SPPU New Syllabus

A Book Of

Python Programming

For MCA(Management) : Semester - II

[Course Code IT-21 : Credit - 03]

CBCS Pattern

As Per New Syllabus, Effective from June 2020

Vijay T. Patil

M.E.(Computer Engineering)

Head, Department of Computer Engineering (NBA Accredited)

Vidyalankar Polytechnic, Mumbai-37

Mrs. Manisha Pokharkar

M.E.(Computer Engineering)

Sr. Lecturer, Department of Computer Engineering (NBA Accredited)

Vidyalankar Polytechnic, Mumbai-37

Price ₹ 310.00

NIRALI PRAKASHAN

ADVANCEMENT OF KNOWLEDGE

N5316

Python Programming **ISBN 978-93-5451-148-6**

First Edition : June 2021

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39, Fax - (020) 25511379
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041
Mobile No. 9404233041/9850046517

➢ DISTRIBUTION CENTRES

PUNE

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

Nirali Prakashan (For orders outside Pune) : S. No. 28/27, Dhayari, Near Asian College Pune 411041
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

MUMBAI

Nirali Prakashan : 385, S.V.P. Road, Rasdhara Co-op. Hsg. Society Ltd.,
Girgaum, Mumbai 400004, Maharashtra; Mobile : 9320129587
Tel : (022) 2385 6339 / 2386 9976, Fax : (022) 2386 9976
Email : niralimumbai@pragationline.com

➢ DISTRIBUTION BRANCHES

JALGAON

Nirali Prakashan : 34, V. V. Golani Market, Navi Peth, Jalgaon 425001, Maharashtra,
Tel : (0257) 222 0395, Mob : 94234 91860; Email : niralijalgaon@pragationline.com

KOLHAPUR

Nirali Prakashan : New Mahadvar Road, Kedar Plaza, 1st Floor Opp. IDBI Bank, Kolhapur 416 012
Maharashtra. Mob : 9850046155; Email : niralikolhapur@pragationline.com

NAGPUR

Nirali Prakashan : Above Maratha Mandir, Shop No. 3, First Floor,
Rani Jhanshi Square, Sitabuldi, Nagpur 440012, Maharashtra
Tel : (0712) 254 7129; Email : niralinagpur@pragationline.com

DELHI

Nirali Prakashan : 4593/15, Basement, Agarwal Lane, Ansari Road, Daryaganj
Near Times of India Building, New Delhi 110002 Mob : 08505972553
Email : niralidelhi@pragationline.com

BENGALURU

Nirali Prakashan : Maitri Ground Floor, Jaya Apartments, No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003, Karnataka; Mob : 9449043034
Email: niralibangalore@pragationline.com

Other Branches : Hyderabad, Chennai

niralipune@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks

Preface ...

We take this opportunity to present this book entitled as **"Python Programming"** to the students of Second Semester - MCA (Management). The object of this book is to present the subject matter in a most concise and simple manner. The book is written strictly according to the New Syllabus (CBCS Pattern).

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts, its intricacies, programs and practices. This book will help the readers to have a broader view on Python Programming. The language used in this book is easy and will help students to improve their knowledge of programming and understand the matter in a better and happier way.

We sincerely thank Shri. Dineshbhai Furia and Shri. Jignesh Furia of Nirali Prakashan, for the confidence reposed in us and giving us this opportunity to reach out to the students of management studies.

We have given our best inputs for this book. Any suggestions towards the improvement of this book and sincere comments are most welcome on niralipune@pragationline.com.

Authors

■■■

Syllabus ...

1. Introduction & Components of Python [Weightage 15%, Sessions 07]

1.1 Understanding Python
1.2 Role of Python in AI and Data science
1.3 Installation and Working with Python
1.4 The Default Graphical Development Environment for Python - IDLE
1.5 Types and Operation
1.6 Python Object Types - Number, Strings, Lists, Dictionaries, Tuples, Files, User Defined Classes
1.7 Understanding Python Blocks
1.8 Python Program Flow Control
1.9 Conditional Blocks using if, else and elif
1.10 Simple for Loops in Python
1.11 for loop using ranges, string, list and dictionaries
1.12 Use of while loop in Python
1.13 Loop Manipulation using pass, continue, break and else
1.14 Programming using Python conditional and loops Block

Extra Reading: Python installation with windows, Linux and MAC OS, creating virtual environment, configuring python on EC2 instance, understanding Python IDE – (VSCode, PyCharm, Spyder), Installing Anaconda and Setting-up environment for Python.

2. Python Functions, Modules & Packages [Weightage 15%, Sessions 07]

2.1 Function Basics - Scope, nested function, non-local statements
2.2 Built-in functions
2.3 Arguments Passing, Anonymous Function: lambda
2.4 Decorators and Generators
2.5 Module basic usage, namespaces, reloading modules - math, random, datetime, etc.
2.6 Package: import basics
2.7 Python namespace packages
2.8 User defined modules and packages

Extra Readings: GUI framework in Python

3. Python Object Oriented Programming [Weightage 15%, Sessions 06]

3.1 Concept of Class, Object and Instances, Method call
3.2 Constructor, class attributes and destructors
3.3 Real time use of class in live projects
3.4 Inheritance, super class and overloading operators
3.5 Static and Class Methods
3.6 Adding and retrieving dynamic attributes of classes
3.7 Programming using OOPS
3.8 Delegation and Container

Extra Readings: Integrating GUI framework with OOP

4. Python Regular Expression **[Weightage 10%, Sessions 04]**

4.1 Powerful Pattern matching and searching
4.2 Power of Pattern Searching using regex in Python
4.3 Real time parsing of data using regex
4.4 Password, e-mail, URL validation using regular expression
4.5 Pattern finding programs using regular expression

Extra Readings: Web scrapping and pattern matching with regex

5. Python Multithreading and Exception Handling **[Weightage 10%, Sessions 05]**

5.1 Exception Handling
5.2 Avoiding code break using exception handling
5.3 Safe guarding file operation using exception handling
5.4 Handling and helping developer with error code
5.5 Programming using Exception handling
5.6 Multithreading
5.7 Understanding threads
5.8 Synchronizing the threads
5.9 Programming using Multithreading

Extra Readings: Multiprocessing, deadlock, synchronization, monitors and messaging queue

6. Python File Operation **[Weightage 05%, Sessions 02]**

6.1 Reading config files in Python
6.2 Writing log files in Python
6.3 Understanding read functions, read(), readline() and readlines()
6.4 Understanding write
6.5 Functions write() and writelines()
6.6 Manipulating file pointer using seek
6.7 Programming using file operations

Extra Readings: Reading and Writing the files on AWS S3 bucket

7. Python Database Interaction **[Weightage 10%, Sessions 05]**

7.1 Introduction to NoSQL database
7.2 Advantages of NoSQL database
7.3 SQL vs. NoSQL
7.4 Introduction to MongoDB with Python
7.5 Exploring Collections and Documents
7.6 Performing basic CRUD operations with MongoDB and Python

Extra Readings: Graph database like Neo4j with python

8. Python for Data Analysis **[Weightage 20%, Sessions 09]**

8.1 NumPy
8.2 Introduction to NumPy
8.3 Creating arrays, Using arrays and Scalars
8.4 Indexing Arrays, Array Transposition
8.5 Universal Array Function
8.6 Array Input and Output
8.7 Pandas
8.8 What are Pandas? Where it is used?
8.9 Series in pandas, pandas DataFrames, Index objects, Reindex
8.10 Drop Entry, Selecting Entries
8.11 Data Alignment, Rank and Sort
8.12 Summary Statics, Missing Data, Index Hierarchy
8.13 Matplotlib
8.14 Python for Data Visualization
8.15 Introduction to Matplotlib
8.16 Visualization Tools

Extra Readings: Text analytics with NLP and Python

■■■

Contents ...

■■■

1... Introduction & Components of Python

Objectives...

- To understand basic concepts in Python Programming.
- To learn Features and Environment for Python Programming.
- To learn Data Types in Python Programming.
- To study Looping in Python Programming.
- To understand Loop Manipulation Statements in Python.

1.1 UNDERSTANDING PYTHON

- Python is a high-level, interpreted, interactive and object-oriented programming language. Today, Python is the trendiest programming language.
- Python is a popular programming language because 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, supports GUI mode, interactive, versatile and interpreted, interfaces to all major commercial databases, and so on.

1.1.1 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, Small-Talk, 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.
- 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 the fundamental errors of the language. In Python 3.0, the print statement has been replaced with a `print( )` function.
- Python widely used in both industry and academic circles because of its simple, concise and extensive support of libraries.
- Python is available for almost all operating systems such as Windows, Mac, Linux/UNIX etc. Python can be downloading from **http://www.python.org/downloads**.

1.1.2 Applications of Python Programming

- Google's App Engine web development framework uses Python as an application language.
- Maya, a powerful integrated 3D modeling and animation system provides a Python scripting API.
- Linux Weekly News published by using a web application written in Python programming.
- Google makes extensive use of Python in its Web Search Systems.
- The popular YouTube Video sharing service is largely written in Python programming.
- The NSA (National Security Agency) uses Python programming for cryptography and intelligence analysis.
- iRobot uses Python programming to develop commercial and military robotic devices.
- The Raspberry Pi single-board computer promotes Python programming as its educational language.
- Nextflix and Yelp have both documented the role of Python in their software infrastructures.
- Industrial Light and Magic, Pixar and others uses Python programming in the production of animated movies.

1.1.3 Features of Python

1. **Easy to Learn and Use:** Python is easy to learn and use. It is developer-friendly and high level programming language. It 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 languages such as C++ and Java.

2. **Interpreted Language:** Python is an interpreted language. The 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, especially 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 an 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 three Greater Than symbols (>>>) appears as shown in Fig. 1.1.

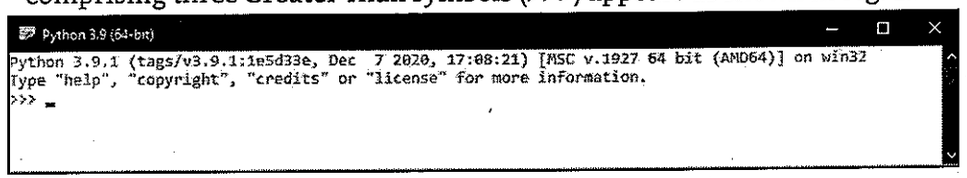


Fig. 1.1: Python Command Prompt

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.

