimal Module:

- Decimal numbers are just the floating-point numbers with fixed decimal points. We can create decimals from integers, strings, floats, or tuples.
 - A Decimal instance can represent any number exactly, round up or down, and apply a limit to the number of significant digits.
-

Example : For decimal module.

```
>>> from decimal import Decimal
>>> Decimal(121)
Decimal('121')
>>> Decimal(0.05)
Decimal('0.050000000000000000277555756156289135105907917022705078125')
>>> Decimal('0.15')
Decimal('0.15')
>>> Decimal('0.012')+Decimal('0.2')
Decimal('0.212')
>>> Decimal(72)/Decimal(7)
Decimal('10.28571428571428571428571429')
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
```

3. Fractions Module:

- A fraction is a number which represents a whole number being divided into multiple parts.
- Python fractions module allows us to manage fractions in our Python programs.

Example: For fractions module.

```
>>> import fractions
>>> for num, decimal in [(3, 2), (2, 5), (30, 4)]:
fract = fractions.Fraction(num, decimal)
print(fract)
3/2
2/5
15/2
```

- It is also possible to convert a decimal into a Fractional number. Let's look at a code snippet:

```
>>> import fractions
>>> for deci in ['0.6', '2.5', '2.3', '4e-1']:
fract = fractions.Fraction(deci)
print(fract)
```

Output:

```
3/5
5/2
23/10
2/5
>>>
```

4. Random Module:

- Sometimes, we want the computer to pick a random number in a given range, pick a random element from a list etc.
- The random module provides functions to perform these types of operations. This function is not accessible directly, so we need to import random module and then we need to call this function using random static object.

Example: For random module.

```
>>> import random
>>> print(random.random())      # It generate a random number in the range
                                (0.0, 1.0)

0.27958089234907935
>>> print(random.randint(10,20)) # It generate a random integer between x
                                and y inclusive

13
```

-
- Python uses library random to generate random numbers.

```
>>> import random
```

Methods:

- (i) **Randrange() Method:** Generates integer between lower to upper argument. By default lower bound is 0.

Example:

```
rno=random.randrange(50) #generates random numbers from 0 to 49
rno=random.randrange(10,20) #generates random numbers from 10 to 20
```

- (ii) **shuffle(list) Method:** Used to shuffle the contents of the list.

Example:

```
import random
color = ['Cyan', 'Magenta', 'Yellow', 'Black']
random.shuffle(Color)
print "Reshuffled color :", Color
#which returns: Reshuffled color : ['Cyan', 'Magenta', 'Yellow', 'Black']
random.shuffle(Color)
print "Reshuffled color :", Color
#which returns: Reshuffled color : ['Yellow', 'Black', 'Magenta', 'Cyan']
```

- (iii) **uniform() Method:** Returns a random floating number between the two specified numbers (both included).

Syntax: random.uniform(x,y)

Where, x is lower limit and y is upper limit of random float and returns random floating point number greater than or equal to x and less than y.

Example:

```
import random
print "uniform no(1,100):", random.uniform(1,100)
# uniform no(1,100) 9.38500684602
```

- (iv) **random() Method:** Tenerates the random numbers from 0 to 1.

Example:

```
>>> from random import random
>>> random() # 0.8407436368337916
>>> random() # 0.3069951291690556
>>> random() # 0.4887997856049814
>>> random() # 0.3534343245685684
```

Each time random generates the different number.

- (v) **randint() Method:** Returns a random number selected element from the specified range.

Example:

```
import random
print random.randint(0,5)    #output either 1,2,3,4 or 5
```

- (vi) **choice() Method:** Returns a random element from the given sequence.

Example:

```
import random
MyChoice = random.choice(['1-Swimming', '2-Badminton',
                          '3-Cricket', '4-Basketball', '5-Hockey'])
print 'My choice is:', MyChoice #returns: My choice is :
                                4-Basketball
```

5. Statistics Module:

- Statistics module provides access to different statistics functions. Example includes mean (average value), median (middle value), mode (most often value), standard deviation (spread of values).

Example: For statistics module.

```
>>> import statistics
>>> statistics.mean([2,5,6,9])           # average
5.5
>>> import statistics
>>> statistics.median([1,2,3,8,9])       # central value
3
>>> statistics.median([1,2,3,7,8,9])
5.0
>>> import statistics
>>> statistics.mode([2,5,3,2,8,3,9,4,2,5,6]) # repeated value
2
>>> import statistics
>>> statistics.stdev([1,1.5,2,2.5,3,3.5,4,4.5,5])
1.3693063937629153
```

(II) Functional Programming Modules:

- These modules provide functions and classes that support a functional programming style and general operations on callable.
- Following are modules which comes under functional programming modules.
 - (i) itertools (functions creating iterators for efficient looping).
 - (ii) functools (higher-order functions and operations on callable objects).
 - (iii) operator (standard operators as functions).

1. itertools Module:

- Python programming itertools module provide us various ways to manipulate the sequence while we are traversing it.
- Python itertools chain() function just accepts multiple iterable and return a single sequence as if all items belongs to that sequence.

Example: For itertools module with chain().

```
>>> from itertools import *
>>> for value in chain([1.3, 2.51, 3.3], ['C++', 'Python', 'Java']):
    print(value)
```

Output:

```
1.3
2.51
3.3
C++
Python
Java
```

-
- Python itertools cycle() function iterate through a sequence upto infinite. This works just like a circular Linked List.
-

Example: For intools module with cycles().

```
>>> from itertools import *
>>> for item in cycle(['C++', 'Python', 'Java']):
    index=index+1
    if index==10:
        break
    print(item)
```

Output:

```
C++
Python
Java
C++
Python
Java
C++
Python
Java
>>>
```

2. functools Module:

- Python functools module provides us various tools which allows and encourages us to write reusable code.
- Python functools partial() functions are used to replicate existing functions with some arguments already passed in. It also creates new version of the function in a well-documented manner.
- Suppose we have a function called multiplier which just multiplies two numbers. Its definition looks like:

```
def multiplier(x, y):
    return x * y
```

- Now, if we want to make some dedicated functions to double or triple a number then we will have to define new functions as:

```
def multiplier(x, y):
    return x * y
def doubleIt(x):
    return multiplier(x, 2)
def tripleIt(x):
    return multiplier(x, 3)
```

- But what happens when we need 1000 such functions? Here, we can use partial functions:

```
from functools import partial
def multiplier(x, y):
    return x * y
double = partial(multiplier, y=2)
triple = partial(multiplier, y=3)
print('Double of 2 is {}'.format(double(5)))
print('Triple of 5 is {}'.format(triple(5)))
```

Output:

```
Double of 5 is 10
Triple of 5 is 15
```

3. Operator Module:

- The operator module supplies functions that are equivalent to Python's operators. These functions are handy in cases where callables must be stored, passed as arguments, or returned as function results.
- Functions supplied by the operator module are listed in following table:

Sr. No.	Function	Signature/Syntax	Behaves Like
1.	abs	abs(a)	abs(a)
2.	add	add(a,b)	a+b
3.	and_	and_(a,b)	a&b
4.	div	div(a,b)	a/b
5.	eq	eq(a,b)	a==b
6.	gt	gt(a,b)	a>b
7.	invert, inv	invert(a), inv(a)	~a
8.	le	le(a,b)	a<=b
9.	lshift	lshift(a,b)	a<<b
10.	lt	lt(a,b)	a<b
11.	mod	mod(a,b)	a%b

contd. ...

12.	mul	mul(a,b)	a*b
13.	ne	ne(a,b)	a!=b
14.	neg	neg(a)	-a
15.	not_	not_(a)	not a
16.	or_	or_(a,b)	a b
17.	pos	pos(a)	+a
18.	repeat	repeat(a,b)	a*b
19.	rshift	rshift(a,b)	a>>b
20.	xor_	xor(a,b)	a^b

4. Time Module:

- There is a popular time module available in Python which provides functions for working with times and for converting between representations.
- Python has a module named time to handle time-related tasks. To use functions defined in the module, we need to import the module first like `import time`.
- Python “time” module has following functions:

(i) **time()**: This function returns the current time instant, a floating-point number of seconds since the epoch. Example. The function `time()` returns the current system time in ticks since 00:00:00 hrs January 1, 1970 (epoch).
`import time; # This is required to include time module.`

`ticks = time.time()`

`print "Number of ticks since 12:00am, January 1, 1970:", ticks`

`#Output Number of ticks since 12:00am, January 1, 1970: 7186862.73399`

(ii) **gmtime(sec)**: This function returns a structure with 9 values each representing a time attribute in sequence. It converts seconds into time attributes(days, years, months etc.) till specified seconds from epoch. If no seconds are mentioned, time is calculated till present.

(iii) **asctime(“time”)**: This function takes a time attributed string produced by `gmtime()` and returns a 24 character string denoting time.

(iv) **ctime(sec)**: This function returns a 24 character time string but takes seconds as argument and computes time till mentioned seconds. If no argument is passed, time is calculated till present.

(v) **sleep(sec)**: This method is used to halt the program execution for the time specified in the arguments.

(vi) **tzset()**: Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

(vii) **clock()**: Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of `time.clock` is more useful than that of `time.time()`.

Datetime Module:

- A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

Example: Import the datetime module and display the current date:

```
import datetime
x = datetime.datetime.now()
print(x)
```

Output:

```
2021-07-24 12:41:12.701464
```

- The date contains year, month, day, hour, minute, second, and microsecond. The datetime module has many methods to return information about the date object.

Example: Return the year and name of weekday:

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

Output:

```
2021
Saturday
```

Creating Date Objects:

- To create a date, we can use the datetime() class (constructor) of the datetime module.
- The datetime() class requires three parameters to create a date: year, month, day.

Example: Create a date object:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

Output:

```
2021-07-24 00:00:00
```

4.2 PACKAGES

- Suppose we have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all as they have similar names or functionality.
- It is necessary to group and organize them by some mean which can be achieved by packages.
- A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages and so on.
- Packages allow for a hierarchical structuring of the module namespace using dot notation. Packages are a way of structuring many packages and modules which help in a well-organized hierarchy of data set, making the directories and modules easy to access.
- A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional `__init__.py` file (For example: Phone/`__init__.py`).

4.2.1 Creating and Importing Packages

- Creating a package is quite easy, since it makes use of the operating system's inherent hierarchical file structure as shown in Fig. 4.5.
- Here, there is a directory named mypkg that contains two modules, p1.py and p2.py. The contents of the modules are:

p1.py

```
def m1():  
    print("first module")
```

p2.py

```
def m2():  
    print("second module")
```

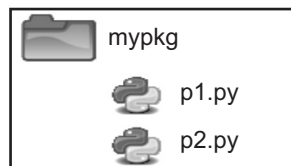


Fig. 4.1

- Here, mypkg a folder /directory which consist of p1.py and p2.py. We can refer these two modules with dot notation (mypkg.p1, mypkg.p2) and import them with the one of the following syntaxes:

Syntax 1: `import <module_name>[, <module_name> ...]`

Example:

```
>>> import mypkg.p1, mypkg.p2  
>>> mypkg.p1.m1()  
first module  
>>> p1.m1()
```

Syntax 2: `from <module_name> import <name(s)>`

Example:

```
>>> from mypkg.p1 import m1  
>>> m1()  
first module  
>>>
```

Syntax 3: `from <module_name> import <name> as <alt_name>`

Example:

```
>>> from mypkg.p1 import m1 as function  
>>> function()  
first module  
>>>
```

Syntax 4: `from <package_name> import <modules_name>[, <module_name> ...]`

Example:

```
>>> from mypkg import p1,p2
>>> p1.m1()
first module
>>> p2.m2()
second module
>>>
```

4.2.2 Standard Packages

- NumPy and SciPy are the standards packages used by Python programming.
- NumPy enriches the programming language Python with powerful data structures, implementing multi-dimensional arrays and matrices.
- SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy needs Numpy, as it is based on the data structures of Numpy and furthermore its basic creation and manipulation functions.
- It extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.
- Both NumPy and SciPy are not part of a basic Python installation. They have to be installed after the Python installation. NumPy has to be installed before installing SciPy.

