THIRD YEAR DIPLOMA
ENGINEERING AND TECHNOLOGY
COMPUTER ENGINEERING GROUP
SEMESTER-VI

PROGRAMMING WITH 'PYTHON'

VIJAY T. PATIL
Dr. MEENAKSHI A. THALOR
Mrs. JYOTI MANTE (KHURPADE)

Scanned with CamScanner

Contents ...

1. Introduction and Syntax of Python Program **1.1 – 1.30**

- 1.0 Introduction 1.1
- 1.1 Features of Python 1.2
  - 1.1.1 Running Python Scripts 1.6
  - 1.1.2 Internal Working of Python 1.7
- 1.2 Python Building Blocks 1.7
  - 1.2.1 Character Set 1.8
  - 1.2.2 Identifiers 1.8
  - 1.2.3 Keywords 1.8
  - 1.2.4 Variables 1.9
  - 1.2.5 Literals 1.10
  - 1.2.6 Indentation 1.12
  - 1.2.7 Commenting in Python 1.13
- 1.3 Python Environment Setup (Installation and Working of IDE) 1.13
- 1.4 Running Simple Python Scripts to Display 'Welcome' Message 1.20
- 1.5 Python Data Types 1.21
  - 1.5.1 Numbers Data Type 1.22
  - 1.5.2 String Data Type 1.24
  - 1.5.3 List Data Type 1.26
  - 1.5.4 Tuple Data Type 1.27
  - 1.5.5 Dictionary 1.27
- 1.6 Input and Output in Python Programming 1.28
  - Practice Questions 1.30

2. Python Operators and Control Flow Statements **2.1 – 2.24**

- 2.0 Introduction 2.1
- 2.1 Operators 2.2
  - 2.1.1 Arithmetic Operators 2.3
  - 2.1.2 Assignment Operators (Augmented Assignment Operators) 2.3
  - 2.1.3 Relational or Comparison Operators 2.4
  - 2.1.4 Logical Operators 2.4
  - 2.1.5 Bitwise Operators 2.4
  - 2.1.6 Identity Operators 2.5
  - 2.1.7 Membership Operators 2.6
  - 2.1.8 Python Operator Precedence and Associativity 2.7
- 2.2 Control Flow 2.8
- 2.3 Conditional Statements/Decision Making Statements 2.8
  - 2.3.1 if Statement 2.8
  - 2.3.2 if-else Statement 2.9
  - 2.3.3 Nested if Statement 2.10
  - 2.3.4 Multi-way if-elif-else (Ladder) Statement 2.11

Scanned with CamScanner

2.4 Looping in Python 2.12
2.4.1 while Loop 2.12
2.4.2 for Loop 2.14
2.4.3 Nested for and while Loops 2.16
2.5 Loop Manipulation/Loop Control Statements 2.18
2.5.1 break Statement 2.18
2.5.2 continue Statement 2.18
2.5.3 pass Statement 2.19
• Practice Questions 2.23

3. Data Structures in Python **3.1 – 3.36**
3.0 Introduction 3.1
3.1 Lists 3.1
3.1.1 Creating a List 3.2
3.1.2 Accessing Values in List 3.3
3.1.3 Deleting Values in List 3.3
3.1.4 Updating Lists (Change or Add Elements to a List) 3.4
3.1.5 Basic List Operations (Indexing and Slicing) 3.6
3.1.5.1 Indexing 3.6
3.1.5.2 List Slicing 3.7
3.1.6 Built-in Functions and Methods for List 3.8
3.2 Tuples 3.10
3.2.1 Creating Tuple 3.11
3.2.2 Accessing Values in Tuple 3.13
3.2.3 Deleting Tuples 3.13
3.2.4 Updating Tuple 3.14
3.2.5 Tuple Operations 3.14
3.2.6 Build-in Functions and Methods of Tuple 3.15
3.3 Sets 3.16
3.3.1 Accessing Values in Sets 3.16
3.3.2 Deleting Values in Set 3.17
3.3.3 Updating Set 3.17
3.3.4 Basic Set Operations 3.18
3.3.5 Built-in Functions and Methods for Set 3.19
3.4 Dictionaries 3.21
3.4.1 Creating Dictionary 3.22
3.4.2 Accessing Values in a Dictionary 3.22
3.4.3 Deleting Elements/Items from Dictionary 3.23
3.4.4 Updating Dictionary 3.23
3.4.5 Basic Operations on Directory 3.24
3.4.6 Built-in Functions and Methods for Dictionary 3.25
• Practice Questions 3.35

4. Python Functions, Modules and Packages **4.1 – 4.38**
4.0 Introduction 4.1
4.1 Use of Python Built-In Functions 4.2
4.1.1 Type Data Conversion Functions 4.2
4.1.2 Built-In Mathematical Functions 4.5

Scanned with CamScanner

4.2 User Defined Functions 4.6
4.2.1 Function Definition 4.7
4.2.2 Function Calling 4.7
4.2.3 Function Arguments 4.9
4.2.3.1 Required Arguments 4.9
4.2.3.2 Keywords Arguments 4.10
4.2.3.3 Default Arguments 4.10
4.2.3.4 Variable Length Arguments 4.11
4.2.4 return Statement 4.11
4.2.5 Scope of Variable 4.12
4.3 Modules 4.14
4.3.1 Writing Module 4.14
4.3.2 Importing Modules 4.15
4.3.3 Aliasing Modules 4.17
4.3.4 Python Built in Modules 4.17
4.3.5 Namespace and Scoping 4.22
4.4 Python Packages 4.23
4.4.1 Introduction 4.23
4.4.2 Writing Python Packages 4.24
4.4.3 Standard Packages 4.24
4.4.3.1 Math 4.25
4.4.3.2 NumPy 4.25
4.4.3.3 SciPy 4.30
4.4.3.4 Matplotlib 4.31
4.4.3.5 Pandas 4.33
4.4.4 User Defined Packages 4.36
- Practice Questions 4.37

5. Object Oriented Programming in Python 5.1 – 5.18

5.0 Introduction 5.1
5.1 Classes 5.2
5.1.1 Creating Classes 5.2
5.1.2 Objects and Creating Objects 5.3
5.1.3 Instance Variable and Class Variable 5.4
5.2 Data Hiding 5.4
5.3 Data Encapsulation and Data Abstraction 5.5
5.4 Method Overloading 5.9
5.5 Inheritance and Composition Class 5.10
5.5.1 Inheritance 5.11
5.5.2 Method Overriding 5.14
5.5.3 Composition Classes 5.15
5.6 Customization via Inheritance Specializing Inherited Methods 5.16
- Practice Questions 5.18

Scanned with CamScanner

6. File I/O Handling and Exception Handling 6.1 – 6.24

6.0 Introduction 6.1
6.1 I/O Operations (Reading Keyboard Input, Printing to Screen) 6.1
6.2 Files 6.4
6.2.1 Opening File in different Modes 6.4
6.2.2 Different Modes of Opening File 6.5
6.2.3 Accessing File Contents using Standard Library Functions 6.6
6.2.4 Closing File 6.7
6.2.5 Writing Data to File 6.7
6.2.6 Reading Data from File 6.8
6.2.7 File Position 6.9
6.2.8 Renaming a File 6.13
6.2.9 Deleting a File 6.13
6.2.10 Directories 6.14
6.2.10.1 Create New Directory 6.14
6.2.10.2 Get Current Directory 6.14
6.2.10.3 Changing Directory 6.14
6.2.10.4 List Directories and Files 6.14
6.2.10.5 Removing Directory 6.15
6.3 Exception Handling 6.17
6.3.1 Introduction 6.17
6.3.2 Exception Handling in Python Programming 6.19
6.3.2.1 try-except 6.19
6.3.2.2 try-except with No Exception 6.20
6.3.2.3 try...finally 6.21
6.3.3 Raise Statement 6.22
6.3.4 User Defined Exception 6.22
• Practice Questions 6.24

Programs P.1 – P.8

■■■

Scanned with CamScanner

1...

Introduction and Syntax of Python Program

Chapter Outcomes...

- Identify the given variables, keywords and constants in Python.
- Use indentation, comments in the given program.
- Install the given Python IDE and editor.
- Develop the python program to display the given text.

Learning Objectives...

- To understand Basic Concepts in Python Programming
- To learn Features and Environment for Python Programming
- To know Python Programming Building Blocks like Keywords, Variables, Identifiers etc.
- To learn Data Types in Python Programming

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.

History of Python Programming Language:

- 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) to alt.sources. 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.

Scanned with CamScanner

- 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.**
- 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. Nextflix 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 FEATURES OF PYTHON

- Python's features include:

1. Easy to Learn and Use:

- Python is easy to learn and use. It is developer-friendly and high level programming language. Python has few keywords, simple structure, and a clearly defined syntax that makes it easily understandable for beginners.
- Python language is more expressive means that it is more understandable and readable for programmers.
- 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 Language:

- Python is an interpreted language i.e., interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners. There are excellent, straightforward tools to work with python code, is interactive interpreter.
- In python, we need not to learn a build system, IDE, special text editor, or anything else to start using python. All we need only a command prompt and the interactive editor.
- Python provides a Python Shell (also known as Python Interactive Shell) which is used to execute a single Python command and get the result as shown below. If python is installed on the PC then to open the Python Shell on Windows, open the command prompt, write python and press enter.
- As we can see, a Python Prompt comprising of three Greater Than symbols (>>>) appears, (See Fig. 1.1).

Scanned with CamScanner

```
Command Prompt - python
Microsoft Windows [Version 10.0.17134.950]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Meenakshi>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Fig. 1.1: Python Command Prompt

3. **Interactive Mode:**

- Python programming language has support for interactive mode, which allows interactive testing and debugging of code. Graphical User Interfaces (GUIs) can be developed using Python.
- Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh and the X Window system of Unix.
- There are many free and commercial editors available for Python. Following table lists Python editors:

Sr. No.	Editor	Description	Icon/Logo
1.	IDLE	• 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.	
2.	PyCharm	• 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.	PC
3.	Spyder	• 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.	SPYDER
4.	PyDev	• 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.	PyDev
5.	Jupyter Notebook	• 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.	jupyter

- 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. 1.2).

Scanned with CamScanner

- 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.

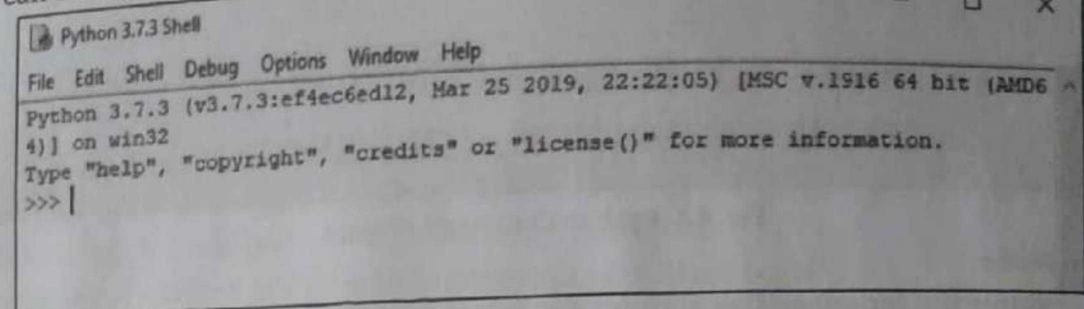


Fig. 1.2: Python Shell

4. **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.

5. **Platform Independence/Cross Platform Language/Portable:**

- Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.
- Fig. 1.3 shows execution of Python code by interpreter.

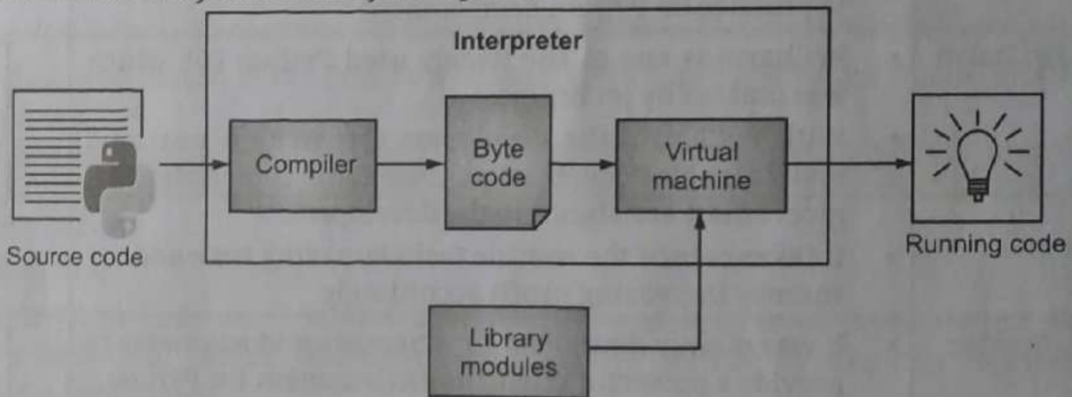


Fig. 1.3: Execution of Python Code

- 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.

6. **Object-Oriented Language:**

- A programming language that can model the real world is said to be object-oriented. It focuses on objects, and combines data and functions. Contrarily, a procedure-oriented language revolves around functions, which are code that can be reused.
- Python supports both procedure-oriented and object-oriented programming which is one of the key python features. It also supports multiple inheritance, unlike Java.
- Python has a powerful but simplistic way of doing OOP, especially when compared to C++ and Java languages. Python supports object oriented language and concepts of classes and objects come into existence.

Scanned with CamScanner

7. **Extensible:**

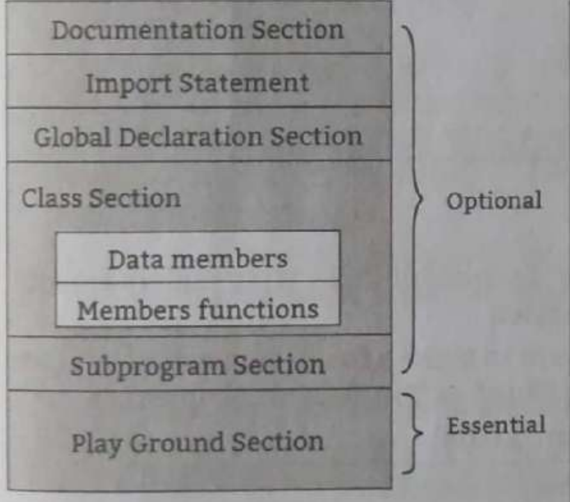
- Python programming implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in the python code.
- Python has a large and broad library and provides rich set of module and functions for rapid application development.
- Python languages bulk library is portable and cross platform compatible with Unix, Windows etc.

Limitations of Python:

1. Python is an interpreter based language. Therefore, it is bit slower than compiler based languages.
2. Python is a high level language like C/C++/Java, it also uses many layers to communicate with the operating system and the computer hardware.
3. Graphics intensive applications such as games make the program to run slower.
4. Due to the flexibility of the data types, Python's memory consumption is also high.

Structure of a Python Program:

- Fig. 1.4 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.




```
#Python Program Structure. Calculate Area and Circumference
of circle using class.   # 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
```

Fig. 1.4: Typical Program Structure of Python Programming with Example

- 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.

Scanned with CamScanner

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.

1.1.1 Running Python Scripts

- Python has two basic modes namely, normal and interactive.
- The **normal script mode** is the mode where the scripted and finished .py files are run in the Python interpreter.
- 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.

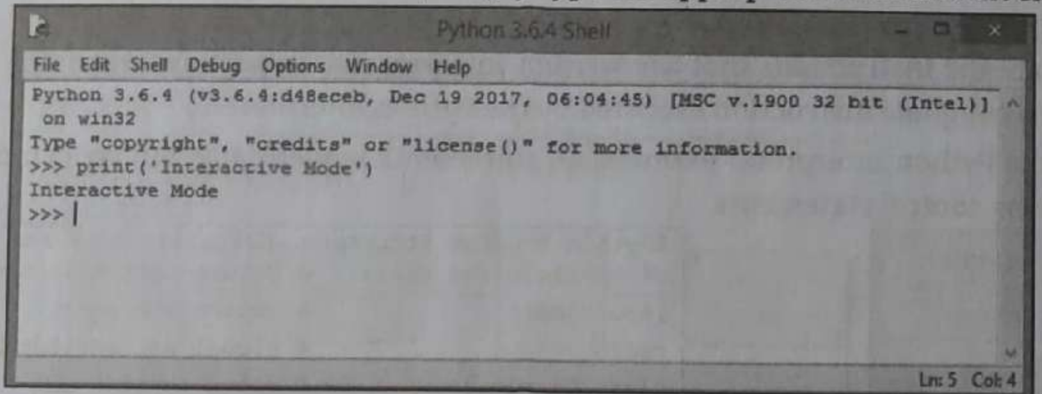


Fig. 1.5

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. 1.6 (a) and 1.6 (b)).

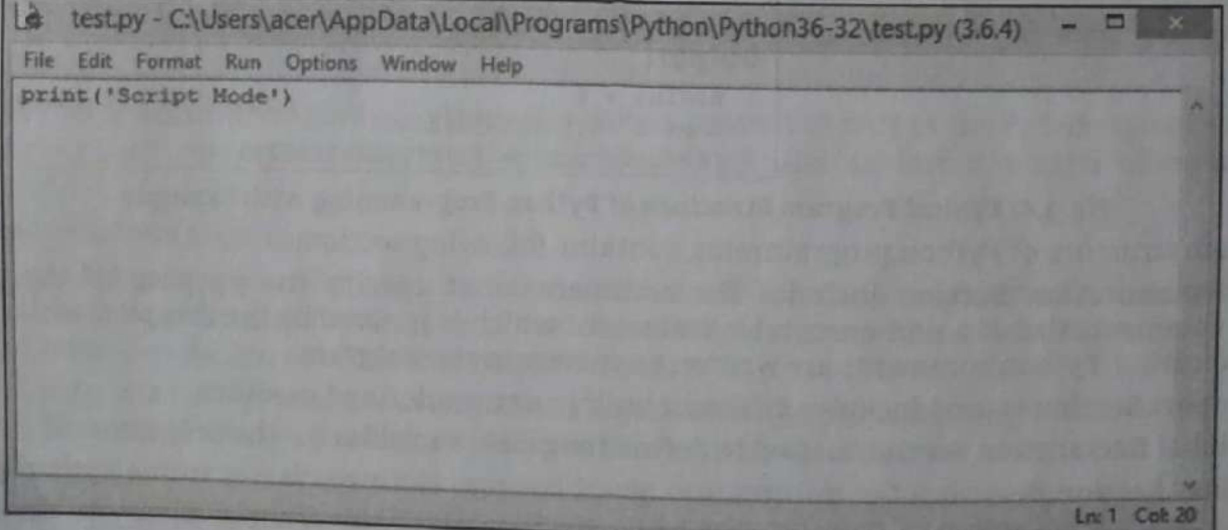


Fig. 1.6 (a)

Scanned with CamScanner

```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\acer\AppData\Local\Programs\Python\Python36-32\test.py ==
Script Mode
>>> |
                                                                    Ln: 6 Col: 4
```

Fig. 1.6 (b)

1.1.2 Internal Working of Python

- Python is an object oriented programming language like C++ and Java. Python is called an interpreted language means Python programs are executed by the interpreter.
- Python uses code modules that are interchangeable instead of a single long list of instructions that was standard for functional programming languages.
- The standard implementation of python is called "CPython". It is the default and widely used implementation of the Python.
- When a programmer tries to run a Python code as instructions in an interactive manner in a Python shell, then Python performs various operations internally.
- All such internal operations can be broken down into a series of steps as shown in Fig. 1.7.

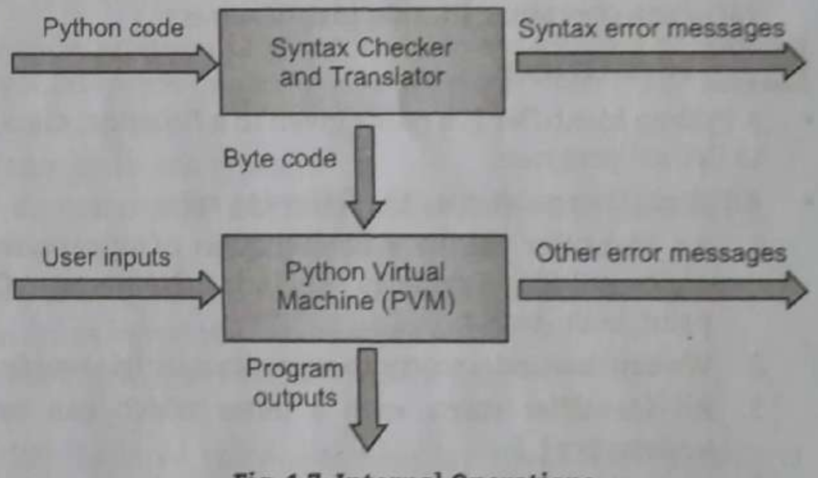


Fig. 1.7: Internal Operations

- The Python interpreter performs following tasks to execute a Python program:
  1. The interpreter reads a Python expression or statement, also called the source code, and verifies that it is well formed. In this step, as soon as the interpreter encounters such an error, it halts translation with an error message.
  2. If a Python expression is well formed, the interpreter then translates it to an equivalent form in a low-level language called byte code. When the interpreter runs a script, it completely translates it to byte code.
  3. This byte code is next sent to another software component, called the Python Virtual Machine (PVM), where it is executed. If another error occurs during this step, execution also halts with an error message.

1.2 PYTHON BUILDING BLOCKS

- In order to write any Python program, we must be aware of its structure, available keywords and data types also have some knowledge of variables, constants, identifiers and so on.
- Keywords, identifiers, variables etc., are the basic building blocks of Python programming. Python uses the character sets as the building block to form the basic program elements such as variables, keywords, constants, etc.

Scanned with CamScanner

1.2.1 Character Set

- The character set is a set of alphabets, letters, symbols and some special characters that are valid in Python programming language.
- Python uses the following character sets. These characters are submitted to the Python interpreter; they are interpreted or uniquely identified in various contexts, such as characters, identifiers, names or constants.
  1. **Lowercase English Letters:** a to z.
  2. **Uppercase English Letters:** A to Z.
  3. **Punctuation and Symbols:** "$", "!", etc.
  4. **Whitespace Characters:** An actual space (" "), as well as a newline, carriage return, horizontal tab, vertical tab, and a few others.
  5. **Non-Printable Characters:** Characters such as backspace, "\b", that cannot be printed literally in the way that the letter A can be printed.
  6. **Delimiter:** Delimiters are symbols that perform a special role in Python like grouping, punctuation and assignment. Following symbols and symbol combination uses as a delimiter in python:

     ( ) [ ] { } , : . ` = ; += -= *= /= %= **= &= |= ^= >>= <<=
- A program in Python contains a sequence of instructions. Python breaks each statement into sequence of lexical components/elements which are identify by the interpreter, known as tokens.
- A token is a smallest unit of the program. Python contains various types of tokens, such as keywords, variables, operators, literals, identifiers etc.

1.2.2 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`, `Salary10`, _PhoneNo.
- Example of invalid identifiers includes: `!count`, `4marks`, `%Loan`.

1.2.3 Keywords

- 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.

Scanned with CamScanner

- 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		

1.2.4 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 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 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.

Scanned with CamScanner

- 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 = "Yogesh"
>>> z = x + y
>>> print(x); print(y); print(name); Pring (z)
10
5
Yogesh
15
```

1.2.5 Literals

- 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
```

Scanned with CamScanner

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. Example: 100	Integers of unlimited size followed by lower-case or uppercase L. Example: 87032845L	Real numbers with both integer and fractional part. Example: 26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of complex number. 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 contain 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) Directory:

- 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 (,).

Scanned with CamScanner

Example: For literal collections.

```
# create list
>>> numbers=[1,2,3,4,5,6,7]
>>> print(numbers)
# create tuples
>>> list=('a','b','c')
>>> print(list1)
# 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'>
```

1.2.6 Indentation

- Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
- 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. 1.8).

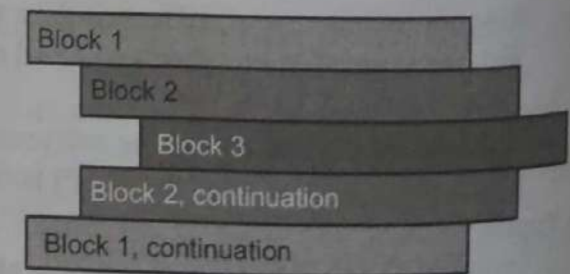


Fig. 1.8: Indentation in Python

- 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.

Example 1: For indentation.