Table 1.1: Python Editors

Editor	Description	Logo
IDLE	IDLE is a popular Integrated Development Environment written in Python and it has been integrated with the default language. Mainly used by beginner's level developers who want to practice Python development.	
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.	

contd. ...

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 features. It is very extensible and has a good plugin system and API.	SPYDER
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 interpreter.	PyDev
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 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.

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.

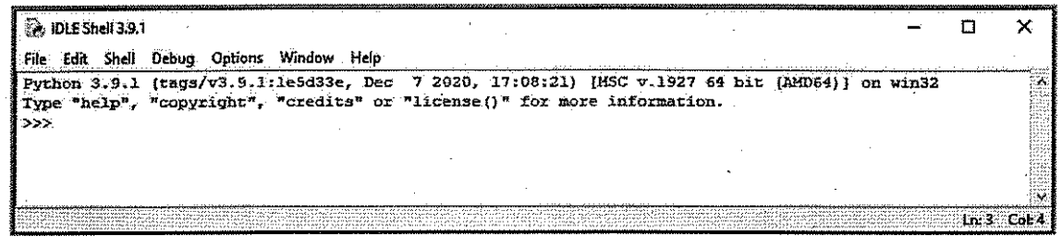


Fig. 1.2: Python Shell

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 anyone can use and read its source code make changes/modify 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.

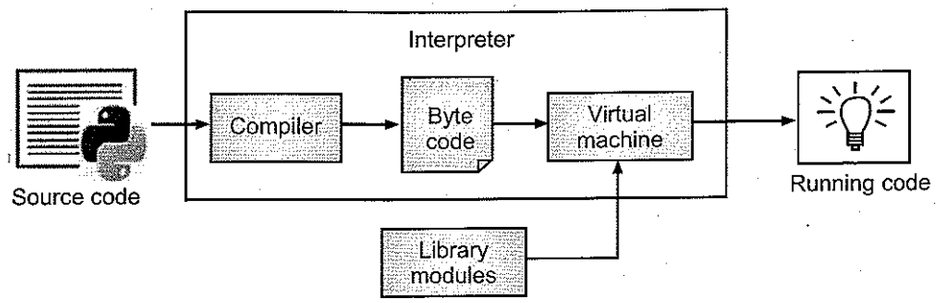


Fig. 1.3: Execution of Python Code

Python source code goes through Compiler which compiles the **source code** into a **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 codes 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.
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 a rich set of modules and functions for rapid application development. Python languages bulk library is portable and cross-platform compatible with UNIX, Windows etc.

1.1.4 Limitations of Python

- Python is an interpreter based language. Therefore, it is a bit slower than compiler based languages.
- 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.
- Graphics intensive applications such as games make the program to run slower.
- Due to the flexibility of the data types, Python's memory consumption is also high.

1.1.5 Structure of a Python Program

- Fig. 1.4 shows a typical program structure of Python programming.

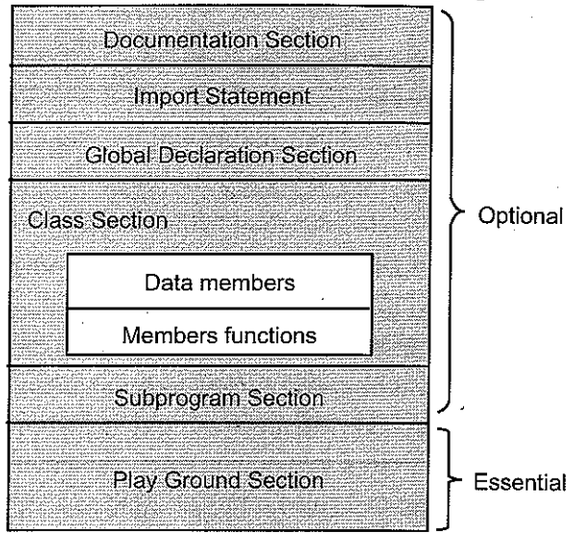


Fig. 1.4: Program structure of Python programming

- Python 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 the order of execution using control statements.
- Program structure of Python programming contains following sections:
  - **Documentation Section** includes the comments that specify the purpose of the program. Comments are a non-executable statement that is ignored by the compiler while program execution. Python comments are written anywhere in the program.
  - **Import Statement Section** is used includes different built-in or user defined modules.
  - **Global Declaration Section** is used to define the global variables for the programs.
  - **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 operates on data members.
  - **Subprogram Section** includes user defined functions. The functions include the set of statements that need to be executed when the function is called from anywhere.
  - **Play Ground Section** is the main section of Python program and the main section starts where the function calling.

1.1.6 Running Python Script

- 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.

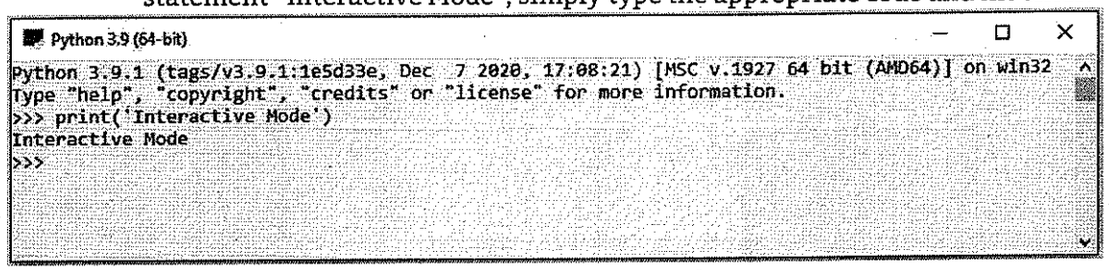


Fig. 1.5 (a): Interactive Mode

    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 output. To run the script, either select "Run" → "Run Module" or press F5 key.

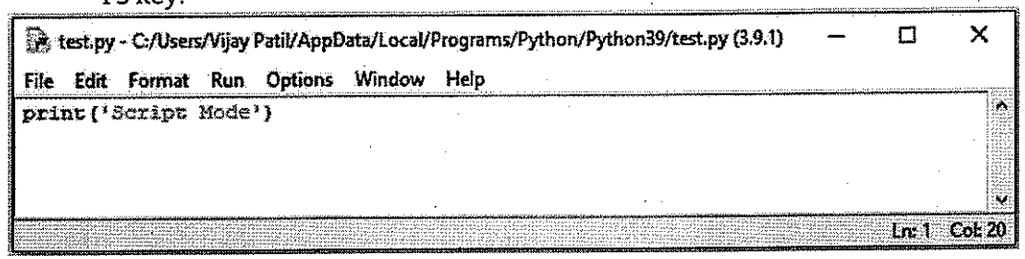


Fig. 1.5 (b): Script Mode

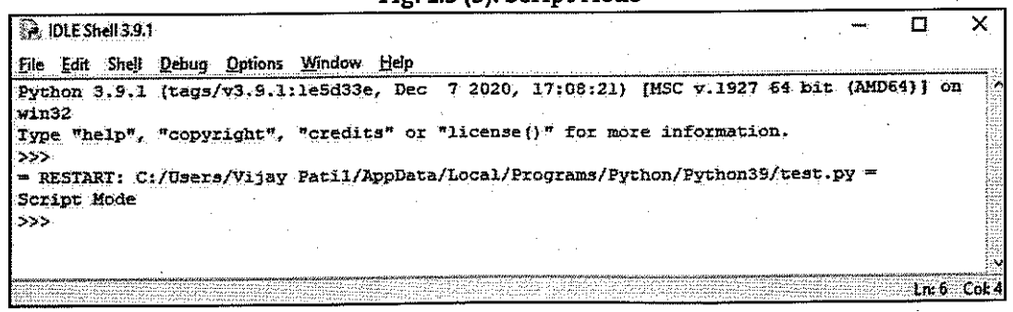


Fig. 1.5 (c) : Execute Script

1.1.7 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 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. 1.6.

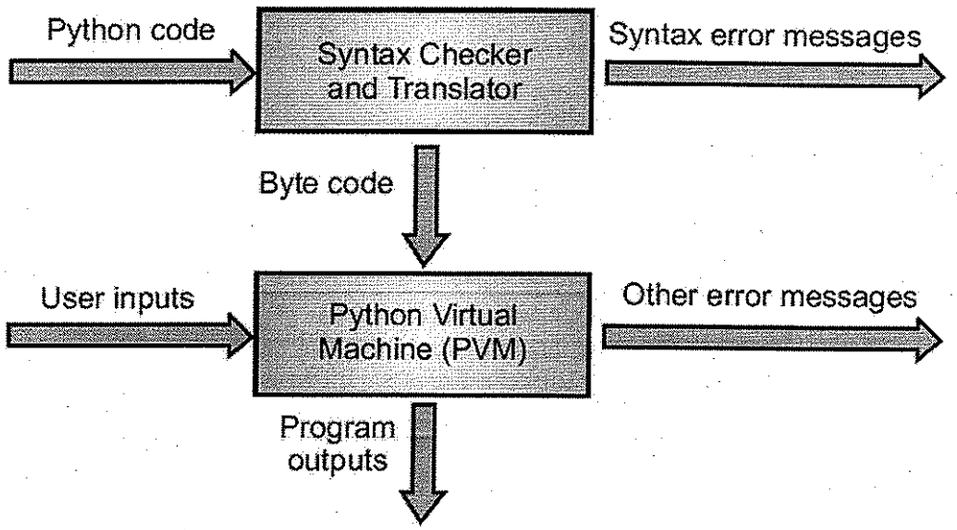


Fig. 1.6: Internal Operations

- The Python interpreter performs the 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 ROLE OF PYTHON IN AI AND DATA SCIENCE

- Artificial Intelligence (AI) is an area of computer science that gives priority to creating intelligent machines that can work like human-beings such as speech recognition, visual perception, translation, and even decision making. *NumPy* for scientific computation, *PyBrain* for Python Machine Language, *SciPy* for advanced Computing, are the libraries to make Python is the suitable language for AI. IDE (Integrated Development Environment) supports to avoid struggles of developers with the use of many algorithms.