Math:

- Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions, representation functions, logarithmic functions and angle conversion functions.
 - Two mathematical constants are also defined in math module.
 - The **Pie (π)** is a well-known mathematical constant, which is defined as the ratio of the circumference to the diameter of a circle and its value is 3.141592653589793.
-

```
>>> import math
>>> math.pi
3.141592653589793
>>>
```

- Another well-known mathematical constant defined in the math module is **e**. It is called **Euler's number** and it is a base of the natural logarithm. Its value is 2.718281828459045.
-

```
>>> import math
>>> math.e
2.718281828459045
>>>
```

NumPy:

- NumPy is the fundamental package for scientific computing with Python. NumPy stands for "Numerical Python". It provides a high-performance multidimensional array object, and tools for working with these arrays.
- An array is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers and represented by a single variable. NumPy's array class is called ndarray. It is also known by the alias array.
- In NumPy arrays, the individual data items are called elements. All elements of an array should be of the same type. Arrays can be made up of any number of dimensions.
- In NumPy, dimensions are called axes. Each dimension of an array has a length which is the total number of elements in that direction.
- The size of an array is the total number of elements contained in an array in all the dimension. The size of NumPy arrays are fixed; once created it cannot be changed again.
- Numpy arrays are great alternatives to Python Lists. Some of the key advantages of Numpy arrays are that they are fast, easy to work with, and give users the opportunity to perform calculations across entire arrays.

SciPy:

- SciPy is a library that uses NumPy for more mathematical functions. SciPy uses NumPy arrays as the basic data structure, and comes with modules for various commonly used tasks in scientific programming, including linear algebra, integration (calculus), ordinary differential equation solving, and signal processing.

Matplotlib:

- The matplotlib.pyplot is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.
- There are various plots which can be created using python matplotlib like bar graph, histogram, scatter plot, area plot, pie plot.

Pandas:

- Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures.
- It is built on the Numpy package and its key data structure is called the DataFrame. DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.

4.2.3 User Defined Packages

- We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily.
- In the same way, a package in Python takes the concept of the modular approach to next logical level.

- As we know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules.
- Physically, a package is actually a folder containing one or more module files. Let's create a package named MyPkg, using the following steps:

Step 1: Create a folder MyPkg on,

C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\

Create modules Message.py and Mathematics.py with following code:

Message.py

```
def SayHello(name):  
    print("Hello " + name)  
    return
```

Mathematics.py

```
def sum(x,y):  
    return x+y  
def average(x,y):  
    return (x+y)/2  
def power(x,y):  
    return x**y
```

Step 2: Create an empty `__init__.py` file in the MyPkg folder. The package folder contains a special file called `__init__.py`, which stores the package's content. It serves two purposes:

- (i) The Python interpreter recognizes a folder as the package if it contains `__init__.py` file.
- (ii) `__init__.py` exposes specified resources from its modules to be imported.

An empty `__init__.py` file makes all functions from above modules available when this package is imported. Note that `__init__.py` is essential for the folder to be recognized by Python as a package. We can optionally define functions from individual modules to be made available.

Step 3: Create P1.py file in MyPkg folder and write following code:

```
from MyPkg import Mathematics  
from MyPkg import Message  
greet.SayHello("Meenakshi")  
x=functions.power(3,2)  
print("power(3,2) : ", x)
```

Output:

```
Hello Meenakshi  
power(3,2) : 9
```

Using `__init__.py` File:

- The `__init__.py` file is normally kept empty. However, it can also be used to choose specific functions from modules in the package folder and make them available for import. Modify `__init__.py` as below:

`__init__.py`

```
from .Mathematics import average, power
from .Message import SayHello
```

- The specified functions can now be imported in the interpreter session or another executable script.
- Create `test.py` in the `MyPkg` folder and write following code:

test.py from `MyPkg` import `power`, `average`, `SayHello`

```
SayHello()
x=power(3,2)
print("power(3,2) : ", x)
```

- Note that functions `power()` and `SayHello()` are imported from the package and not from their respective modules, as done earlier. The output of above script is:

```
Hello world
power(3,2) : 9
```

4.2.4 Examples

- We have included a `__init__.py`, file inside a directory to tell Python that the current directory is a package.
- Whenever we want to create a package, then we have to include `__init__.py` file in the directory. we can write code inside it.
- Let's create a simple package that has the following structure:

Package (university)

- `__init__.py`
- `student.py`
- `faculty.py`

`student.py`

```
class Student:
```

```
    def __init__(self, student):
        self.name = student['name']
        self.gender = student['gender']
        self.year = student['year']
    def get_student_details(self):
        return f"Name: {self.name}\nGender: {self.gender}\nYear: {self.year}"
```

```
# faculty.py
class Faculty:
    def __init__(self, faculty):
        self.name = faculty['name']
        self.subject = faculty['subject']
    def get_faculty_details(self):
        return f"Name: {self.name}\nSubject: {self.subject}"
```

- We have the above in the student.py and faculty.py files. Let's create another file to access those classed inside it. Now, inside the package directory create a file named testing.py and include the following code:
-

```
# testing.py
# importing the Student and Faculty classes from respective files
from student import Student
from faculty import Faculty
# creating dicts for student and faculty
student_dict = {'name' : 'John', 'gender': 'Male', 'year': '3'}
faculty_dict = {'name': 'Emma', 'subject': 'Programming'}
# creating instances of the Student and Faculty classes
student = Student(student_dict)
faculty = Faculty(faculty_dict)
# getting and printing the student and faculty details
print(student.get_student_details())
print()
print(faculty.get_faculty_details())
```

Output:

```
Name: John
Gender: Male
Year: 3
Name: Emma
Subject: Programming
```

- We have seen how to create and to access a package in Python. And this is a simple package. There might be plenty of sub-packages and files inside a package. Let's see how to access subpackage modules.
-

- Create a directory with the following structure:
 - package (university)
 - `__init__.py`
 - Subpackage (student)
 - `__init__.py`
 - `main.py`
 - ...
 - `testing.py`
 - Copy the above student code and place it here. Now, let's see how to access it in the `testing.py` file. Add the following in the `testing.py` file.

Example:

```
# testing.py
from student.main import Student
# creating dicts for student
student_dict = {'name' : 'John', 'gender': 'Male', 'year': '3'}
# creating instances of the Student class
student = Student(student_dict)
# getting and printing the student details
print(student.get_student_details())
```

Output: After running `testing.py`.

```
Name: John
Gender: Male
Year: 3
```

Example: For NumPy with array object.

```
>>> import numpy as np
>>> a=np.array([1,2,3])           # one dimensional array
>>> print(a)
[1 2 3]
>>> arr=np.array([[1,2,3],[4,5,6]]) # two dimensional array
>>> print(arr)
[[1 2 3]
 [4 5 6]]
>>> type(arr)
<class 'numpy.ndarray'>
>>> print("No. of dimension: ", arr.ndim)
```

```
No. of dimension: 2
>>> print("Shape of array: ", arr.shape)
Shape of array: (2, 3)
>> >print("size of array: ", arr.size)
size of array: 6
>>> print("Type of elements in array: ", arr.dtype)
Type of elements in array: int32
>>> print("No of bytes:", arr.nbytes)
No of bytes: 24
```

Example 1: Using linalg sub package of SciPy.

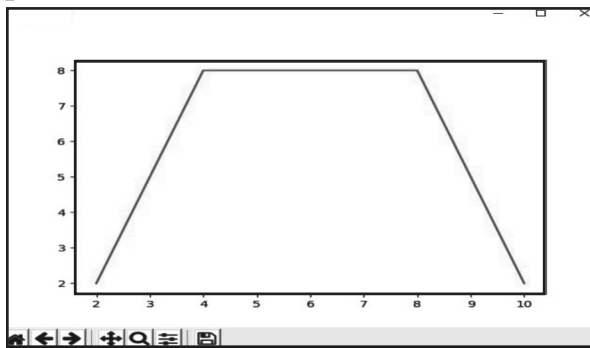
```
>>> import numpy as np
>>> from scipy import linalg
>>> a = np.array([[1., 2.], [3., 4.]])
>>> linalg.inv(a) # find inverse of array
array([[ -2. ,  1. ],
       [ 1.5, -0.5]])
>>>
```

Example 2: Using linalg sub package of SciPy.

```
>>> import numpy as np
>>> from scipy import linalg
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(a) # find determinant of array
0.0
>>>
```

Example 3: For line plot.

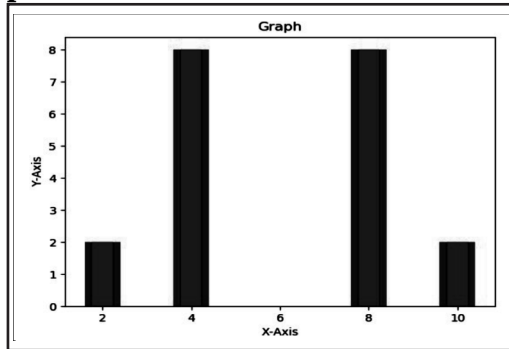
```
>>> from matplotlib import pyplot as plt
>>> x=[2,4,8,10]
>>> y=[2,8,8,2]
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x02E69B70>]
>>> plt.show()
```

Output:

Example 4: For bar graph.

```
>>> from matplotlib import pyplot as plt
>>> x=[2,4,8,10]
>>> y=[2,8,8,2]
>>> plt.xlabel('X-Axis')
Text(0.5, 0, 'X-Axis')
>>> plt.ylabel('Y-Axis')
Text(0, 0.5, 'Y-Axis')
>>> plt.title('Graph')
Text(0.5, 1.0, 'Graph')
>>> plt.bar(x,y,label="Graph",color='r',width=.5)
<BarContainer object of 4 artists>
>>> plt.show()
```

Output:



Example 5: Using series data structure of Panda.

```
>>> import pandas as pd
>>> import numpy as np
>>> numpy_arr = array([2, 4, 6, 8, 10, 20])
>>> si = pd.Series(arr)
>>> print(si)
0      2
1      4
2      6
3      8
4     10
5     20
dtype: int32
```

Example 6: Using series data structure of Panda.

```
>>> import pandas as pd
>>> data=[10,20,30,40,50]
```

```
>>> index=['a','b','c','d','e']
>>> si=pd.Series(data,index)
>>> si
a    10
b    20
c    30
d    40
e    50
dtype: int64
>>>
```

4.3 WORKING WITH FILES

- File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since, Random Access Memory (RAM) is volatile which loses its data when computer is turned off. We use files for future use of the data permanently.
- To read from or write to a file, we must first open it when we are done with it, we should close it to free up the resources it holds.
- Python has several functions for file handling to perform operations like creating, reading, updating, and deleting files.
- Python has extensive support for manipulating files and directories. These manipulations can be carried out by operating system commands using the built-in Python functions for file and directory.
- Files are divided into following two categories:
 1. **Text Files:** Text files are simple texts in human readable format. A text file is structured as sequence of lines of text.
 2. **BinaryFiles:** Binary files have binary data (0s and 1s) which is understood by the computer.
- When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- Hence, in Python a file operation takes place in the following order:
 - Open a file.
 - Read or write (perform operation).
 - Close the file.