```
>>> for i in range (1,11):
        print(i)
```

Output:

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

Example 2: For indentation in python.

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

Output:

```
1
2
3
4
5
```

1.2.7 Commenting in Python

- Comments are meant for computer programmers for better understanding a program. Python interpreter ignores the comment in the program.

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: For multiline comment.

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

1.3 PYTHON ENVIRONMENT SETUP (INSTALLATION AND WORKING OF IDE)

- 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. 1.9).

Fig. 1.9: Home Page

Scanned with CamScanner

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

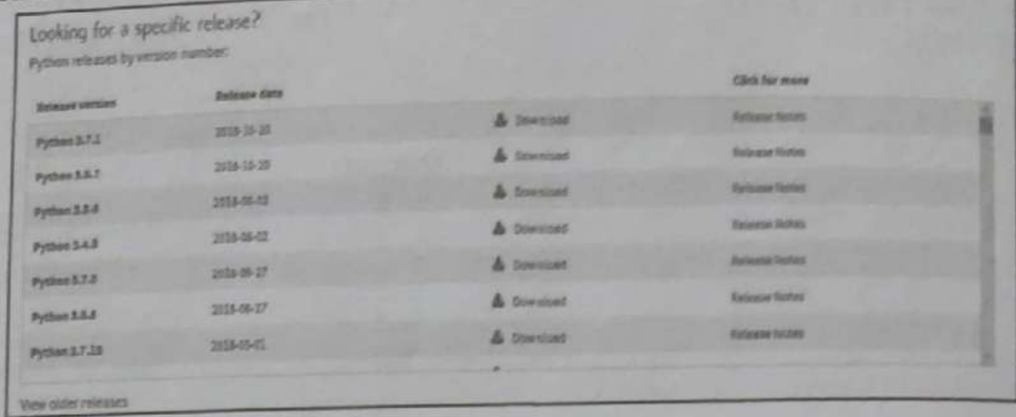


Fig. 1.10: Python Release Versions

Step 3 : Once Python version is demanded (e.g. 3.7.1.)

Open the python 3.7.1 version pack and double click on it to start installation.

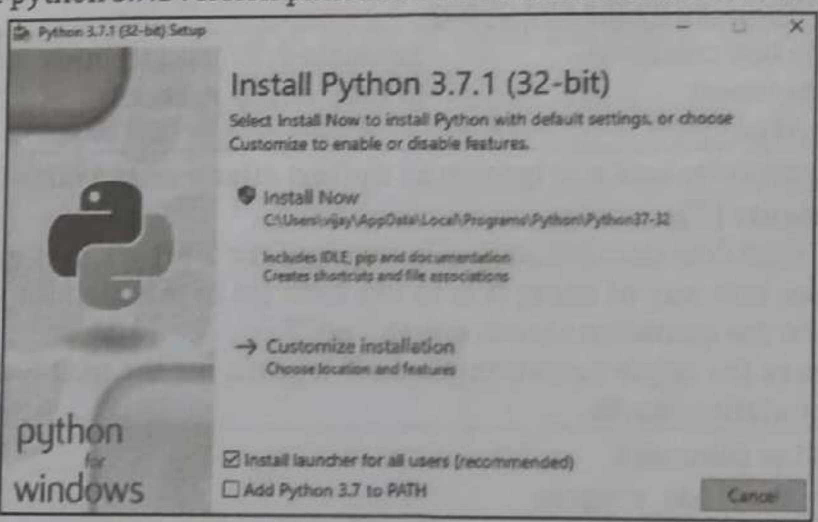


Fig. 1.11

Step 4 : Click on "install now" option for installation.

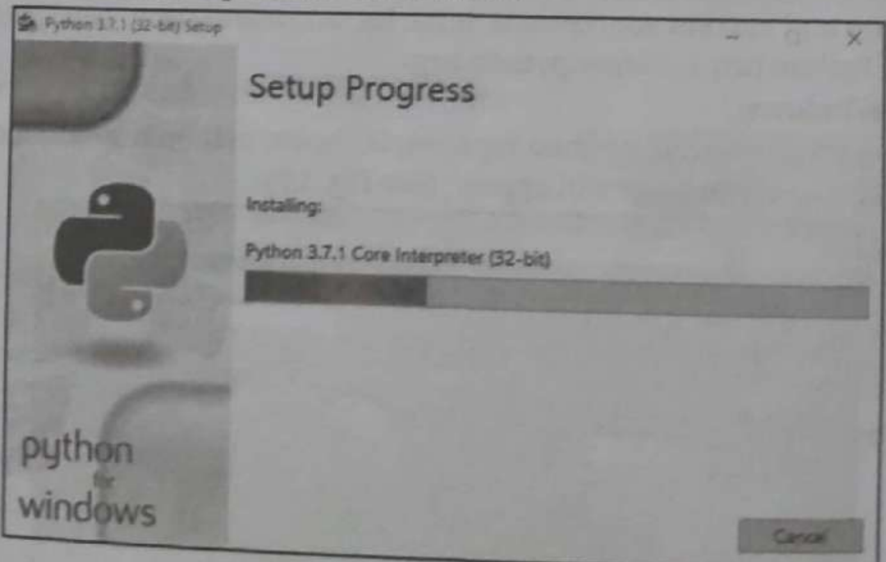


Fig. 1.12

Scanned with CamScanner

Step 5 : After complete the installation close the windows.

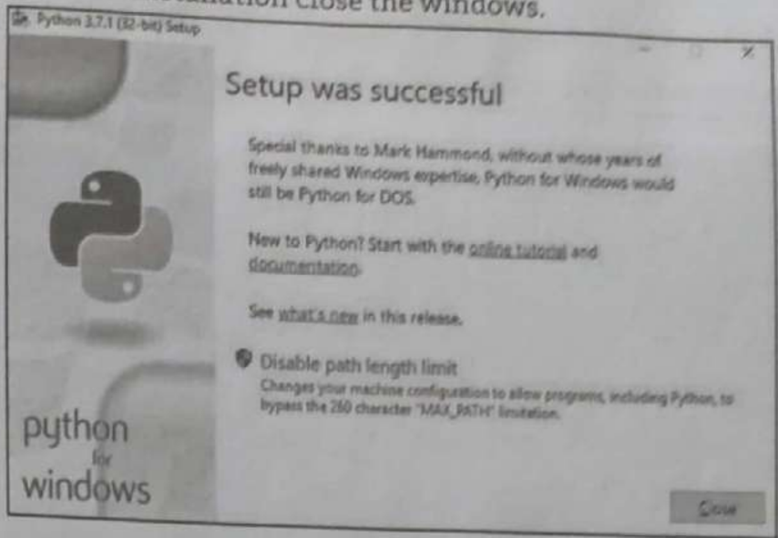


Fig. 1.13

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. 1.14).

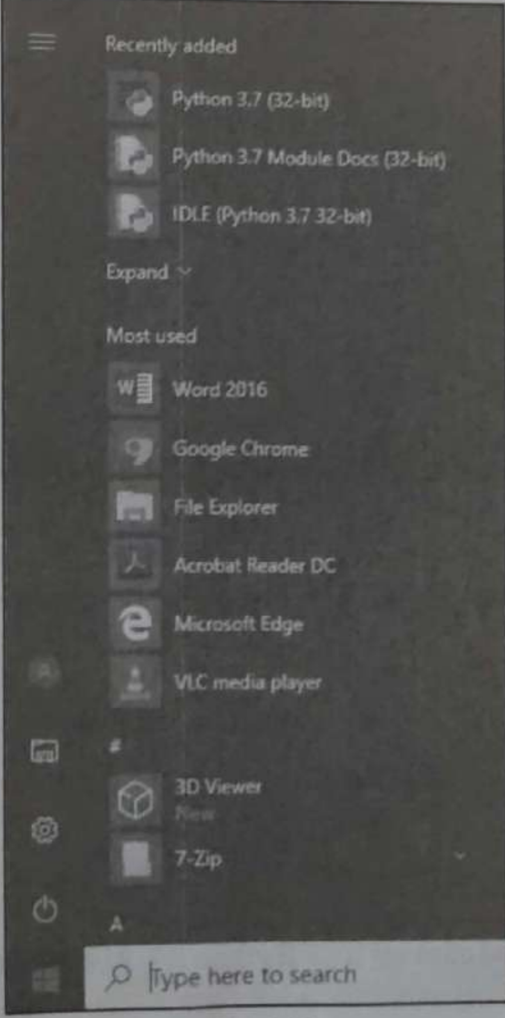


Fig. 1.14

Scanned with CamScanner

Step 2 : Click on All Programs and then click on Python 3.7 (32 bit) as shown in Fig. 1.14. We will see the Python interactive prompt in Python command line.

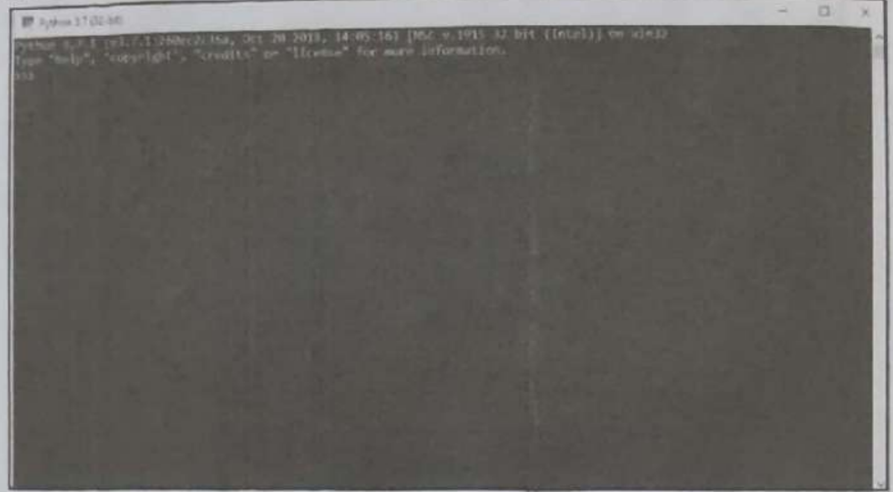


Fig. 1.15

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. 1.16.

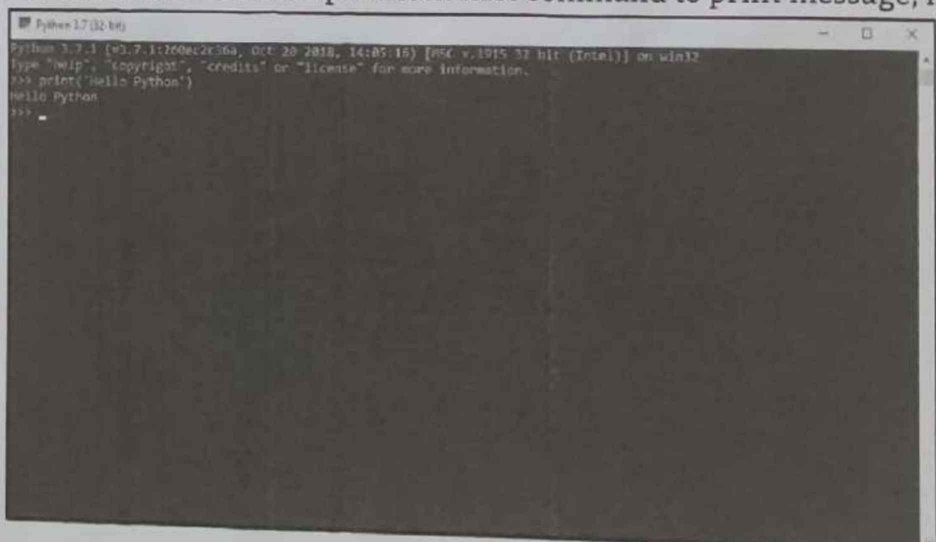


Fig. 1.16

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 3.7, 32-bit) options.

Scanned with CamScanner

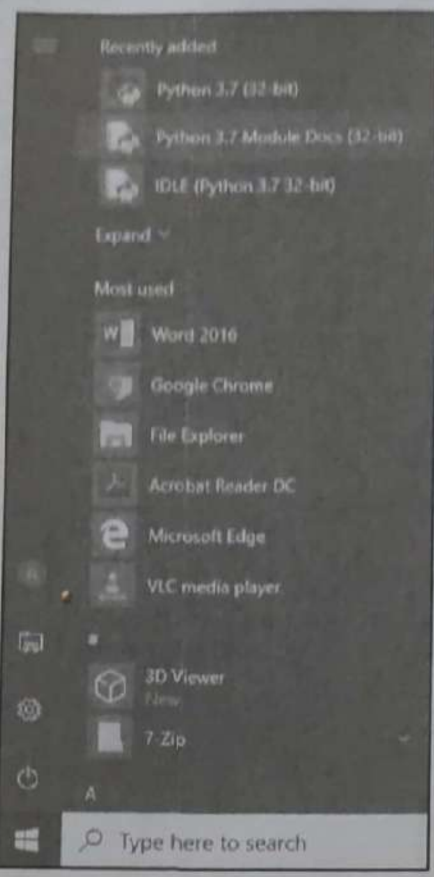


Fig. 1.17

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

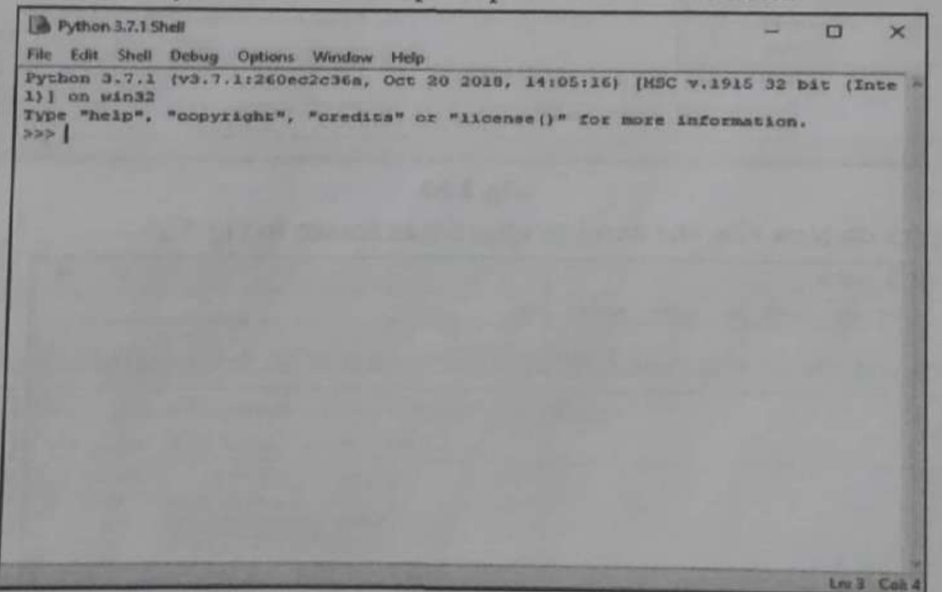


Fig. 1.18

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.

Scanned with CamScanner

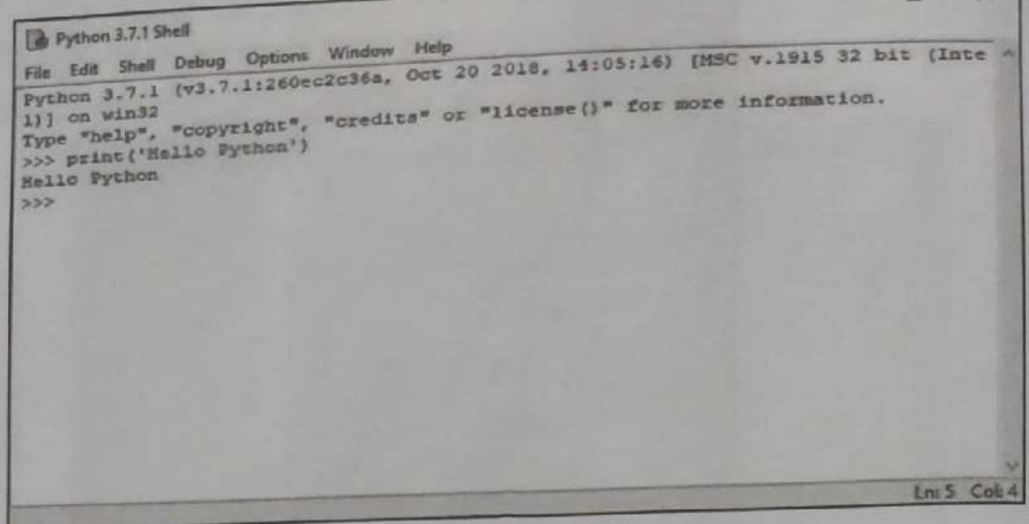


Fig. 1.19

Executing Python Programs Scripts:

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

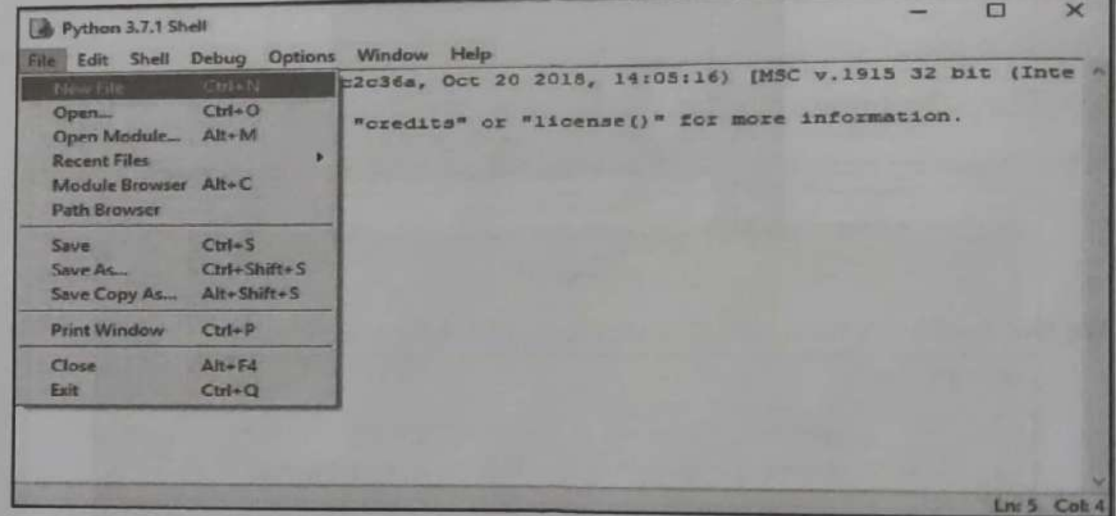


Fig. 1.20

- As soon as we click on New File, the window appears as shown in Fig. 1.21.

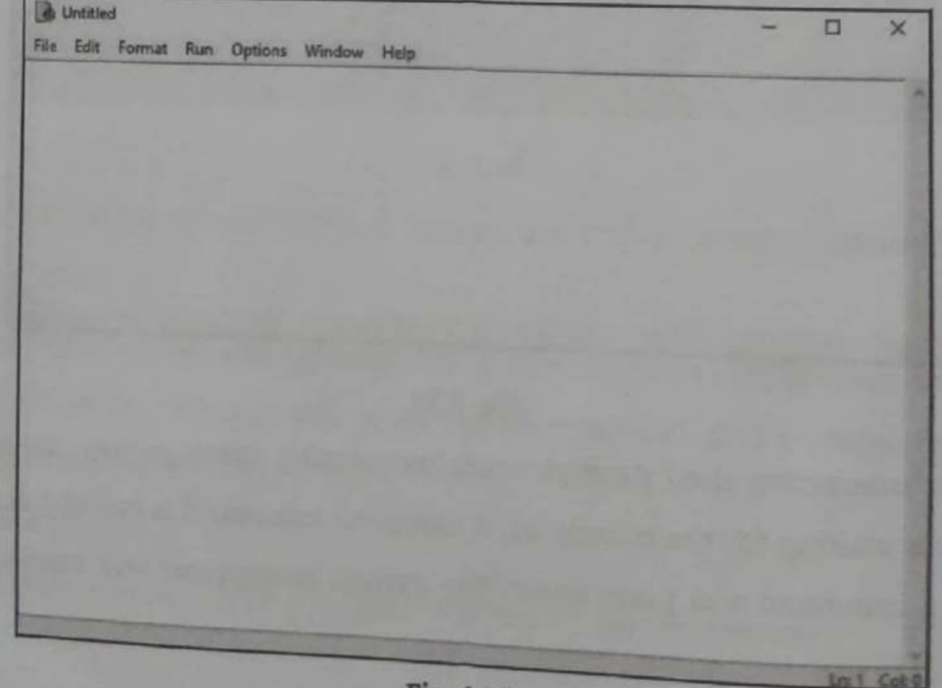


Fig. 1.21

Scanned with CamScanner

- Write Python program in script mode, (See Fig. 1.22).

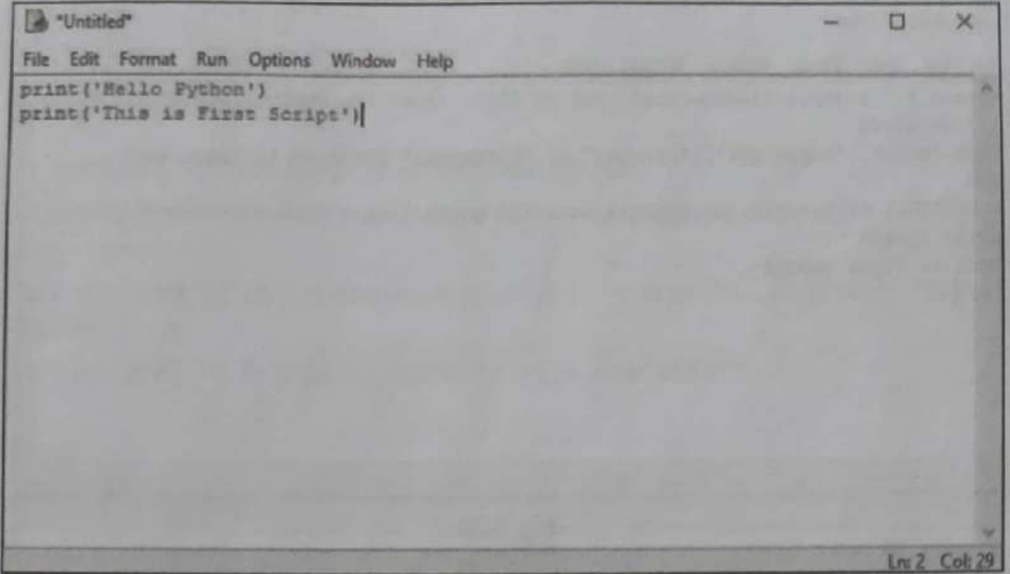


Fig. 1.22

- 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.

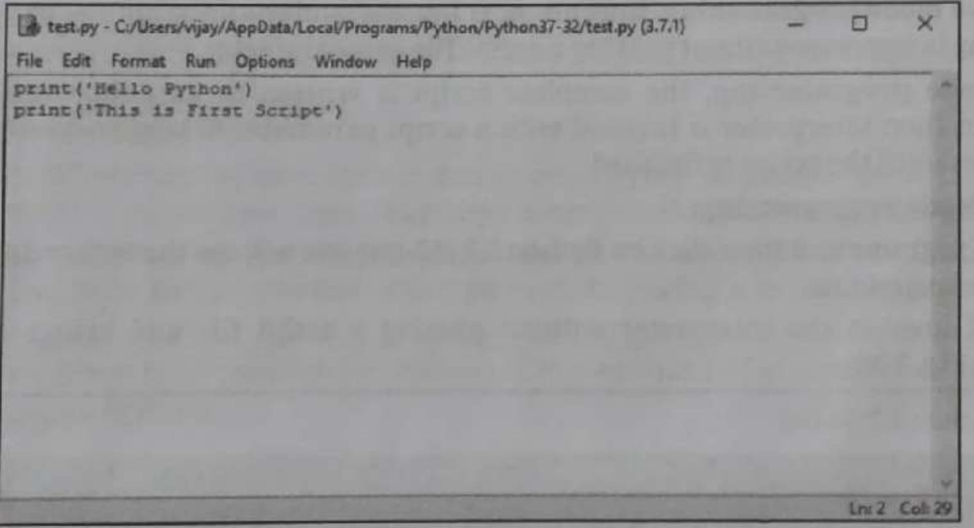


Fig. 1.23

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

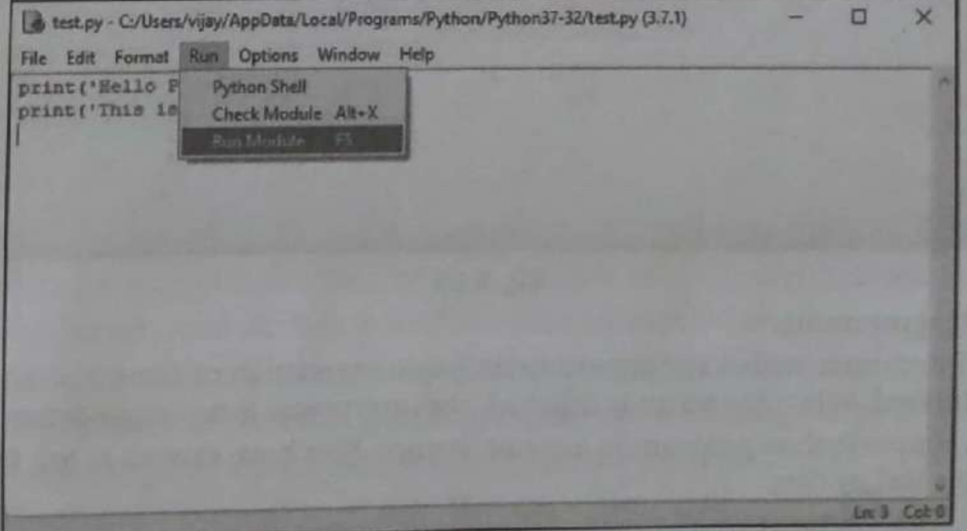


Fig. 1.24

Scanned with CamScanner

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

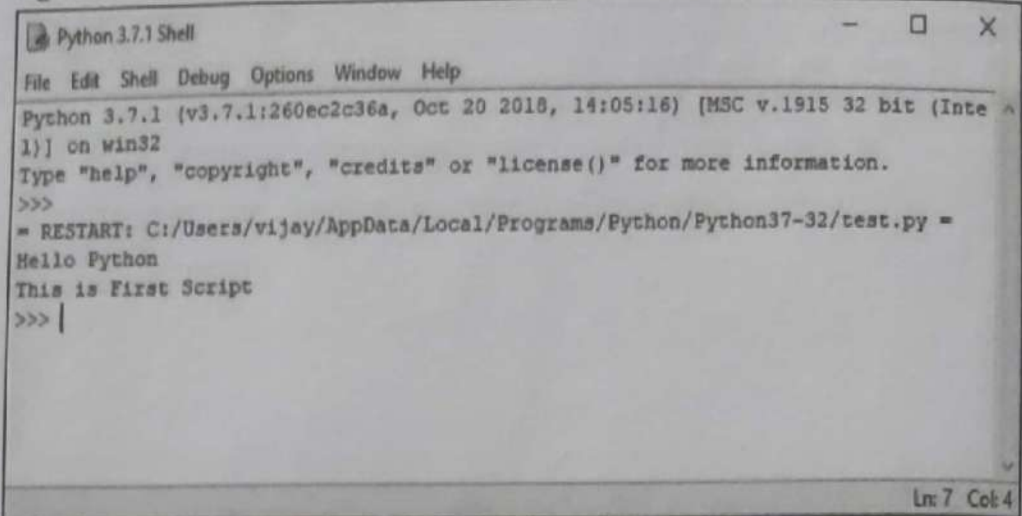


Fig. 1.25

1.4 RUNNING SIMPLE PYTHON SCRIPTS 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 3.7 (32-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. 1.26).