- Python has various library packages to develop and decode the AI project and Data Science. Some of the specialized packages as follows:
  - **NumPy:** NumPy stands for Numerical Python. NumPy is a Python library that provides mathematical functions to handle large dimension arrays. It is used as a container for generic data comprising of an N-Dimensional Array Object, Integrating tools for C/C++ Codes, Random Number Capacity, and other functions. The library provides vectorization of mathematical operations on the NumPy array type, which enhances performance and speeds up the execution.
  - **Pandas:** It gives easy-to-use data structures and analytics tools for Python and it is an open-source library. Pandas is one of the most popular Python library for data manipulation and analysis. Pandas provide useful functions to manipulate large amount of structured data. Pandas provide the easiest method to perform analysis. It provides large data structures and manipulating numerical tables and time series data. Pandas is a perfect tool for data wrangling. Pandas is designed for quick and easy data manipulation, aggregation, and visualization. There are two data structures in Pandas:
    - **Series:** It handles and stores data in one-dimensional data.
    - **DataFrame:** It handles and stores two-dimensional data.
  - **Matplotlib:** It is a 2D plotting library to create publication-quality images. Matplotlib is another useful Python library for Data Visualization. Matplotlib allows making quick line graphs, pie charts, histograms, and other professional grade figures. Matplotlib has interactive features like zooming and planning and saving the Graph in graphics format.
  - **SciPy:** SciPy is another popular Python library for data science and scientific computing. SciPy provides great functionality to scientific mathematics and computing programming. SciPy contains sub-modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers, Statmodel and other tasks common in science and engineering.
  - **AIMA:** "Artificial Intelligence – A Modern Approach" by Russel and Norvig used for implementing algorithms for general AI.
  - **pyDatalog:** It is a logic programming engine in Python.
  - **EasyAI:** It used to create a simple Python engine for two-player games with AI. (Ex: Negamax)

- **PyBrain:** A flexible, simple but effective algorithm for Machine Learning which provides predefined environments to test and compare algorithms.
- **PyML:** It is a bilateral framework for SVMs and other kernel methods. And also it supported on Linux and Mac OS X.
- **Scikit-Learn/ Sklearn:** It is the most popular library for Machine Learning that is a highly effective tool for data analysis. Sklearn is built on NumPy, SciPy, and matplotlib. Sklearn provides easy and simple tools for data mining and data analysis. It provides a set of common machine learning algorithms to users through a consistent interface. Scikit-Learn help to quickly implement popular algorithms on datasets and solve real-world problems.
- **MDP-Toolkit:** It has both supervised and unsupervised learning algorithm which can be expanded to the use of network architecture.
- **NLTK:** Natural Language Toolkit (NLTK) is used to do linguistic data and documentation for research and development in text analysis and Natural Language Processing.

1.3 INSTALLATION AND WORKING WITH PYTHON

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

1.3.1 Installation of 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 as shown in Fig. 1.7 (a).

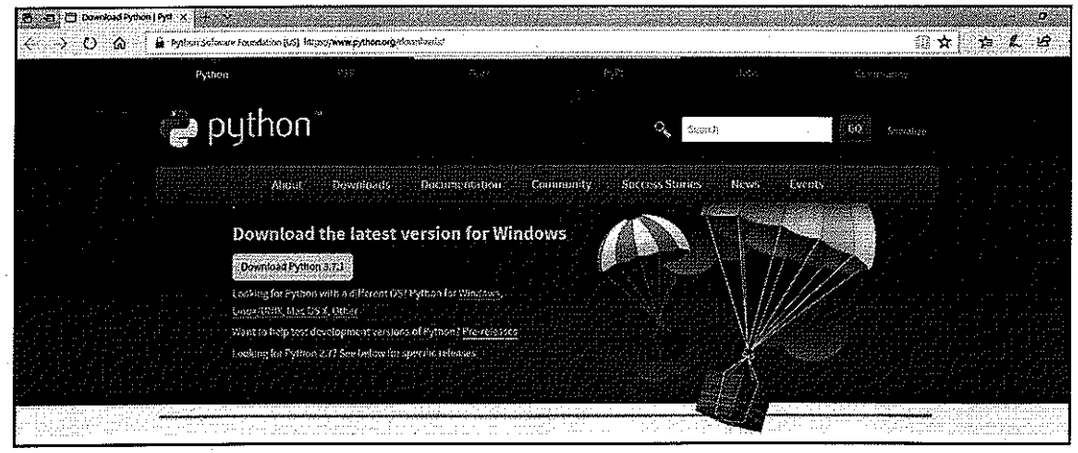


Fig. 1.7 (a): Home Page

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

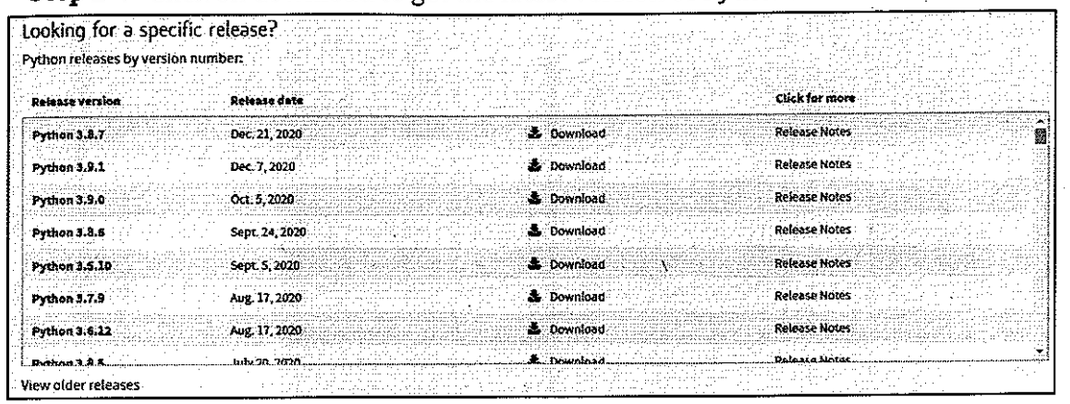


Fig. 1.7 (b): Python Release Versions

Step 3 : Once Python version is demanded (e.g. 3.7.1. User can download latest versions). Open the Python 3.7.1 version pack and double click on it to start installation.

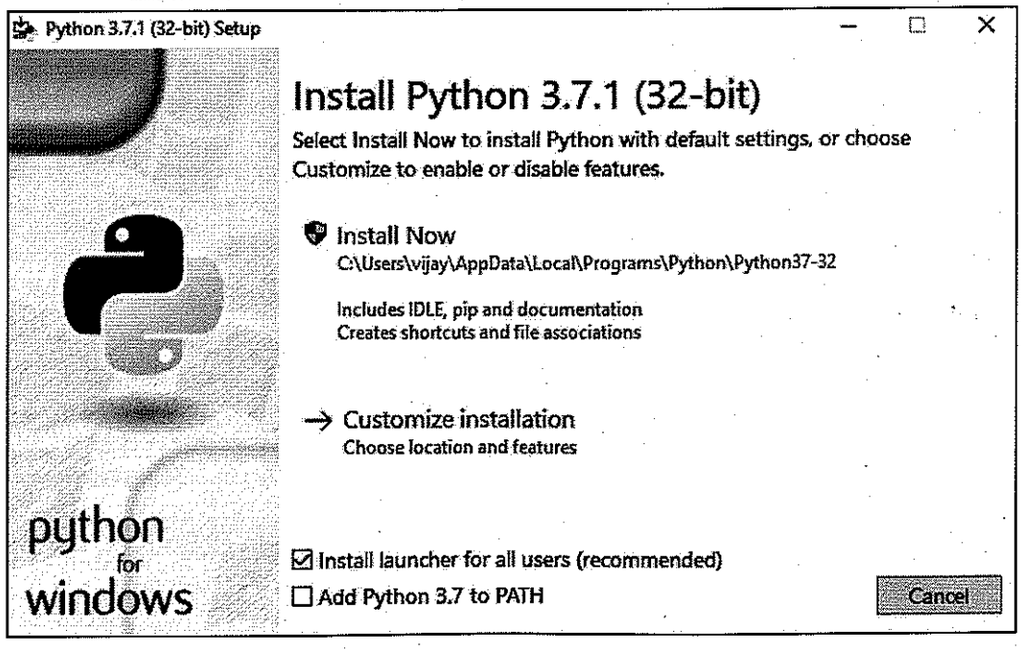


Fig. 1.7 (c): Python Installation

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

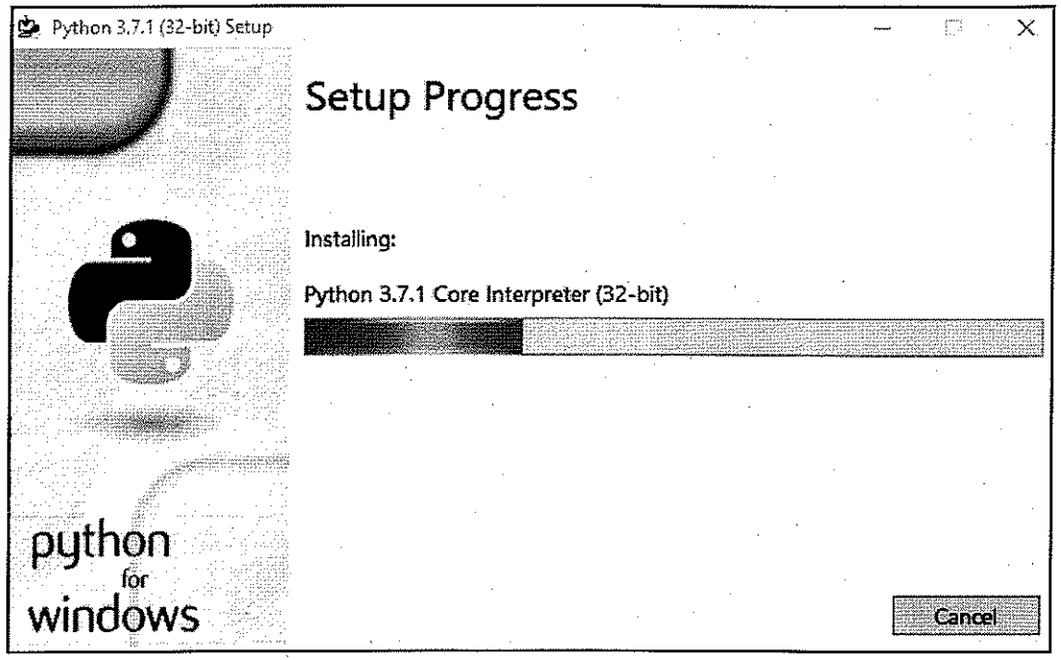


Fig. 1.7 (d): Setup Progress

Step 5 : After complete the installation close the windows.