4.3.1 Creating Files

- Python provides basic functions and methods necessary to manipulate files. The file manipulation is performed using a file object.

- To create a new file in Python, use the `open()` method, with one of the following parameters.

"x" - create - will create a file, returns an error if the file exist.

"a" - append - will create a file if the specified file does not exist.

"w" - write - will create a file if the specified file does not exist.

Example: Create a file called "xyz.txt":

```
f = open("xyz.txt", "x")          # Here a new empty file is created
```

Example: Create a new file if it does not exist:

```
f = open("xyz.txt", "w")
```

Write to an Existing File

- To write to an existing file, you must add a parameter to the `open()` function

"a" - Append - will append to the end of the file.

"w" - Write - will overwrite any existing content.

Example: Open the file "demofile.txt" and append content to the file:

```
f = open("abc.txt", "a")
f.write("Now you can append line to a file")
```

Example:

```
file = open("testfile.txt", "w")
file.write("Hello World")
file.write("This is our new text file")
file.write("and this is another line.")
file.write("Why? Because we can.")
```

- There are two types of files that can be handled in python, normal text files and binary files.
 - Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in Python by default.
 - Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

Accessing File Contents using Standard Library Functions:

- Once, a file is opened and we will have one file object, we can get various information related to that file. Here, is a list of all attributes related to file object:

Sr. No.	Attribute	Description	Example
1.	<code>file.closed</code>	Returns true if file is closed, false otherwise.	<pre>>>> f=open("abc.txt") >>> f.closed False</pre>

contd. ...

2.	<code>file.mode</code>	Returns access mode with which file was opened.	<pre>>>> f=open("abc.txt","w") >>> f.mode 'w'</pre>
3.	<code>file.name</code>	Returns name of the file.	<pre>>>> f=open("abc.txt","w") >>>f.name 'abc.txt'</pre>
4.	<code>file.encoding</code>	The encoding of the file.	<pre>>>> f=open("abc.txt","w") >>>f.encoding cp1252</pre>

Example: For file object attribute.

```
f=open("sample.txt","r")
print(f.name)
print(f.mode)
print(f.encoding)
f.close()
print(f.closed)
```

Output:

```
sample.txt
r
cp1252
True
```

4.3.2 Operations on Files

- Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- This section describes the file operations of all types of files include opening a file, reading from a file, writing to a file, and closing a file.

1. **open() Function:**

- We have to open the file before you can read or write a file, using Python's built-in `open()` function.
- The `open()` function creates a file object which is called "handle". This would be utilized to call other support methods associated with it.

Syntax: `file_object = open(filename [, access_mode][, buffering])`

where,

- **filename:** The `file_name` argument is a string value that contains the name of the file that you want to access.

- **access_mode:** The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).
- **buffering:** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).
- The file can be open in Text mode or Binary mode.
 - (i) The **text mode** returns strings while reading from the file. The default is reading in text mode.
 - (ii) The **binary mode** returns bytes and this is the mode to be used when dealing with non-text files like image or executable files.
- The text and binary modes are used in conjunction with the r, w, and a modes. The list of all the modes used in Python are given in following table:

Sr. No.	Mode	Description
1.	r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2.	rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3.	r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4.	rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5.	w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6.	wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7.	w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8.	wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

contd. ...

9.	a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10.	ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11.	a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12.	ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
13.	t	Opens in text mode (default).
14.	b	Opens in binary mode.
15.	+	Opens a file for updating (reading and writing).

Example: For different modes of opening a file.

```
f = open("test.txt")           # opening in r mode (reading only)
f = open("test.txt", 'w')      # opens in w mode (writing mode)
f = open("img.bmp", 'rb+')     # read and write in binary mode
```

2. close() Function:

- When the all operations related to the file are complete, then we need to close the file. The close() function of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() function to close a file.

Syntax: fileObject.close()

Example:

```
# Open a file
f = open("test.txt", "w")
print ("Name of the file: ", f.name)
# Close opened file
f.close()
```

Output:

```
Name of the file: test.txt
```

3. Reading and Writing Files:

- The methods read() and write() are used to read and write files.

Writing Data to File:

- In order to write into a file in Python, we need to open it in write 'w', append 'a' or exclusive creation 'x' mode.
- We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased. There are following two ways to write in a file:

Using write() Function:

- Writing a string or sequence of bytes (for binary files) is done using write() method. This method returns the number of characters written to the file.

Example: Inserts the string str1 in a single line in the text file. The write() method does not add a newline character ('\n') to the end of the string.

```
File_object.write(str1)
```

Example: For write(string) method.

```
f=open("sample.txt")
print("**content of file1**")
print(f.read())
f=open("sample.txt","w")
f.write("first line\n")
f.write("second line\n")
f.write("third line\n")
f.close()
f=open("sample.txt","r")
print("**content of file1**")
print(f.read())
```

Output:

```
**content of file1**
Hello,I am There
**content of file1**
first line
second line
third line
```

Using writelines() Function:

- It writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value.

- For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

```
File_object.writelines(L)          for L = [str1, str2, str3]
```

Example: For writelines() method.

```
fruits=["Orange\n", "Banana\n", "Apple\n"]
f=open("sample.txt", mode="w+", encoding="utf-8")
f.writelines(fruits)
f.close()
f=open("sample.txt", "r")
print(f.read())
```

Output:

```
Orange
Banana
Apple
```

Reading Data From File:

- To read a file in python, we must open the file in reading mode (r or r+).
- There are following three ways to read data from a text file:
 1. **read():** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

Syntax: File_object.read([n])

Example: for read() method.

```
f=open("sample.txt", "r")
print(f.read(5))          # read first 5 data
print(f.read(5))          # read next five data
print(f.read())           # read rest of the file
print(f.read())
```

Output:

```
first
line
second line
third line
```

2. **readline():** Reads a line of the file and returns in form of a string. For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

Syntax: File_object.readline([n])

Example: For readline() method.

```
f=open("sample.txt","r")
print(f.readline())    # read first line followed by\n
print(f.readline(3))
print(f.readline(5))
print(f.readline())
print(f.readline())
```

Output:

```
first line
sec
ond l
ine
third line
```

3. readlines(): Reads all the lines and return them as each line a string element in a list.

Syntax: file_object.readlines()

Example: For readlines() method.

```
f=open("sample.txt","r")
print(f.readlines())
```

Output:

```
['first line\n', 'second line\n', 'third line\n']
```

File Related Standard Functions:

Sr. No.	Function	Description	Example
1.	file.close()	Close the file. We need to reopen it for further access.	f.close()
2.	file.flush()	Flush the internal buffer.	f.flush() print(f.read()) f.close()
3.	file.fileno()	Returns an integer file descriptor.	print(f.fileno()) f.close()
4.	file.isatty()	It returns true if file has a <tty> attached to it.	print(f.isatty()) f.close()
5.	file.next()	Returns the next line from the last offset.	try: while f.next(): print(f.next()) except: f.close()

contd. ...

6.	<code>file.read()</code>	This function reads the entire file and returns a string.	<code>lines =f.read() f.write(lines) f.close()</code>
7.	<code>file.read(size)</code>	Reads the given number of bytes. It may read less if EOF is hit.	<code>text =f.read(10) print(text) f.close()</code>
8.	<code>file.readline()</code>	Reads a single line and returns it as a string.	<code>text =f.readline() print(text) f.close()</code>
9.	<code>file.readline(size)</code>	It will read an entire line (trailing with a new line char) from the file.	<code>text =f.readline(20) print(text) f.close()</code>
10.	<code>file.readlines()</code>	Reads the content of a file line by line and returns them as a list of strings.	<code>lines =f.readlines() f.writelines(lines) f.close()</code>
11.	<code>file.readlines (size_hint)</code>	It calls the <code>readline()</code> to read until EOF. It returns a list of lines read from the file. If you pass <code><size_hint></code> , then it reads lines equalling the <code><size_hint></code> bytes.	<code>text =f.readlines(25) print(text) f.close()</code>
12.	<code>file.seek(offset [, from])</code>	Sets the file's current position.	<code>position =f.seek(0,0); print(position) f.close()</code>
13.	<code>file.tell()</code>	Returns the file's current position.	<code>lines =f.read(10) #tell() print(f.tell()) f.close()</code>
14.	<code>file.truncate(size)</code>	Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.	<code>f.truncate(10) f.close()</code>
15.	<code>file.write(string)</code>	It writes a string to the file. And it doesn't return any value.	<code>line ='Welcome Geeks\n' f.write(line) f.close()</code>
16.	<code>file.writelines (sequence)</code>	Writes a sequence of strings to the file. The sequence is possibly an iterable object producing strings, typically a list of strings.	<code>lines =f.readlines() #writelines() f.writelines(lines) f.close()</code>

Example: For file object methods.

```
f = open("sample.txt","w+")
f.write("Line one\nLine two\nLine three")
f.seek(0)
print(f.read())
print("Is readable:",f.readable())
print("Is writeable:",f.writable())
print("File no:",f.fileno())
print("Is connected to tty-like device:",f.isatty())
f.truncate(5)
f.flush()
f.close()
```

Output:

```
Line one
Line two
Line three
Is readable: True
Is writeable: True
File no: 3
Is connected to tty-like device: False
```

Handling Files through OS Module:

- The OS module of Python allows us to perform Operating System (OS) dependent operations such as making a folder, listing contents of a folder, know about a process, end a process etc.
- It has methods to view environment variables of the operating system on which Python is working on and many more.

Directory related Standard Functions:

Sr. No.	Function	Description	Example
1.	os.getcwd()	Show current working directory.	import os os.getcwd()
2.	os.path.getsize()	Show file size in bytes of file passed in parameter.	size =os.path.getsize("sample.txt")
3.	os.path.isfile()	Is passed parameter a file.	print(os.path.isfile ("sample.txt"))

contd. ...

4.	<code>os.path.isdir()</code>	Is passed parameter a folder.	<code>print(os.path.isdir("sample.txt"))</code>
5.	<code>os.listdir()</code>	Returns a list of all files and folders of present working directory.	<code>print("***Contents of Present working directory ***\n ",os.listdir())</code>
6.	<code>os.listdir(path)</code>	Return a list containing the names of the entries in the directory given by path.	<code>print("***Contents of given directory***\n", os.listdir("testdir"))</code>
7.	<code>os.rename(current,new)</code>	Rename a file.	<code>os.rename("sample.txt", "sample1.txt")</code>
8.	<code>os.remove(file_name)</code>	Delete a file.	<code>os.remove("sample.txt")</code>
9.	<code>os.mkdir()</code>	Creates a single subdirectory.	<code>os.mkdir("testdir")</code>
10.	<code>os.chdir(path)</code>	Change the current working directory to path.	<code>os.chdir("d:\IT")</code>

Example: For handling files through OS module.

```
import os
os.getcwd()
print("***Contents of Present working directory***\n ", os.listdir())
print(os.path.isfile("sample.txt"))
print(os.path.isdir("sample.txt"))
***Contents of Present working directory***
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs',
'LICENSE.txt', 'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe',
'python3.dll', 'python37.dll', 'pythonw.exe', 'sample.txt', 'Scripts',
'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll', '_ _pycache_ _']
True
False
```

4.3.3 File Object Attributes

- A file object allows us to use, access and manipulate all the user accessible files.
- One can read and write any such files. Once a file is opened and he/she have one file object, he/she can get various information related to that file.
- Following is a list of all the attributes related to a file object:

Table 4.1: List of Attributes of file object

Sr. No.	Attribute and Description
1.	file.closed: Returns true if file is closed, false otherwise.
2.	file.mode: Returns access mode with which file was opened.
3.	file.name: Returns name of the file.
4.	file.softspace: Returns false if space explicitly required with print, true otherwise.