```
Python 3.7 (64-bit)
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.19
16 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>>
>>>
>>>
>>> print("Welcome Python")
Welcome Python
>>>
```

Fig. 1.26

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!"
```

Scanned with CamScanner

- 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!
```

1.5 PYTHON 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.

Declaration and Use of Data Types:

- One of the main differences between Python and strongly-typed languages like C, C++ or Java is the way it deals with types. In strongly-typed languages every variable must have a unique data type.
- For example, if a variable is of type integer, solely integers can be saved in the variable. In Java or C, every variable has to be declared before it can be used. Declaring a variable means binding it to a data type.
- Declaration of variables is not required in Python. If there is need of a variable, we think of a name and start using it as a variable.
- In the following line of code, we assign the value 42 to a variable:

```
i = 42
```

- The equal "=" sign in the assignment shouldn't be seen as "is equal to". It should be "read" or interpreted as "is set to", meaning in our example "the variable i is set to 42". Now we will increase the value of this variable by 1:

```
>>> i = i + 1
>>> print i
43
>>>
```

- 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.
  2. **String:** Represents text characters, special symbols or alphanumeric data.
  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).

Scanned with CamScanner

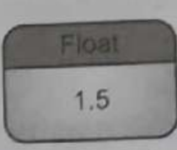
1.5.1 Numbers Data Type

- Number data types store numeric values. Number objects are created when we assign a value to them.
- Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex in Python.

1. **Integers:**
- 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.

2. **Floating Point Numbers:**
- 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.

3. **Complex Numbers:**
- 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.
- For example, in C=–1–5.5j, the complex number is –1–5.5j and is assigned to the variable C. Hence, the Python interpreter takes the data type of the variable C as a complex type.



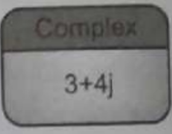


Fig. 1.27: Types of Numbers Data Type

- We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

1. **Integers (int Data Type):**
- 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
>>> b=-10
>>> b
-10
```

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

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

- In Python 3, there is no limit to how long an integer value can be.It can grow to have as many digits as the computer's memory space allows.
- 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

Scanned with CamScanner

Example: Integers in binary, octal and hexadecimal formats.

```
>>> print(0b10111011) # binary number
187
>>> print(0o10)       # octal number
8
>>> print(0xFF)       # hexadecimal number
255
```

2. **Boolean (bool Data Type):**

- In addition, Boolean is a sub-type of integers. The simplest build-in type in Python is the bool type, it represents the two values namely, False and True. Internally the true value is represented as 1 and false is 0.
- The bool type can only take the two values True or False, which are 1 or 0 in numeric context. For example, a = 18 > 5 assign True value to a and creates a as a Boolean variable.

Example: For bool data type.

```
>>> x=True
>>> type(x)
<class 'bool'>
>>> y=False
>>> type(y)
<class 'bool'>
>>> 3==4
False
>>> 2<3
True
```

3. **Floating-Point/Float Numbers (Float Data Type):**

- 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
```

4. **Complex Numbers (Complex Data Type):**

- 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
```

Scanned with CamScanner

1.5.2 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"'
>>>
```

- We can convert almost any object in Python to a string using a type constructor called str() function.

Example: For string with str().

```
>>> S=str(42)
>>> S
'42'
>>> type(S)
<class 'str'>
>>>
```

Accessing the String:

- 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'
```

Scanned with CamScanner

```
>>> 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'
>>>
```

nd Join a String:

split() function in Python breaks a string into individual letters. Use split() method to chop up a ng into a list of substrings, around a specified delimiter. The outcome of split() methods gives us t.

mple: For split() function.

```
>>> s="Python programming is easy"
>>> s
'Python programming is easy'
>>> l=s.split()                 # split() without any argument
>>> l
['Python', 'programming', 'is', 'easy']
>> s="Python,programming,is,easy"
>> s
Python,programming,is,easy'
>> l=s.split(',')                # split() with a argument
>> l
'Python', 'programming', 'is', 'easy']
>>
> type(l)
lass 'list'>
```

oin() method to join the list back into a string, with a specified delimiter in between. The me of join() method gives us a string.

ple: For join() function/method.

```
>> l
Python', 'programming', 'is', 'easy']
>
> type(l)
ass 'list'>
 s='  '.join(l)
 s
thon programming is easy'
 s='.'.join(l)
 s
thon programming is easy'
```

Scanned with CamScanner

String Built-in Methods:

- String objects also have several useful methods to report various characteristics of the string, such as whether it consists of digits or alphabetic characters or is all uppercase or lowercase.

Sr. No.	String Operation	Explanation	Example
1.	+	Adds two strings together.	`X = "hello"+"world"`
2.	*	Replicates a string.	`X = " "*20`
3.	upper	Converts a string to uppercase.	`x.upper()`
4.	lower	Converts a string to lowercase.	`x.lower()`
5.	title	Capitalizes the first letter of each word in a string.	`x.title()`
6.	find, index	Searches for the target in a string.	`x.find(y) x.index(y)`
7.	rfind, rindex	Searches for the target in a string, from the end of the string.	`x.rfind(y) x.rindex(y)`
8.	startswith, endswith	Checks the beginning or end of a string for a match.	`xstartswith(y) x.endswith(y)`
9.	replace	Replaces the target with a new string.	`x.replace(y,z)`
10.	strip, rstrip, lstrip	Removes whitespace or other characters from the ends of a string.	`x.strip()`
11.	encode	Converts a Unicode string to a bytes object.	`x.encode("utf_8")`

1.5.3 List Data Type

- List is an ordered sequence of items. It is one of the most used datatype 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 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']
```

Scanned with CamScanner

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]
```

nd Strings:

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

ample: For conversion of string to a list.

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

Tuple Data Type

le is an ordered sequence of items same as list. The only difference is that tuples are immutable. les once created cannot be modified.

les are used to write-protect data and are usually faster than list as it cannot change amically. It is defined within parentheses ( ) where items are separated by commas (,).

ple data type in python programming is similar to a list data type, which also contains rogeneous items/elements.

nple: For tuple.

```
>> a=(10,'abc',1+3j)
>> a
10, 'abc', (1+3j))
> a[0]

> a[0]=20                           # tuples are immutable/unchangeable
aceback (most recent call last):
File "<pyshell#12>", line 1, in <module>
  a[0]=20
eError: 'tuple' object does not support item assignment
```

ctionary

ary is an unordered collection of key-value pairs. It is the same as the hash table type.

er of elements in a dictionary is undefined, but we can iterate over the following:

key

value

Scanned with CamScanner

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'])
>>>
```

.6 INPUT AND OUTPUT IN PYTHON PROGRAMMING

Input means the data entered by the user of the program. In python, the input() function is used to accept an input from a user. The raw_input ( ) function available for Input on older version.

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"
```

Scanned with CamScanner

Additional Programs:

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
```

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 Rectange= ",area)
```

Output:

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

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
```

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
```

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
```

Scanned with CamScanner

```
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

1. What is Python programming language?
2. Give short history for Python.
3. Enlist applications for Python programming.
4. What are the features of Python?
5. List any four editors used for Python programming.
6. 'Python programming language is interpreted and intractive' comment this sentence.
7. How to run python scripts? Explain in detail.
8. What is interpreter? How it works?
9. Explain the following features of Python programming:
   (i) Simple
   (ii) Platform independent
   (iii) Interactive
   (iv) Object Oriented.
10. Explain about the need for learning Python programming and its importance.
11. Describe the internal working of Python diagrammatically.
12. Write in brief about characters set of Python.
13. Write in brief about any five keywords in Python.
14. Write the steps to install Python and to run Python code.
15. What is the role of indentation in Python?
16. How to comment specific line(s) in Python program?
17. What is variable? What are the rules and conventions for declaring a variables?
18. What are the various data types available in Python programming.
19. What are four built-in numeric data types in Python? Explain.
20. What is the difference between interactive mode and script mode of Python.
21. Python has developed as an open source project. Justify this statement.
22. Define the following terms:
    (i) Identifier
    (ii) Literal
    (iii) Data type
    (iv) Tuple
    (v) List.
23. Explain dictionary data type in detail.

■■■

Scanned with CamScanner

2...

Python Operators and Control Flow Statements

Chapter Outcomes...

- Write simple Python program for the given arithmetic expressions.
- Use different types of operators for writing the arithmetic expressions.
- Write a 'Python' program using decision making structure for two-way branching to solve the given problem.
- Write a 'Python' program using decision making structure for multi-way branching to solve the given problem.

Learning Objectives...

- To understand Basic Operators in Python Programming
- To learn Control Flow and Conditional Statements in Python
- To study Looping in Python Programming
- To understand Loop Manipulation Statements in Python

2.0 INTRODUCTION

- 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. The Python language provides a rich set of operators.
- The operator and operand when combined to perform a certain operation, it becomes an expression. For example, in expression x + y, x and y are the variables (operands) and the plus (+) sign is the operator that specifies the type of operation performed on the variables.
- 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, loop/iteration control and jump or loop control.

Scanned with CamScanner

2.1 OPERATORS

- An operator is a symbol which specifies a specific action. An operator is a special symbol that tells the interpreter to perform a specific operation on the operands. The operands can be literals, variables or expressions.
- An operand is a data item on which operator acts. Operators are the symbol, which can manipulate the value of operands. Some operators require two operands while others require only one.
- Consider the expression 5 + 2 = 7. Here, 5 and 2 are called the operands and + is called the operator.

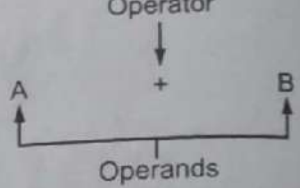


Fig. 2.1: Concept of Operator and Operands

- 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. +, –, ~ are called unary operators.

Syntax: `operator operand`

Example:

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

- The invert (~) operator returns the bitwise inversion of long integer arguments. Inversion of x can be computed as ~ (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:

- 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 types of operators supported by Python includes Arithmetic operators, Assignment operators, Relational or Comparison operators, Logical operators, Bitwise operators, Identity operators and Membership operators.

Scanned with CamScanner

2.1.1 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 30
2.	-	Subtraction	Subtracts the value of the right operand from the value of the left operand.	>>> b-a 10
3.	*	Multiplication	Multiplies the value of the left and right operand.	>>> a*b 200
4.	/	Division	Divides the value of the left operand by the right operand.	>>> b/a 2.0
5.	**	Exponent	Performs exponential calculation.	>>> a**2 100
6.	%	Modulus	Returns the remainder after dividing the left operand with the right operand.	>>> a%b 10
7.	//	Floor Division	Division of operands where the solution is a quotient left after removing decimal numbers.	>>> b//a 2

2.1.2 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

Scanned with CamScanner

2.1.3 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.	== (Equality Operator)	If the values of two operands are equal, then the condition becomes true.	`>>> (a==b)` `False`
2.	!= (Not Equality Operator)	If values of two operands are not equal, then condition becomes true.	`>>> (a!=b)` `True`
3.	> (Greater Than Operator)	If the value of left operand is greater than the value of right operand, then condition becomes true.	`>>> (a>b)` `False`
4.	< (Less Than Operator)	If the value of left operand is less than the value of right operand, then condition becomes true.	`>>> (a<b)` `True`
5.	>= (Greater Than Equal to Operator)	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	`>>> (a>=b)` `False`
6.	<= (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)` `True`

2.1.4 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 (Logical AND Operator)	If both the operands are true then condition becomes true.	(a and b) is False.
2.	OR (Logical OR Operator)	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
3.	NOT (Logical NOT Operator)	Used to reverse the logical state of its operand.	Not(a and b) is True.

2.1.5 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.

Scanned with CamScanner

- Following table shows bitwise operators assume a=10 (1010) and b=4 (0100).

Sr. No.	Operator	Description	Example
1.	& (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.	| (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.	^ (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
4.	~ (Bitwise Ones Complement Operator)	It is unary operator and has the effect of 'flipping' bits i.e. opposite the bits of operand.	~a= ~ 1010 = 0101
5.	<< (Bitwise Left Shift Operator)	The left operand's value is moved left by the number of bits specified by the right operand.	a<<2 = 1010 << 2 =101000 = 40
6.	>> (Bitwise Right Shift Operator)	The left operand's value is moved right by the number of bits specified by the right operand.	a>>2 = 1010 >> 2 =0010 = 2

- 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

2.1.6 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.	>>> a=3
			>>> b=3
			>>> print(a is b)
			True

2.	is not	Return false, if the variables on either side of the operator point to the same object and true otherwise.	>>> a=3
			>>> b=3
			>>> print(a is not b)
			False

Scanned with CamScanner

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 tables but not identical.

2.1.7 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 it item is not in list or in sequence	`>>> x="Hello World"` `>>> print('H' in x)` `True`
2.	not in	True if value is not found in list or in sequence, and false it item is in list or in sequence.	`>>> x="Hello World"` `>>> print("Hello" not in x)` `False`

Scanned with CamScanner

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
```

2.1.8 Python Operator Precedence and Associativity

- 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 identity 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 Pythons 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
```

Scanned with CamScanner

- 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.	= %= /= //= -= += *= **=	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

2.2 CONTROL FLOW

- A program's control flow is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops and loop manipulation (jump) 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 is refer to statement sequencing in a program to get desire result. In this section, we introduce statements that allow us to change the control flow, using logic about the values of program variables.

2.3 CONDITIONAL STATEMENTS/DECISION MAKING STATEMENTS

- Decision making statements are those block of statements that executes a particular block according to the Boolean expression evaluation, (True or False).
- In Python, conditional statements are used to determine if a specific condition is met by testing whether a condition is True or False. Conditional statements are used to determine how a program is executed.
- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
- Decision structures evaluate multiple expressions which produce True or False as outcome. We need to determine which action to take and which statements to execute if outcome is True or False otherwise.
- Python decision making statements includes, if statements, if-else statements, nested-if statements, multi-way if-elif-else statements.

2.3.1 if Statement

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

Syntax:

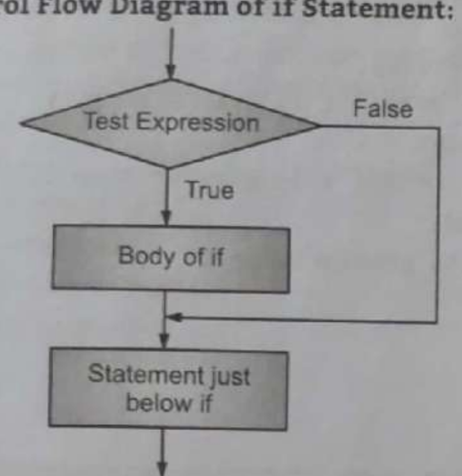
```
if condition:
    statement(s)
```

OR

```
if condition:
    block
```

- Note that indentation is required for statements which are under if condition.

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
```

2.3.2 if-else Statement

- if statements executes when the conditions following if is true and it does nothing when the condition is false. The if-else statement takes care of a true as well as false condition.

Syntax:

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

OR

```
if condition:
   if_block
else:
   else_block
```

Scanned with CamScanner

Control flow Diagram of if-else Statement:

Test Expression
False
True
Body of if
Body of else
Statement just below if

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 of 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!
```

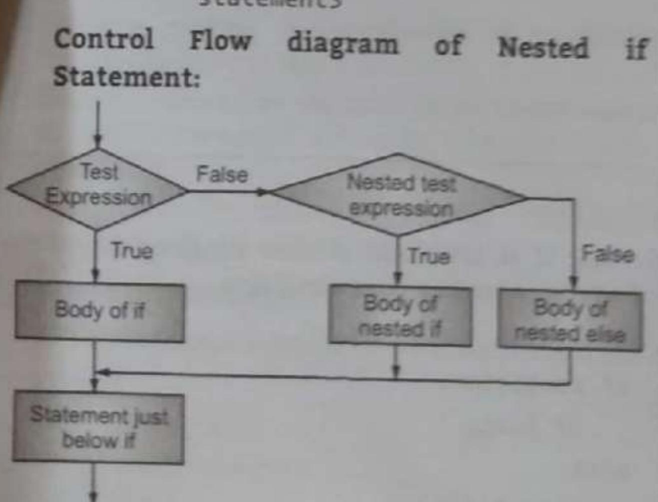
2.3.3 Nested if Statements

- 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 than b and c")
else:
    print("a is less than b and c")
print("End of Nested if")
```

Output:

```
a is greater than b and c
End of Nested if
```

Scanned with CamScanner

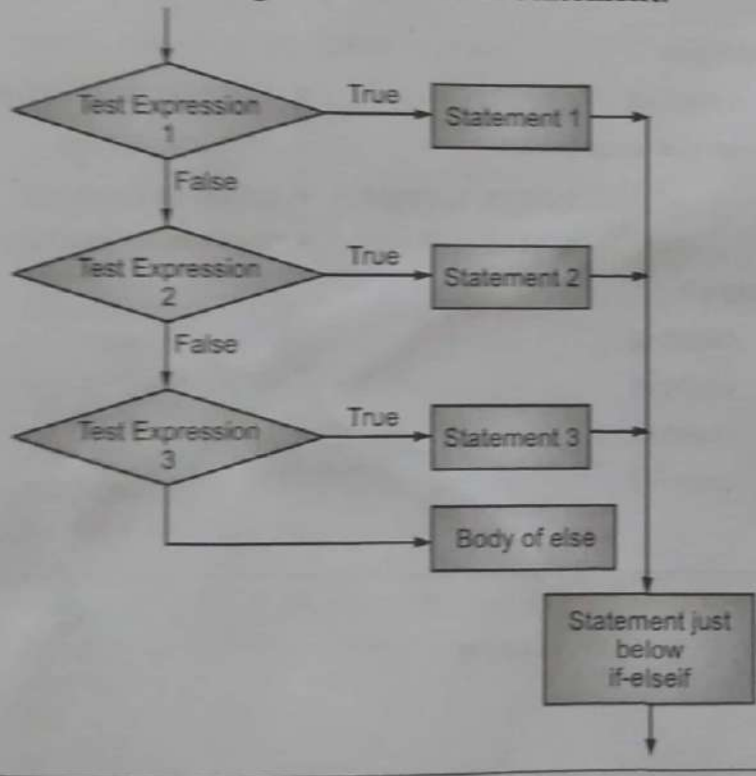
2.3.4 Multi-way if-elif-else (Ladder) Statements

- Here, a user can decide among multiple options. 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: 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)
```

Scanned with CamScanner

Output:

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

2.4 LOOPING IN PYTHON

- 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 execute 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.

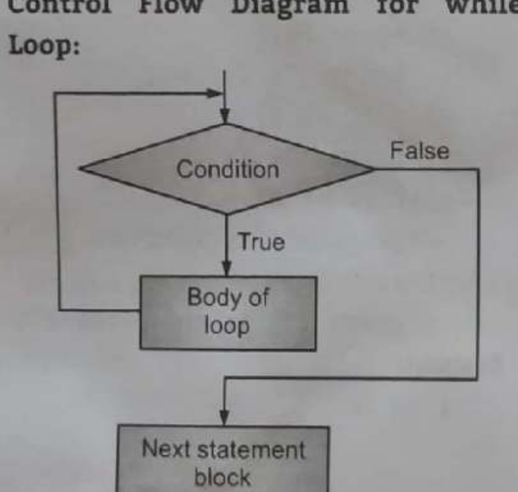
2.4.1 while Loop

- It is the most basic looping statement in Python programming. It executes a sequence of statements repeatedly as long as a condition is true.

Syntax:

```
while expression:
    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
```

Scanned with CamScanner

Output:

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

Additional Programs:

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
```

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
```

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
```

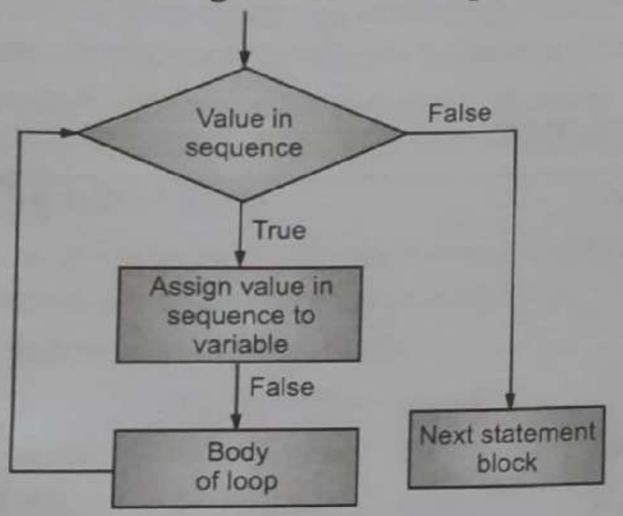
Scanned with CamScanner

2.4.2 for Loop

- In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is similar to for each loop in other languages.
- The Python for loop iterates through a sequence of objects, i.e. it iterates through each value in a sequence, where the sequence of object holds multiple items of data stored one after another.

Syntax:
```
for var in sequence:
    Statements
```

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
```

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

Scanned with CamScanner

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.

Additional Programs:

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
```

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
```

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)
```

Scanned with CamScanner

put:

```
of Even numbers=  110
```

Nested for and while Loops

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

ax:

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

AND

```
while expression:
    while expression:
        statement(s)
statement(s)
```

mple: Nested for loop.

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

put:

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

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
```

rint() will be executed at the end of inner for while loop.

l Programs:

am to print following pyramid:

```
4
4 5
```

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()
```

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()
```

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()
```

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()
```

Scanned with CamScanner

2.5 LOOP MANIPULATION/LOOP CONTROL STATEMENTS

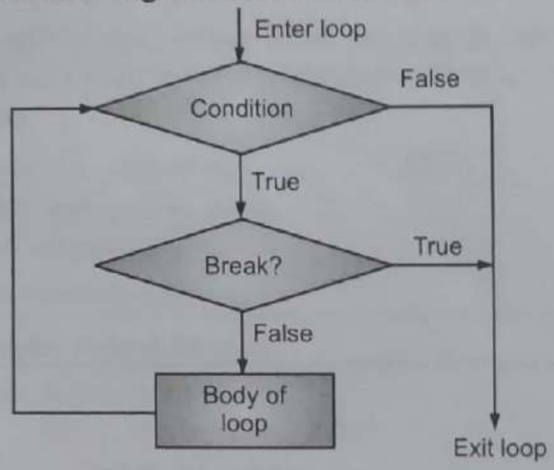
- In Python, loop statements give us a way execute the block of code repeatedly. But sometimes, we may want to exit a loop completely or skip specific part of loop when it meets a specified condition. It can be done using loop control mechanism.
- Loop control statements change execution from its normal 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.

2.5.1 break Statement

- The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

  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:

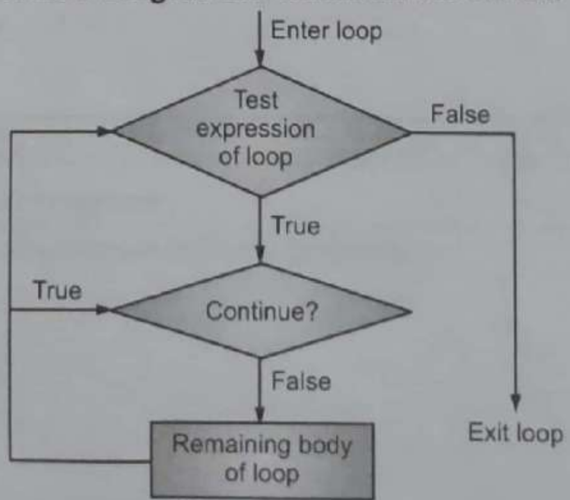
```
i=1
i=2
i=3
i=4
```

2.5.2 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
```

Scanned with CamScanner

2.5.3 Pass Statement

- It is used when a statement is required syntactically but we do not want any command or code to execute. A pass statement in Python also refers to as a null statement.
- The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet

  Syntax: pass

Example: 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
```

Additional Programs:

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
```

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
```

Scanned with CamScanner

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
```

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
```

Scanned with CamScanner

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
elifch == '-':
    result = num1 - num2
elifch == '*':
    result = num1 * num2
elifch == '/':
    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
```

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
```

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")
```

Scanned with CamScanner

Output:

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

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]
```

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)
```

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)
```

Scanned with CamScanner

11. Print patterns of (*) using loop.

(i)
```
for i in range(0, 5):
    for j in range(0, i+1):
        print("* ",end="")
    print("\r")
```
Output:
```
*
* *
* * *
* * * *
* * * * *
```
(ii)
```
for i in range(0, 5):
    num = 1
    for j in range(0, i+1):
        print(num, end=" ")
    num = num + 1
    print("\r")
```
Output:
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Practice Questions

1. What is operator? Which operators used in Python?
2. What is meant by control flow of a program?
3. Define the terms: (i) Loop, (ii) Program, (iii) Operator, (iv) Control flow.
4. What are the different loops available in Python?
5. What happens if a semicolon (;) is placed at the end of a Python statement?
6. Explain about different logical operators in Python with appropriate examples.
7. Explain about different relational operators in Python with examples.
8. Explain about membership operators in Python.
9. Explain about Identity operators in Python with appropriate examples.
10. Explain about arithmetic operators in Python.
11. List different conditional statements in Python.
12. What are the different nested loops available in Python?
13. What are the different loop control (manipulation) statements available in Python? Explain with suitable examples.
14. Explain if-else statement with an example.
15. Explain continue statement with an example.
16. Explain use of break statement in a loop with example.
17. Predict output and justify your answer: (i) -11%9 (ii) 7.7//7 (iii) (200-70)*10/5 (iv) 5*1**2.