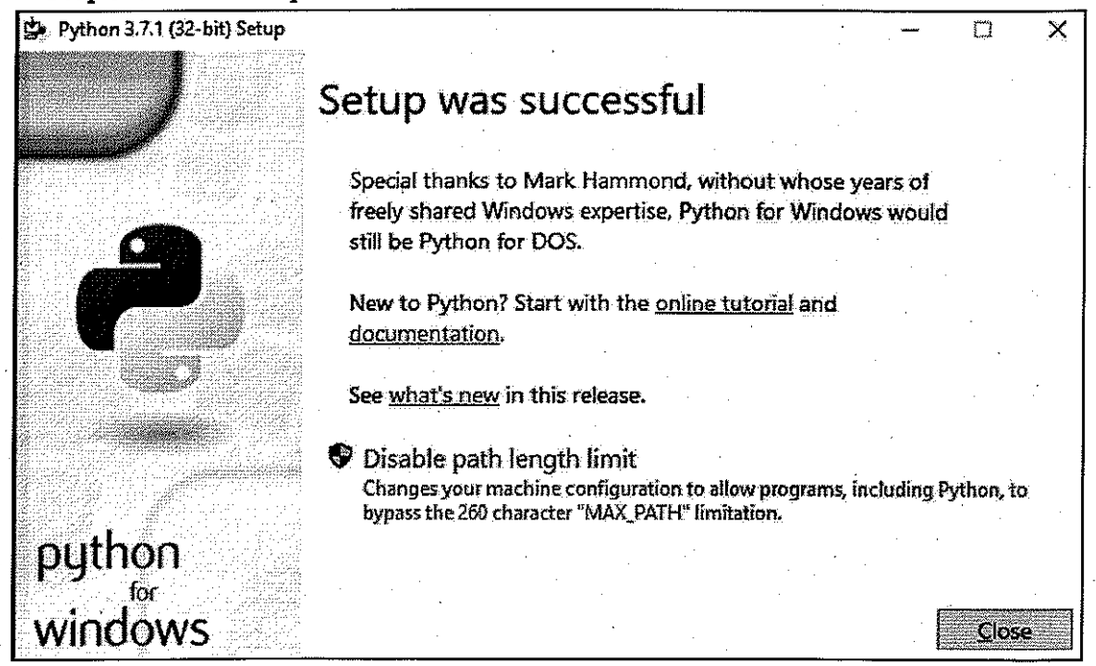


Fig. 1.7 (e): Complete the Installation

1.3.2 Starting Python in Command Line Modes

- 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. These steps are given below:

Step 1 : Press Start button.

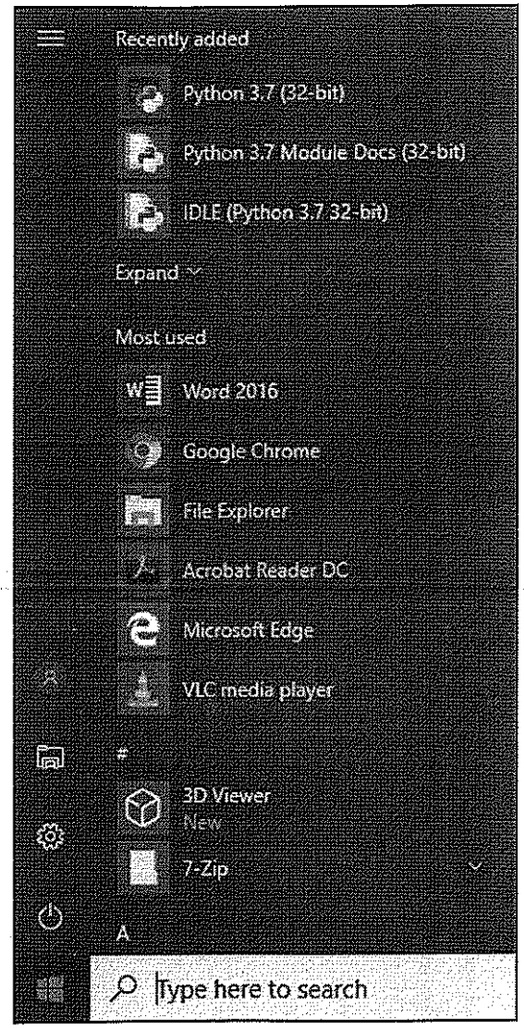


Fig. 1.8 (a): Python 3.7 (32-bit) options

Step 2 : Click on **All programs** section and then click on Python 3.7 (32-bit). You will see the Python interactive prompt in Python command line.

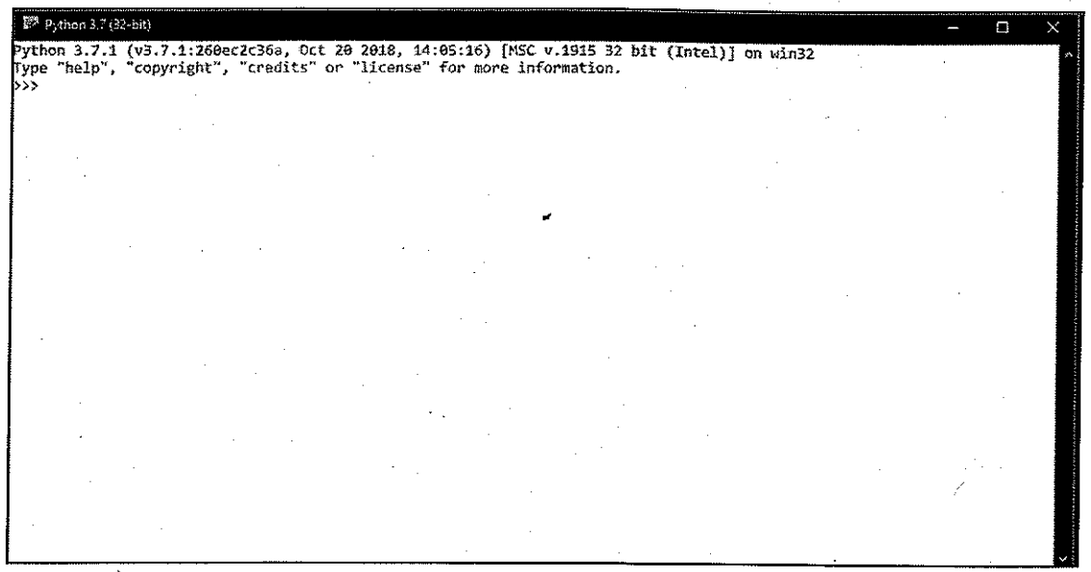


Fig. 1.8 (b): Command Prompt

Python command line 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.

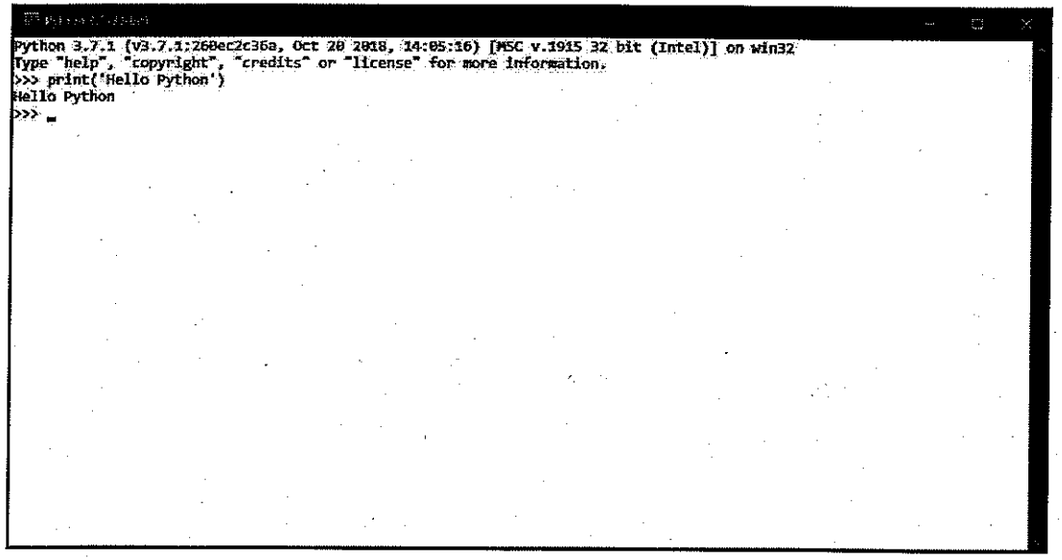


Fig. 1.8 (c): Python command line

Step 3 : To exit from the command line of Python, press Ctrl+Z keys followed by Enter key or type exit() or quit() and press Enter key.

1.4 THE DEFAULT GRAPHICAL DEVELOPMENT ENVIRONMENT FOR PYTHON - IDLE

- **Starting Python IDLE:** When we install Python3, we also get IDLE (Integrated Development Environment). IDLE includes a color syntax-highlighting editor, a debugger, the Python Shell, and a complete copy of Python3's online documentation set. The steps are given below:

  Step 1 : Press start button and click on IDLE (Python 3.7 32-bit) options.