Example:

```
# Open a file
f = open("test.txt", "wb")
print ("Name of the file: ", f.name)
print ("Closed or not: ", f.closed)
print ("Opening mode: ", f.mode)
print("Soft space flag: ", f.softspace)
f.close()
```

Output:

```
Name of the file: test.txt
Closed or not: False
Opening mode: wb
Softspace flag: 0
```

4.3.4 File Positions

- To move directly to the position we use two built-in methods tell() and seek(). The tell() method tells us the current position within the file. The seek() method changes the current file position.
- We can change the current file cursor (position) using the seek() method. Similarly, the tell() method returns the current position (in number of bytes) of file cursor/pointer.
- To change the file object's position use f.seek(offset, reference_point). The position is computed from adding offset to a reference point.
- The reference_point can be omitted and defaults to 0, using the beginning of the file as the reference point.

- The reference points are 0 (the beginning of the file and is default), 1 (the current position of file) and 2 (the end of the file).
- The `f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- In other words, the `tell()` is used to find the current position of the file pointer in the file while the `seek()` used to move the file pointer to the particular position.

Example 1: For file position.

```
f=open("sample.txt","r")
print(f.tell())
print(f.read())
print(f.tell())
print(f.read())          # print blank line
print(f.seek(0))
print(f.read())
```

Output:

```
0
first line
second line
third line

37

0
first line
second line
third line
>>>
```

Example 2:

```
fruits=["Orange\n","Banana\n","Apple\n"]
f=open("sample.txt",mode="w+",encoding="utf-8")
for fru in fruits:
    f.write(fru)
print("Tell the byte at which the file cursor is:",f.tell())
f.seek(0)
for line in f:
    print(line)
```

Output:

```
Tell the byte at which the file cursor is: 23
Orange
Banana
Apple
```

Example 3:

```
# Open a file
f = open("test.txt", "r+")
str = f.read(18)
print ("Read String is: ", str)
# Check current position
position = f.tell()
print ("Current file position: ", position)
# Reposition pointer at the beginning once again
position = f.seek(0, 0)
str = f.read(18)
print ("Again read String is: ", str)
# Close opened file
f.close()
```

Output:

```
Read String is: Python programming
Current file position: 18
Again read String is: Python programming
```

4.3.5 Listing Files in a Directory

- Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.
- To use this module, we need to import it first and then we can call any related functions.
- The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the shutil functions, we will first need to use import shutil.
- The os.listdir() is used to get a list of entries in a directory. Pass in the directory for which you need the entries; use a "." for the current directory of the process.

Syntax: os.listdir(path)

where, path is the directory, which needs to be explored.

Return Value: This method returns a list containing the names of the files, directories in the directory given by path. Path may be either of type str or type bytes.

Example:

```
import os, sys
#Open a file
path = "d:\\tybsc\\"
```

```
dirs = os.listdir( path )

# This would print all the files and directories

for file in dirs:

    print (file)
```

Output: We will get list of all files in tybsc directory.

4.3.6 Testing File Types

- The `os.path` module includes functions that are useful to perform path-related operations.
- We can access the `os.path` module from the `os` module. The methods available in this module are useful for operations such as file path inquiries and retrieving information about files and directories.
- The functions `isfile()`, `isdir()`, and `islink()` in the `os.path` module are used to test if a string reflects the name of a regular file, a directory, or a link.
 1. `os.path.isfile(path)`: Return True if path is an existing regular file.
 2. `os.path.isdir(path)`: Returns True if path is an existing directory.
 3. `os.path.islink(path)`: Returns True if path refers to a directory entry that is a symbolic link.

Example:

```
import os.path

FILENAMES = [

    file 1

    os.path.dirname( file ), 1/1,

    './broken link',

]

for file in FILENAMES:

    print('File          : (!r}'.format(file))

    print('Absolute      : ', os.path.isabs(file))

    print('Is File?       : ', os.path.isfile(file))

    print('Is Dir?        : ', os.path.isdir(file))

    print('Is Link?        : ', os.path.islink(file))

    print('Mountpoint?    : ', os.path.ismount(file))

    print('Exists?         : ', os.path.exists(file))

    print('Link Exists?    : ', os.path.lexists(file))

    print()
```

Output:

```
File           : '/'
Absolute       : True
Is File?       : False
Is Dir?        : True
Is Link?       : False
Mountpoint?    : True
Exists?        : True
Link Exists?   : True
File           : './broken_link'
Absolute       : False
Is File?       : False
Is Dir?        : False
Is Link?       : True
Mountpoint?    : False
Exists?        : False
Link Exists?   : True
```

4.3.7 Removing Files and Directories

- We can use the `remove()` method to delete files by supplying the name of the file to be deleted as the argument.

Syntax: `os.remove(file_name)`

Example:

```
import os
# Delete file test1.txt
os.remove("text1.txt")
```

Example:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

rmdir() Method:

- The `rmdir()` method deletes or removes the directory, which is passed as an argument in the method.

- Before removing a directory, all the contents in it should be removed.

Syntax: `os.rmdir('dirname')`

Example:

```
import os

# This would remove "/tmp/test" directory.
os.rmdir( "/tmp/test" )
```

Delete directory and contents recursively:

- Use `shutil.rmtree()` function for deleting contents and directory recursively.

Example: `import shutil`

```
shutil.rmtree("/home/ismail/cache")
```

4.3.8 Copying and Renaming Files

- Python provides in-built functions for easily copying files using the Operating System Shell utilities. The `shutil` module offers a number of high-level operations on files and collections of files.
- Python `rename()` file is a method used to rename a file or a directory in Python programming.

Copying Files and Folders:

- Following are the ways through which Python copy a file.

1. Using Python OS Module:

- There are two ways to copy a file in Python- the `popen()` method and the `system()` method.

(i) `popen()` Method:

- This method creates a pipe to or from the command. It returns an open file object which connects to a pipe. We can use it for reading or writing according to the file open mode i.e. 'r' (default) or 'w'.

```
os.popen(command[, mode[, bufsize]])
```

where, **mode:** It can be 'r' (default) or 'w'.

bufsize: If buffering value is 0, then no buffering will occur. If it is set to 1, then line buffering will take place while accessing the file. If you provide a value greater than 1, then buffering will occur with the specified buffer size. However, for a negative value, the system will assume the default buffer size. This method returns an open file object connected to the pipe.

For Windows OS:

```
import os

os.popen('copy 1.txt.py 2.txt.py')
```

For Linux OS:

```
import os
os.popen('cp 1.txt.py 2.txt.py')
```

(ii) system() Method:

- It is the most common way of running any system command. With the **system()** method, you can call any command in a subshell. Internally, this method will call the standard C library function.
- The system() method returns the exit status of the command.

For Windows OS.

```
import os
os.system('copy 1.txt.py 2.txt.py')
```

For Linux OS.

```
import os
os.system('cp 1.txt.py 2.txt.py')
```

2. Using Python Threading Library:

- We can borrow the Thread module from the threading library to Python copy a file. This does the copying in an async manner.
- While using this method, please make sure to employ locking to avoid deadlocks. We may face it if the application is using multiple threads reading/writing a file.

```
import shutil
from threading import Thread
src="1.txt.py"
dst="3.txt.py"
Thread(target=shutil.copy, args=[src, dst]).start()
```

3. Using Python Subprocess Module:

- The subprocess module lets us work with child processes- launch them, attach to their pipes for input, output, and error, and retrieve return values.
- From this module, we can use the methods call() and check_output(), for Python copy a file.

Subprocess's call() Method to Copy a File in Python:

- The subprocess module gives a simple interface to work with child processes. It enables us to launch subprocesses, attach to their input/output/error pipes, and retrieve the return values.
- The subprocess module aims to replace the legacy modules and functions like – os.system, os.spawn*, os.popen*, popen2.*.

- It exposes a `call()` method to invoke system commands to execute user tasks.

```
import subprocess
src="1.txt.py"
dst="2.txt.py"
cmd='copy "%s" "%s"' % (src, dst)
status = subprocess.call(cmd, shell=True)
if status != 0:
    if status < 0:
        print("Killed by signal", status)
    else:
        print("Command failed with return code - ", status)
    else:
        print('Execution of %s passed!\n' % cmd)
```

Subprocess's `check_output()` Method to Copy a File in Python:

- With subprocess's `check_output()` method, you can run an external command or a program and capture its output. It also supports pipes.

```
import os, subprocess
src=os.path.realpath(os.getcwd()+"https://cdn.techbeamers.com/1.txt.py")
dst=os.path.realpath(os.getcwd()+"https://cdn.techbeamers.com/2.txt.py")
cmd='copy "%s" "%s"' % (src, dst)
status = subprocess.check_output(['copy', src, dst], shell=True)
print("status: ", status.decode('utf-8'))
```

4. Using Python Shutil Module:

- The `shutil` module provides functions for copying files, as well as entire folders.
- Calling `shutil.copy(source, destination)` will copy the file at the path source to the folder at the path destination (both source and destination are strings).
- If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.
- There are four methods we will consider in this module to help Python copy a file- `copyfile()`, `copy()`, `copy2()`, and `copyfileobj()`.
- This method copies the content of the source to the destination only if the target is writable. If we don't have the right permissions, then it will raise an `IOError`.

(i) `shutil copyfile()` Method:

- This method copies the content of the source to the destination only if the target is writable. If we don't have the right permissions, then it will raise an `IOError`.
- It works by opening the input file for reading while ignoring its file type. Next, it doesn't treat special files any differently and won't copy them as new special files.

- The `copyfile()` method makes use of lower-level function `copyfileobj()` below. It takes file names as arguments, opens them and passes file handles to `copyfileobj()`.
- In this method, one optional third argument can use to specify the buffer length. It'll then open the file for reading in chunks of the specified buffer size. However, the default behavior is to read the entire file in one go.

Syntax: `copyfile(source_file, destination_file)`

Example: Sample code for copy file.

```
from shutil import copyfile
from sys import exit

source = input("Enter source file with full path: ")
target = input("Enter target file with full path: ")
# adding exception handling
try:
    copyfile(source, target)
    except IOError as e:
        print("Unable to copy file. %s" % e)
    exit(1)
except:
    print("Unexpected error:", sys.exc_info())
    exit(1)
    print("\nFile copy done!\n")
while True:
    print("Do you like to print the file ? (y/n): ")
    check = input()
    if check == 'n':
        break
    elif check == 'y':
        file = open(target, "r")
        print("\nHere follows the file content:\n")
        print(file.read())
        file.close()
        print()
        break
    else:
        continue
```

(ii) copy() Method:

- The copy() method is like the "cp" command in Unix. It means if the target is a folder, then it'll create a new file inside it with the same name (basename) as the source file.