Scanned with CamScanner

3...

Data Structures in Python

Chapter Outcomes...

- Write Python program to use and manipulate lists for the given problem.
- Write Python program to use and manipulate tuples for the given problem.
- Write Python program to use and manipulate sets for the given problem.
- Write Python program to use and manipulate dictionaries for the given problem.

Learning Objectives...

- To learn Concepts of Lists like Defining, Accessing, Deleting, Updating and so on
- To understand Basic List Operations and Built-in List Functions
- To Study Concept of Tuples with Accessing, Deleting, Updating Values in Tuples
- To study Basic Tuple Operations and Built-in Tuple Functions
- To understand Concepts of Sets with Accessing, Deleting, Updating Values in Sets
- To understand Basic Set Operations and Built-in Set Functions
- To know Concepts of Dictionaries with Accessing, Deleting, Updating Values in Dictionary
- To study Basic Dictionary Operations and Built-in Dictionaries Functions

3.0 INTRODUCTION

- A data structure is a specialized format for organizing and storing data, so that various operations can be performed on it efficiently and easily.
- Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate and systematic manner/way. There are four data structures in Python namely, list, tuple, dictionary and set.
- A data structure that stores an ordered collection of items in Python is called a list. In other words, a list holds a sequence of items or elements.
- Similar to a list, the tuple is an ordered sequence of items. A set is an unordered collection of unique items in Python. A dictionary in Python is an unordered collection of key value pairs.
- The data types that are most used in Python are strings, tuples, lists and dictionaries. These are collectively called as data structures.

3.1 LISTS

- A list in Python is a linear data structure. 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.

Scanned with CamScanner

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** is: `<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. It is also a dynamic data type. Lists can contain any sort of object. It can be numbers, strings, tuples and even other lists.

3.1.1 Creating a List

- 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: Creating an empty list.

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

Example: 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: 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: 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: Create a list using inbuilt range() function.

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

Scanned with CamScanner

Example: Create a list with in-built characters A, B and C.

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

3.1.2 Accessing Values in 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.

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 Deleting Values in List

- Python provides many ways in which the elements in a list can be deleted.
  1. 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 stacks (First In, Last Out 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. 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]
```

Scanned with CamScanner

```
>>> 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. 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']
>>>
```

3.1.4 Updating Lists (Change or Add Elements to a List)

- List 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]
```

Scanned with CamScanner

- 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 use 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]
```

Scanned with CamScanner

4. The concatenation operation in Python programming is used to combine the elements/items of 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. 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]
```

3.1.5 Basic List Operations (Indexing and Slicing)

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

3.1.5.1 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 them 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.

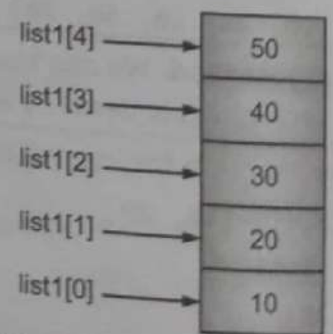


Fig. 3.1: List has Five Elements with Index 0 to 4

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]
```

Scanned with CamScanner

```
>>> 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.

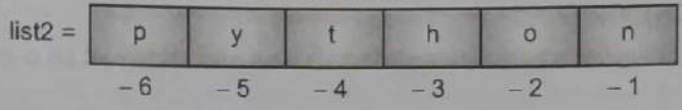


Fig. 3.2: List with Negative Index

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
```

3.1.5.2 List 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]`

Scanned with CamScanner

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 o and step size with -1
[50, 40, 30, 20]
```

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
@ #$ &
```

3.1.6 Built-in Functions and Methods for List

- Python provide certain method and function work with list. Following table shows different built-in functions (which one can perform on list) with their description and example.

Sr. No.	Built-in Function	Description	Example

1.	len(list)	It returns the length of the list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> len(list1)
			5

contd. ...

2.	max(list)	It returns the item that has the maximum value in a list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> max(list1)
			5

3.	sum(list)	Calculates sum of all the elements of list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> sum(list1)
			15

4.	min(list)	It returns the item that has the minimum value in a list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> min(list1)
			1

5.	list(seq)	It converts a tuple into a list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> list(list1)
			[1, 2, 3, 4, 5]

Methods of List Class:

Sr. No.	Methods	Description	Example

1.	list.append(item)	It adds the item to the end of the list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> list1.append(6)
			>>> list1
			[1, 2, 3, 4, 5, 6]

2.	list.count(item)	It returns number of times the item occurs in the list.	>>> list1
			[1, 2, 3, 4, 5, 6, 3]
			>>> list1.count(3)
			2

3.	list.extend(seq)	It adds the elements of the sequence at the end of the list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> list2
			['A', 'B', 'C']
			>>> list1.extend(list2)
			>>> list1
			[1, 2, 3, 4, 5, 'A', 'B', 'C']

4.	list.index(item)	It returns the index number of the item. If item appears more than one time, it returns the lowest index number.	>>> list1=[1,2,3,4,5,3]
			>>> list1
			[1, 2, 3, 4, 5, 3]
			>>> list1.index(3)
			2

5.	list.insert(index,item)	It inserts the given item onto the given index number while the elements in the list take one right shift.	>>> list1
			[1, 2, 3, 4, 5, 3]
			>>> list1.insert(2,7)
			>>> list1
			[1, 2, 7, 3, 4, 5, 3]

contd. ...

Scanned with CamScanner

6.	list.pop(item=list[-1])	It deletes and returns the last element of the list.	>>> list1
			[1, 2, 7, 3, 4, 5, 3]
			>>> list1.pop()
			3
			>>> list1.pop(2)
			7

7.	list.remove(item)	It deletes the given item from the list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> list1.remove(3)
			>>> list1
			[1, 2, 4, 5]

8.	list.reverse()	It reverses the position (index number) of the items in the list.	>>> list1
			[1, 2, 3, 4, 5]
			>>> list1.reverse()
			>>> list1
			[5, 4, 3, 2, 1]

9.	list.sort()	Sorts items in the list.	>>> list1
			[1, 3, 2, 4, 5]
			>>> list1.sort()
			>>> [1, 3, 2, 4, 5]

10.	list.sort([func])	It sorts the elements inside the list and uses compare function if provided.	>>> list1
			[1, 3, 2, 5, 4]
			>>> list1.sort()
			>>> list1
			[1, 2, 3, 4, 5]

3.2 TUPLES

- A tuple is also a linear data structure. 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 you can not change their values.

Difference between Tuples and Lists:

1. A values in the list can be replaced with another any time after its creation, whereas in tuples, the values in it cannot be replaced with another, once tuples are created.
2. Lists allows us to add new items to it, but tuple does not allow us to add new items, once it is created.
3. We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) data types.

Scanned with CamScanner

4. Since, tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
5. Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
6. Tuples can be used as values in sets whereas lists can not.
7. Some tuples can be used as dictionary keys (specifically, tuples that contain immutable values like strings, numbers, and other tuples). Lists can never be used as dictionary keys, because lists are not immutable.

- 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.

3.2.1 Creating Tuple

- 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: `<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)`
- 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])
```

Scanned with CamScanner

```
>>> 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'>
```

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
```

Scanned with CamScanner

3.2.2 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.

3.2.3 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: 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: 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)
```

- Or 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)
```

Scanned with CamScanner

3.2.4 Updating Tuple

- 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)
```

3.2.5 Tuple Operations

- A set of operations can be apply 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)
```

Scanned with CamScanner

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
>>>
```

3.2.6 Build-in Functions and Methods of Tuple

- Following table built-in tuple functions in Python programming.

Sr. No.	Function	Description	Example

1.	len(tuple)	Gives the total length of the tuple.	>>> tup1
			(1, 2, 3)
			>>> len(tup1)
			3

2.	max(tuple)	Returns item from the tuple with max value.	>>> tup1
			(1, 2, 3)
			>>> max(tup1)
			3

3.	min(tuple)	Returns item from the tuple with min value.	>>> tup1
			(1, 2, 3)
			>>> min(tup1)
			1

4.	zip(tuple1,tuple2)	It zips elements from two tuples into a list of tuples.	>>> tup1=(1,2,3)
			>>> tup2=('A','B','C')
			>>> tup3=zip(tup1,tup2)
			>>> list(tup3)
			[(1, 'A'), (2, 'B'), (3, 'C')]

5.	tuple(seq)	Converts a list into tuple.	>>> tuple1 = (1, 2, 3, 4, 5)
			>>> list1 = list (tuple1)
			>>> list1
			[1,2,3,4,5]

Scanned with CamScanner

Methods of Tuple:

Sr. No.	Function	Description	Example

1.	count()	Returns the number of times a specified value occurs in a tuple	>>> tup1
			(1, 2, 3, 2, 4)
			>>> tup1.count(2)
			2

2.	index()	Searches the tuple for a specified value and returns the position of where it was found	>>> tup1
			(1, 2, 3)
			>>> tup1.index(3)
			2

3.3 SETS

- The set data structure in Python programming is implemented to support mathematical set operations. Set is an unordered collection of unique items. Items stored in a set are not kept in any particular order.
- 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.

3.3.1 Accessing Values in Sets

- 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'}
>>>
```

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}
>>>
```

Scanned with CamScanner

2. 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}
>>>
```

3.3.2 Deleting Values in 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. pop() method removes random item from a set and returns it.
  4. 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()
```

3.3.3 Updating Set

- 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}
```

Scanned with CamScanner

```
>>> 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 accepts 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'}
```

3.3.4 Basic Set Operations

- Union, Intersection, Difference and Symmetric Difference are some operations which are performed on sets.

1. **Union:**

- Union operation performed on two sets returns all the elements from both the sets.
- The union of A and B is 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}
```

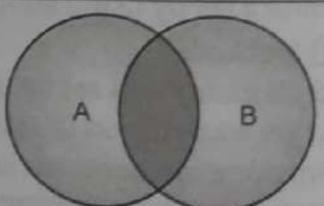


Fig. 3.3: Union Operation

2. **Intersection:**

- 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.

Scanned with CamScanner

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}
```

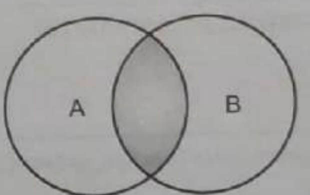


Fig. 3.4: Intersection Operation

3. **Difference:**

- 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}
```

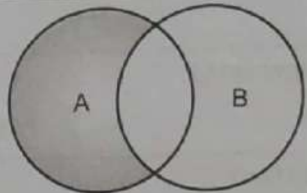


Fig. 3.5: Difference Operation

4. **Symmetric Difference:**

- Symmetric Difference operation performed by using ^ 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}
```

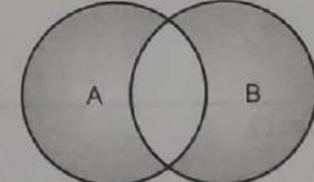


Fig. 3.6: Symmetric Difference Operation

3.3.5 Built-in Functions and Methods for Set

Methods for Set:

Sr. No.	Method	Description	Syntax
1.	union()	Return a new set containing the union of two or more sets	set.union(set1,set2...)
2.	intersection()	Returns a new set which is the intersection of two or more sets	set.intersection(set1,set2...)
3.	intersection_update()	Removes the items from this set that are not present in other sets	set.intersection_update(set1,set2...)
4.	difference()	Returns a new set containing the difference between two or more sets	set.difference(set1,set2...)
5.	difference_update()	Removes the items from this set that are also included in another set	set.difference_update(set1,set2...)

contd. ...

Scanned with CamScanner

6.	`symmetric_difference()`	Returns a new set with the symmetric differences of two or more sets	`set.symmetric_difference(set)`
7.	`symmetric_difference_update()`	Modify this set with the symmetric difference of this set and other set	`set.symmetric_difference_update(set)`
8.	`isdisjoint()`	Determines whether or not two sets have any elements in common	`set.isdisjoint(set)`
9.	`issubset()`	Determines whether one set is a subset of the other	`set.issubset(set)`
10.	`issuperset()`	Determines whether one set is a superset of the other	`set.issuperset(set)`
11.	`add(item)`	It adds an item to the set. It has no effect if the item is already present in the set.	`set.add(set)`
12.	`discard(item)`	It removes the specified item from the set.	`set.discard(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.	`set.remove(set)`
14.	`pop()`	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.	`set.pop(set)`
15.	`update()`	Updates the set with the union of itself and others.	`set.update(set)`

- 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.	`all()`	Return True if all elements of the set are true (or if the set is empty).	`>>> all (A) True`
2.	`any()`	Return True if any element of the set is true. If the set is empty, return False.	`>>> any (A) True`
3.	`len()`	Return the length (the number of items) in the set.	`>>> len(A)` `4`
4.	`max()`	Return the largest item in the set.	`>>> max(A)` `7`
5.	`min()`	Return the smallest item in the set.	`>>> min(A)` `1`
6.	`sorted()`	Return a new sorted list from elements in the set(does not sort the set itself).	`>>> sorted(A)` `[1, 3, 6, 7]`
7.	`sum()`	Retrun the sum of all elements in the set.	`>>> sum(A)` `17`

Scanned with CamScanner

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
```

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}`

Scanned with CamScanner

- 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 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={}
>>> dict1                                          # Empty dictionary
{}
>>> dict2={1:"Orange", 2:"Mango", 3:"Banana"} # Dictionary with integer keys
>>> dict2
{1: 'Orange', 2: 'Mango', 3: 'Banana'}
>>> dict3={"name":"vijay", 1:[10,20]}
>>> dict3                                          # Dictionary with mixed keys
{'name': 'vijay', 1: [10, 20]}
```
  2. Python provides a build-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.

Scanned with CamScanner

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/Items From Dictionary

- We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.
- The method, popitem() can be used to remove and return an arbitrary item (key, value) form 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 is a mutable data type.
- The dictionary are 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
```

Scanned with CamScanner

```
{'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 Directory

- 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 a 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
```

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:
    1. 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'}
```

Scanned with CamScanner

2. 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'
```

3.4.6 Built-in Functions and Methods for Dictionary

- 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.	clear()	Removes all the elements from the dictionary.	dict={1:'Vijay',2:'Amar',3:'Santosh'}
			>>> dict
			{1: 'Vijay', 2: 'Amar', 3: 'Santosh'}
			>>> dict.clear()
			>>> dict
			{}

2.	copy()	Returns a copy of the dictionary.	>>> dict={1: 'Vijay', 2: 'Amar', 3: 'Santosh'}
			>>> X=dict.copy()
			>>> X
			{1: 'Vijay', 2: 'Amar', 3: 'Santosh'}

3.	fromkeys()	The fromkeys() method creates a new dictionary with default value for all specified keys.
		If default value is not specified, all keys are set to None.
			>>> dict=dict.fromkeys(['Vijay','Meenakshi'],'Author')
			>>> dict
			{'Vijay': 'Author', 'Meenakshi': 'Author'}
			>>>

4.	get()	Returns the value of the specified key.	>>> dict1={'name':'vijay','age':40}
			>>> dict1.get('name')
			'vijay'

5.	items()	Returns a list containing the a tuple for each key value pair.	dict{1: 'Vijay', 2: 'Amar', 3: 'Santosh'} >>> for i in dict.items(): print(i) (1, 'Vijay') (2, 'Amar') (3, 'Santosh')
6.	keys()	Returns a list containing the dictionary's keys.	dict={1:'Vijay',2:'Amar',3:'Santosh'} >>> dict.keys() dict_keys([1, 2, 3])
7.	pop()	Removes the element with the specified key.	dict={1:'Vijay',2:'Amar',3:'Santosh'} >>> print(dict.pop(2)) Amar

contd. ...

Scanned with CamScanner

8.	`popitem()`	Removes the last inserted key-value pair.	`dict={1:'Vijay',2:'Amar',3:'Santosh'}`
			`>>> dict.popitem()`
			`(3, 'Santosh')`

9.	`setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value.	`>>> dict`
			`{2: 'Amar', 3: 'Santosh'}`
			`>>> dict.setdefault(1,'Vijay')`
			`>>> dict`
			`{2: 'Amar', 3: 'Santosh', 1: 'Vijay'}`

10.	`update()`	Updates the dictionary with the specified key-value pairs.	`>>> 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'}`

11.	`values()`	Returns a list of all the values in the dictionary.	`>>> dict`
			`{1: 'Vijay', 2: 'Amar', 3: 'Santosh'}`
			`>>> dict.values()`
			`dict_values(['Vijay', 'Amar', 'Santosh'])`

Built-in Functions of Directory:

Sr. No.	Function	Description	Example

1.	`all()`	Return True if all keys of the dictionary are true (or if the dictionary is empty).	`>>> dict`
			`{1: 'Vijay', 2: 'Amar', 3: 'Santosh'}`
			`>>> all(dict)`
			`True`

2.	`any()`	Return True if any key of the dictionary is true. If the dictionary is empty, return False.	`>>> dict={}`
			`>>> any(dict)`
			`False`

3.	`len()`	Return the length (the number of items) in the dictionary.	`>>> dict`
			`{1: 'Vijay', 2: 'Amar', 3: 'Santosh'}`
			`>>> len(dict)`
			`3`

4.	`sorted()`	Return a new sorted list of keys in the dictionary.	`>>> dict1`
			`{2: 'Amar', 1: 'Vijay', 4: 'Umesh', 3: 'Amar'}`
			`>>> sorted(dict1)`
			`[1, 2, 3, 4]`

5.	`type()`	Returns the type of the passed variable.	`dict = {'Name': 'Zara', 'Age': 7};` `print "Variable Type: %s" % type (dict)`

Strings:

- A string is a linear data structure in Python. Strings are the group of characters.
- In Python, a string type object is a sequence (left-to-right order) of characters. Strings start and end with single or double quotes. Python strings are immutable means string can not be modified according to requirement.

Scanned with CamScanner

- Single and double quoted strings are same and we can use a single quote within a string when it is surrounded by double quote and vice versa.

Example: For string.

```
>>> a='Hello Python'
>>> a
'Hello Python'
>>> a="Hello Python"
>>> a
'Hello Python'
>>> type(a)
<class 'str'>
```

str Class:

- Strings are objects of the str class. We can create a string using the constructor of str class

```
S1=str()              # creates an empty string object
S2=str("Hello")       # creates a string object for Hello
```

- An alternative wat to create a string object is by assigning a string value to a variable.

Example: For str class.

```
S1=""                 # create an empty string
S2="Hello"            # equivalent to S2=str("Hello")
```

str() Function:

- The str function is used to convert a number into a string.

Example: For str().

```
>>> a=10
>>> type(a)
<class 'int'>
>>>str(a)             # converting int number to string
'10'
```

Python Built-in Functions for String:

Sr. No.	Function	Example

1.	len() function return the number of characters in a string.	`>>> a="PYTHON"`
		`>>> len(a)`
		`6`

2.	min() function return the smallest character in a string.	`>>> a="PYTHON"`
		`>>> min(a)`
		`'H'`

3.	max() function return the largest character in a string.	`>>> a="PYTHON"`
		`>>> max(a)`
		`'Y'`

Deleting Entire String:

- Deletion of entire string is possible with the use of del keyword.

Example: For deletion of string.

```
>>> a="PYTHON"
>>> a
'PYTHON'
>>> del a
```

Scanned with CamScanner

```
>>> a
Traceback (most recent call last):
 File "<pyshell#9>", line 1, in <module>
  a
NameError: name 'a' is not defined
```

+ operator (String Concatenation):

- The concatenation operator (+) is used to join two strings.

Example: For + operator in string.

```
>>> "Hello" + "Python"
'HelloPython'
>>> s1="Hello"
>>> s2="Python"
>>> s1+s2
'HelloPython'
```

*** Operator (String Multiplication):**

- The multiplication (*) operator is used to concatenate the same string multiple times, it is called repetition operator.

Example: for * operator in string.

```
>>> s1="Hello "
>>> s2=3*s1
>>> s2
'Hello Hello Hello '
>>> "**"*5
'**********'
>>> a="**"
>>> a*5
'**********'
>>>
```

String Traversal (Traversing String with for Loop and while Loop):

- Traversal is a process in which we access all the elements of the string one by one using for and while loop.

Example: Traversing using for loop.

```
>>> s="Python Programming"
>>> for ch in s:
        print(ch,end="")

Python Programming
>>> for ch in range(0,len(s),2):
        print(s[ch],end="")
```

Example: Traversing using while loop.

```
>>> s="Python Programming"
>>> index=0
>>> while index<len(s):
        print(s[index],end="")
        index=index+1
```

Output:

```
Python Programming
```

Scanned with CamScanner

```
>>> a
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

+ operator (String Concatenation):

- The concatenation operator (+) is used to join two strings.

Example: For + operator in string.

```
>>> "Hello" + "Python"
'HelloPython'
>>> s1="Hello"
>>> s2="Python"
>>> s1+s2
'HelloPython'
```

*** Operator (String Multiplication):**

- The multiplication (*) operator is used to concatenate the same string multiple times, it is called repetition operator.

Example: for * operator in string.

```
>>> s1="Hello "
>>> s2=3*s1
>>> s2
'Hello Hello Hello '
>>> "**"*5
'**********'
>>> a="**"
>>> a*5
'**********'
>>>
```

String Traversal (Traversing String with for Loop and while Loop):

- Traversal is a process in which we access all the elements of the string one by one using for and while loop.

Example: Traversing using for loop.

```
>>> s="Python Programming"
>>> for ch in s:
        print(ch,end="")

Python Programming
>>> for ch in range(0,len(s),2):
        print(s[ch],end="")
```

Example: Traversing using while loop.

```
>>> s="Python Programming"
>>> index=0
>>> while index<len(s):
        print(s[index],end="")
        index=index+1
```

Output:

```
Python Programming
```

Scanned with CamScanner

Immutable Strings:

- Strings are immutable which means that we cannot change any element of a string. If we want to change an element of a string, we have to create a new string.

Example: For immutable string.

```
>>>str="Python"
>>>str
'Python'
>>>str[0]="H"
Traceback (most recent call last):
 File "<pyshell#33>", line 1, in <module>
str[0]="H"
TypeError: 'str' object does not support item assignment
```

- Here, when we try to change the 0th index of string to a character "H", but the python interpreter generates an error. The solution to this problem is to generate a new string rather than change the old string.

Example:

```
>>>str="Python"
>>> str1='H'+str[1:]
>>> str1
'Hython'
```

- Consider the following two similar strings:

```
Str1="Python"
Str2="Python"
```

Here, Str1 and Str2 have the same content. Thus python uses one object for each string which has the same content. Both Str1 and St2 refers to the same string object, whereas Str1 and Str2 have the same ID number.

Example:

```
>>> Str1="Python"
>>> Str2="Python"
>>> id(Str1)
54058464
>>> id(Str2)
54058464
```

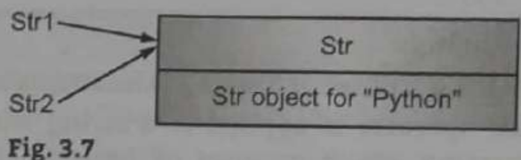


Fig. 3.7

String Indices and Accessing String Elements:

- Strings are arrays of characters and elements of an array can be accessed using indexing. Indices start with 0 from left side and –1 when starting from right side.

```
S1="Hello Python"
```

Character	H	e	l	l	o		P	y	t	h	o	n
Index (From Left)	0	1	2	3	4	5	6	7	8	9	10	11
Index (From Right)	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

Example:

```
>>> s1="Hello Python"
>>> print(s1[0])          # print first character
H
>>> print(s1[11])         # print last character
n
>>> print(s1[-12])        # print first character
```

Scanned with CamScanner

```
H
>>> print(s1[-1])          # print last character
n
>>> print(s1[15])          # out of index range
Traceback (most recent call last):
 File "<pyshell#65>", line 1, in <module>
  print(s1[15])
IndexError: string index out of range
>>>
```

'in' and 'not in' Operator in Strings:

- The 'in' operator is used to check whether a character or a substring is present in a string or not. The expression returns a Boolean value.

Example: For in and out operators.

```
>>> s1="Hello Python"
>>> "Hello" in s1
True
>>> "java" in s1
False
>>> "Hello" not in s1
False
>>> 'H' in s1
True
>>> 'B' in s1
False
>>> 'Py' in s1
True
>>>
```

String Slicing:

- A piece or subset of a string is known as slice. To cut a substring from a string is called string slicing.
- Slice operator is applied to a string with the use of square braces([]). Operator [min] will give a substring which consists of letters between m and n indices, including letter at index m but excluding that at n, i.e. letter from m^{th} index to $(n-1)^{th}$ index.
- Here two indices are used separated by a colon (:). A slice 3:7 means indices characters of 3rd, 4th, 5th and 6th positions. The second integer index i.e. 7 is not included. You can use negative indices for slicing.

Syntax:

```
stringname[start_index:end_index]
stringname[start_index:end_index:step_size]
```

Example:

```
>>> s1="Hello Python"
>>> s1[0:4]
'Hell'
>>> s1[:2]
'He'
>>> s1[2:]
'llo Python'
```

Scanned with CamScanner

```
>>> s1[0:len(s1):2]
'HloPto'
>>> s1[-1:0:-1]
'nohtyPolle'
>>> s1[:-1]
'Hello Pytho'
```

String Comparison:

- Operators such as ==, <, >, <=, >= and != are used to compare the strings.

Example: For string competition.