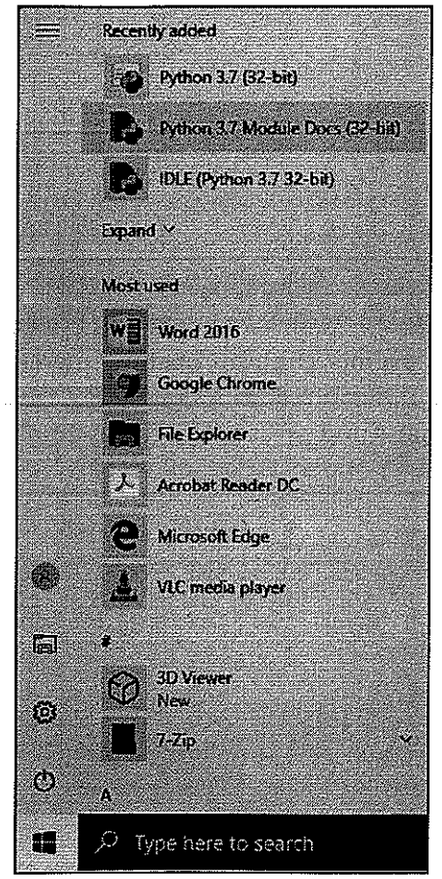


Fig. 1.9 (a): IDLE (Python 3.7 32-bit) options

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

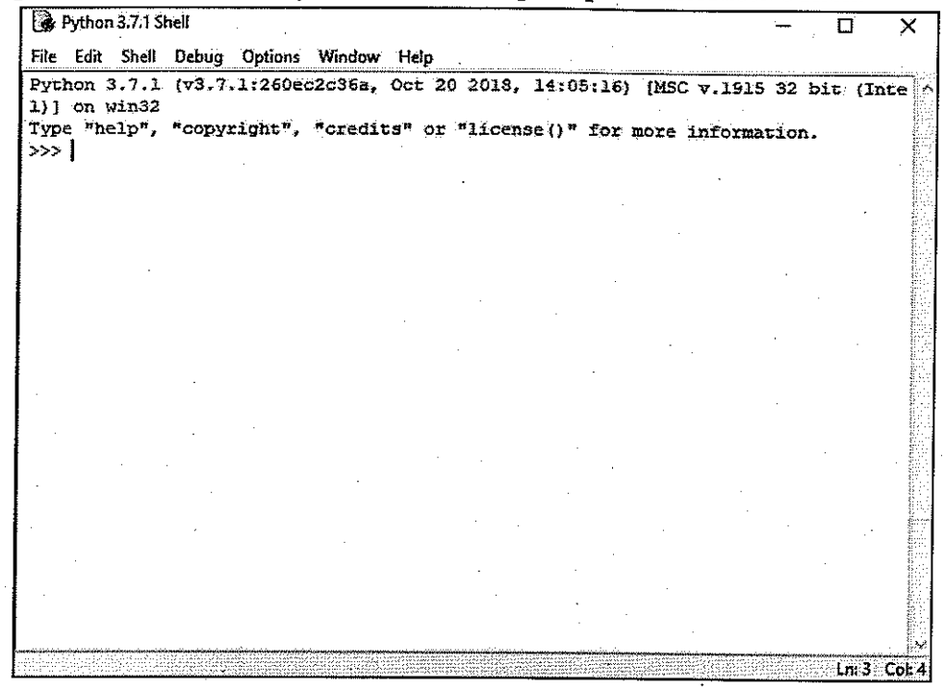


Fig. 1.9 (b): Python Interactive Prompt

- 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 you write a command and press enter, the python interpreter will immediately display the result.

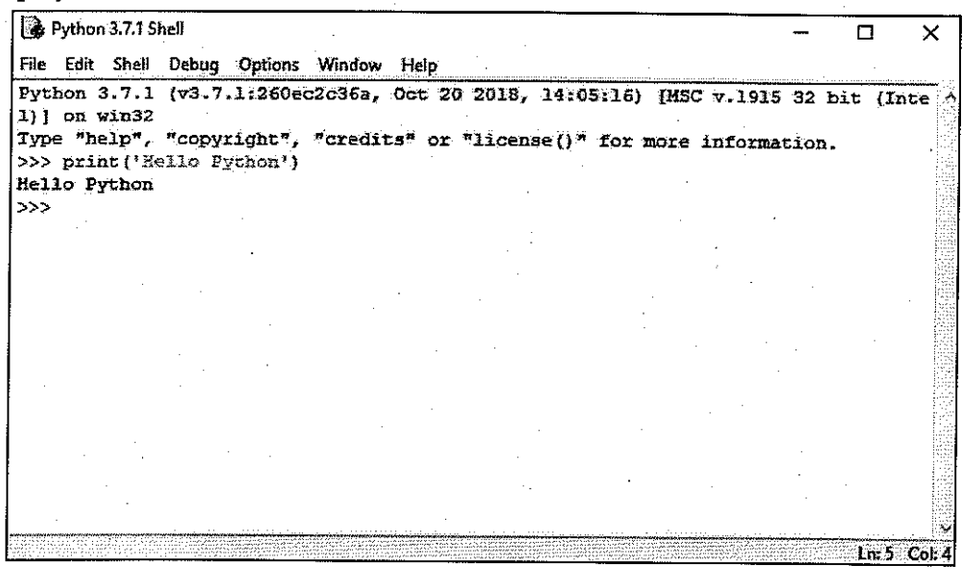


Fig. 1.9 (c): Display result by Interpreter

Executing Python Programs:

- In Python IDLEs-shell window, click on **File menu**, and then on **New File** option or press **Ctrl+N keys.**

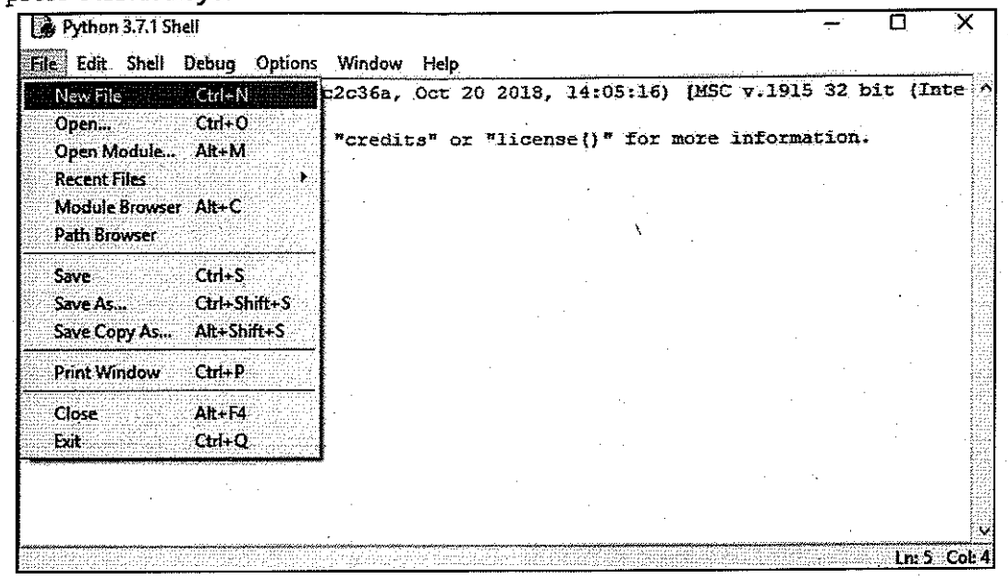


Fig. 1.9 (d): New file

- As soon as you click on **New File**, the window appears as:

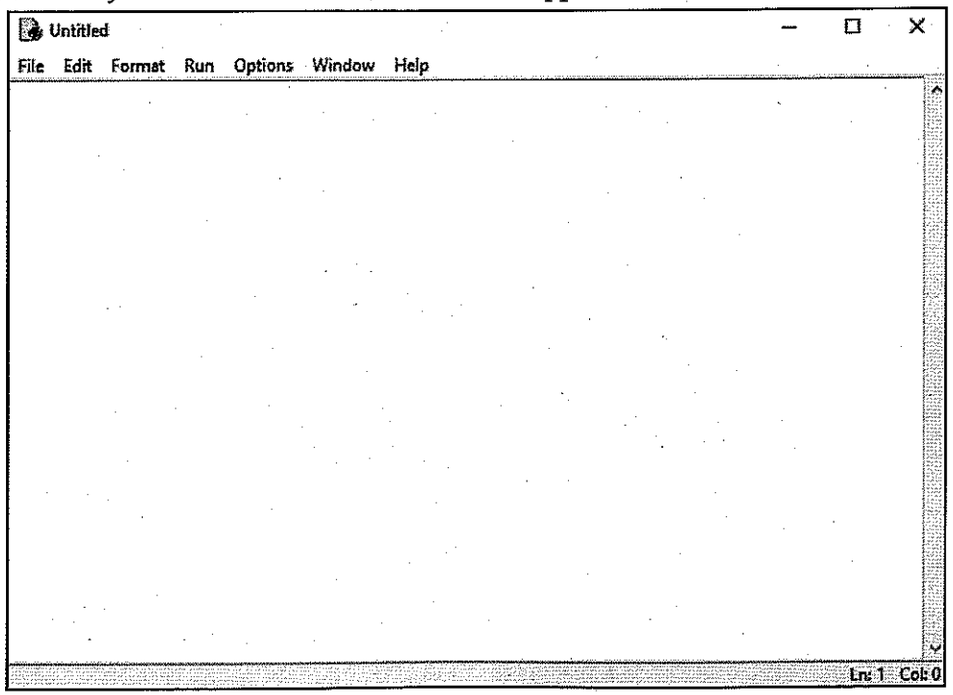


Fig. 1.9 (e): Script Mode

- Write Python program in script mode.

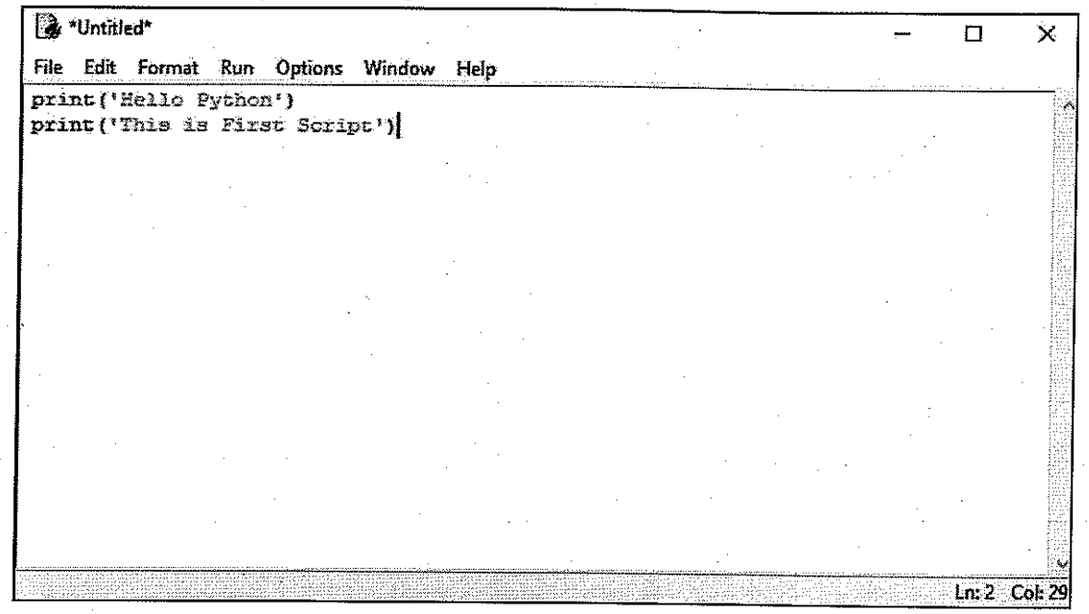


Fig. 1.9 (f): Program in script mode

- 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.