Syntax: shutil.copy(source, destination)

- Also, this method will sync the permissions of the target file with the source after copying its content. It too throws the SameFileError if you are copying the same file.

```
import os
import shutil
#store the file path in the variable 'source' and 'target'
source = 'current/test/test.py'
target = '/prod/new'
assert not os.path.isabs(source)
target = os.path.join(target, os.path.dirname(source))
# create the folders if not already exists
os.makedirs(target)
# adding exception handling
try:
    shutil.copy(source, target)
except IOError as e:
    print("Unable to copy file. %s" % e)
except:
    print("Unexpected error:", sys.exc_info())
```

(iii) Copyfileobj() Method:

- This method copies the file to a target path or file object. If the target is a file object, then you need to close it explicitly after the calling the Copyfileobj().
- It assumes an optional argument (the buffer size) which we can use to supply the buffer length. It is the number of bytes kept in memory during the copy process. The default size that system uses is 16KB.

Example:

```
from shutil import copyfileobj
status = False
if isinstance(target, string_types):
    target = open(target, 'wb')
    status = True
try:
    copyfileobj(self.stream, target, buffer_size)
finally:
    if status:
        target.close()
```

(iv) Shutil copy2() Method:

- However, the copy2() method functions like the copy(). But it also gets the access and modification times added in the meta-data while copying the data.
- Copying the same file would result in SameFileError.

```
from shutil import *
import os
import time
from os.path import basename
def displayFileStats(filename):
    file_stats = os.stat(basename(filename))
    print('\tMode:', file_stats.st_mode)
    print('\tCreated:', time.ctime(file_stats.st_ctime))
    print('\tAccessed:', time.ctime(file_stats.st_atime))
    print('\tModified:', time.ctime(file_stats.st_mtime))
    os.mkdir('test')
    print('SOURCE:')
    displayFileStats(__file__)
    copy2(__file__, 'testfile')
    print('TARGET:')
    displayFileStats(os.path.realpath(os.getcwd() + '/test/testfile'))
```

Renaming Files and Folders:

- The os.rename() is used to python rename file or directory. The rename() method takes two arguments, the current filename and the new filename.

Syntax: os.rename(current_file_name, new_file_name)

Example:

```
import os
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
The following example shows the usage of rename() method.
# !/usr/bin/python3
import os, sys
os.chdir("d:\\tmp")
# listing directories
print ("The dir is: %s"%os.listdir(os.getcwd()))
```

```
# renaming directory 'tutorialsdir'
os.rename("python3","python2")
print ("Successfully renamed.")
# listing directories after renaming "python3"
print ("the dir is: %s" %os.listdir(os.getcwd()))
```

Output:

```
The dir is: [
  'Applicationdocs.docx', 'book.zip', 'foo.txt',
  'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files',
  'java.ppt', 'Python3'
]
Successfully renamed.
the dir is: [
  'Applicationdocs.docx', 'book.zip', 'foo.txt',
  'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files',
  'java.ppt', 'python2'
]
```

Example: For remaining files.

```
import os
print("***Contents of Present working directory***\n ",os.listdir())
os.rename("sample.txt","sample1.txt")
print("***Contents of Present working directory after rename***\n ",
                                              os.listdir())
```

Output:

```
***Contents of Present working directory***
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs',
'LICENSE.txt', 'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe',
'python3.dll', 'python37.dll', 'pythonw.exe', 'sample.txt', 'Scripts',
'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll', '_ _pycache_ _']
***Contents of Present working directory after rename***
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs',
'LICENSE.txt', 'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe',
'python3.dll', 'python37.dll', 'pythonw.exe', 'sample1.txt', 'Scripts',
'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll', '_ _pycache_ _']
>>>
```

4.3.9 Splitting Pathnames

- A full file path is composed of following two components:
 1. directory name
 2. base name.
- Following methods splits a full pathname.

basename() Method:

- The `os.path.basename()` method takes a path name as an argument and returns the leaf name of the specified path.
- For example, we have created a file called `file1` in a directory called `user1` under the home directory.
- Now, we can use the `basename()` method to retrieve only the filename from the `path/home/user1/file1` by using the following code:

```
>>> import os
>>> os.path.basename('/home/user1/file1')
'file1'
```

dirname() Method:

- We can use the `os.path.dirname()` method to retrieve the directory name from a path name.
- For example, the following code returns the directory name of the path `'/home/user1/file1'`:

```
>>> import os
>>> os.path.dirname('/home/user1/file1')
'/home/user1'
```

join() Method:

- The `os.path.join()` method joins two or more path components into a single path name. This method takes the path components as arguments.
- The following example illustrates the use of the `join()` method:

```
>>>import os
>>>current_dir=os.getcwd()
>>>print current_dir
/home/
>>>join_dir=os.path.join(current_dir,'testfile')
>>>print join_dir
/home/testfile
```

- This example joins the path of the current working directory with the second argument, `testfile`, and returns the joined path.

split() Method:

- The `os.path.split()` method splits a path, which is passed as an argument, into a directory name and a base name and returns them as a tuple.
- The following example splits the joined path obtained in the previous example:

```
>>> import os.path
>>> os.path.split(join_dir)
('/home', 'testfile')
```

splitdrive() Method:

- The `splitdrive()` method is used to split the drive name and the path name of the argument passed in the method.
- The following examples illustrate the use of the `splitdrive()` method in both Unix and Windows versions:

- **Unix Version:**

```
>>> import os.path
>>> os.path.splitdrive('/home/testfile')
(' ', '/home/ testfile')
```

- **Windows Version:**

```
>>> import os.path
>>> os.path.splitdrive('c:/Python')
('c:', '/Python')
```

Table 4.2: The Information Methods of the `os.path` Module

Method	Description
<code>getsize(file_name)</code>	Returns the size of a file in bytes
<code>getatime(file_name)</code>	Returns the time when a file was last accessed
<code>getmtime(file_name)</code>	Returns the time when a file was last modified

splittext() Method:

- The `splittext` method separates the first name and the extension name of a filename. Consider the following code:

```
>>> import os.path
>>> os.path.splitext('testfile.txt')
('testfile', '.txt')
```

- Let (name be a complete path to a file, say
`/usr/home/hp1/scripting/python/intro/hw.py`

- Occasionally you need to split such a filepath into the basename `hw.py` and the directory name `/usr/home/hpl/scripting/python/intro`. In Python this is accomplished by,

```
basename = os.path.basename(fname)
dirname = os.path.dirname(fname)
# or
dirname, basename = os.path.split(fname)
```

- The extension is extracted by the `os.path.splitext` function,

```
root, extension = os.path.splitext(fname)
```
- Yielding `'py'` for extension and the rest of `f` name for root.
- The extension without the leading dot is easily obtained by `os.path.splitext(fname)[1][1:]`.
- Changing some arbitrary extension of a file with name `f` to a new extension `ext` can be done by,

```
newfile = os.path.splitext(f)[0] + ext
```

- Here is a specific example:

```
>>> f = '/some/path/case2.data_source'
>>> moviefile = os.path.basename(os.path.splitext(f)[0] + '.mpg')
>>> moviefile
'case2.mpg'
```

4.4 DIRECTORIES

- A directory or folder is nothing more than a location on a disk used for storing information about files.
- All directories in Python are organized into a hierarchical structure that you can imagine as a family tree.
- A directory or folder is a collection of files and sub directories.
- Python has the `os` module, which provides us with many useful methods to work with directories (and files as well).
- All files are contained within various directories, and Python has no problem handling these too.
- The `os` module has several methods that help us to create, remove, and change directories.
- The `os` module contains the functions `mkdir()` for creating directories and `chdir()` for moving to directories.

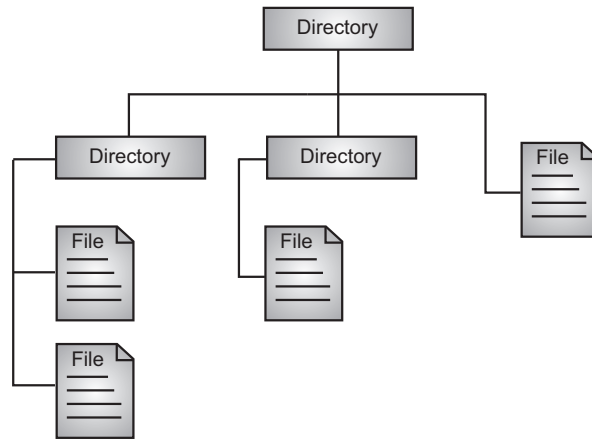


Fig. 4.2: Directory Structure

4.4.1 Create New Directory

- A directory or folder is a collection of files and sub directories. Python has the `os` module, which provides us with many useful methods to work with directories (and files as well).
- We can make a new directory using the `mkdir()` method.
- This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

Syntax: `os.mkdir("newdir")`

Example:

```
>>> import os
>>> os.mkdir("testdir")
```

4.4.2 Get Current Directory

- We can get the present working directory using the `getcwd()` method.
- The `getcwd()` method returns the current working directory in the form of a string.

Syntax: `os.getcwd()`

Example:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\ Meenakshi \\AppData\\Local\\Programs\\Python\\Python37-32'
```

4.4.3 Changing Directory

- We can change the current working directory using the `chdir()` method.
- The new path that we want to change must be supplied as a string to this method. We can use both forward slash (/) and the backward slash (\) to separate path elements.

Syntax: `os.chdir("dirname")`

Example:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Meenakshi\\AppData\\Local\\Programs\\Python\\Python37'
>>> os.chdir("d:\\IT")
>>> os.getcwd()
'd:\\IT'
>>>
```

4.4.4 List Directories and Files

- All files and sub directories inside a directory can be known using the `listdir()` method.
- This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

Example:

```
>>> os.listdir()
['DLLs', 'Doc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt',
'python.exe', 'python3.dll', 'python37.dll', 'pythonw.exe', 'Scripts',
'tcl', 'test.py', 'testdir', 'Tools', 'vcruntime140.dll']
```

4.4.5 Removing Directory

- The `rmdir()` method is used to remove directories in the current directory.