```
>>> S1="abcd"
>>> s2="abcd"
>>> s1>s2
True
```

- **Note:** Python compares the numeric value of each character. The ASCII value of 'a' is 97 and ASCII numeric value of 'A' is 65. It means 97>65, thus it returns True. However, character by character comparison goes on till the end of the string.

```
>>> s1="abcd"
>>> s2="abcd"
>>> s1==s2
True
>>>s1="abcd"
>>> s2="defg"
>>> s1>s2
False
```

Special Characters in String:

- The backslash (\) character is used to introduce a special character or escape character. It converts difficult-to-type characters into a string.

Sr. No.	Escape Character	Meaning	Example
1.	\ newline	Ignored	`>>> print("line1\line2\line3")` `line1line2line3`
2.	\\	Backslash (\)	`>>> print("This is a backslash (\\) mark")` `This is a backslash (\) mark`
3.	\'	Single Quote (')	`>>> print("These are \'single quote\'")` `These are 'single quote'`
4.	\"	Double Quote (")	`>>> print("These are \"double quote\"")` `These are "double quote"`
5.	\a	ASCII Bell (BELL)	`>>>  print("\a")`
6.	\b	ASCII Backspace (BS)	`>>> print("Hello \b World!")` `Hello World!`
7.	\f	ASCII Formfeed (FF)	`>>> print("Hello \f World!")` `Hello  World!`

8.	\n	ASCII Linefeed (LF), Newline	`>>> print("This is a first line \n This is a second line")`
			`This is a first line`
			`This is a second line`

contd. ...

Scanned with CamScanner

9.	\r	ASCII Carriage return (CR)	`>>> print("Hello \r World!")` `Hello   World!`
10.	\t	ASCII Horizontal tab (TAB)	`>>> print("This is tav \t key")` `This is tav    key`

11.	\v	ASCII Vertical tab (VT)	`>>> print("Hello \v World!")`
			`Hello`
			` World!`

12.	\ooo	ASCII Character with octal value ooo	`>>> print("\110\145\154\154\157\40\127 \157\162\154\144\41")` `Hello World!`
13.	\xhhh...	ASCII Character with hex value hh...	`>>> print("\x48\x65\x6c\x6c\x6f\x20\x57 \x6f\x7\x6c\x64\x21")` `Hello World!`

String Formatting Operator:

- The string in Python have a unique built-in operation, the % operator (modulo). This is also called the string formatting operator. This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

  Example:

```
>>> print("My name is %s and weight is %d kg!"%('Vijay',60))
My name is Vijay and weight is 60 kg!
```

Sr. No.	Format Symbol	Conversion
1.	%c	Character.
2.	%s	string conversion via str() prior to formatting.
3.	%i	signed decimal integer.
4.	%d	signed decimal integer.
5.	%u	unsigned decimal integer.
6.	%o	octal integer.
7.	%x	hexadecimal integer (lowercase letters).
8.	%X	hexadecimal integer (UPPERcase letters).
9.	%e	exponential notation (with lowercase 'e').
10.	%E	exponential notation (with UPPERcase 'E').
11.	%f	floating point real number.
12.	%g	the shorter of %f and %e.
13.	%G	the shorter of %f and %E.

String Formatting Functions:

- Python includes the following built-in functions to manipulate strings.

Sr. No.	Method	Description	Example

1.	capitalize()	Makes the first letter of the string capital.	`>>> s1="python programming"`
			`>>> s1.capitalize()`
			`'Python programming'`

2.	center(width, fillchar)	Returns a space padded string with the original string centered to a total width columns.	`>>> s1="python programming"`
			`>>> print(s1.center(30,'*'))`
			`******python programming******`

contd. ...

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.	>>> s1="python programming"
			>>> s1.count('o')
			2
			>>> s1.count('o',5,18)
			1

4.	decode()	Decodes the string using the codec registered for encoding.	>>> s1
			'python Programming'
			>>> s1=s1.encode()
			>>> s1
			b'python Programming'
			>>> s1=s1.decode()
			>>> s1
			'python Programming'

5.	encode()	Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.	s1="python Programming" >>> s1.encode() b'python Programming'

6.	endswith(suf ix, beg, end)	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.	>>> s1="python programming is easy to learn."
			>>> s1.endswith('learn')
			False
			>>> s1.endswith('learn.')
			True
			>>> s1.endswith('is',7,26)
			False
			>>> s1.endswith('easy',7,26)
			True

7.	expandstab (tabsize)	It returns a copy of the string in which tab characters ie. '\t' are expanded using spaces, optionally using the given tabsize (default 8).	>>> s1="python\tprogramming"
			>>> s1
			'python\tprogramming'
			>>> s1.expandtabs()
			'python programming'
			>>> s1.expandtabs(16)
			'python programming'

8.	enumerate()	The enumerate() method adds counter to an iterable and returns it (the enumerate object).	>>> s1=['Orange','Mango','Banana','Pineapple']
			>>> s2=enumerate(s1)
			>>> print(list(s2))
			[(0, 'Orange'), (1, 'Mango'), (2, 'Banana'), (3, 'Pineapple')]

9.	find(str, beg, end)	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.	>>> s1="python programming"
			>>> s1.find("prog")
			7
			>>> s1.find("prog",5)
			7
			>>> s1.find("prog",10)
			-1

contd. ...

Scanned with CamScanner

10.	`index(str, beg, end)`	Same as find(), but raises an exception if str not found.	`>>> s1="python programming"`
			`>>> s1.index("prog")`
			`7`
			`>>> s1.index("prog",10)`
			`Traceback (most recent call last):`
			`File "<pyshell#64>", line 1, in <module>`
			`s1.index("prog",10)`
			`ValueError: substring not found`

11.	`isalnum()`	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.	`>>> s1="python3"`
			`>>> s1.isalnum()`
			`True`
			`>>> s1="1234"`
			`>>> s1.isalnum()`
			`True`
			`>>> s1="python programming"`
			`>>> s1.isalnum()`
			`False`

12.	`isalpha()`	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.	`>>> s1="python"`
			`#no space and digit in this string`
			`>>> s1.isalpha()`
			`True`
			`>>> s1="python3"`
			`>>> s1.isalpha()`
			`False`
			`>>> s1="python programming"`
			`>>> s1.isalpha()`
			`False`

13.	`isdigit()`	Returns true if string contains only digits and false otherwise.	`>>> s1="python programming"`
			`>>> s1.isdigit()`
			`False`
			`>>> s1="python3"`
			`>>> s1.isdigit()`
			`False`
			`>>> s1="12345"`
			`#only digit in string`
			`>>> s1.isdigit()`
			`True`

14.	`islower()`	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.	`>>> s1="python"`
			`>>> s1.islower()`
			`True`
			`>>> s1="Python"`
			`>>> s1.islower()`
			`False`

contd. ...

Scanned with CamScanner

15.	isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise.	>>> s1="12345"
			>>> s1.isnumeric()
			True
			>>> s1="\u00B2345"
			>>> s1.isnumeric()
			True
			>>> s1="python3"
			>>> s1.isnumeric()
			False

16.	isspace()	Returns true if string contains only whitespace characters and false otherwise.	>>> s1=' '
			>>> s1.isspace()
			True
			>>> s1="python programming"
			>>> s1.isspace()
			False
			>>> s1=' \t'
			>>> s1.isspace()
			True

17.	istitle()	Returns true if string is properly "titlecased" and false otherwise.	>>> s1="Python Programming"
			>>> s1.istitle()
			True
			>>> s1="Python programming"
			>>> s1.istitle()
			False
			>>> s1="PYTHON"
			>>> s1.istitle()
			False
			>>> s1="123 Is A Number"
			>>> s1.istitle()
			True

18.	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.	>>> s1="PYTHON PROGRAMMING"
			>>> s1.isupper()
			True
			>>> s1="Python Programming"
			>>> s1.isupper()
			False

Practice Questions

1. What is data structure? Which data structure used by Python?
2. How to define and access the elements of list?
3. What is list? How to create list?
4. What are the different operations that can be performed on a list? Explain with examples
5. Explain any two methods under lists in Python with examples.
6. Write a Python program to describe different ways of deleting an element from the given List.
7. What is tuple in Python? How to create and access it?
8. What are mutable and immutable types?

Scanned with CamScanner

9. Is tuple mutable? Demonstrate any two methods of tuple.
10. Write in brief about Tuple in Python. Write operations with suitable examples
11. Write in brief about Set in Python. Write operations with suitable examples.
12. Explain the properties of dictionary keys.
13. Explain directory methods in Python.
14. How to create directory in Python? Give example.
15. Write in brief about Dictionary in python. Write operations with suitable examples.
16. What is the significant difference between list and dictionary?
17. Compare List and Tuple.
18. Explain sort() with an example
19. How append() and extend() are different with reference to list in Python?
20. Write a program to input any two tuples and interchange the tuple variable.
21. Write a Python program to multiply two matrices
22. Write a Python code to get the following dictionary as output:
    {1:1, 3:9,5:25,7:49,9,81}
23. Write the output for the following:
    (i)
    ```
    >>>a=[1,2,3]
    >>>b=[4,5,6]
    >>> c=a+b
    ```
    (ii)
    ```
    >>>[1,2,3]*3
    ```
    (iii)
    ```
    >>>t=['a','b','c','d','e','f']
    >>>t[1:3]=['x','y']
    >>>print t
    ```
24. Give the output of Python code:
    ```
    Str='Maharashtra State Board of Technical Education'
    print(x[15::1])
    print(x[-10:-1:2])
    ```
25. Give the output of following Python code:
    ```
    t=(1,2,3,(4,),[5,6])
    print(t[3])
    t[4][0]=7
    print(t)
    ```
26. Write the output for the following if the variable fruit='banana':
    ```
    >>>fruit[:3]        o/p='ban'
    >>>fruit[3:]        o/p='ana'
    >>>fruit[3:3]       o/p=' '
    >>>fruit[:]         o/p='banana'
    ```
27. What is string? How to create it? Enlist various operations on strings.

■■■

4...

Python Functions, Modules and Packages

Chapter Outcomes...

- Use the Python standard functions for the giving problem.
- Develop relevant user defined functions for the given problem using the Python code.
- Write Python module for the given problem.
- Write Python package for the given problem.

Learning Objectives...

- To learn Basic Concepts of Functions
- To study use of Python Built-in Functions
- To understand User Defined Functions with its Definition, Calling, Arguments Passing etc.
- To study Scope of Variables like Global and Local
- To learn Module Concept with Writing and Importing Modules
- To study Python Built-in Modules like Numeric, Mathematical, Functional Programming Module
- To learn Python Packages with its Basic Concepts and User Defined Packages

4.0 INTRODUCTION

- Functions, modules and packages are all 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 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.
- 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 have 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.

Scanned with CamScanner

4.1 USE OF PYTHON BUILT-IN FUNCTIONS

- Functions are the self-contained block of statements that act like a program that performs specific task.
- The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.
- Python built-in (library) functions can be used to perform specific tasks. Some of these functions comes in category of mathematical functions and some are called type conversion functions and so on.

4.1.1 Type Data Conversion Functions

- Sometimes it's necessary to perform conversions between the built-in types. To convert between types we simply use the type name as a function.
- In addition, several built-in functions are supplied to perform special kinds of conversions. All of these functions return a new object representing the converted value.
- Python defines type conversion functions to directly convert one data type to another. Data conversion in Python can happen in following two ways:
  1. Either we tell the compiler to convert a data type to some other type explicitly, and/or
  2. The compiler understands this by itself and does it for us.

1. Python Implicit Data Type Conversion:

- Implicit conversion is when data type conversion takes place either during compilation or during run time and is handled directly by Python.

Example: For implicit data type conversion.

```
>>> a=10
>>> b=25.34
>>> sum=a+b
>>> print sum
35.34
>>>
```

- In the above example, an int value a is added to float value b, and the result is automatically converted to a float value sum without having to tell the compiler. This is the implicit data conversion.
- In implicit data conversion the lowest priority data type always get converted to the highest priority data type that is available in the source code.

2. Python Explicit Data Type Conversion:

- Explicit conversion also known as type casting where we force an expression to be of a specific type.
- While developing a program, sometimes it is desirable to convert one data type into another. In Python, this can be accomplished very easily by making use of built-in type conversion functions.
- The type conversion functions result into a new object representing the converted value. A list of data type conversion functions with their respective description is given in following table.

Sr. No.	Function	Description	Example
1.	int(x [,base])	Converts x to an integer. base specifies the base if x is a string.	x=int('1100',base=2)=12 x=int('1234',base=8)=668
2.	long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.	x=long('123'base=8)=83L x=long('11',base=16)=17L
3.	float(x)	Converts x to a floating-point number.	x=float('123.45')=123.45

contd. ...

4.	complex(real[,imag])	Creates a complex number.	x=complex(1,2) = (1+2j)
5.	str(x)	Converts object x to a string representation.	x=str(10) = '10'
6.	repr(x)	Converts object x to an expression string.	x=repr(3) = 3
7.	eval(str)	Evaluates a string and returns an object.	x=eval('1+2') = 3
8.	tuple(s)	Converts s to a tuple.	x=tuple('123') = ('1', '2', '3') x=tuple([123]) = (123,)
9.	list(s)	Converts s to a list.	x=list('123') = ['1', '2', '3'] x=list(['12'] = ['12']
10.	set(s)	Converts s to a set.	x=set('Python') = {'y', 't', 'o', 'P', 'n', 'h'}
11.	dict(d)	Creates a dictionary. d must be a sequence of (key, value) tuples.	dict={'id':'11','name':'vijay'} print(dict) ={'id': '11', 'name': 'vijay'}
12.	chr(x)	Converts an integer to a character.	x=chr(65) = 'A'
13.	unichr(x)	Converts an integer to a Unicode character.	x=unichr(65) =u'A'
14.	ord(x)	Converts a single character to its integer value.	x=ord('A')= 65
15.	hex(x)	Converts an integer to a hexadecimal string.	x=hex(12) = 0xc
16.	oct(x)	Converts an integer to an octal string.	x=oct(8) = 0o10

Formatting Numbers and Strings:

- The format() function formats a specified value into a specified format.

Syntax: format(value, format)

Example: For string and number formation

```
>>> x=12.345
>>> format(x,".2f")
'12.35'
>>>
```

Parameter Values:

Sr. No.	Format	The Format we want to Format the Value Into.	Example

1.	<	Left aligns the result (within the available space).	>>> x=10.23456
			>>> format(x,"<10.2f")
			'10.23 '

2.	>	Right aligns the result (within the available space).	>>>x=10.23456 >>> format(x,">10.2f") ' 10.23'
3.	^	Center aligns the result (within the available space).	>>>x=10.23456 >>> format(x,"^10.2f") ' 10.23 '

contd. ...

Scanned with CamScanner

4.	+	Use a sign to indicate if the result is positive or negative.	`>>> x=123`
			`>>> format(x,"+")`
			`'+123'`
			`>>>x=-123`
			`>>> format(x,"+")`
			`'-123'`

5.	-	Use a sign for negative values only.	`>>> x=-123`
			`>>> format(x,"+")`
			`'-123'`

6.	,	Use a comma as a thousand separator.	`>>> x=10000000`
			`>>> format(x,",")`
			`'10,000,000'`

7.	_	Use a underscore as a thousand separator.	`>>> x=100000`
			`>>> format(x,"_")`
			`'100_000'`

8.	b	Binary format.	`>>> x=10`
			`>>> format(x,"b")`
			`'1010'`

9.	c	Converts the value into the corresponding unicode character.	`>>> x=10`
			`>>> format(x,"c")`
			`'\n'`

10.	d	Decimal format.	`>>> x=10`
			`>>> format(x,"d")`
			`'10'`

11.	e	Scientific format, with a lower case e.	`>>> x=10`
			`>>> format(x,"e")`
			`'1.000000e+01'`

12.	E	Scientific format, with an upper case E.	`>>> x=10`
			`>>> format(x,"E")`
			`'1.000000E+01'`

13.	f	Fix point number format.	`>>> x=10`
			`>>> format(x,"f")`
			`'10.000000'`

14.	g	General format.	`>>> x=10`
			`>>> format(x,"g")`
			`'10'`

15.	o	Octal format.	`>>> x=10`
			`>>> format(x,"o")`
			`'12'`

16.	x	Hex format, lower case.	`>>> x=10`
			`>>> format(x,"x")`
			`'a'`

17.	X	Hex format, upper case.	`>>> x=10`
			`>>> format(x,"X")`
			`'A'`

18.	%	Percentage format.	`>>> x=100`
			`>>> format(x,"%")`
			`'10000.000000%'`

19.	`>10s`	String with width 10 in left justification.	`>>> x="hello"`
			`>>> format(x,">10s")`
			`'     hello'`

20.	`<10s`	String with width 10 in right justification.	`>>> x="hello"`
			`>>> format(x,"<10s")`
			`'hello     '`

Scanned with CamScanner

4.1.2 Built-In Mathematical 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.

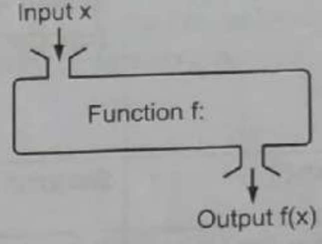


Fig. 4.1

- 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.	`min()`	Returns smallest value among supplied arguments.	`>>> min(20, 10, 30)` `10`
2.	`max()`	Returns largest value among supplied arguments.	`>>> max(20, 10, 30)` `30`

3.	`pow()`	The pow() 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.	`>>> pow(2,3)`
			`8`
			`>>> pow(2,3,2)`
			`0`

4.	`round()`	The round() 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.	`>>> round(10.2345)`
			`10`
			`>>> round(5.76543)`
			`6`
			`>>> round(5.76543,2)`
			`5.77`

5.	`abs()`	Absolute function, also known as Modulus (not to be confused with Modulo), returns the non-negative value of the argument value.	`>>> abs(-5)`
			`5`
			`>>> abs(5)`
			`5`

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`

contd. ...

Scanned with CamScanner

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))` `10.067661995777765`
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`

10.	log(a,(Base)):	This function is used to compute the natural logarithm (Base e) of a.	`>>> print(math.log(14))` `2.6390573296152584`
11.	log2(a)	This function is used to compute the logarithm base 2 of a. Displays more accurate result than log(a,2).	`>>> rint(math.log2(14) )` `3.807354922057604`
12.	log10(a)	This function is used to compute the logarithm base 10 of a. Displays more accurate result than log(a,10).	`>>> print(math.log10(14))` `1.146128035678238`

13.	sqrt()	The method sqrt() returns the square root of x for x > 0.	`>>> math.sqrt(100)`
			`10.0`
			`>>> math.sqrt(5)`
			`2.23606797749979`

14.	trunc()	This function returns the truncated integer of x.	`>>> math.trunc(3.354)` `3`

4.2 USER DEFINED FUNCTIONS

- Functions in Python programming are self-contained programs that perform some particular tasks. Once, a function is created by the programmer for a specific task, this function can be called anytime to perform that task.
- Python gives us many built-in functions like print(), len() etc. but we can also create our own functions. These functions are called user-defined functions.
- User defined function are the self-contained block of statements created by users according to their requirements.
- A user-defined function is a block of related code statements that are organized to achieve a single related action or task. A key objective of the concept of the user-defined function is to encourage modularity and enable reusability of code.

Scanned with CamScanner

4.2.1 Function Definition

- 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.

  Syntax:
```
def function-name(parametes):
    "function_docstring"
    function_statements
    return [expression]
```

Defining Function:

- 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
- The basic syntax for a Python function definition is explained in Fig. 4.2.

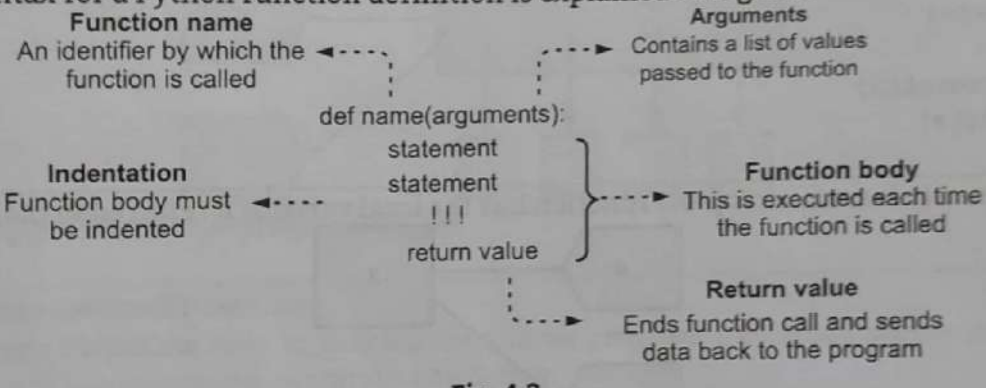


Fig. 4.2

4.2.2 Function Calling

- The def statement only creates a function but does not call it. After the def has run, we can call (run) the function by adding parentheses after the function's name.

Example: For calling a function.
```
>>> def square(x):  # function definition
        return x*x

>>> square(4)          # function call
16
>>>
```

Concept of Actual and 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).

Scanned with CamScanner

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)
```

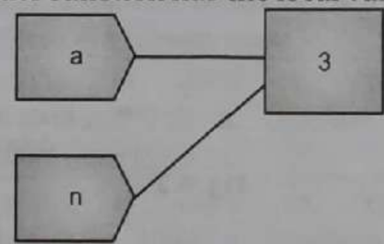
Call by object reference

- Most programming languages have a formal mechanism for determining if a parameter receives a copy of the argument (call by value) or a reference to the argument (call by name or call by reference) but Python uses a mechanism, which is known as "Call-by-Object/Call by Object Reference/Call by Sharing".

Example:

```
>>> def increment(n):
        n=n+1
>>> a=3
>>> increment(3)
>>> print(a)
3
```

- When we pass a to increment (n), the function has the local variable n referred to the same object:



- When control comes to n =n+1 as integer is immutable, by definition we are not able to modify the object's value to 4 in place: we must create a new object with the value 4. We may visualize it like below:

- All this time, the variable a continues to refer to the object with the value 3, since we did not change the reference:

- We can still "modify" immutable objects by capturing the return of the function.

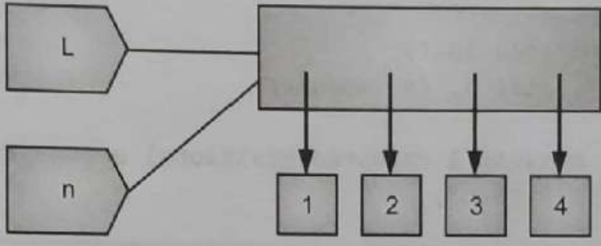
```
>>> def increment(n):
        n=n+1
        return n

>>> a=3
>>> a=increment(3)
>>> print(a)
4
```

Scanned with CamScanner

- By assigning the return value of the function to a, we have reassigned a to refer to the new object with the value 4 created in the function. Note the object a initially referred to never change — it is still 3 — but by having a point to a new object created by the function, we are able to "modify" the value of a.
- The same increment() function generates a different result when we passing a mutable object: Here L is a list which is mutable

```
>>> def increment(n):
        n.append([4])
>>> L=[1,2,3]
>>> increment(L)
>>> print(L)
[1, 2, 3, [4]]
```

- Here, the statement L = [1,2,3] makes a variable L(box) that points towards the object [1,2,3].
- On the function being called, a new box n is created. The contents of n are the SAME as the contents of box L. Both the boxes contain the same object. That is, both the variables point to the same object in memory. Hence, any change to the object pointed at by n will also be reflected by the object pointed at by L.



Hence the output of the above program will be:
[1, 2, 3, 4]

Advantages of User-Defined Functions:

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. By using functions, we can avoid rewriting same logic/code again and again in a program.
3. We can call python functions any number of times in a program and from any place in a program.
4. We can track a large python program easily when it is divided into multiple functions.
5. Reusability is the main achievement of Python functions.
6. Functions in Python reduces the overall size of the program.

4.2.3 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.

4.2.3.1 Required Arguments

- Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Example: For required argument.

```
>>> def display(str):
        "This print a string passed as argument"
        print(str)
        return
>>> display()                       # required argument
```

Scanned with CamScanner

```
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
  display()
TypeError: display() missing 1 required positional argument: 'str'
>>>
```

- There is an error because we did not pass any argument to the function display. We have to pass argument like display ("hello").

4.2.3.2 Keywords Arguments

- Keyword arguments are related to the function calls. When we use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows us to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example 1: For keywords arguments.

```
>>> def display(str):
        "This print a string passed as argument"
        print(str)
        return
>>> display()
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    display()
TypeError: display() missing 1 required positional argument: 'str'
>>> display(str="Hello Python")
Hello Python
```

Example 2:

```
>>> def display(name,age):
        print("Name:",name)
        print("Age:",age)
        return
>>> display(name="meenakshi",age=35)
Name: meenakshi
Age: 35
>>> display(age=35,name="meenakshi")
Name: meenakshi
Age: 35
>>>
```

4.2.3.3 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: For default arguments.

```
>>> def display(name,age=20):
        print("Name:",name)
        print("Age:",age)
        return
>>> display(name="meenakshi")
Name: meenakshi
Age: 20
>>> display(name="meenakshi",age=35)
Name: meenakshi
Age: 35
>>>
```

Scanned with CamScanner

4.2.3.4 Variable Length Arguments

- In many cases where we are required to process a function with more number of arguments than we specified in the function definition. These types of arguments are known as variable length arguments.

Syntax:

```
def function_name([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_statements
    return [expression]
```

- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example: For variable length argument.

```
>>> def display(arg1,*list):
        "This prints a variable passed arguments"
        print(arg1)
        for i in list:
                print(i)
                return
>>> display(10,20,30)
10
20
30
>>>
```

4.2.4 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
>>>
```

Fruitful Function:

- 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
```

- **void Functions:** void functions are those functions which do not return any value.