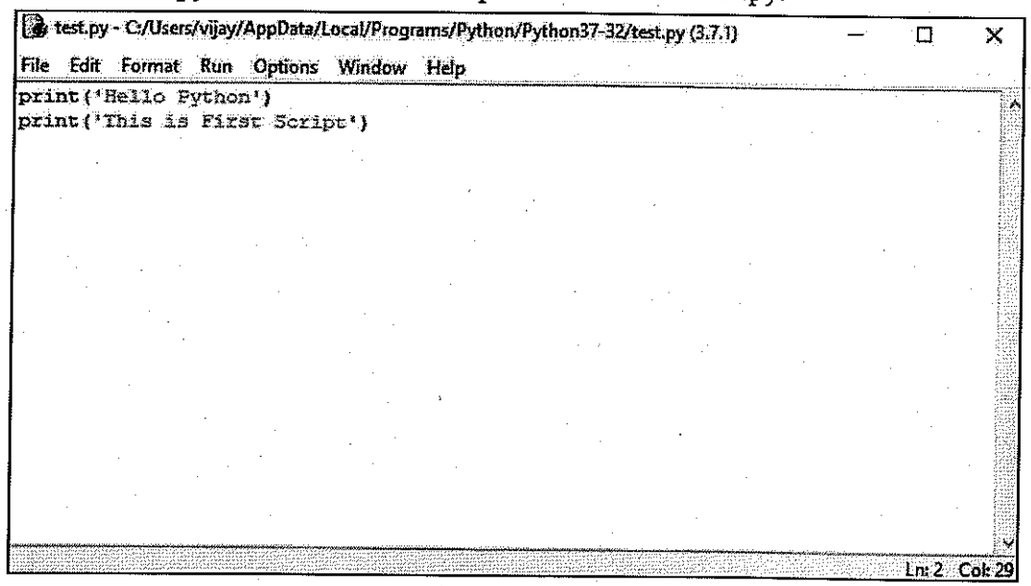


Fig. 1.9 (g): Save the script with file name

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

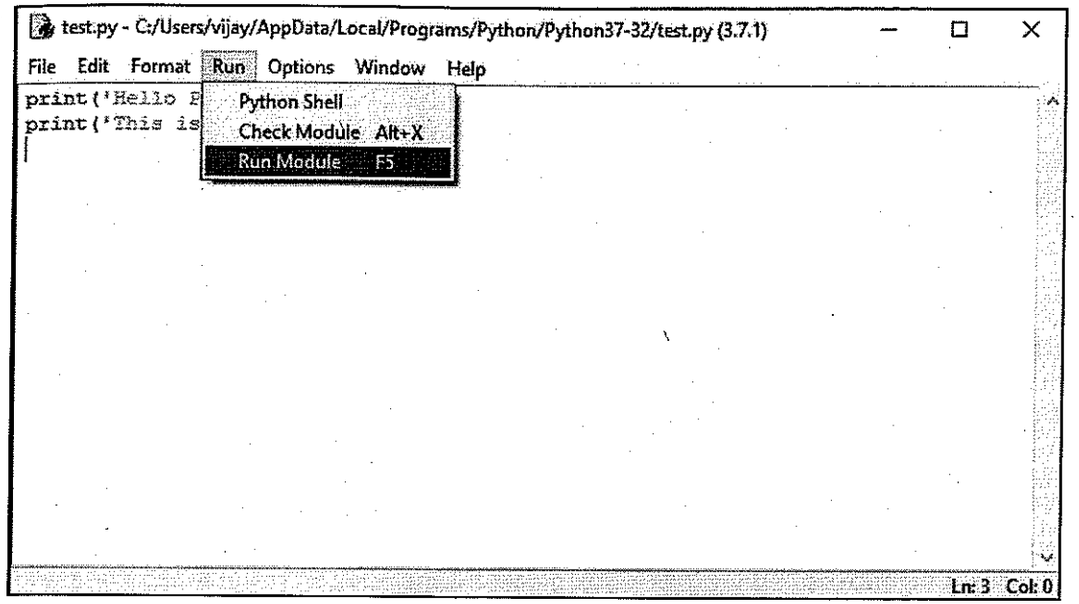


Fig. 1.9 (h): Run Module

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

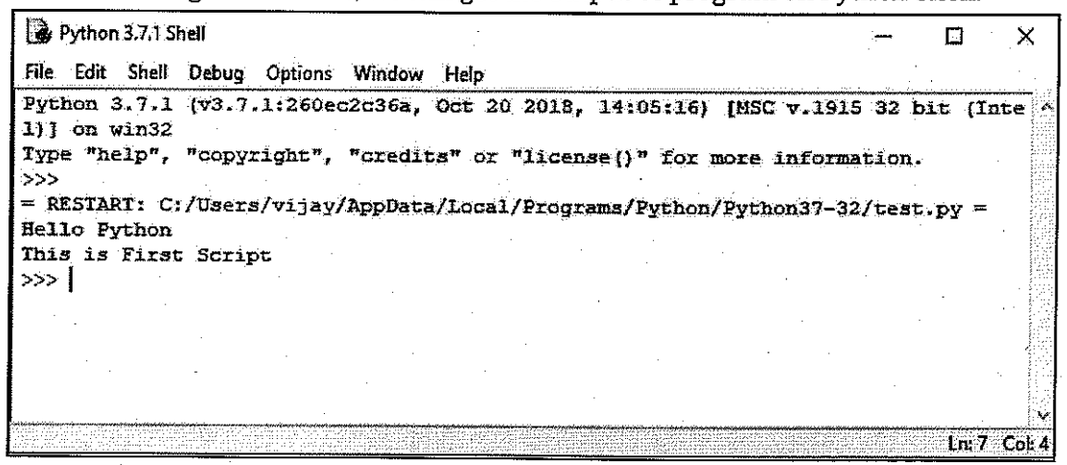


Fig. 1.9 (i): Output on Python shell

1.5 TYPES AND OPERATION

- An operator is a symbol that 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.

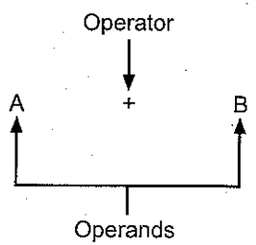


Fig. 1.10: Concept of Operator and Operands

- In Python, the operators can be unary operators or binary operator.

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).

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
30
>>> -x
-10
>>> 2+3
5
```


Expression:

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

  Example:

```
1 + 8
(3 * 9) / 5
a * b + c * 3
```

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

1.5.1 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.

Table 1.2: Arithmetic Operators

Operator Type	Operator	Description	Example
+	Addition	Adds the value of the left and right operands	>>> a + b 30
–	Subtraction	Subtracts the value of the right operand from the value of the left operand	>>> b - a 10
*	Multiplication	Multiplies the value of the left and right operand	>>> a * b 200
/	Division	Divides the value of the left operand by the right operand	>>> b / a 2.0
**	Exponent	Performs exponential calculation	>>> a ** 2 100
%	Modulus	Returns the remainder after dividing the left operand with the right operand	>>> a % b 10
//	Floor Division	Division of operands where the solution is a quotient left after removing decimal numbers	>>> b // a 2

1.5.2 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:

Table 1.3: Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand.	c = a + b assigns value of a + b into c
+=	It adds right operand to the left operand and assigns the result to left operand.	c += a is equivalent to c = c + a
-=	It subtracts right operand from the left operand and assigns the result to left operand.	c -= a is equivalent to c = c - a
*=	It multiplies right operand with the left operand and assigns the result to left operand.	c *= a is equivalent to c = c * a
/=	It divides left operand with the right operand and assigns the result to left operand.	c /= a is equivalent to c = c / a
%=	It takes modulus using two operands and assigns the result to left operand.	c %= a is equivalent to c = c % a
**=	Performs exponential (power) calculation on operators and assign value to the left operand.	c **= a is equivalent to c = c ** a
//=	Performs exponential (power) calculation on operators and assign value to the left operand.	c //= a is equivalent to c = c // a

1.5.3 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 hold the value 10 and variable b hold the value 20.

Table 1.4: Relational or Comparison Operators

Operator	Description	Example
== (Equality Operator)	If the values of two operands are equal, then the condition becomes true.	>>> (a==b) False
!= (Not Equality Operator)	If values of two operands are not equal, then condition becomes true.	>>> (a!=b) True
> (Greater than Operator)	If the value of left operand is greater than the value of right operand, then condition becomes true.	>>> (a>b) False
< (Less than Operator)	If the value of left operand is less than the value of right operand, then condition becomes true.	>>> (a<b) True
>= (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
<= (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

1.5.4 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).
- For following table, assume the variable **a** hold True and variable **b** hold False value.

Table 1.5: Logical Operators

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

1.5.5 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.
- Following table shows bitwise operators assume a=10 (1010) and b=4 (0100).

Table 1.6: Bitwise Operators

Operator	Description	Example
& (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
| (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
^ (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
~ 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
<< (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
>> (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

Table 1.7: The outcome of each bit operation

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

1.5.6 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 do not imply that they are identical.

Table 1.8: Identity Operators

Operator	Description	Example

Is	Return true, if the operands are same. Return false, if the operands are not same.	>>> a=3
		>>> b=3
		>>> print(a is b)
		True

Is not	Return, false, if the operands are same. Return true, if the operands are not same.	>>> a=3
		>>> b=3
		>>> print(a is not b)
		False

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.

1.5.7 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.

Table 1.9: Membership Operators

Operator	Description	Example

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`

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`

Example:

```
x = 'Hello world'
y = {1:'a',2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)

# Output: False
print('a' in y)
```

1.5.8 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.

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.

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
```