Syntax: `os.rmdir("dirname")`

Example:

```
>>> os.listdir()
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs',
'LICENSE.txt', 'mydir', 'mypkg', 'NEWS.txt', 'p1.py', 'p2.py',
'python.exe', 'python3.dll', 'python37.dll', 'pythonw.exe',
'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '_ _pycache_ _']
>>> os.rmdir("mydir")
>>> os.listdir()
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs',
'LICENSE.txt', 'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe',
'python3.dll', 'python37.dll', 'pythonw.exe', 'sample.txt', 'Scripts',
'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll', '_ _pycache_ _']
>>>
```

- If the directory is not empty then we will get the “The directory is not empty” error. To remove a directory, first remove all the files inside it using `os.remove()` method.

```
>>> os.chdir("C:\\Users\\Meenakshi\\AppData\\Local\\Programs
\\Python\\Python37")

>>> os.rmdir("mydir1")
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
  os.rmdir("mydir1")
OSError: [WinError 145] The directory is not empty: 'mydir1'
>>> os.chdir("C:\\Users\\Meenakshi\\AppData\\Local\\Programs
\\Python\\Python37\\mydir1")

>>> os.listdir()
['sample.txt']
>>> os.remove("sample.txt")
>>> os.listdir()
[]
>>> os.chdir("C:\\Users\\Meenakshi\\AppData\\Local\\Programs
\\Python\\Python37")

>>> os.rmdir("mydir1")
>>>
```

- In order to remove a non-empty directory we can use the `rmtree()` method inside the `shutil` module.

```
>>> import shutil
>>> shutil.rmtree('test')
```

PROGRAMS

Program 1: Program to create a simple file and write some content in it.

```
print("Enter 'x' for exit.");
filename = input("Enter file name to create and write content: ");
if filename == 'x':
    exit();
else:
    c = open(filename, "w");
    print("\nThe file,", filename, "created successfully!");
```

```
print("Enter sentences to write on the file: ");
sent1 = input();
c.write(sent1);
c.close();
print("\nContent successfully placed inside the file.!!");
```

Output:

```
Enter 'x' for exit.
Enter file name to create and write content: file1
The file, file1 created successfully!
Enter sentences to write on the file:
good morning
Content successfully placed inside the file.!!
>>>
```

Program 2: Program to open a file in write mode and append some content at the end of a file.

```
print("Enter 'x' for exit.");
filename = input("Enter file name to append content: ");
if filename == 'x':
    exit();
else:
    c=open(filename, "a+");
print("Enter sentences to append on the file: ");
sent1=input();
c.write("\n");
c.write(sent1);
c.close();
print("\nContent appended to file.!!");
```

Output:

```
Enter 'x' for exit.
Enter file name to append content: file1
Enter sentences to append on the file:
good afternoon
Content appended to file.!!
```

Program 3: Program to open a file in read mode and print number of occurrences of characters 'a'.

```
fname = input("Enter file name: ")
l=input("Enter letter to be searched:")
k = 0
with open(fname, 'r') as f:
    for line in f:
        words = line.split()
        for i in words:
            for letter in i:
                if(letter==l):
                    k=k+1
print("Occurrences of the letter:")
print(k)
```

Output:

```
Enter file name: file1
Enter letter to be searched:o
Occurrences of the letter:
7
```

Program 4: Program to read the contents of a file.

```
a=str(input("Enter the name of the file with .txt extension:"))
file2=open(a,'r')
line=file2.readline()
while(line!=""):
    print(line)
    line=file2.readline()
file2.close()
```

Output:

```
Enter the name of the file with .txt extension: data1.txt
```

Program 5: Program to count the number of words and lines in a text file.

```
fname = input("Enter file name: ")
num_words = 0
lines=0
with open(fname, 'r') as f:
    for line in f:
        words = line.split()
        num_words += len(words)
lines += 1
print("Number of words:")
```

```
print(num_words)
print("Number of lines:")
print(num_lines)
```

read.txt

Contents of file:
Hello world

Output:

```
Enter file name: data1.txt
Number of words:
2
Number of lines:
1
```

Program 6: Program to count the occurrences of a word in a text file.

```
fname = input("Enter file name: ")
word=input("Enter word to be searched:")
k = 0
with open(fname, 'r') as f:
    for line in f:
        words = line.split()
        for i in words:
            if(i==word):
                k=k+1
        print("Occurrences of the word:")
print(k)
```

read.txt

Let Contents of file:
hello world hello
hello

Output:

```
Enter file name: test.txt
Enter word to be searched:hello
Occurrences of the word:
3
```

Program 7: Program to copy the contents of one file into another.

```
with open("test.txt") as f:
    with open("out.txt", "w") as f1:
        for line in f:
            f1.write(line)
```

Program 8: Program to reads a text file and counts the number of times a certain letter appears in the text file.

```
fname = input("Enter file name: ")
l=input("Enter letter to be searched:")
k = 0
with open(fname, 'r') as f:
    for line in f:
        words = line.split()
        for i in words:
            for letter in i:
                if(letter==l):
                    k=k+1
print("Occurrences of the letter:")
print(k)
```

read.txt:

```
Case 1:
Contents of file:
hello world hello
hello
```

Output:

```
Enter file name: out.txt
Enter letter to be searched:o
Occurrences of the letter:
5
```

Program 9: Program to read a file and capitalize the first letter of every word in the file.

```
fname = input("Enter file name: ")
with open(fname, 'r') as f:
    for line in f:
        l=line.title()
    print(l)
```

read.txt:

```
Case 1:
Contents of file:
hello world
hello
```

Output:

```
Enter file name: read.txt
Hello World
Hello
```

Program 10: Program to read the contents of a file in reverse order.

```
filename=input("Enter file name: ")
for line in reversed(list(open(filename))):
    print(line.rstrip())
```

read.txt:

```
Contents of file:
hello world
hello
```

Output:

```
Enter file name: read.txt
hello
hello word
```

4.5 REGULAR EXPRESSIONS

- A Regular Expression or RegEx or REs is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern.
- The regular expressions can be defined as the sequence of characters which are used to search for a pattern in a string.
- The module 're' provides the support to use RegEx in the python program. The 're' module must be imported to use the RegEx functionalities in python like import re.
- The 're' module included with Python primarily used for string searching and manipulation.
- When we have imported the re module, we can start using regular expressions. The re module offers a set of functions like findall(), search(), split(), match() etc. that allows us to search a string for a match.

4.5.1 Concept of Regular Expression

- A regular expression is a special sequence of characters that helps us match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- Regular expressions are widely used in UNIX world. The Python module **re** provides full support for Perl-like regular expressions in Python.
- The re module raises the exception re.error if an error occurs while compiling or using a regular expression.
- We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression.
- To avoid any confusion while dealing with regular expressions, we would use Raw Strings as r'expression'.

- Regular Expressions are a sequence of characters used to check whether a pattern exists in a given text (string) or not.

4.5.2 Types of Regular Expression

- A regular expression is a special sequence of characters that helps we match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- There are two types of regular expression namely, Basic and Extended expressions. These regular expressions are two variations on the syntax of the specified pattern.
 1. **Basic Regular Expression:** The simplest regular expression consists of the exact characters of the string that it is intended to match. The regular language defined by the expression consists of only that one string. Upper and lower case letters are regarded as different symbols. In basic regular expressions the meta-characters '?', '+', '{', '|', '(', and ')' lose their special meaning; instead use the backslashed versions '\?', '\+', '\{', '\|', '\(', and '\)'.
 2. **Extended Regular Expression:** An extended regular expression specifies a set of strings to be matched. The expression contains both text characters and operator characters. Text characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features. An extended regular expression ordinary character, a special character preceded by a <backslash,> or a <period> shall match a single character.
- The real power of RegEx matching in Python emerges when it contains special characters called metacharacters.
- Metacharacters are special characters that affect how the regular expression finds the patterns and are mostly used to define the pattern of search or manipulation.
- In a regex, a set of characters specified in square brackets ([]) makes up a character class. This metacharacter sequence matches any single character that is in the class.

Metacharacter	Description
^	Matches the start of the string.
.	Matches a single character, except a newline. But when used inside square brackets, a dot is matched.
[]	A bracket expression matches a single character from the ones inside it. [abc] matches 'a', 'b', and 'c'. [a-z] matches characters from 'a' to 'z'. [a-cx-z] matches 'a', 'b', 'c', 'x', 'y', and 'z'.
[^]	Matches a single character from those except the ones mentioned in the brackets[^abc] matches all characters except 'a', 'b' and 'c'.

contd. ...

()	Parentheses define a marked subexpression, also called a block, or a capturing group.
\t, \n, \r, \f	Tab, newline, return, form feed.
*	Matches the preceding character zero or more times. ab*c matches 'ac', 'abc', 'abbc', and so on. [ab]* matches '', 'a', 'b', 'ab', 'ba', 'aba', and so on. (ab)* matches '', 'ab', 'abab', 'ababab', and so on.
{m,n}	Matches the preceding character minimum m times, and maximum n times. a{2,4} matches 'aa', 'aaa', and 'aaaa'.
{m}	Matches the preceding character exactly m times.
?	Matches the preceding character zero or one times. ab?c matches 'ac' or 'abc'.
+	Matches the preceding character one or one times. ab+c matches 'abc', 'abbc', 'abbbc', and so on, but not 'ac'.
	The choice operator matches either the expression before it, or the one after abc def matches 'abc' or 'def'

- There are some basic predefined character classes, which are represented by the special sequence. The special sequence consists of alphabetic characters lead by \ (backlash).
- Each special sequence has a unique meaning that helps us find or match other strings or sets of strings using a specialized syntax present in a pattern.
- A special sequence is a \ followed by one of the characters in the list below, and has a special meaning. RegEx special sequences and their meanings are given in the following table:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string.	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string").	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string").	r"\Bain" r"ain\B"

contd. ...

\d	Returns a match where the string contains digits (numbers from 0-9).	"\d"
\D	Returns a match where the string DOES NOT contain digits.	"\D"
\s	Returns a match where the string contains a white space character.	"\s"
\S	Returns a match where the string DOES NOT contain a white space character.	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character).	"\w"
\W	Returns a match where the string DOES NOT contain any word characters.	"\W"
\Z	Returns a match if the specified characters are at the end of the string.	"Spain\Z"

- Character sets are a predefined range of characters enclosed by a square bracket. With the use of sets, we can match only one out of several characters.
- A set is a set of characters inside a pair of square brackets [] with a special meaning.

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present.
[a-n]	Returns a match for any lower case character, alphabetically between a and n.
[^arn]	Returns a match for any character EXCEPT a, r, and n.
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present.
[0-9]	Returns a match for any digit between 0 and 9.
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59.
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case.
[+]	In sets, +, *, ., , (), \$, {} has no special meaning, so [+] means: return a match for any + character in the string.

4.5.3 Methods of re Package

- The "re" package provides methods such as re.match(), re.search() and re.findall().

Using match() Function:

- A Match Object is an object containing information about the search and the result.

- The `match()` function attempts to match RE pattern to string with optional flags.

Syntax: `re.match(pattern, string, flags=0)`

Where, `pattern` is the regular expression to be matched. `String` is the string, which would be searched to match the pattern at the beginning of string. We can specify different flags using bitwise OR (`|`).

Example:

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
    print "matchObj.group() : ", matchObj.group()
    print "matchObj.group(1) : ", matchObj.group(1)
    print "matchObj.group(2) : ", matchObj.group(2)
else:
    print "No match!!"
```

Output:

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

-
- The Match object has properties and methods used to retrieve information about the search and the result:
 1. `.span()` returns a tuple containing the start-, and end positions of the match.
 2. `.string` returns the string passed into the function.
 3. `.group()` returns the part of the string where there was a match.

Example:

```
import re
# Search for an upper case "S" character in the beginning of a word,
# and print its position:
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

Output:

```
(12, 17)
```

Using `search()` Function:

- The `search()` function searches the string for a match, and returns a Match object if there is a match.

- This function searches for first occurrence of RE pattern within string with optional flags.