Scanned with CamScanner

nple: For void function.

```
>>> def show():
        str="hello"
        print(str)
>>> show()
ello
>>
```

Scope of Variable

ariables in a program may not be accessible at all locations in that program. This depends on e we have declared a variable.

scope of a variable determines the portion of the program where you can access a particular ble/identifier.

vailability/accessibility of a variable in different parts of a program is referred to as its scope.

e are two basic scopes of variables in on:

- **lobal Variables:** Global variables can be ccessed throughout (outside) the program ody by all functions.
- **ocal Variables:** Local variables can be ccessed only inside the function in which ey are declared.

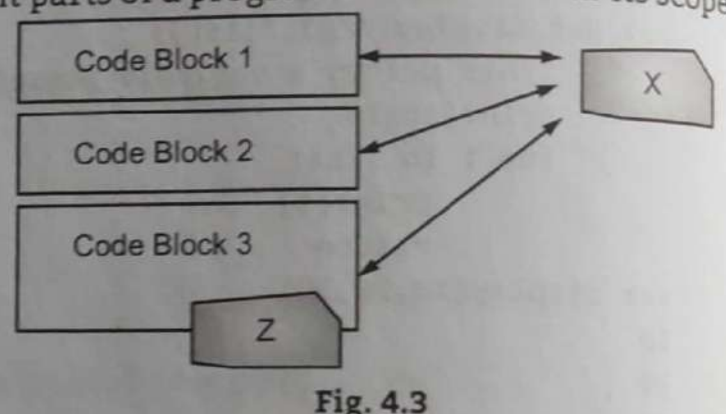


Fig. 4.3

4.3 shows, global variable (x) can be reached and modified anywhere in the code, local ble (z) exists only in block 3.

ple: 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()
ocal variable= 20
lobal variable= 10
>
```

ce between Local Variables and Global Variables:

	Local Variables	Global Variables
	Local variables are declared inside a function.	Global variables are declared outside any function.
	Accessed only by the statements, inside a function in which they are declared.	Accessed by any statement in the entire program.
	Local variables are alive only for a function	Global variables are alive till the end of the program
	A local variable is destroyed when the control of the program exit out of the block	Global variable is destroyed when the entire

- A function is said to be a recursive if it calls itself. For example, lets 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)
```

Scanned with CamScanner

```
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
```

4.3 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.

4.3.1 Writing Module

- Writing a module means simply creating a file which can 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 perform 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
```

Scanned with CamScanner

```
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
```

4.3.2 Importing Modules

- Import statement is used to imports a specific module by using its name. 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.

Scanned with CamScanner

- We can access attribute / mehods 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
```

Scanned with CamScanner

4.3.3 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
```

4.3.4 Python Built in 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).

Scanned with CamScanner

(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)
```

Scanned with CamScanner

(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)
```

Scanned with CamScanner

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('0.05000000000000000277555756156289135105907917022705078125')
>>> 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
```

Scanned with CamScanner

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 callables.
- 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)
```

Scanned with CamScanner

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 creats 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

contd. ...

Scanned with CamScanner

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
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

4.3.5 Namespace and Scoping

- A namespace is a system to have a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary.
- Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace. So, the division of the word itself gives little more information: Name (which means name, an unique identifier) + Space(which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.
- A namespace in python is a collection of names. So, a namespace is essentially a mapping of names to corresponding objects.
- At any instant, different python namespaces can coexist completely isolated- the isolation ensures that there are no name collisions/problem.
- A scope refers to a region of a program where a namespace can be directly accessed, i.e. without using a namespace prefix.
- Scoping in Python revolves around the concept of namespaces. Namespaces are basically dictionaries containing the names and values of the objects within a given scope.

Types of Namespaces:

- When a user creates a module, a global namespace gets created, later creation of local functions creates the local namespace. The built-in namespace encompasses global namespace and global namespace encompasses local namespace.
  1. **Local Namespace:** This namespace covers the local names inside a function. Python creates this namespace for every function called in a program. It remains active until the function returns.
  2. **Global Namespace:** This namespace covers the names from various imported modules used in a project. Python creates this namespace for every module included in the program. It will last until the program ends.
  3. **Built-in Namespace:** This namespace covers the built-in functions and built-in exception names. Python creates it as the interpreter starts and keeps it until we exit.

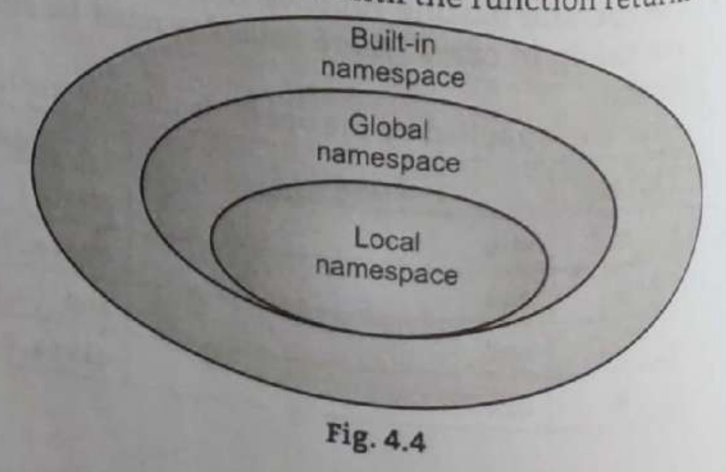


Fig. 4.4

Scanned with CamScanner

- Namespaces help us uniquely identify all the names inside a program. According to Python's documentation "a scope is a textual region of a Python program, where a namespace is directly accessible." Directly accessible means that when we are looking for an unqualified reference to a name Python tries to find it in the namespace.
- Scopes are determined statically, but actually, during runtime, they are used dynamically. This means that by inspecting the source code, we can tell what the scope of an object is, but this does not prevent the software from altering that during runtime.

Python Variable Scoping:

- Scope is the portion of the program from where a namespace can be accessed directly without any prefix.
- Namespaces are a logical way to organize variable names when a variable inside a function (a local variable) shares the same name as a variable outside of the function (a global variable).
- Local variables contained within a function (either in the script or within an imported module) and global variables can share a name as long as they do not share a namespace.
- At any given moment, there are at least following three nested scopes:
  1. Scope of the current function which has local names.
  2. Scope of the module which has global names.
  3. Outermost scope which has built-in names.
- When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.
- If there is a function inside another function, a new scope is nested inside the local scope. Python has two scopes.
  1. **Local Scope Variable:** All those variables which are assigned inside a function known as local scope Variable
  2. **Global Scope Variable:** All those variables which are outside the function termed as global variable.

Example: For global scope and local scope.

```
global_var = 30      # global scope
def scope():
    local_var = 40          # local scope
    print(global_var)
    print(local_var)
scope()
print(global_var)
```

Output:

```
30
40
3
```

4.4 PYTHON 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.

4.4.1 Introduction

- 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).

Scanned with CamScanner

4.4.2 Writing Python Packages

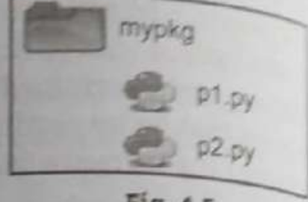


Fig. 4.5

- 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")
```

- 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
>>>
```

4.4.3 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.

Scanned with CamScanner

- 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.

4.4.3.1 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.
- **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
>>>
```

- Different mathematical functions of Math module already explained in Section 4.1.2.

4.4.3.2 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.
- Fig. 4.6 shows the axes (or dimensions) and lengths of two example arrays; (a) is a one-dimensional array and (b) is a two-dimensional array.

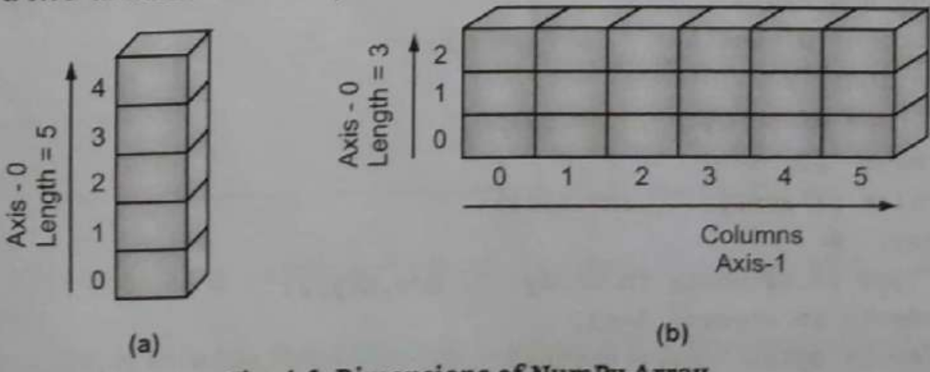


Fig. 4.6: Dimensions of NumPy Array

Scanned with CamScanner

- A one dimensional array has one axis indicated by Axis-0. That axis has five elements in it, so we say it has length of five.
- A two dimensional array is made up of rows and columns. All rows are indicated by Axis-0 and all columns are indicated by Axis-1. If Axis-0 in two dimensional array has three elements, so its length it three and Axis-1 has six elements, so its length is six.
- Execute Following command to install numpy in window, Linux and MAC OS:

```
python -m pip install numpy
```

- To use NumPy you need to import Numpy:

```
import numpy as np          # alias np
```

- Using NumPy, a developer can perform the following operations:
  1. Mathematical and logical operations on arrays.
  2. Fourier transforms and routines for shape manipulation.
  3. Operations related to linear algebra.
  4. NumPy has in-built functions for linear algebra and random number generation.

Array Object:

- NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called axes. The number of axes is rank. NumPy's array class is called ndarray. It is also known by the alias array.
- Basic attributes of the ndarray class as follow:

Sr. No.	Attributes	Description
1.	shape	A tuple that specifies the number of elements for each dimension of the array.
2.	size	The total number elements in the array.
3.	ndim	Determines the dimension an array.
4.	nbytes	Number of bytes used to store the data.
5.	dtype	Determines the datatype of elements stored in array.

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
```

Scanned with CamScanner

Basic Array Operations:

- In NumPy, arrays allow a wide range of operations which can be performed on a particular array or a combination of Arrays.
- These operations include some basic mathematical operation as well as Unary and Binary operations. In case of +=, −=, *= operators, the existing array is modified.
  1. **Unary Operators:** Many unary operations are provided as a method of ndarray class. This includes sum, min, max, etc. These functions can also be applied row-wise or column-wise by setting an axis parameter.
  2. **Binary Operators:** These operations apply on array elementwise and a new array is created. You can use all basic arithmetic operators like +, −, /, , etc. In case of +=, −=, = operators, the existing array is modified.

Example: For basic array operators.

```
>>> arr1=np.array([1,2,3,4,5])
>>> arr2=np.array([2,3,4,5,6])
>>> print(arr1)
[1 2 3 4 5]
>>> print("add 1 in each element:",arr1+1)
add 1 in each element: [2 3 4 5 6]
>>> print("subtract 1 from each element: ", arr1-1)
subtract 1 from each element:  [0 1 2 3 4]
>>> print("multiply 10 with each element in array: ",arr1*10)
multiply 10 with each element in array:  [10 20 30 40 50]
>>> print("sum of all array elements: ",arr1.sum())
sum of all array elements:  15
>>> print("array sum=:", arr1+arr2)
array sum=: [ 3  5  7  9 11]
>>> print("Largest element in array: ",arr1.max())
Largest element in array:  5
```

Reshaping of Array:

- We can also perform reshape operation using python numpy operation. Reshape is when you change the number of rows and columns which gives a new view to an object.

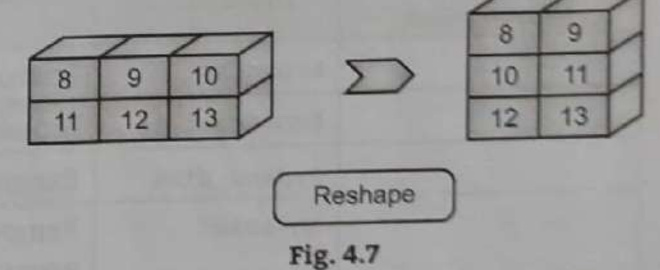


Fig. 4.7

Example:

```
>>> arr=np.array([[1,2,3],[4,5,6]])
>>> a=arr.reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Slicing of Array:

- Slicing is basically extracting particular set of elements from an array. Consider an array ([(1,2,3,4), (5,6,7,8)]).
- Here, the array (1,2,3,4) is at index 0 and (3,4,5,6) is at index 1 of the python numpy array. We need a particular element (say 3) out of a given array.

Scanned with CamScanner

- Let us consider the below example:

```
>>> import numpy as np
>>> a=np.array([(1,2,3,4),(5,6,7,8)])
>>> print(a[0,2])
3
```

- Now we need the 2nd element from the zeroth and first index of the array. The code will be as follows:

```
import numpy as np
a=np.array([(1,2,3,4),(5,6,7,8)])
print(a[0:,2])
[3 7]
```

- Here, colon represents all the rows, including zero.

Array Manipulation Functions:

- Several routines are available in NumPy package for manipulation of elements in ndarray object. They can be classified into the following types

Changing Shape	Shape	Description
	reshape	Gives a new shape to an array without changing its data.
	flat	A 1-D iterator over the array.
	flatten	Returns a copy of the array collapsed into one dimension.
	ravel	Returns a contiguous flattened array.
Transpose Operations	**Operation**	**Description**
	transpose	Permutes the dimensions of an array.
	ndarray.T	Same as self.transpose().
	rollaxis	Rolls the specified axis backwards.
	swapaxes	Interchanges the two axes of an array.
Changing Dimensions	**Dimension**	**Description**
	broadcast	Produces an object that mimics broadcasting.
	broadcast_to	Broadcasts an array to a new shape.
	expand_dims	Expands the shape of an array.
	squeeze	Removes single dimensional entries from the shape of an array.
Joining Arrays	**Array**	**Description**
	concatenate	Joins a sequence of arrays along an existing axis.
	stack	Joins a sequence of arrays along a new axis.
	hstack	Stacks arrays in sequence horizontally (column wise).
	vstack	Stacks arrays in sequence vertically (row wise).
Splitting Arrays	**Array**	**Description**
	split	Splits an array into multiple sub-arrays.
	hsplit	Splits an array into multiple sub-arrays horizontally (column wise).
	vsplit	Splits an array into multiple sub-arrays vertically (row wise).

contd. ...

Scanned with CamScanner

Addint/Removing Elements	Elements	Description
	resize	Returns a new array with the specified shape.
	append	Appends the values to the end of the array.
	insert	Inserts the value along the given axis before the given indices.
	delete	Returns a new array with sub array along an axis deleted.
	unique	Finds the unique elements in the array.
Bitwise Operations	**Operation**	**Description**
	bitwise_and	Compute bitwise AND operation of array elements.
	bitwise_or	Compute bitwise OR operation of array elements.
	invert	Compute bitwise NOT.
	left_shift	Shifts bits of a binary representation to the left.
	right_shift	Shifts bits of a binary representation to the right.

- Using numpy, we can deal with 1D polynomials by using the class poly1d. This class takes coefficients or roots for initialization and forms a polynomial object. When we print this object we will see it prints like a polynomial.
- Let us have a look at example code:

```
from numpy import poly1d
poly1 = poly1d([1,2,3])
print(poly1)
print("\nSquaring the polynomial: \n")
print(poly1* poly1)
print("\nIntegrating the polynomial: \n")
print(poly1.integ(k=3))
print("\nFinding derivative of the polynomial: \n")
print(poly1.deriv())
print("\nSolving the polynomial for 2: \n")
print(poly1(2))
```

Output:

```
   2
1 x + 2 x + 3
Squaring the polynomial:
   4     3      2
1 x + 4 x + 10 x + 12 x + 9
Integrating the polynomial:
         3      2
0.3333 x + 1 x + 3 x + 3
Finding derivative of the polynomial:
2 x + 2
Solving the polynomial for 2:
11
```

Scanned with CamScanner

4.4.3.3 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.
- Following command execute to install SciPy in Window, Linux and MAC OS:

```
python -m pip install scipy
```

- SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

Sr. No.	Subpackage	Description
1.	cluster	Clustering algorithms.
2.	constants	Physical and mathematical constants.
3.	fftpack	Fast Fourier Transform routines.
4.	integrate	Integration and ordinary differential equation solvers.
5.	interpolate	Interpolation and smoothing splines.
6.	io	SciPy has many modules, classes, and functions available to read data from and write data to a variety of file formats.
7.	linalg	Linear algebra.
8.	ndimage	N-dimensional image processing.
9.	odr	Orthogonal distance regression.
10.	optimize	Optimization and root-finding routines.
11.	signal	Signal processing.
12.	sparse	Sparse matrices and associated routines.
13.	spatial	Spatial data structures and algorithms.
14.	special	Special functions.
15.	stats	Statistical distributions and functions.

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
>>>
```

Scanned with CamScanner

Example 3: Using special sub package of SciPy.

```
>>> from scipy.special import cbrt
>>> cb=cbrt([81,64])                    # find cube root
>>> cb
array([4.32674871, 4.          ])
```

4.4.3.4 Matplotlib

- 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.
- Following command execute to install matplotlib in Window, Linux and MAC OS:

```
python -m pip install matplotlib
```

Importing matplotlib:

```
from matplotlib import pyplot as plt
```

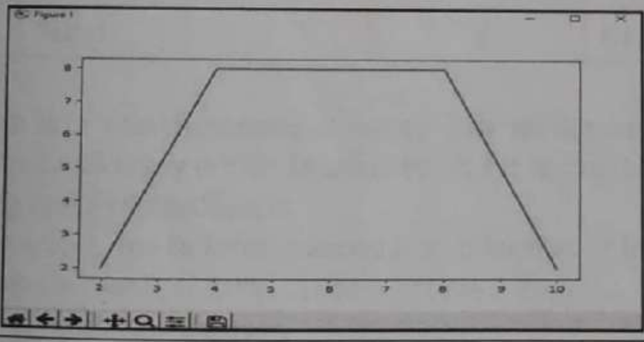
OR

```
import matplotlib.pyplot as plt
```

Example: 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:



Bar Graph:

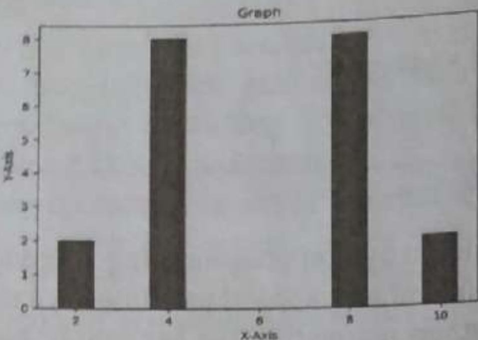
- A bar graph uses bars to compare data among different categories. It is well suited when you want to measure the changes over a period of time. It can be represented horizontally or vertically.

Example: 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()
```

Scanned with CamScanner

Output:



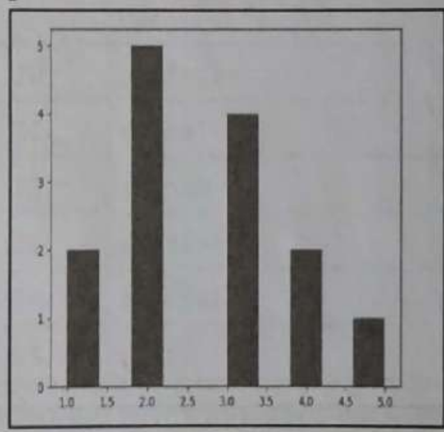
Histogram:

- Histograms are used to show a distribution whereas a bar chart is used to compare different entities. Histograms are useful when we have arrays or a very long list.
- A Histogram is a special graph that uses vertical columns to show frequencies (how many times each score occurs):

Example: For histogram.

```
>>> from matplotlib import pyplot as plt
>>> y=[1,1,2,2,2,2,2,3,3,3,3,4,4,5]
>>> plt.hist(y)
(array([2., 0., 5., 0., 0., 4., 0., 2., 0., 1.]), array([1. , 1.4, 1.8, 2.2, 2.6,
                              3. , 3.4, 3.8, 4.2, 4.6, 5. ]), <a list of 10 Patch objects>)
>>> plt.show()
```

Output:



Scatter Plot:

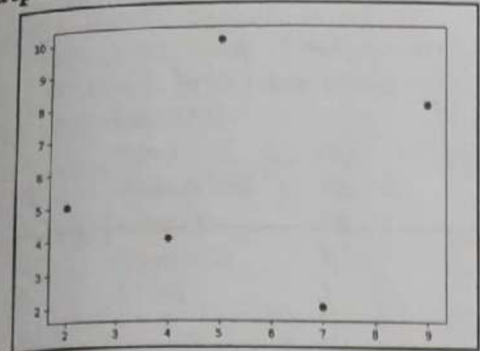
- Usually we need scatter plots in order to compare variables, for example, how much one variable is affected by another variable to build a relation out of it.
- The data is displayed as a collection of points, each having the value of one variable which determines the position on the horizontal axis and the value of other variable determines the position on the vertical axis.

Example: For scatter plot.

```
>>> from matplotlib import pyplot as plt
>>> x = [5, 2, 9, 4, 7]
>>> y = [10, 5, 8, 4, 2]
>>> plt.scatter(x,y)
<matplotlib.collections.PathCollection object at 0x05792770>
>>> plt.show()
```

Scanned with CamScanner

Output:



4.4.3.5 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.

Installing Pandas:

```
pip install pandas
```

Data structures supported by Pandas:

- Pandas deals with the following three data structures:

Sr. No.	Data Structure	Dimensions	Description
1.	Series	1	1D labeled homogeneous array, size immutable.
2.	Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
3.	Panel	3	General 3D labeled, size-mutable array.

Series:

- Series is a one-dimensional array like structure with homogeneous data. The Series is a one dimensional array which is Labelled and it is capable of holding array of any type like Integer, Float, String and Python Objects.
- For example, the following series is a collection of integers 10, 22, 30, 40,... The syntax is as follows:

```
pandas.Series(data, index, dtype, copy)
```

- It takes four arguments:
  1. **data:** It is the array that needs to be passed so as to convert it into a series. This can be Python lists, NumPy Array or a Python Dictionary or Constants.
  2. **index:** This holds the index values for each element passed in data. If it is not specified, default is numpy.arange(length_of_data).
  3. **dtype:** It is the datatype of the data passed in the method.
  4. **copy:** It takes a Boolean value specifying whether or not to copy the data. If not specified, default is false.
- Here data is only mandatory argument of Series.

Example 1: 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)
```

Scanned with CamScanner

```
0       2
1       4
2       6
3       8
4       10
5       20
dtype: int32
```

Example 2: 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
>>>
```

Data Frames:

- Data Frame is a two-dimensional array with heterogeneous data. We can convert a Python's list, dictionary or Numpy array to a Pandas data frame.
- For example:

Roll No	Name	City
11	Vijay	Thane
12	Amar	Pune
13	Santosh	Mumbai

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

- It takes following arguments:
  1. **data:** The data that is needed to be passed to the DataFrame() Method can be of any form line ndarray, series, map, dictionary, lists, constants and another DataFrame.
  2. **index:** This argument holds the index value of each element in the DataFrame. The default index is np.arange(n).
  3. **columns:** The default values for columns is np.arange(n).
  4. **dtype:** This is the data type of the data passed in the method.
  5. **copy:** It takes a Boolean value to specify whether or not to copy the data. The default value is false.
- Here data is only mandatory argument of DataFrame.

Example:

```
>>> import pandas as pd
>>> li = [1, 2, 3, 4, 5, 6]
>>> df = pd.DataFrame(li)
>>> print(df)
0  1
1  2
2  3
3  4
4  5
5  6
```

Scanned with CamScanner

Example: Using DataFrame data structure of Panda.

```
>>> import pandas as pd
>>> dict={"Name":["Meenakshi","Anurag","Khwahish","Aarul"],"Age":[35,39,7,3]}
>>> df=pd.DataFrame(dict)
>>> print(df)
      Name       Age
0     Meenakshi  35
1     Anurag     39
2     Khwahish   7
3     Aarul      3
>>>
```

Panel:

- Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

```
pandas.Panel(data, item, major_axis, minor_axis, dtype, copy)
```

- It takes following arguments:
  1. **data:** The data can be of any form like ndarray, list, dict, map, DataFrame.
  2. **item:** axis 0, each item corresponds to a dataframe contained inside.
  3. **major_axis:** axis 1, it is the index (rows) of each of DataFrames.
  4. **minor_axis:** axis 2, it is the columns of each DataFrames.
  5. **dtype:** The data type of each column
  6. **copy:** It takes a Boolean value to specify whether or not to copy the data. The default value is false.