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

Table 1.10: Python Operator Precedence

Operator	Description
(),[]	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
<=, <, >, >=	Comparison Operator
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment Operators
is, is not	Identity
in, not in	Membership operators
Not, OR, AND	Logical Operators NOT, AND, OR

1.6 PYTHON OBJECT TYPES - Number, Strings, Lists, Dictionaries, Tuples, Files, User defined Classes

- 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
>>>
```

Data types:

- 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).
7. **Set:** Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements.

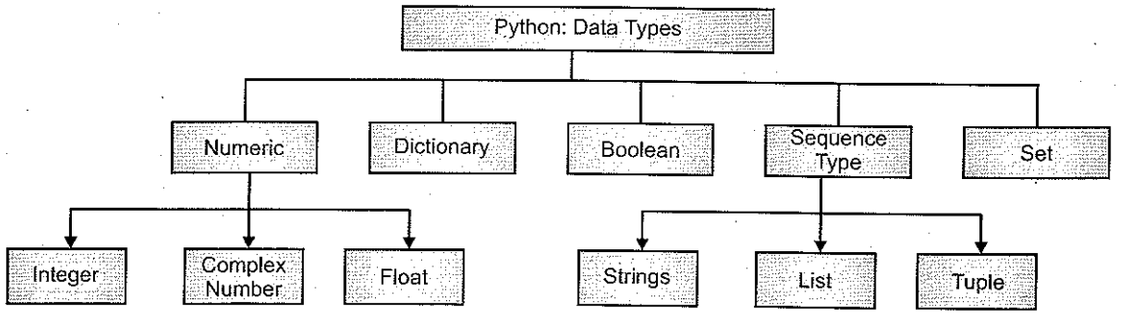


Fig. 1.11: Python-Data Types

1.6.1 Numeric Data Type

- Numeric data types store number values. Number objects are created when we assign a value to them. Integers, floating point numbers and complex numbers fall under Python numeric category. They are defined as Int, Float and Complex in Python.

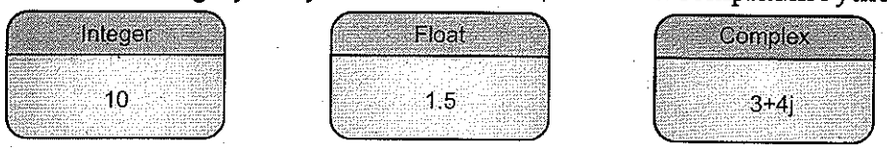


Fig. 1.12: Types of Number Data Types

- **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.
- **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.
- **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.
- 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 Python3, 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 prefix to the integer.

Table 1.11: Prefixes to the integer

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

Example: Integers in binary, octal and hexadecimal formats.

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

2. 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
```

3. 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
```

1.6.2 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'
>>> 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'
>>>
```

Split and Join a String:

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

Example: 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 an argument
>>> l
['Python', 'programming', 'is', 'easy']
>>>
>>> type(l)
<class 'list'>
```

- Use `join()` method to join the list back into a string, with a specified delimiter in between. The outcome of `join()` method gives us a string.

  Example: For `join()` function/method.

```
>>> l
['Python', 'programming', 'is', 'easy']
>>>
>>> type(l)
<class 'list'>
>>> s=' '.join(l)
>>> s
'Python programming is easy'
>>> s='.'.join(l)
>>> s
'Python.programming.is.easy'
>>> type(s)
<class 'str'>
```

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.

Table 1.12: String Operations

String Operation	Explanation	Example
+	Adds two strings together.	`x = "hello"+"world"`
*	Replicates a string.	`x = " "*20`
upper	Converts a string to uppercase.	`x.upper()`
lower	Converts a string to lowercase.	`x.lower()`
title	Capitalizes the first letter of each word in a string.	`x.title()`
find, index	Searches for the target in a string.	`x.find(y) x.index(y)`
rfind, rindex	Searches for the target in a string, from the end of the string.	`x.rfind(y) x.rindex(y)`
startswith, endswith	Checks the beginning or end of a string for a match.	`x.startswith(y)` `x.endswith(y)`
replace	Replaces the target with a new string.	`x.replace(y,z)`
strip, rstrip, lstrip	Removes whitespace or other characters from the ends of a string.	`x.strip()`
encode	Converts a Unicode string to a bytes object.	`x.encode("utf_8")`

1.6.3 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']
```

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

  Example: For list with updation/alteration/modification.

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

List and Strings:

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

  Example: For conversion of string to a list.

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

1.6.4 Tuple Data Type

- Tuple is an ordered sequences of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and that are usually faster than list as it cannot change dynamically.
- It is defined within parentheses ( ) where items are separated by comma (,). A tuple data type in Python programming is similar to a list data type, which also contains heterogeneous items/elements.

Example: For tuple.

```
>>> a=(10,'abc',1+3j)
>>> a
(10, 'abc', (1+3j))
>>> a[0]
10
>>> a[0]=20                  # tuples are immutable/unchangeable
Traceback (most recent call last):
File "<pyshell#12>", line 1, in <module>
a[0]=20
TypeError: 'tuple' object does not support item assignment
```

1.6.5 Dictionary

- Dictionary is an unordered collection of key-value pairs. It is the same as the hash table type. The order of elements in a dictionary is undefined, but we can iterate over the following:
  - The key
  - The value
  - The items (key-value pairs) in a dictionary.
- When we have the large amount of data, the dictionary data type is used. The dictionary data type is mutable in nature which means we can update modify/update any value in the dictionary. Items in dictionaries are enclosed in curly braces { } and separated by the comma (,). A colon (:) is used to separate key from value. Values can be assigned and accessed using square braces ([]).

Example: For dictionary data type.

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


1.6.6 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
```

1.6.7 Set

- Set is an unordered collection of unique items. The sets in Python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access its elements and carry out these mathematical operations.

Accessing values in set:

- Set is defined by values separated by comma inside braces { }. Items in a set are not ordered. A set does not contain any duplicate values or elements.

Example:

```
>>> a={1,3,5,4,2}
>>> print("a=",a)
a= {1, 2, 3, 4, 5}
>>> print(type(a))
<class 'set'>
>>>
```

Updating set:

- We can add elements to a set by using `add()` method. Again as discussed there is no specific index attached to the newly added element.

```
>>> a={1,2,3,4}
>>> a.add(6)
>>> print(a)
{1, 2, 3, 4, 6}
```

Deleting values in set:

- We can remove elements from a set by using `discard()` method.

```
>>> b={'a','b','c'}
>>> print(b)
{'b', 'a', 'c'}
>>> b.discard('b')
>>> print(b)
{'a', 'c'}
```

1.6.8 Files

- A file object allows us to use, access and manipulate all the user accessible files. One can read and write any such files. A file can be opened with a built-in function called `open()`. This function takes in the file's address and the `access_mode` and returns a file object.

Syntax: `file object = open(file name[, access_mode][, buffersize])`

Table 1.13: The valid values of Access mode parameters

Access Modes	Description
r	Opens a file for reading only.
rb	Opens a file for reading only in binary format.
r+	Opens a file for both reading and writing.
rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only.
wb	Opens a file for writing only in binary format.
w+	Opens a file for both writing and reading.
wb+	Opens a file for both writing and reading in binary format.
a	Opens a file for appending.
ab	Opens a file for appending in binary format.
a+	Opens a file for both appending and reading.
ab+	Opens a file for both appending and reading in binary format.

Writing to file:

- It writes the contents of string to the file.

```
f=open("D:\test.txt","w")
f.write("Python Programming")
f.close()
```

Writing lines to file:

```
lines=["Hello world.\n", "Welcome to Python Programming.\n"]
f=open("D:\test.txt","w")
f.writelines(lines)
f.close()
```


Reading from a file:

- Three different methods are provided to read data from file.
  1. **readline():** Reads the characters starting from the current reading position upto a newline character.
  2. **read(chars):** Reads the specified number of characters starting from the current position.
  3. **readlines():** Reads all lines until the end of file and returns a list object.

Example:

```
f=open("test.txt","r")
line=f.readline()
print(line)
f.close()
```

1.6.9 User defined Classes

- A class is a user-defined data type or blueprint or prototype from which objects are created. Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

Points to remember:

- Classes are created by keyword **class**.
- Attributes are the variables that belong to a class.

Class definition Syntax:

```
Class ClassName:
 # Statement-1
 .
 .
 .
 # Statement-N
```

- Attributes are always public and can be accessed using the dot (.) operator.

Syntax: `Class_Name.Class_Attribute`

Example:

```
Class Student:
    RollID=10
    Name="Vijay"
    def display(self):
        print(self.RollID,self.Name)
```

- Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.
- **Class Object:** An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. An object consists of:
  - **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
  - **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
  - **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Creating Instance of Class:

Syntax: `<object_name> = <class_name> (<arguments>)`

Example:

```
class Student:
    RollID=10
    Name="Vijay"
    def display(self):
        print("RollID=%d\n Name=%s"%(self.RollID,self.Name))
#Creating Stud1 instance of Student class
Stu1= Student()
Stu1.display()
```

Output:

```
RollID=10
Name=Vijay
```

1.6.10 Data Structures in Python

1.6.10.1 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.

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 comma.

  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, "Vijay", '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.

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 1: Creating an empty list.

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

Example 2: Creating a list with any integer elements.

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

Example 3: Creating a list with string elements.

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

Example 4: Creating a list with mixed data.

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

Example 5: Create a list using inbuilt `range()` function.

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

Example 6: Create a list with in-built characters A, B and C.

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

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. 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].

  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 m^th index to n-1 index
['two', 3]
>>> list1[3:]   # get element from m^th index to last index
[10, 'six', 20]
>>> list1[:4]   # get element from 0^th index to n-1 index
['one', 'two', 3, 10]
>>>
>>>
```

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, 20]
```

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

Updating Lists (Change or Add Elements to a List)

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

Example: For updating lists.

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

List methods used for updating list.

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().

   Syntax: list.append(item)

   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.

   Syntax: list1.extend(list2)

   Example: Program for extend() method.

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

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

Example: Program for insert() method.

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

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

   Example: Using + operator to combine with list.

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

5. * **operator to repeat list**: 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]
   ```

7. **List indexing:** 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.