Syntax: re.search(pattern, string, flags=0)

Example:

```
import re
line = "Cats are smarter than dogs";
searchObj = re.search( r'(.*) are (.*) .*', line, re.M|re.I)
if searchObj:
    print "searchObj.group() : ", searchObj.group()
    print "searchObj.group(1) : ", searchObj.group(1)
    print "searchObj.group(2) : ", searchObj.group(2)
else:
    print "Nothing found!!"
```

Output:

```
searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter
```

Example:

```
import re
# Check if the string starts with "Pune" and ends with "city":
txt = "Pune is a smart city"
x = re.search("^Pune.*city$", txt)
if x:
    print("YES! We have a match!")
else:
    print("No match")
```

Output:

```
YES! We have a match!")
```

Using findall() Function:

- This module is used to search for “all” occurrences that match a given pattern.
 - In contrast, search() module will only return the first occurrence that matches the specified pattern.
 - The findall() will iterate over all the lines of the file and will return all non-overlapping matches of pattern in a single step.
-

Example:

```
import re
txt = " Pune is a smart city "
# Check if "Mumbai" is in the string:
x = re.findall("Mumbai", txt)
print(x)
if (x):
    print("Yes, there is at least one match!")
else:
    print("No match")
```

Output:

No match

4.6 EXCEPTION HANDLING

- Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- An exception is also called as runtime error that can halt the execution of the program.
- An exception is an error that happens/occurs during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids the normal flow of the program's instructions.
- An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids the program to crash.
- When we execute a Python program, there may be a few uncertain conditions which occur, known as errors. Errors also referred to as bugs that are incorrect or inaccurate action that may cause the problems in the running of the program or may interrupt the execution of program.
- There are following three type of error occurs:
 1. **Compile Time Errors:** Occurs at the time of compilation, include due error occur to the violation of syntax rules like missing of a colon (:).
 2. **Run Time Errors:** Occurs during the runtime of a program, example, include error occur due to wrong input submitted to program by user.
 3. **Logical Errors:** Occurs due to wrong logic written in the program.
- Errors occurs at runtime are known as exception. Errors detected during execution of program. Python provides a feature (Exception handling) for handling any unreported errors in program.

- When exception occurs in the program, execution gets terminated. In such cases we get system generated error message.
- By handling the exceptions, we can provide a meaningful message to the user about the problem rather than system generated error message, which may not be understandable to the user.
- Exception can be either built-in exceptions or user defined exceptions.
- The interpreter or built-in functions can generate the built-in exceptions while user defined exceptions are custom exceptions created by the user.

Example: For exceptions.

```
>>> a=3
>>> if (a<5)
SyntaxError: invalid syntax
>>> 5/0
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
5/0
ZeroDivisionError: division by zero
```

- Python provides two very important features to handle any unexpected error in Python programs and to add debugging capabilities in them namely, Exception Handling and Assertions.

4.6.1 Built-in Exceptions

- Errors detected during execution are called exceptions. An exception is an event (usually an error), which occurs during the execution of a program that disrupts the normal flow of execution of the program (or program's instructions).
- In Python programming we can handle exceptions using try-except statement, try-finally statement and raise statement.
- Following table lists all the standard built-in exceptions available in Python programming language:

Sr. No.	Exception	Cause of Error
1.	ArithmeticError	Base class for all errors that occur for numeric calculation.
2.	AssertionError	Raised in case of failure of the assert statement.

contd. ...

3.	AttributeError	Raised in case of failure of attribute reference or assignment.
4.	Exception	Base class for all exceptions.
5.	EOFError	Raised when there is no input from either the <code>raw_input()</code> or <code>input()</code> function and the end of file is reached.
6.	EnvironmentError	Base class for all exceptions that occur outside the Python environment.
7.	FloatingPointError	Raised when a floating point calculation fails.
8.	ImportError	Raised when an import statement fails.
9.	IndexError	Raised when an index is not found in a sequence.
10.	IOError	Raised when an input/ output operation fails, such as the <code>print</code> statement or the <code>open()</code> function when trying to open a file that does not exist.
11.	IndentationError	Raised when indentation is not specified properly.
12.	KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
13.	KeyError	Raised when the specified key is not found in the dictionary.
14.	LookupError	Base class for all lookup errors.
15.	NameError	Raised when an identifier is not found in the local or global namespace.
16.	NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
17.	OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
18.	OSError	Raised for operating system-related errors.
19.	RuntimeError	Raised when a generated error does not fall into any category.
20.	StopIteration	Raised when the <code>next()</code> method of an iterator does not point to any object.
21.	SystemExit	Raised by the <code>sys.exit()</code> function.
22.	StandardError	Base class for all built-in exceptions except <code>StopIteration</code> and <code>SystemExit</code> .

contd. ...

23.	SyntaxError	Raised when there is an error in Python syntax.
24.	SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25.	SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26.	TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
27.	UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
28.	ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
29.	ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.

4.6.2 Handling Exceptions

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- The exception handling is a process that provides a way to handle exceptions that occur at runtime. The exception handling is done by writing exception handlers in the program.
- The exception handlers are blocks that execute when some exception occurs at runtime. Exception handlers displays same message that represents information about the exception.
- For handling exception in Python, the exception handler block needs to be written which consists of set of statements that need to be executed according to raised exception. There are three blocks that are used in the exception handling process, namely, try, except and finally.
 1. **try Block:** A set of statements that may cause error during runtime are to be written in the try block.

2. **except Block:** It is written to display the execution details to the user when certain exception occurs in the program. The except block executed only when a certain type as exception occurs in the execution of statements written in the try block.
3. **finally Block:** This is the last block written while writing, an exception handler in the program which indicates the set of statements that are used to clean up the resources used by the program.

Exception Handling Using try-except:

- In Python, exceptions can be handled using a try statement. A try block consisting of one or more statements is used by programmers to partition code that might be affected by an exception.
- A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.
- The associated except blocks are used to handle any resulting exceptions thrown in the try block. If any statement within the try block throws an exception, control immediately shifts to the catch block. If no exceptions is thrown in the try block, the catch block is skipped.
- There can be one or more except blocks. Multiple except blocks with different exception names can be chained together.
- The except blocks are evaluated from top to bottom in the code, but only one except block is executed for each exception that is thrown.
- The first except block that specifies the exact exception name of the thrown exception is executed. If no except block specifies a matching exception name then an except block that does not have an exception name is selected, if one is present in the code.

Syntax:

```
try:
    certain operations here
    .....
except Exception1:
    If there is Exception1, then execute this block.
except Exception2:
    If there is Exception2, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

Example 1: For try-except clause/statement.

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
```

```
except IOError:
    print ("Error: can\'t find file or read data")
else:
    print ("Written content in the file successfully")
    fh.close()
```

Example 2: For try statement.

```
n=10
m=0
try:
    n/m
except ZeroDivisionError:
    print("Divide by zero error")
else:
    print (n/m)
```

Output:

Divide by zero error

Exception Handling Using try-except with No Exception:

- We can use try-except clause with no exception. All types of exceptions that occur are caught by the try-except statement.
- However, because it catches all exceptions, the programmer cannot identify the root cause of a problem that may occur. Hence, this type of programming approach is not considered good.

Syntax:

```
try:
    certain operations here
    .....
except:
    If there is any exception, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

Example: For try-except statement with no exception.

```
while True:
    try:
        a=int(input("Enter an integer: "))
        div=10/a
        break
```

```
except:
    print("Error Occurred")
    print("Please enter valid value")
    print()
print("Division is: ",div)
```

Output:

```
Enter an integer: b
Error Occurred
Please enter valid value

Enter an integer: 2.5
Error Occurred
Please enter valid value

Enter an integer: 0
Error Occurred
Please enter valid value

Enter an integer: 5
Division is:  2.0
```

Exception Handling Using try...finally:

- The try statement in Python can have an optional finally clause. This clause is executed always and is generally used to release external resources.
- The statement written in finally clause will always be executed by the interpreter, whether the try statement raises an exception or not.
- A finally block is always executed before leaving the try statement, whether an exception is occurred or not.
- When an exception is occurred in try block and has not been handled by an except block, it is re-raised after the finally block has been executed.
- The finally clause is also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement.

Syntax:

```
try:
    certain operations here;
    .....
    Due to any exception, this may be skipped.
finally:
    This would always be executed.
    .....
```

Example 1: For try-finally.

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
finally:
    print ("file is closing")
fh.close()
```

Example 2: Program to check for ZeroDivisionError Exception.

```
x=int(input("Enter first value:"))
y=int(input("Enter second value:"))
try:
    result=x/y
except ZeroDivisionError:
    print("Division by Zero")
else:
    print("Result is:",result)
finally:
    print("Execute finally clause")
```

Output 1:

```
Enter first value:5
Enter second value:0
Division by Zero
Execute finally clause
```

Output 2:

```
Enter first value:10
Enter second value:5
Result is: 2.0
Execute finally clause
```

raise Statement:

- We can raise an existing exception by using raise keyword. So, we just simply write raise keyword and then the name of the exception.
 - The raise statement allows the programmer to force a specified exception to occur.
-

Example: We can use raise to throw an exception if age is less than 18 conditions occurs.

```
while True:
    try:
        age = int(input("Enter your age for election: "))
```

```
if age < 18:
    raise Exception
else:
    print("you are eligible for election")
    break
except Exception:
    print("This value is too small, try again")
```

Output:

```
Enter your age for election: 11
This value is too small, try again
Enter your age for election: 18
you are eligible for election
>>>
```

- The raise statement can be complemented with a custom exception as explained in next section.

4.6.3 Exception with Arguments

- An exception is an error that happens during the execution of a program. An exception can have an argument, which is a value that gives additional information about the problem.
- The contents of the argument vary from exception to exception. We capture an exception's argument by supplying a variable in the except clause as follows:

```
try:
    We do your operations here;
    .....
except ExceptionType, Argument:
    We can print value of Argument here...
```

Example:

```
try:
    b=float(56+78/0)
except Exception, Argument:
    print 'This is the Argument\n', Argument
```

Output:

```
This is the Argument
integer division or modulo by zero
```

-
- If we write the code to handle a single exception, we can have a variable follow the name of the exception in the except statement. If we are trapping multiple exceptions, we can have a variable follow the tuple of the exception.
-

- This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.
- Following is an example for a single exception:

```
# Define a function here.
def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "The argument does not contain numbers\n", Argument
# Call above function here.
temp_convert("xyz");
```

Output:

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

4.6.4 User Defined Exception

- Python has many built-in exceptions which forces the program to output an error when something in it goes wrong. However, sometimes we may need to create custom exceptions that serves the purpose.
- Python allow programmers to create their own exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly. Most of the built-in exceptions are also derived from Exception class.
- User can also create and raise his/her own exception known as user defined exception. User defined exceptions are also known as custom exceptions.
- In the following example, we create custom exception class AgeSmallException that is derived from the base class Exception.

Example 1: Raise a user defined exception if age is less than 18.

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""          # empty class
    pass
class AgeSmallException(Error):
    """Raised when the input value is too small"""  # empty class
    pass
# main program
```

```
while True:
    try:
        age = int(input("Enter your age for election: "))
        if age < 18:
            raise AgeSmallException
        else:
            print("you are eligible for election")
            break
    except AgeSmallException:
        print("This value is too small, try again!")
        print()
```

Output:

```
Enter your age for election: 11
This value is too small, try again!
Enter your age for election: 15
This value is too small, try again!
Enter your age for election: 18
you are eligible for election
```

Example 2: Raise a user defined exception id password is incorrect.