Example 1:

```
import pandas as pd
import numpy as np
information = np.random.rand(1, 2, 3)
pandas_panel = pd.Panel(information)
print(pandas_panel)
```

Output:

```
<class 'pandas.core.panel.Panel'>
Dimensions: 1 (items) x 2 (major_axis) x 3 (minor_axis)
Items axis: 0 to 0
Major_axis axis: 0 to 1
Minor_axis axis: 0 to 2
```

Example 2:

```
import pandas as pd
importnumpy as np
l1=[1,2,3,4,5,6]
l2=["one","two","three","four","five","six"]

data = {'Item1' : pd.DataFrame(l1),'Item2' : pd.DataFrame(l2)}
p = pd.Panel(data)
print(p)
print(p.major_xs(1))
```

Output:

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 6 (major_axis) x 1 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 0 to 5
Minor_axis axis: 0 to 0
   Item1 Item2
0      2   two
```

Scanned with CamScanner

User Defined Packages

organize a large number of files in different folders and subfolders based on some criteria, so we can find and manage them easily. In the same way, a package in Python takes the concept of nodular approach to next logical level.

e know, a module can contain multiple objects, such as classes, functions, etc. A package can ain one or more relevant modules.

ically, a package is actually a folder containing one or more module files. Let's create a package ed MyPkg, using the following steps:

reate a folder MyPkg on "C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\".

iodules Message.py and Mathematics.py with following code:

.py

```
ef SayHello(name):
      print("Hello " + name)
      return
```

iatics.py

```
ef sum(x,y):
     return x+y
ef average(x,y):
     return (x+y)/2
ef power(x,y):
     return x**y
```

reate an empty __init__.py file in the MyPkg folder. The package folder contains a special file _init__.py, which stores the package's content. It serves two purposes:

he Python interpreter recognizes a folder as the package if it contains __init__.py file.

_init__.py exposes specified resources from its modules to be imported.

mpty __init__.py file makes all functions from above modules available when this package is orted. Note that __init__.py is essential for the folder to be recognized by Python as a package. an optionally define functions from individual modules to be made available.

reate P1.py file in MyPkg folder and write following code:

```
rom MyPkg import Mathematics
rom MyPkg import Message
reet.SayHello("Meenakshi")
=functions.power(3,2)
rint("power(3,2) : ", x)
```

```
ello Meenakshi
ower(3,2) :  9
```

_init__.py File:

__init__.py file is normally kept empty.However, it can also be used to choose specific functions modules in the package folder and make them available for import. Modify __init__.py as v:

- 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
```

Practice Questions

1. What is function?
2. What is module?
3. What is package?
4. Define function. Write syntax to define function. Give example of function definition.
5. Can a Python function return multiple values? If yes, how it works?
6. How function is defined and called in Python.
7. Explain about void functions with suitable examples.
8. What is actual and formal parameter? Explain the difference along with example.
9. Explain about fruitful functions with suitable examples.
10. Discuss the difference between local and global variable.
11. Explain any five basic operations performed on string.
12. Explain math module with its any five functions.
13. Differentiate between match() and search() function. Explain with example.
14. Explain type conversion of variable in Python.
15. Write a function that takes single character and prints 'character is vowel' if it is vowel, 'character is not vowel' otherwise.
16. Explain various string operations that can be performed using operators in Python.
17. Explain with an example, how + and * operators work with strings.
18. Explain str.find() function with suitable example.
19. Define is module? What are the advantages of using module?
20. How to create a module and use it in a python program explain with an example.
21. Explain various functions of math module.
22. List and explain any four built in string manipulation functions supported by Python.
23. Explain string slicing in Pyhton. Show with example.
24. Explain the concept of namespaces with an example.
25. Write about the concept of scope of a variable in a function.

Scanned with CamScanner

5...

Object Oriented Programming in Python

Chapter Outcomes...

- Create classes and objects to solve the given problem.
- Write Python code for data hiding for the given problem.
- Write Python code using data abstraction for the given problem.
- Write Python program using Inheritance for the given problem.

Learning Objectives...

- To learn Object Oriented Programming Concepts in Python programming
- To Creating Classes and Objects in Python
- To learn Method Overloading, Method Overriding, Data Hiding, Data Abstraction, Inheritance etc.

5.0 INTRODUCTION

- Python is an Object-Oriented Programming Language (OOPL) follows an Object-Oriented Programming (OOP) paradigm. It deals with declaring Python classes and objects which lays the foundation of OOPs concepts.
- Python programming offers OOP style programming and provides an easy way to develop programs. Python programming uses the OOPs concepts that makes Python more powerful to help design a program that represents real-world entities.
- Python also supports OOP concepts such as Inheritance, Method overriding, Data abstraction and Data hiding.

Important terms in OOP/Terminology of OOP:

1. **Class:** Classes are defined by the user. The class provides the basic structure for an object. It consists of data members and method members that are used by the instances, (objects) of the class.
2. **Object:** A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods. Class itself does nothing but the real functionality is achieved through their objects. Object is an instance or occurrence of the class. It takes the properties (variables) and uses the behavior (methods) defined in the class.
3. **Data Member:** A variable defined in either a class or an object; it holds the data associated with the class or object.
4. **Instance Variable:** A variable that is defined in a method; it scope is only within the object that defines it.
5. **Class Variable:** A variable that is defined in the class and can be used by all the instances of that class.

Scanned with CamScanner

6. **Instance:** An object is an instance of the class.
7. **Instantiation:** The process of creation of an object of a class.
8. **Method:** Methods are the functions that are defined in the definition of class and are used by various instances of the class.
9. **Function Overloading:** A function defined more than one time with different behavior's is known as function overloading. The operation performed varies by the types of objects or arguments involved.
10. **Encapsulation:** Encapsulation is the process of binding together the methods and data variables as a single entity i.e., class. This keeps both the data and functionality code safe from the outside world. It hides the data within the class and makes it available only through the methods.
11. **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it. A class 'A' that can use the characteristics of another class 'B' is said to be derived class i.e. a class inherited from B. This process is called inheritance.
12. **Polymorphism:** Polymorphism allows one interface to be used for a set of actions i.e., one name may refer to different functionality. The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.
13. **Data Abstraction**: The basic idea of data abstraction is to visible only the necessary information, unnecessary information will be hidden from the outside world. Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where we type the text and send the message. We don't know the internal processing about the message delivery.

5.1 CLASSES

- Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods.
- Object is simply a collection of data (variables) and methods (functions) that act on those data.
- A class is like an object constructor or a "blueprint" for creating objects. A class defines the properties and behavior (variables and methods) that is shared by all its objects.

5.1.1 Creating Classes

- A class is a block of statements that combine data and operations, which are performed on the data, into a group as a single unit and acts a blueprint for the creation of objects.
- To create a class, use the keyword 'class'. Here's the very basic structure of python class definition.

Syntax:

```
class ClassName:
  'Optional class documentation string'
    # list of python class variables
    # python class constructor
    # python class method definitions
```

- Following is an example of creation of an empty class:

```
class Car:
    pass
```

Here, the pass statement is used to indicate that this class is empty.
- In a class we can define variables, functions, etc. While writing any function in class we have to pass atleast one argument that is called self Parameter.
- The self parameter is a reference to the class itself and is used to access variables that belongs to the class. It does not have to be named self, we can call it whatever we like, but it has to be the first parameter of any function in the class.

Scanned with CamScanner

- We can write a class on interactive interpreter or in a.py file

Class on Interactive Interpreter:

```
>>> class student:
        def display(self):          # defining method in class
            print("Hello Python")
```

Class in a .py file:

```
class student:
    def display(self):              # defining method in class
        print("Hello Python")
```

- In Python programming self is a default variable that contains the memory address of the instance of the current class. So we can use self to refer to all the instance variables and instance methods.

5.1.2 Objects and Creating Objects

- An object is an instance of a class that has some attributes and behavior.
- Objects can be used to access the attributes of the class.

Syntax: `obj_name=class_name()`

Example:

```
s1=student()
s1.display()
```

- Complete program with class and objects on interactive interpreter is given below:

```
>>> class student:
        def display(self):          # defining method in class
            print("Hello Python")
>>> s1=student()                    # creating object of class
>>> s1.display()                    # calling method of class using object
Hello Python
```

- Complete program with class and objects on interactive interpreter in .py file is given below:

```
class student:
    def display(self):
        print("Hello Python")
s1=student()
s1.display()
```

Output:

```
Hello Python
```

Example : Class with get and put method.

```
class Car:
    def get(self, color, style):
        self.color = color
        self.style = style
    def put(self):
        print(self.color)
        print(self.style)
c = Car()
c.get('Sedan', 'Black')
c.put()
```

Output:

```
Sedan
Black
```

Scanned with CamScanner

5.1.3 Instance Variable and Class Variable

- Instance variable is defined in a method and its scope is only within the object that defines it. Instance attribute is unique to each object (instance). Every object of that class has its own copy of that variable. Any changes made to the variable don't reflect in other objects of that class.
- Class variable is defined in the class and can be used by all the instances of that class. Class attribute is same for all objects. And there's only one copy of that variable that is shared with all objects. Any changes made to that variable will reflect in all other objects.
- Instance variables are unique for each instance, while class variables are shared by all instances. Following example demonstrates the use of instance and class variable.

Example: For instance and class variables.

```
class Sample:
    x = 2                        # x is class variable
    def get(self, y):            # y is instance variable
        self.y = y
s1 = Sample()
s1.get(3)                        # Access attributes
print(s1.x ," ",s1.y)
s2 = Sample()
s2.y=4                           # Modify attribute
print(s2.x," ",s2.y)
```

Output:

```
2 3
2 4
```

5.2 DATA HIDING

- Data hiding is a software development technique specifically used in Object-Oriented Programming (OOP) to hide internal object details (data members).
- Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.
- Data hiding is also known as information hiding. An object's attributes may or may not be visible outside the class definition.
- We need to name attributes with a double underscore (_ _) prefix, and those attributes then are not be directly visible to outsiders. Any variable prefix with double underscore is called private variable which is accessible only with class where it is declared.

Example: For data hiding.

```
class Counter:
   _ _secretCount = 0 #private variable
   def count(self): #public method
      self._ _secretCount += 1
      print ("count=",self._ _secretCount) # accessible in the same class
c1= Counter()
c1.count() #invoke method
c1.count()
print ("Total count=",c1._ _secretCount) #cannot access private variable directly
```

Output:

```
count= 1
count= 2
AttributeError: 'Counter' object has no attribute '_ _secretCount'
```

Scanned with CamScanner

5.3 DATA ENCAPSULATION AND DATA ABSTRACTION

- We can restrict access of methods and variables in a class with the help of encapsulation. It will prevent the data from being modified by accident.
- Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- The terms encapsulation and abstraction (data hiding) are often used as synonyms. Data abstraction we can achieve through encapsulation.
- Encapsulation is a process to bind data and functions together into a single unit i.e., class while abstraction is a process in which the data inside the class is the hidden from the outside world. It hides the sensitive information.
- In short, hiding internal details and showing functionality is known as data abstraction.
- To support encapsulation, declare the methods or variables as private in the class. The private methods cannot be called by the object directly. It can be called only from within the class in which they are defined.
- Any function prefix with double underscore is called private method which is accessible only with class where it is declared.
- Following table shows the access modifiers for variables and methods:

Sr. No.	Types	Description
1.	Public methods	Accessible from anywhere i.e. inside the class in which they are defined, in the sub class, in the same script file as well as outside the script file.
2.	Private methods	Accessible only in their own class. Starts with two underscores.
3.	Public variables	Accessible from anywhere.
4.	Private variables	Accessible only in their own class or by a method if defined. Starts with two underscores.

Example: For access modifiers with data abstraction.

```
class student:
    _ _a=10 #private variable
    b=20 #public variable
    def _ _private_method(self): #private method
        print("private method is called")
    def public_method(self): #public method
        print("public method is called")
        print("a=",self._ _a) #can be accessible in same class
s1=student()
# print("a=",s1._ _a) #generate error
print("b=",s1.b)
# s1._ _private_method() #generate error
s1.public_method()
```

Output:

```
b=20
public method is called
a=10
```

Constructor:

- A constructor is a special method i.e., is used to initialize the instance variable of a class.

Creating Constructor in Class:

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of class is created.
- Python class constructor is the first piece of code to be executed when we create a new object of a class.
- In Python the _ _init_ _() method is called the constructor and is always called when an object is created.
- Primarily, the constructor can be used to put values in the member variables. We may also print messages in the constructor to be confirmed whether the object has been created.

Syntax:

```
def _ _init_ _(self):
    # body of the constructor
```

- _ _init_ _ is a special method in Python classes, which is the constructor method for a class. In the following example you can see how to use it.

Example 1: For creating constructor.

```
class Person:
    def _ _init_ _(self, rollno, name, age):
        self.rollno=rollno
        self.name = name
        self.age = age
        print("Student object is created")
p1 = Person(11,"Vijay", 40)
print("Rollno of student= ", p1.rollno)
print("Name of student= ",p1.name)
print("Age of student= ",p1.age)
```

Output:

```
Student object is created
Rollno of student= 11
Name of student= Vijay
Age of student= 40
```

Example 2: Define a class named Rectangle which can be constructed by a length and width. The Rectangle class has a method which can compute the area.

```
class Rectangle(object):
    def _ _init_ _(self, l, w):
        self.length = l
        self.width = w

    def area(self):
        return self.length*self.width

r = Rectangle(2,10)
print(r.area())
```

Output:

```
20
```

Scanned with CamScanner

Example 3: Create a Cricle class and intialize it with radius. Make two methods getArea and getCircumference inside this class.

```
class Circle():
    def _ _init_ _(self,radius):
        self.radius = radius
    def getArea(self):
        return 3.14*self.radius*self.radius
    def getCircumference(self):
        return self.radius*2*3.14
c=Circle(5)
print("Area",c.getArea())
print("Circumference",c.getCircumference())
```

Output:

```
Area 78.5
Circumference 31.400000000000002
```

- The types of constructors includes default constructor and parameterized constructor.

1. **Default Constructor:**

- The default constructor is simple constructor which does not accept any arguments. It's definition has only one argument which is a reference to the instance being constructed.

Example 1: Display Hello message using default constructor.

```
class Student:
    def _ _init_ _(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
s1 = Student()
s1.show("Meenakshi")
```

Output:

```
This is non parametrized constructor
Hello Meenakshi
```

Example 2: Counting the number of objects of a class.

```
class Student:
    count=0;
    def _ _init_ _(self):
        Student.count=Student.count+1
s1=Student()
s2=Student()
print("The number of student objects",Student.count)
```

Output:

```
The number of student objects: 2
```

2. **Parameterized Constructor:**

- Constructor with parameters is known as parameterized constructor.
- The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example: For parameterized constructor.

```
class Student:
    def _ _init_ _(self,name):
        print("This is parametrized constructor")
        self.name = name
```

Scanned with CamScanner

```
        def show(self):
            print("Hello",self.name)
    s1 = Student("Meenakshi")
    s1.show()
```

Output:

```
This is parametrized constructor
Hello Meenakshi
```

Destructor:

- A class can define a special method called a destructor with the help of _ _del_ _(). It is invoked automatically when the instance (object) is about to be destroyed.
- It is mostly used to clean up any non-memory resources used by an instance (object).

Example: For destructor.

```
class Student:
    def __init__(self):
        print('non parameterized constructor-student created ')
    def __del__(self):
        print('Destructor called, student deleted.')
s1=Student()
s2=Student()
del s1
```

Output:

```
non parameterized constructor-student created
non parameterized constructor-student created
Destructor called, student deleted.
```

Built-in Class Attributes:

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:
  - **_ _dict_ _**: It displays the dictionary in which the class's namespace is stored.
  - **_ _name_ _**: It displays the name of the class.
  - **_ _bases_ _**: It displays the tuple that contains the base classes, possibly empty. It displays them in the order in which they occur in the base class list.
  - **_ _doc_ _**: It displays the documentation string of the class. It displays none if the docstring isn't given.
  - **_ _module_ _**: It displays the name of the module in which the class is defined. Generally the value of this attributes is "_ _main_ _" in interactive mode.

Example: For default built-in class attribute.

```
class test:
    'This is a sample class called Test.'
    def _ _init_ _(self):
        print("Hello from _ _init_ _ method.")
# class built-in attribute
print(test._ _doc_ _)
print(test._ _name_ _)
print(test._ _module_ _)
print(test._ _bases_ _)
print(test._ _dict_ _)
```

Scanned with CamScanner

Output:

```
This is a class called Test.
test
__main__
(<class 'object'>,)
{'__module__': '__main__', '__doc__': 'This is a sample class called Test.', '__init__': <function test.__init__ at 0x013AC618>, '__dict__': <attribute '__dict__' of 'test' objects>, '__weakref__': <attribute '__weakref__' of 'test' objects>}
```

5.4 METHOD OVERLOADING

- Method overloading is the ability to define the method with the same name but with a different number of arguments and data types.
- With this ability one method can perform different tasks, depending on the number of arguments or the types of the arguments given.
- Method overloading is a concept in which a method in a class performs operations according to the parameters passed to it.
- As in other languages we can write a program having two methods aith same name but with different number of arguments or order of arguments but in python if we will try to do the same we will get the following issue with method overloading in Python:

```
# to calculate area of rectangle
def area(length, breadth):
    calc = length * breadth
    print calc
# to calculate area of square
def area(size):
    calc =  size * size
    print calc
area(3)
area(4,5)
```

Output:

```
9
TypeError: area() takes exactly 1 argument (2 given)
```

- Python does not support method overloading, i.e., it is not possible to define more than one method with the same name in a class in Python.
- This is because method arguments in python do not have a type. A method accepting one argument can be called with an integer value, a string or a double as shown in next example.

```
class Demo:
    def method(self, a):
        print(a)
obj= Demo()
obj.method(50)
obj.method('Meenakshi')
obj.method(100.2)
```

Output:

```
50
Meenakshi
100.2
```

Scanned with CamScanner

- Same method works for three different data types. Thus, we cannot define two methods with the same name and same number of arguments but having different type as shown in the above example. They will be treated as the same method.
- It is clear that method overloading is not supported in python but that does not mean that we cannot call a method with different number of arguments. There are a couple of alternatives available in python that make it possible to call the same method but with different number of arguments.

Using Default Arguments:

- It is possible to provide default values to method arguments while defining a method. If method arguments are supplied default values, then it is not mandatory to supply those arguments while calling method as shown in next example.

Example 1: Method overloading with default arguments.

```
class Demo:
    def arguments(self, a = None, b = None, c = None):
        if(a != None and b != None and c != None):
            print("3 arguments")
        elif (a != None and b != None):
            print("2 arguments")
        elif a != None:
            print("1 argument")
        else:
            print("0 arguments")
obj = Demo()
obj.arguments("Meenakshi","Anurag","Thalor")
obj.arguments("Anurag","Thalor")
obj.arguments("Thalor")
obj.arguments()
```

Output:

```
3 arguments
2 arguments
1 argument
0 arguments
```

Example 2: With a method to perform different operations using method overloading.

```
class operation:
    def add(self,a,b):
        return a+b

op1=operation()
# To add two integer numbers
print("Addition of integer numbers=",op1.add(10,20))
# To add two floting point numbers
print("Addition of integer numbers=",op1.add(11.12,12.13))
# To add two strings
print("Addition of integer numbers=",op1.add("Hello","Python"))
```

Output:

```
Addition of integer numbers= 30
Addition of integer numbers= 23.25
Addition of integer numbers= HelloPython
```

5.5 INHERITANCE AND COMPOSITION CLASS

- The inheritance feature allows us to make it possible to use the code of the existing class by simply creating a new class and inherits the code of the existing class.

Scanned with CamScanner

5.5.1 Inheritance

- In inheritance objects of one class procure the properties of objects of another class. Inheritance provide code reusability, which means that some of the new features can be added to the code while using the existing code. The mechanism of designing or constructing classes from other classes is called inheritance.
- The new class is called derived class or child class and the class from which this derived class has been inherited is the base class or parent class.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

Syntax:

```
class A:
    # properties of class A
class B(A):
    # class B inheriting property of class A
    # more properties of class B
```

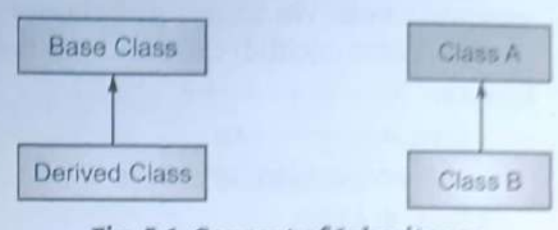


Fig. 5.1: Concept of Inheritance (Single Inheritance)

Example 1: Inheritance without using constructor.

```
class Vehicle: #parent class
    name="Maruti"
    def display(self):
        print("Name= ",self.name)
class Category(Vehicle): #derived class
    price=2000
    def disp_price(self):
        print("Price=$",self.price)
car1=Category()
car1.display()
car1.disp_price()
```

Output:

```
Name= Maruti
Price=$ 2000
```

Example 2: Inheritance using constructor.

```
class Vehicle: #parent class
    def _ _init_ _(self,name):
        self.name=name
    def display(self):
        print("Name= ",self.name)
class Category(Vehicle): #derived class
    def __init__(self,name,price):
        Vehicle.__init__(self,name) # passing data to base class constructor
        self.price=price
    def disp_price(self):
        print("Price=$ ",self.price)
car1=Category("Maruti",2000)
car1.display()
car1.disp_price()
car2=Category("BMW",5000)
car2.display()
car2.disp_price()
```

Scanned with CamScanner

5.5.1 Inheritance

- In inheritance objects of one class procure the properties of objects of another class. Inheritance provide code reusability, which means that some of the new features can be added to the code while using the existing code. The mechanism of designing or constructing classes from other classes is called inheritance.
- The new class is called derived class or child class and the class from which this derived class has been inherited is the base class or parent class.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

Syntax:

```
class A:
    # properties of class A
class B(A):
    # class B inheriting property of class A
    # more properties of class B
```

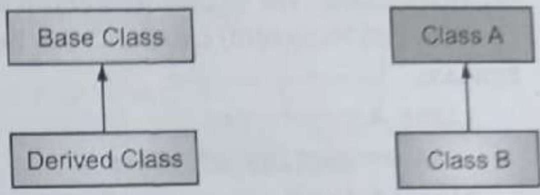


Fig. 5.1: Concept of Inheritance (Single Inheritance)

Example 1: Inheritance without using constructor.

```
class Vehicle: #parent class
    name="Maruti"
    def display(self):
        print("Name= ",self.name)
class Category(Vehicle): #derived class
    price=2000
    def disp_price(self):
        print("Price=$",self.price)
car1=Category()
car1.display()
car1.disp_price()
```

Output:

```
Name= Maruti
Price=$ 2000
```

Example 2: Inheritance using constructor.

```
class Vehicle: #parent class
    def _ _init_ _(self,name):
        self.name=name
    def display(self):
        print("Name= ",self.name)
class Category(Vehicle): #derived class
    def __init__(self,name,price):
        Vehicle.__init__(self,name) # passing data to base class constructor
        self.price=price
    def disp_price(self):
        print("Price=$ ",self.price)
car1=Category("Maruti",2000)
car1.display()
car1.disp_price()
car2=Category("BMW",5000)
car2.display()
car2.disp_price()
```

Scanned with CamScanner

Output:

```
Name= Maruti
Price=$ 2000
Name= BMW
Price=$ 5000
```

Multilevel Inheritance:

- Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.
- In multilevel inheritance (See Fig. 5.2), we inherit the classes at multiple separate levels. We have three classes A, B and C, where A is the super class, B is its sub(child) class and C is the sub class of B.

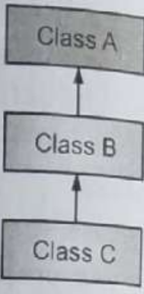


Fig. 5.2

Syntax:

```
class A:
    # properties of class A
class B(A):
    # class B inheriting property of class A
    # more properties of class B
class C(B):
    # class C inheriting property of class B
    # thus, class C also inherits properties of class A
    # more properties of class C
```

Example: For multilevel inheritance.

```
class Grandfather:
    def display1(self):
        print("Grand Father")
# The child class Father inherits the base class Grandfather
class Father(Grandfather):
    def display2(self):
        print("Father")
# The child class Son inherits another child class Father
class Son(Father):
    def display3(self):
        print("Son")
s1=Son()
s1.display3()
s1.display2()
s1.display1()
```

Output:

```
Son
Father
Grand Father
```

Multiple Inheritance:

- Python provides us the flexibility to inherit multiple base classes in the child class.
- Multiple Inheritance means that we are inheriting the property of multiple classes into one. In case we have two classes, say A and B, and we want to create a new class which inherits the properties of both A and B.
- So it just like a child inherits characteristics from both mother and father, in python, we can inherit multiple classes in a single child class.

Scanned with CamScanner

Syntax:

```
class A:
    # variable of class A
    # functions of class A
class B:
    # variable of class A
    # functions of class A
class C(A, B):
    # class C inheriting property of both classA and B
    # add more properties to class C
```

Base Class 1 Base Class 2 Base Class 3
Derived Class

(a)

Class A Class B
Class C

(b)

Fig. 5.3

Example:

```
# base class
class Father:
    def display1(self):
        print("Father")
# base class
class Mother:
    def display2(self):
        print("Mother")
# derived class
class Son(Father,Mother):
    def display3(self):
        print("Son")
s1=Son()
s1.display3()
s1.display2()
s1.display1()
```

Output:

```
Son
Mother
Father
```

Hierarchical Inheritance:

- When more than one derived classes are created from a single base – it is called hierarchical inheritance.
- In following program, we have a parent (base) class name Email and two child (derived) classes named Gmail and yahoo.

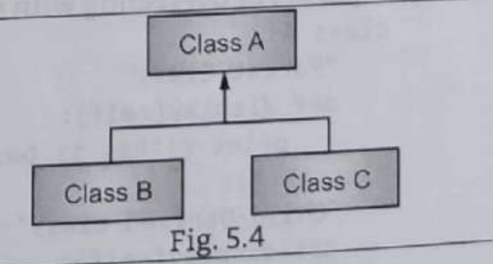


Fig. 5.4

Example: For hierarchical inheritance.

```
class Email:
    def send_email(self, msg):
        print()
class Gmail(Email):
    def send_email(self, msg):
        print( "Sending `{}` from Gmail".format(msg) )
```

Scanned with CamScanner

```
class Yahoo(Email):
    def send_email(self, msg):
        print( "Sending `{}` from Yahoo".format(msg) )
client1 = Gmail()
client1.send_email("Hello!")
client2 = Yahoo()
client2.send_email("Hello!")
```

Output:

```
Sending 'Hello!' from Gmail
Sending 'Hello!' from Yahoo
```

5.5.2 Method Overriding

- Overriding is the ability of a class to change the implementation of a method provided by one of its base class. Method overriding is thus a strict part of the inheritance mechanism.
- To override a method in the base class, we must define a new method with same name and same parameters in the derived class.
- Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power. Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it.