```
class InvalidPassword(Exception):
    pass
def verify_password(pswd):
    if str(pswd) != "abc":
        raise InvalidPassword
    else:
        print('Valid Password: '+str(pswd))
# main program
verify_password("abc") # won't raise exception
verify_password("xyz") # will raise exception
```

Output:

```
Valid Password: abc
Traceback (most recent call last):
File
"C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\p1.py",
line 12, in <module>
```

```
verify_password("xyz") # will raise exception
File
"C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\p1.py", line
6, in verify_password
    raise InvalidPassword
InvalidPassword
```

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

- Which is a named location on disk to store related information?
(a) file (b) dictionary
(c) directory (d) All of the mentioned
- The key function for working with files in Python is,
(a) read() (b) write()
(c) open() (d) close()
- The files can be accessed using various modes like,
(a) read (b) write
(c) append (d) All of the mentioned
- Which mode of file opens the file to read-only mode?
(a) w (b) r
(c) a (d) a+
- Writing a string or sequence of bytes (for binary files) is done using the,
(a) read() (b) readline()
(c) write() (d) None of the mentioned
- Which method tells you the current position within the file?
(a) tell() (b) read()
(c) write() (d) close()
- To read three characters from a file object file1, we use ...
(a) file.readline() (b) file1.readlines()
(c) file1.read(3) (d) file1.read()
- To create a new file in Python, use the open() method, with one of the following parameters:
(a) "x" (create - will create a file, returns an error if the file exist)
(b) "a" (append - will create a file if the specified file does not exist)
(c) "w" (write - will create a file if the specified file does not exist)
(d) All of the mentioned

9. A python program file which contains a python code including python functions, class, or variables is called as,
- (a) file (b) module
(c) directory (d) dictionary
10. To create a module just save the code we want in a file with the file extension,
- (a) .py (b) .pi
(c) .pyf (d) .phy
11. We can use the module we just created, by using the,
- (a) create statement (b) package statement
(c) import statement (d) None of the mentioned
12. We can create an alias when you import a module, by using the _____ keyword.
- (a) as (b) in
(c) pkg (d) All of the mentioned
13. The datetime() class requires three parameters to create a date namely,
- (a) year (b) month
(c) day (d) All of the mentioned
14. Which file mode used to read file only in binary format?
- (a) r (b) rb
(c) rb+ (d) r
15. Python has also a built-in module called math, which extends the list of mathematical functions such as,
- (a) math.sqrt() (b) math.pi
(c) math.ceil() (d) All of the mentioned
16. To exploring modules use, in Python the dir() function is used.
- (a) mod() function (b) mkdir() function
(c) dir() function (d) rmdir() function
17. The sequence of characters which are used to search for a pattern in a string called as,
- (a) regular expression (b) regular statement
(c) Both (a) and (b) (d) None of the mentioned
18. Which Python module work named with RegEx?
- (a) r (b) re
(c) rx (d) None of the mentioned
19. Which method looks for the first location where the RegEx pattern produces a match with the string.
- (a) match() (b) split()
(c) search() (d) findall()

20. A regular expression can be formed by using the mix of,
- (a) meta-characters like \, ^, (), [], \$ etc.
 - (b) special sequences like \s, \d, \A, \A etc.
 - (c) sets like [a-n], [0-9], [arn] etc.
 - (d) All of the mentioned
21. Which allow us to create a hierarchical file directory structure of modules?
- (a) module
 - (b) package
 - (c) file
 - (d) RegEx
22. A package is basically a directory with Python files and a file with the name,
- (a) `__file__.py`
 - (b) `__package__.py`
 - (c) `__init__.py`
 - (d) `__name__.py`
23. Standard packages includes,
- (a) NumPy
 - (b) SciPy
 - (c) Pandas
 - (d) All of the mentioned
24. Which is an unusual condition in a program resulting in the interruption in the flow of the program?
- (a) error
 - (b) exception
 - (c) Both (a) and (b)
 - (d) None of the mentioned
25. Following which keywords are used for exception handling,
- (a) try
 - (b) except
 - (c) finally
 - (d) All of the mentioned
26. An exception can have an _____, which is a value that gives additional information about the problem.
- (a) parameter
 - (b) variable
 - (c) argument
 - (d) None of the mentioned
27. To throw (or raise) an exception use the, raise keyword.
- (a) raise keyword
 - (b) try
 - (c) except
 - (d) else
28. Which is the following statement is true:
- (a) The try block lets you test a block of code for errors
 - (b) The except block lets you handle the error
 - (c) The finally block lets you execute code, regardless of the result of the try- and except blocks
 - (d) All of the mentioned
29. Assertions are carried out by,
- (a) try statement
 - (b) except
 - (c) assert statement
 - (d) finally statement

30. What is the output of following code:

```
import re
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
if x:
    print("YES! We have a match!")
else:
    print("No match")
```

- (a) YES! We have a match! (b) No match
(c) Run time error (d) typeError

31. Is the following Python code valid?

```
try:
    # Do something
except:
    # Do something
finally:
    # Do something
```

- (a) no, there is no such thing as finally
(b) no, finally cannot be used with except
(c) no, finally must come before except
(d) Yes

32. What is the output of following code:

```
import math
x = math.sqrt(64)
print(x)
```

- (a) 8.0 (b) 4.0
(c) 7.0 (d) 6.0

Answers

1. (a)	2. (c)	3. (d)	4. (b)	5. (c)	6. (a)	7. (c)	8. (d)	9. (b)	10. (a)
11. (c)	12. (a)	13. (d)	14. (b)	15. (d)	16. (c)	17. (a)	18. (b)	19. (c)	20. (d)
21. (b)	22. (c)	23. (d)	24. (b)	25. (d)	26. (c)	27. (a)	28. (d)	29. (c)	30. (a)
31. (b)	32. (a)								

Q.II Fill in the Blanks:

- _____ are named locations on disk to store related information.
- When we want to read from or write to a file, we need to _____ it first.

3. We can use _____ function to get a list containing all attributes of a module.
4. A Regular Expression (RegEx) is a sequence of characters that defines a _____ pattern.
5. An _____ is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
6. We can specify the _____ while opening a file.
7. A directory must contain a file named _____ in order for Python to consider it as a package.
8. The _____ method returns the number of characters written to the file.
9. The critical operation which can raise an exception is placed inside the _____ clause. The code that handles the exceptions is written in the _____ clause.
10. We can use the _____ method of the os module to create directories in the current directory.
11. Python has a module named _____ to work with RegEx.
12. The _____ method displays the current working directory.
13. To read a file in Python, we must open the file in reading _____ mode.
14. The python code file saved with the extension (.py) is treated as the _____.
15. We can see that the read() method returns a _____ as '\n'. Once the end of the file is reached, we get an empty string on further reading.
16. _____ characters such as [] . ^ \$ * + ? { } () \ | etc. are characters that are interpreted in a special way by a RegEx engine.
17. We can change our current file cursor (position) using the _____ method.
18. We can import modules from packages using the _____ operator.
19. The _____ module has a class named dateclass that can contain information from both date and time objects.
20. The _____ method returns our current position (in number of bytes).
21. A period (.) in RegEx matches any _____ character, (except newline '\n').
22. The _____ keyword is used to raise/throw an exception.
23. We can use the readline() method to read _____ lines of a file.
24. The re.findall() method returns a list of strings containing _____ matches.
25. The _____ method closes an opened file.
26. In Python programming, exceptions are raised when errors occur at _____.
27. A python _____ is a collection of modules.

28. We can import the definitions inside a module to another module using _____ keyword.
29. Using the _____ statement, we can instruct a program to execute a certain block of code only in the absence of exceptions.
30. We can use the _____ method to change the current directory.

Answers

1. Files	2. open	3. dir()	4. search
5. exception	6. mode	7. __init__.py	8. write()
9. try, except	10. mkdir()	11. re	12. getcwd()
13. r	14. module	15. newline	16. Meta
17. seek()	18. dot (.)	19. datetime	20. tell()
21. single	22. raise	23. individual	24. all
25. close()	26. runtime	27. package	28. import
29. else	30. chdir()		

Q.III State True or False:

1. A file is an internal storage on hard disk.
2. A module is a single source code file have the .py file extension.
3. File open() function return file object is called a handle.
4. The try and except block in Python is used to catch and handle exceptions.
5. To open the file, use the built-in open() function (returns a file object, which has a read() method for reading the content of the file).
6. The r+ mode of file opens a file for both reading and writing.
7. The rmdir() method deletes the directory, which is passed as an argument in the method.
8. Default file access mode is write mode.
9. A package is a directory of Python modules that contains an additional __init__.py file, which distinguishes a package from a directory that is supposed to contain multiple Python scripts.
10. The search() function in RegEx searches the string for a match, and returns a Match object if there is a match.
11. The raise keyword allows us to throw an exception at any time.
12. A package is a sequence of characters that forms a search pattern.
13. Python has a built-in package called re, which can be used to work with Regular Expressions.
14. In mode, we specify whether we want to read r, write w or append a to the file.

15. The mode attribute of file returns access mode with which file was opened.
16. The exit() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
17. A file containing .py python code is called a module.
18. The write() method writes any string to an open file.
19. In the try, all statements are executed until an exception is encountered while the except is used to catch and handle the exception(s) that are encountered in the try clause.
20. A file can be removed (deleted) using the delete() method. F
21. The findall() function returns a list containing all matches.
22. The read() method reads a string from an open file.
23. The seek() method changes the current file position.
24. A except clause can have any number of try clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
25. Python os module provides methods that help you perform file-processing operations, such as renaming (changing/modifying) and deleting files.
26. when a Python script encounters a situation that it cannot cope with, it raises an exception.
27. We can import a module using the import statement and access the definitions inside it using the dot operator (math.pi).
28. To modify the names of modules and their functions within Python by using the 'in' keyword.
29. The random module provides functions to perform operations on random numbers.
30. In Python, exceptions can be handled using a try statement.
31. We can use the dir() function to find out names that are defined inside a module.
32. Python offers regex capabilities through the re module bundled as a part of the standard library.

Answers

1. (F)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)	7. (T)	8. (F)	9. (T)	10. (T)
11. (T)	12. (F)	13. (T)	14. (T)	15. (T)	16. (F)	17. (T)	18. (T)	19. (T)	20. (F)
21. (T)	22. (T)	23. (T)	24. (F)	25. (T)	26. (T)	27. (T)	28. (F)	29. (T)	30. (T)
31. (T)	32. (T)								

Q.IV Answer the following Questions:**(A) Short Answer Questions:**

1. Define module.
2. Define package.
3. What is exception?
4. Which function is used for creating files?
5. Define directory.
6. Define file.
7. What is RegEx?
8. What is user defined exception?
9. List functions in math module.
10. Give the purpose of match().
11. Give the syntax for handling exceptions.
12. List any two built-in exceptions.
13. What is the purpose of datetime module.
14. List standard types of packages.
15. Define text and binary files.

(B) Long Answer Questions:

1. What is module? How to create and exploring it? Explain with example.
2. What is file? How to read and write to a file?
3. What is package? How to create and it? Explain with example.
4. What is directory? How to create it? Explain with example.
5. What is exception? How to handle it?
6. What is regular expression? Explain with example.
7. State the methods of re package in python?
8. What is the difference between exception and syntax error?
9. State the use of search method of re package with an example.
10. Write the use of seek() and tell() function.
11. Which methods are used to read from file? Explain two of them with example.
12. How will you copy and rename file in Python?
13. What is user defined function? Explain with example.
14. Describe various types of regular expressions.
15. How to listing files and directories? Describe with example.
16. Write short note on: Operations on file.
17. With the help of example describe how exploring a package?
18. List methods for math and random module.

19. What is the output of following code:

```
def square(x):
    assert x>=0, 'Only positive numbers are allowed'
    return x*x
try:
    square(-2)
except AssertionError as msg:
    print(msg)
```

20. What is the output of following code:

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
    print ('Out of try except blocks')
```

21. What is the output of following code:

```
from re import split
string = "Flat is better than nested. Sparse is better than dense."
words = split(r' ', string)
print(words)
```

■■■