Example 1: For method overriding.

```
class A:                          # parent class
    "Parent Class"
    def display(self):
        print ('This is base class.')
class B(A):        #derived class
    "Child/Derived class"
    def display(self):
        print ('This is derived class.')
obj = B()                         # instance of child
obj.display()                     # child calls overridden method
```

Output:

```
This is derived class.
```

- In Inheritance delegation occurs automatically, but if a method is overridden the implementation of the parent is not considered at all. So, if you want to run the implementation of one or more of the ancestors of your class, you have to call them explicitly.

Using super() Method:

- The super() method gives you access to methods in a superclass from the subclass that inherits from it.
- The super() method also returns a temporary object of the superclass that then allows you to call that superclass's methods.

Example 2: For overriding with super().

```
class A:                          # parent class
    "Parent Class"
    def display(self):
        print ('This is base class.')
class B(A):                       # derived class
    "Child/Derived class"
    def display(self):
        super().display()
            print ('This is derived class.')
obj = B()                         # instance of child
obj.display()                     # child calls overridden method
```

Output:

```
This is base class.
This is derived class.
```

Scanned with CamScanner

5.5.3 Composition Classes

- In composition, we do not inherit from the base class but establish relationships between classes through the use of instance variables that are references to other objects.
- Composition also reflects the relationships between parts, called a "has-a" relationships. Some OOP design texts refer to composition as aggregation.
- It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component.
- UML represents composition as shown in Fig. 5.5.

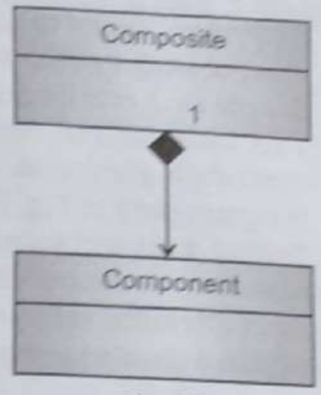


Fig. 5.5

- The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain. Cardinality can be expressed in the following ways:
  - A number indicates the number of Component instances that are contained in the Composite. The * symbol indicates that the Composite class can contain a variable number of Component instances.
  - A range 1..4 indicates that the Composite class can contain a range of Component instances. The range is indicated with the minimum and maximum number of instances, or minimum and many instances like in 1..*.

Syntax:

```
Class GenericClass:
    define some attributes and methods
class ASpecificClass:
    Instance_variable_of_generic_class=GenericClass
    # use this instance somewhere in the class
    some_method(Instance_variable_of_generic_class)
```

- In following program, we have three classes Email, Gmail and yahoo. In email class we are referring the Gmail and using the concept of Composition.

Example: For composition.

```
class Gmail:
    def send_email(self, msg):
        print("Sending `{}` from Gmail".format(msg))
class Yahoo:
    def send_email(self, msg):
        print( "Sending `{}` from Yahoo".format(msg) )
class Email:
    provider=Gmail()
    def set_provider(self,provider):
        self.provider=provider
    def send_email(self, msg):
        self.provider.send_email(msg)
client1 = Email()
client1.send_email("Hello!")
client1.set_provider(Yahoo())
client1.send_email("Hello!")
```

Output:

```
Sending `Hello!` from Gmail
Sending `Hello!` from Yahoo
```

Scanned with CamScanner

5.6 CUSTOMIZATION VIA INHERITANCE SPECIALIZING INHERITED METHODS

- In Python, every time we use an expression of the form object.attr, (where object is an instance or class object), Python searches the namespace tree from bottom to top, beginning with object, looking for the first attr it can find.
- This includes references to self attributes in the methods. Because lower definitions in the tree override higher ones, inheritance forms the basis of specialization.
- Program code in Fig. 5.6 create a tree of objects in memory to be searched by attribute inheritance.
- Calling a class creates a new instance that remembers its class, running a class statement creates a new class and superclasses are listed in parentheses in the class statement header.
- Each attribute reference triggers a new bottom - up tree search - even references to self attributes within a class's methods.

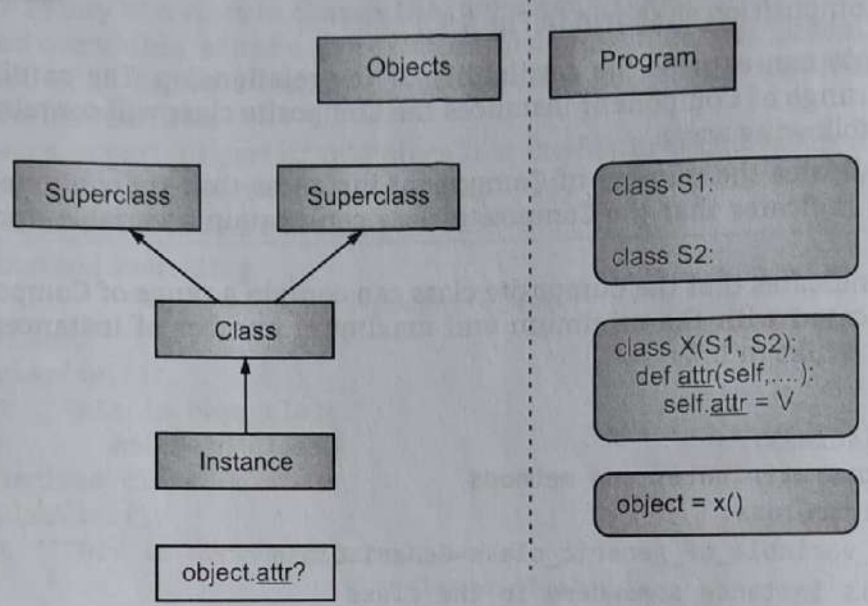


Fig. 5.6

- Fig. 5.6 summarizes the way namespace trees are constructed and populated with names. Generally:
  1. Instance attributes are generated by assignments to self attributes in methods.
  2. Class attributes are created by statements (assignments) in class statements.
  3. Superclass links are made by listing classes in parentheses in a class statement header.
- The net result is a tree of attribute namespaces that leads from an instance, to the class it was generated from, to all the superclasses listed in the class header.
- Python searches upward in this tree, from instances to superclasses, each time we use qualification to fetch an attribute name from an instance object.

Specializing Inherited Methods:

- The tree-searching model of inheritance just described turns out to be a great way to specialize systems. Because inheritance finds names in derived classes before it checks base classes, derived classes can replace default behavior by redefining their base classes' attributes.
- In fact, we can build entire systems as hierarchies of classes, which are extended by adding new external derived classes rather than changing existing logic in-place. The idea of redefining inherited names leads to a variety of specialization techniques.
- For instance, derived classes may replace inherited attributes completely, provide attributes that a base class expects to find, and extend base class methods by calling back to the base class from an overridden method. Here is an example that shows how extension works.

Example : For specialized inherited methods.

```
class A:                          # parent class
    "Parent Class"
    def display(self):
        print ('This is base class.')
```

Scanned with CamScanner

```
class B(A):                            # derived class
    "Child/Derived class"
    def display(self):
        A.display(self)
        print ('This is derived class.')
obj = B()                              # instance of child
obj.display()                          # child calls overridden method
```

- The derived class replaces base's method function with its own specialized version, but within the replacement, derived calls back to the version exported by base class to carry out the default behavior.
- In other words, derived class.display() just extends base class.display() behavior, rather than replacing it completely.
- Extension is only one way to interface with a superclass.
- Following program defines multiple classes that illustrate a variety of common techniques:
  1. **Super:** Defines a method function and a delegate that expects an action in a subclass.
  2. **Inheritor:** Doesn't provide any new names, so it gets everything defined in Super.

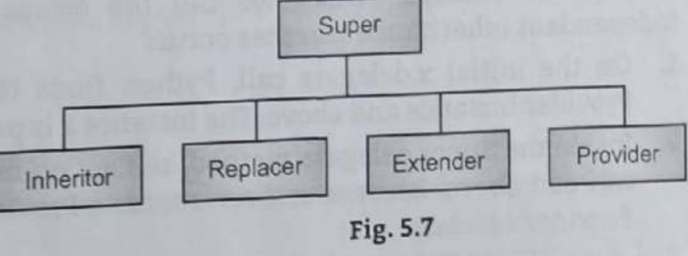


Fig. 5.7

  3. **Replacer:** Overrides Super's method with a version of its own.
  4. **Extender:** Customizes Super's method by overriding and calling back to run the default.
  5. **Provider:** Implements the action method expected by Super's delegate method.

Example: Give a feel for the various ways to customize a common superclass.

```
class Super:
    def method(self):
            print('in Super.method')       # Default behavior
    def delegate(self):
        self.action()                      # Expected to be defined
class Inheritor(Super):                    # Inherit method verbatim
    pass
class Replacer(Super):                     # Replace method completely
    def method(self):
            print('in Replacer.method')
class Extender(Super):                     # Extend method behavior
    def method(self):
        Super.method(self)
        print('in Extender.method')
class Provider(Super):                     # Fill in a required method
    def action(self):
            print('in Provider.action')
for klass in (Inheritor, Replacer, Extender):
            print('\n' + klass._ _name_ _ + '...')
            klass().method()
            print('\nProvider...')
x = Provider()
x.delegate()
```

Scanned with CamScanner

Output:

```
Inheritor...
in Super.method
Replacer...
in Replacer.method
Extender...
in Super.method
in Extender.method
Provider...
in Provider.action
```

- At the end of this program instances of three different classes are created in a for loop. Because classes are objects, you can put them in a tuple and create instances generically.
- Classes also have the special __name__ attribute, which preset to a string containing the name in the class header.
- In previous example, when we call the delegate method through a provider instance, two independent inheritance searches occur:
  1. On the initial x.delegate call, Python finds the delegate method in Super by searching the Provider instance and above. The instance x is passed into the method's self argument as usual.
  2. Inside the Super.delegate method, self.action invokes a new, independent inheritance search of self and above. Because self references a Provider instance, the action method is located in the Provider subclass.
- The superclass in this example is what is sometimes called an abstract superclass – a class that expects parts of its behavior to be provided by its subclasses.

Practice Questions

1. What is OOP?
2. List the features and explain about different Object Oriented features supported by Python.
3. List and explain built in class attributes with example.
4. Design a class that store the information of student and display the same.
5. What are basic overloading methods?
6. What is method overriding? Write an example.
7. Explain class inheritance in Python with an example.
8. How to declare a constructor method in python? Explain.
9. How operator overloading can be implemented in Python? Give an example.
10. Write a Python program to implement the concept of inheritance.
11. Create a class employee with data members: name, department and salary. Create suitable methods for reading and printing employee information.
12. What is data abstraction? Explain in detail.
13. Define the following terms:
    (i) Object
    (ii) Class
    (iii) Inheritance
    (iv) Data abstraction.
14. Describe the term composition classes with example.
15. Explain customization via inheritance specializing inherited methods.

■■■

Scanned with CamScanner

6... File I/O Handling and Exception Handling

Chapter Outcomes...

- Write Python code for the given reading values from keyboard.
- Read data from the given file.
- Write the given data to a file.
- Handle the given exceptions through Python program.

Learning Objectives...

- To understand File, I/O and Exception
- To study I/O Operations like Reading Input, Printing Output etc.
- To learn File Handling Concepts such as Opening, Reading, Writing, Renaming, Deleting, Accessing File Contents etc.
- To study Directories in Python, File and Directory related Standard Functions
- To understand Exception Handling in Python Programming

6.0 INTRODUCTION

- 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.
- A text file is a file that stores information in the term 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.

6.1 I/O OPERATIONS (READING KEYBOARD INPUT, PRINTING TO SCREEN)

- In any programming language an interface plays a very important role. It takes data from the user (input) and displays the output.
- One of the essential operations performed in Python language is to provide input values to the program and output the data produced by the program to a standard output device (monitor).
- The output generated is always dependent on the input values that have been provided to the program. The input can be provided to the program statically and dynamically.

Scanned with CamScanner

- In static input, the raw data does not change in every run of the program. While in dynamic input, the raw data has a tendency to change in every run of the program.
- Python language has predefined functions for reading input and displaying output on the screen. Input can also be provided directly in the program by assigning the values to the variables. Python language provides numerous built in functions that are readily available to us at Python prompt.
- Some of the functions like input() and print() are widely used for standard Input and Output (I/O) operations, respectively.

1. Output (Printing to Screen):

- The function print() is used to output data to the standard output devices i.e., monitor/screen. The output can redirect or store on to a file also.
- The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax: `print(object(s), separator=separator, end=end, file=file, flush=flush)`

Parameter Values:

(i) **object(s):** It can be any object but will be converted to string before printed.

(ii) **sep='separator'**: Optional. Specify how to separate the objects, if there is more than one. Default is ''.

(iii) **end='end'**: Optional. Specify what to print at the end. Default is '\n' (line feed).

(iv) **file:** Optional. An object with a write method. Default is sys.stdout.

(v) **flush:** Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False.

Example: For output using print().

```
>>> print("Hello", "how are you?", sep=" ---")
Hello ---how are you?
>>> print(10,20,30,sep='-')
10-20-30
```

- To make the output more attractive formatting is used. This can be done by using the str.format() method.

Example: For output using format().

```
>>> a=10
>>> b=20
>>> print('Value of a is {} and b is {}'.format(a,b))
Value of a is 10 and b is 20
>>> print('I will visit {0} and {1} in summer'.format('Jammu','Kashmir')
I will visit Jammu and Kashmir in summer
>>>
```

- Just like old sprint() style used in C programming language, we can format the output in Python language also. The % operator is used to accomplish this.

Example: For output with %.

```
>>> x=12.3456789
>>> print('The value of x=%3.2f'%x)
The value of x=12.35
>>> print('The value of x=%3.4f'%x)
The value of x=12.3457
```

Scanned with CamScanner

- The various format symbols available in Python programming are:

Sr. No.	Format Symbol	Conversion
1.	%c	Character.
2.	%s	String conversion via str() prior to formatting.
3.	%i	Signed decimal integer.
4.	%d	Signed decimal integer.
5.	%u	Unsigned decimal integer.
6.	%o	Octal integer.
7.	%x	Hexadecimal integer (lowercase letters).
8.	%X	Hexadecimal integer (UPPERcase letters).
9.	%e	Exponential notation (with lowercase 'e').
10.	%E	Exponential notation (with UPPERcase 'E').
11.	%f	Floating point real number.
12.	%g	The shorter of %f and %e.
13.	%G	The shorter of %f and %E.

2. **Input (Reading Keyboard Input):**

- Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard.

(i) **input(prompt):** The input(prompt) function allows user input. It takes one argument. The syntax is as follows:

Syntax: `input(prompt)`

where, Prompt is a String, representing a default message before the input.

Example: For input (prompt) method.

```
x = input('Enter your name:')
print('Hello, ' + x)
```

Output:

```
Enter your name:vijay
Hello, vijay
```

(ii) **input():** The function input() always evaluate the input provided by user and return same type data. The syntax is as follows:

Syntax: `x=input()`

- If input value is integer type then its return integer value. If input value is string type then its return string value.

Example: For reading input from keyboard.

```
>>> x=input()
5
>>> type(x)
<class 'int'>
>>> x=input()
5.6
>>> type(x)
<class 'float'>
>>> x=int(input())
```

Scanned with CamScanner

```
5
>>> type(x)
<class 'int'>
>>> x=float(input())
2.5
>>> type(x)
<class 'float'>
```

6.2 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.
- 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.

6.2.1 Opening File in different Modes

- All files in Python programming are required to be open before some operation (read or write) can be performed on the file. In Python programming while opening a file, file object is created, and by using this file object we can perform different set as operations on the opened file.
- Python has a built-in function open() to open a file. This function returns a file object also called a handle, as it is used to read or modify the file accordingly.

  Syntax: `file object = open(file_name [, access_mode][, buffering])`

Parameters:

file_name: The file_name argument is a string value that contains the name of the file that we 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. This is 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 we 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).

- If the path is in current working directory, we can just provide the filename, just like in the following examples:

```
>>> file=open("sample.txt")
>>> file.read()
'Hello I am there\n'   # content of file
>>>
```

Scanned with CamScanner

- If still we are getting "no such file or directory" error then use following command to confirm the proper file name and extension on current working directory (PWD).

```
>>> import os
>>> os.listdir()        # display file and folder of current working directory
['DLLs', 'Doc', 'etc', '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 file resides in a directory other than PWD, we have to provide the full path with the file name:

```
>>> file=open("D:\\files\\sample.txt")
>>> file.read()
'Hello I am there\n'
>>>
```

- We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode. In this mode, we get strings when reading from the file. The binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.
- The mode of the file specifies the possible operations that can be performed on the file i.e., what purpose we are opening a file.

6.2.2 Different Modes of Opening File

- Like, C, C++, and Java, a file in Python programming can be opened in various modes depending upon the purpose. For that, the programmer needs to specify the mode whether read 'r', write 'w', or append 'a' mode.
- Apart from this, two other modes exist, which specify to open the file in text mode or binary mode.
  1. The **text mode** returns strings while reading from the file. The default is reading in text mode.
  2. 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.

contd. ...

Scanned with CamScanner

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.
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
```

6.2.3 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.	file.closed	Returns true if file is closed, false otherwise.	>>> f=open("abc.txt")
			>>> f.closed
			False

2.	file.mode	Returns access mode with which file was opened.	>>> f=open("abc.txt","w")
			>>> f.mode
			'w'

3.	file.name	Returns name of the file.	>>> f=open("abc.txt","w") >>>f.name 'abc.txt'
4.	file.encoding	The encoding of the file.	>>> f=open("abc.txt","w") >>>f.encoding cp1252

Example: For file object attribute.

```
f=open("sample.txt","r")
print(f.name)
print(f.mode)
print(f.encoding)
f.close()
print(f.closed)
```

Scanned with CamScanner

Output:

```
sample.txt
r
cp1252
True
```

6.2.4 Closing File

- When we are done with operations to the file, we need to properly close the file.
- Closing a file will free up the resources that were tied with the file and is done using Python close() method.

Syntax: `fileObject.close()`

Example: For closing a file.

```
f=open("sample.txt")
print("Name of the file: ",f.name)
f.close()
```

Output:

```
Name of the file:  sample.txt
```

6.2.5 Writing Data to File

- The write() method writes any string to an open 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.
- The write() method writes the contents onto the file. It takes only one parameter and returns the number of characters writing to the file.
- The write() method is called by the file object onto which we want to write the data. 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.
- We can use three methods to write to a file in Python namely, write(string) (for text), write(byte_string) (for binary) and writelines (list).

1. **write(string) Method:**

- The write(string) method writes the contents of string to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

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())
```

Scanned with CamScanner

Output:

```
**content of file1**
Hello,I am There
**content of file1**
first line
second line
third line
```

2. **writelines(list) Method:**

- 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.

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
```

6.2.6 Reading Data from File

- To read a file in Python, we must open the file in reading mode (r or r+). The read() method in Python programming reads the contents of the file. It returns the characters read from the file.
- The read() is also called by the file object from which we want to read the data.
- There are following three methods available for reading data purpose:

1. **read([n]) Method:**

- The read() method just outputs the entire file if number of bytes are not given in the argument. If we execute read(3), we will get back the first three characters of the file.

Note: [n] means optional.

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([n]) Method:**

- The readline() method just output the entire line whereas readline(n) outputs at most n bytes of a single line of a file. It does not read more than one line. Once, the end of file is reached, we get empty string on further reading.

Scanned with CamScanner

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():** This method maintains a list of each line in the file.

Example: For roadlines() method.

```
f=open("sample.txt","r")
print(f.readlines())
```

Output:

```
['first line\n', 'second line\n', 'third line\n']
```

6.2.7 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
>>>
```

Scanned with CamScanner

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
```

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:`
			`whilef.next():`
			`print(f.next())`
			`except:`
			`f.close()`

6.	`file.read()`	This function reads the entire file and returns a string.	`lines =f.read()`
			`f.write(lines)`
			`f.close()`

7.	`file.read(size)`	Reads the given number of bytes. It may read less if EOF is hit.	`text =f.read(10)`
			`print(text)`
			`f.close()`

8.	`file.readline()`	Reads a single line and returns it as a string.	`text =f.readline()`
			`print(text)`
			`f.close()`

9.	`file.readline(size)`	It will read an entire line (trailing with a new line char) from the file.	`text =f.readline(20)`
			`print(text)`
			`f.close()`

contd. ...

Scanned with CamScanner

10.	`file.readlines()`	Reads the content of a file line by line and returns them as a list of strings.	`lines =f.readlines()`
			`f.writelines(lines)`
			`f.close()`

11.	`file.readlines(size_hint)`	It calls the readline() to read until EOF. It returns a list of lines read from the file. If you pass <size_hint>, then it reads lines equalling the <size_hint> bytes.	`text =f.readlines(25)`
			`print(text)`
			`f.close()`

12.	`file.seek(offset[, from])`	Sets the file's current position.	`position =f.seek(0,0);`
			`print(position)`
			`f.close()`

13.	`file.tell()`	Returns the file's current position.	`lines =f.read(10)`
			`#tell()`
			`print(f.tell())`
			`f.close()`

14.	`file.truncate(size)`	Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.	`f.truncate(10)` `f.close()`

15.	`file.write(string)`	It writes a string to the file. And it doesn't return any value.	`line ='Welcome Geeks\n'`
			`f.write(line)`
			`f.close()`

16.	`file.writelines(sequence)`	Writes a sequence of strings to the file. The sequence is possibly an iterable object producing strings, typically a list of strings.	`lines =f.readlines()`
			`#writelines()`
			`f.writelines(lines)`
			`f.close()`

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
```

Scanned with CamScanner

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"))
4.	os.path.isdir()	Is passed parameter a folder.	print(os.path.isdir("sample.txt"))
5.	os.listdir()	Returns a list of all files and folders of present working. directory.	print("***Contents of Present working directory***\n ",os.listdir())
6.	os.listdir(path)	Return a list containing the names of the entries in the directory given by path.	print("***Contents of given directory***\n ",os.listdir("testdir"))
7.	os.rename(current,new)	Rename a file.	os.rename("sample.txt","sample1.txt")
8.	os.remove(file_name)	Delete a file.	os.remove("sample.txt")
9.	os.mkdir()	Creates a single subdirectory.	os.mkdir("testdir")
10.	os.chdir(path)	Change the current working directory to path.	os.chdir("d:\IT")

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
```

Scanned with CamScanner

6.2.8 Renaming a File

- Renaming a file in Python is done with the help of the rename() method. To rename a file in Python, the OS module needs to be imported.
- The rename() method takes two arguments, the current filename and the new filename.

Syntax: `os.rename(current_file_name, new_file_name)`

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_ _']
>>>
```

6.2.9 Deleting a File

- We can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.
- To remove a file, the OS module need to be imported. The remove() in Python programming in used to remove the existing file with the file name.

Syntax: `os.remove(file_name)`

Example: For deleting files.

```
import os
print("***Contents of Present working directory***\n ",os.listdir())
os.remove("sample.txt")
print("***New Contents of Present working directory***\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_ _']
***New 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', 'Scripts', 'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll',
'_ _pycache_ _']
```

- The is file() method in Python programming checks whether the file passed in the method exists or not. It returns true if the file exist otherwise it returns false.

Scanned with CamScanner

6.2.10 Directories

- If there are a large number of files to handle in the Python program, we can arrange the code within different directories to make things more manageable.
- 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).

6.2.10.1 Create New Directory

- 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")
```

6.2.10.2 Get Current Directory

- We can get the present working directory using the getcwd() method. This 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'
```

6.2.10.3 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 (/) or 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'
>>>
```

6.2.10.4 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']
```

Scanned with CamScanner

6.2.10.5 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')
```

Additional Programs:

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: ");
```

Scanned with CamScanner

```
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.!!
>>>
```

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.!!
```

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)
```

Scanned with CamScanner

Output:

```
Enter file name: file1
Enter letter to be searched:o
Occurrences of the letter:
7
```

6.3 EXCEPTION HANDILNG

- When we executes 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
```

6.3.1 Introduction

- 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.
- 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.

Scanned with CamScanner

- Following table lists all the standard 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.
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 raw_input() or input() 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 print statement or the open() 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 next() method of an iterator does not point to any object.
21.	SystemExit	Raised by the sys.exit() function.
22.	StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
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.

Scanned with CamScanner

6.3.2 Exception Handling in Python Programming

- 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.

6.3.2.1 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.
```

Scanned with CamScanner

Example: 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: 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
```

6.3.2.2 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)
```

Scanned with CamScanner

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
```

6.3.2.3 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: 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: 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")
```

Scanned with CamScanner

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
```

6.3.3 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 condition 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.

6.3.4 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.
- 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
```

Scanned with CamScanner

```
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
```

Scanned with CamScanner

Practice Questions

1. What is file? Enlist types of files in Python programming.
2. What is exception?
3. Explain the term exception handling in detail.
4. Explain different modes of opening a file.
5. Write the syntax of fopen() with example.
6. What are various modes of file object? Explain any five as them.
7. Explain exception handling with example using try, except, raise keywords.
8. Explain try...except blocks for exception handling in Python.
9. Explain various built in functions and methods.
10. Explain open() and close() methods for opening and closing a file.
11. Explain any three methods associated with files in Python.
12. List and explain any five exceptions in Python.
13. List out keywords used in exception handling.
14. How python handles the exception? Explain with an example program.
15. Differentiate between an error and exception.
16. How to create a user defined exception?
17. Give the syntax and significance of input() method.
18. Give syntax of the methods which can be used to take input from the user in Python program.
19. Write a Python program which will throw exception if the value entered by user is less than zero.
20. Write a Python program to accept an integer number and use try/except to catch the exception if a floating point number is entered.
21. Write a Python program to read contents of first.txt file and write same content in second.txt file.
22. Write a Python program to append data to an existing file 'python.py'. Read data to be appended from the use. Then display the contents of entire file.
23. Write a Python program to read a text file and print number of lines, words and characters.
24. Describe the term file I/O handling in detail.

■■■

Scanned with CamScanner