Example: For list index in list.

```
>>> list1=[10,20,30,40,50]
>>> list1[0]
10
>>> list1 [3:]        # list[m:] will return elements indexed from mth
                      index to last index
[40, 50]
>>>list1 [:4]        # list[:n] will return elements indexed from
                      first index to n-1th index
[10, 20, 30, 40]
>>> list1 [1:3]       # list[m:n] will return elements indexed from m
                      to n-1.
[20, 30]
>>> list1[5]
Traceback (most recent call last):
File "<pyshell#71>", line 1, in <module>
list1[5]
IndexError: list index out of range
```

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.

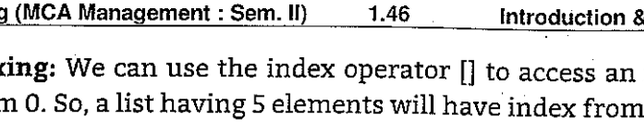


Fig. 1.13: Negative Indexing

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
```


8. **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]`

Example: 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']
```

9. **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.

Program 1:

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

Output:

```
10
20
30
40
```

Program 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
@ # $ &
```

Built-in Functions for List:

Table 1.14: Built-in Functions for List

Built-in functions	Description	Example

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

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

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

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

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

Table 1.15: Methods of List Class

Methods	Description	Example

`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]`

`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`

`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']`

contd. ...

`list.index(item)`	It returns the index number of the item. It item appears one 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`

`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]`

`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`

`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]`

`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]`

`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]`

1.6.10.2 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.

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, "Vijay", 'M', 40).
- A tuple in Python is an immutable data type which means a tuple once created cannot be altered/modified. Tuples can have any number of different data items (integer, float, string, list etc.).
- The simplest method to create a tuple in Python is simply assigning a set of values to the tuple using assignment operator (=). For example: t( ) # creates an empty tuple with name 't'.

Example: For creating tuples.

```
>>> tuple1=(10,20,30)                    # A tuple with integer values
>>> tuple1
(10, 20, 30)
>>> tuple2=(10,"abc",11.22,'X')          # A tuple with different data types
>>> tuple2
(10, 'abc', 11.22, 'X')
>>> tuple3=("python",[10,20,30],[11,"abc",22.33])     # Nested tuple
>>> tuple3
('python', [10, 20, 30], [11, 'abc', 22.33])
>>> tuple4=10,20,30,40,50                # Tuple can be created without
                                         parenthesis
>>> tuple4
(10, 20, 30, 40, 50)
>>> type(tuple4)
<class 'tuple'>
```

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'
```

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.
- 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.

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 m^th index to n-1^th
                    index.
(20, 30, 40)
>>> tuple[:2]       # tuple[:n] will return elements from 0^th index to n-1^th
                    index.
(10, 20)
>>> tuple[2:]       # tuple[m:] will return elements from m^th index to last
                    index.
(30, 40, 50)
>>> tuple[-1]       # access value at last index
50
```

Deleting Tuple:

- 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)
```

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)
>>> tuple1
(10, 20, 30)
>>> tuple2=('A', 'B', 'C')
>>> 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)
```

Tuple Operations:

- A set of operations can be applied to Python programming tuple. The operations described below are supported by most sequence types, both mutable and immutable. Here we will consider different operations in context of immutable sequence.

1. **Concatenation and Repetition:** We can use + operator to combine two tuples. This is also called concatenation operation. We can also repeat the elements in a tuple for a given number of times using the * operator.

   Example: For tuple operations using + and * operators.

   ```
   >>> t1= (10, 20)
   >>> t2= (30, 40)
   >>> t1+t2
   (10, 20, 30, 40)
   >>> t1*2
   (10, 20, 10, 20)
   ```

2. **Membership Function:** We can test if an item exists in a tuple or not, using the keyword in. The in operator evaluates to true if it finds a variable in the specified sequence and false otherwise.

   Example: For membership function in tuple.

   ```
   >>> tuple
   (10, 20, 30, 40, 50)
   >>> 30 in tuple
   True
   >>> 25 in tuple
   False
   ```

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
```

Table 1.16: Built-in Functions and Methods of Tuple

Function	Description	Example

`cmp(tuple1, tuple2)`	Compares elements of both tuples.	`>>> tup1=(1,2,3)`
		`>>> tup2=(1,2,3)`
		`>>> cmp(tup1,tup2)`
		`0`

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

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

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

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

`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')]`

`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`

1.6.10.3 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.

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 1: `Emp={20, "Vijay", 'M', 40}`

     Example 2: For creating sets with { }.

```
>>> a={1,3,5,4,2}
>>> print("a=",a)
a = {1, 2, 3, 4, 5}
>>> print(type(a))
<class 'set'>
>>> colors = {'red', 'green', 'blue', 'red'}
>>> colors
{'blue', 'red', 'green'}
>>>
```

  2. We can also create a set from a list by using the Python's built-in `set()` function.

     Example 3:

```
num_set = set([1, 2, 3, 4, 5, 6])
print(num_set)
```

     Output:

```
{1, 2, 3, 4, 5, 6}
```

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()
```

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}
>>> 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'}
```

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}
```

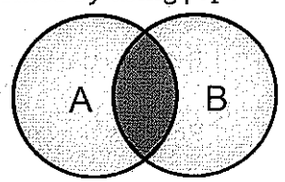


Fig. 1.14 (a): Union

2. **Intersection:** Intersection operation performed on two sets returns all the elements which are common or in both the sets. The intersection of sets A and B is defined as the set composed of all elements that belong to both A and B. It is performed by using & operator.

   Example: For intersection operation in sets.

```
>>> A={1,2,4,6,8}
>>> B={1,2,3,4,5}
>>> C=A & B
>>> C
{1, 2, 4}
```

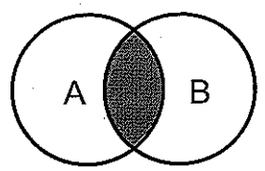


Fig. 1.14 (b): Intersection

3. **Difference:** Difference operation on two sets set1 and set2 returns all the elements which are present on set A but not in set B. 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}
```

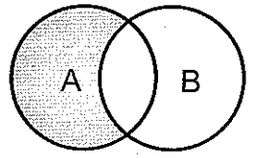


Fig. 1.14 (c): Difference operation

4. **Symmetric Difference:** Symmetric Difference operation performed by using ^ operation. The symmetric difference of two sets A and B is the set (A – B) ^ (B – A). It represents set of all elements which belongs either to set A or B but not both.

Built-in functions for 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}.

Table 1.17: Built-in functions for set

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

1.6.10.4 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":"Vijay", "Gender":Male, "SalaryPerHr":40}`

- 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.

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= {} # Empty dictionary
>>> dict1
{}
>>> dict2={1:"Orange", 2:"Mango", 3:"Banana"}  # Dictionary with
                                               integer keys

>>> dict2
{1: 'Orange', 2: 'Mango', 3: 'Banana'}
>>> dict3={"name":"Vijay", 1:[10,20]}          # Dictionary with mixed
                                               keys

>>> dict3
{'name': 'Vijay', 1: [10, 20]}
```

2. Python provides a built-in function `dict()` for creating a dictionary.

Example: Creating directory using dict().

```
>>> d1=dict({1:"Orange", 2:"Mango", 3:"Banana"})
>>> d2=dict([(1,"Red"), (2,"Yellow"), (3,"Green")])
>>> d3=dict(one=1, two=2, three=3)
>>> d1
{1: 'Orange', 2: 'Mango', 3: 'Banana'}
>>> d2
{1: 'Red', 2: 'Yellow', 3: 'Green'}
>>> d3
{'one': 1, 'two': 2, 'three': 3}
```

Accessing Values in a Dictionary:

- We can access the items of a dictionary by following ways:

1. Referring to its key name, inside square brackets ([]).

Example: For accessing dictionary items [ ] using.

```
>>> dict1={'name':'Vijay','age':40}
>>> dict1['name']
'Vijay'
>>> dict1['adr']
Traceback (most recent call last):
File "<pyshell#79>", line 1, in <module>
dict1['adr']
KeyError: 'adr'
>>>
```

Here, if we refer to a key that is not in the dictionary, you'll get an exception. This error can be avoided by using `get()` method.

2. Using `get()` method returns the value for key if key is in the dictionary, else None, so that this method never raises a KeyError.

Example: For accessing dictionary elements by get().

```
>>> dict1={'name':'Vijay','age':40}
>>> dict1.get('name')
'Vijay'
>>> dict1.get('adr')
```

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
```

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 is mutable means changeable. We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key:value pair is added to the dictionary.

Example: For updating dictionary items/elements.

```
>>> dict1
{'name': 'Vijay', 'age': 40}
>>> dict1['age']=35             # updating values in dictionary
>>> dict1
{'name': 'Vijay', 'age': 35}
>>> dict1['address']='thane'    # add new item in dictionary
>>> dict1
{'name': 'Vijay', 'age': 35, 'address': 'thane'}
```

Basic Operations on Directory:

- In previous sections we studied 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
```

Dictionary Methods:

- Python has a set of built-in methods that you can use on dictionaries.

Table 1.18: Built-in dictionary Methods

Method	Description	Example

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
		{}

contd. ...

copy()	Returns a copy of the dictionary.	car = {
		"brand": "Ford",
		"model": "Figo",
		"year": 1920
		}
		x = car.copy()
		print(x)
		Output:
		{'brand': 'Ford', 'model': 'Figo', 'year': 1920}

fromkeys()	Returns a dictionary with the specified keys and values.	#Create a dictionary from a sequence of keys with value
		keys = {'a', 'e', 'i', 'o', 'u'
		}
		value = 'vowel'
		vowels = dict.fromkeys(keys, value)
		print(vowels)
		Output:
		{'a': 'vowel', 'u': 'vowel', 'o': 'vowel', 'e': 'vowel', 'i': 'vowel'}

get()	Returns the value of the specified key.	car = {
		"brand": "Ford",
		"model": "Figo",
		"year": 2021
		}
		x = car.get("model")
		print(x)
		Output:
		Figo

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')
keys()	Returns a list containing the dictionary's keys	dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> dict.keys() dict_keys([1, 2, 3])

contd. ...

pop()	Removes the element with the specified key.	dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> print(dict.pop(2)) Amar
popitem()	Removes the last inserted key-value pair.	dict={1:'Vijay',2:'Amar', 3:'Santosh'} >>> dict.popitem() (3, 'Santosh')

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')
		'Vijay'
		>>> dict
		{2: 'Amar', 3: 'Santosh', 1: 'Vijay'}

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'}

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'])

all()	Returns 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

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

contd. ...

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

cmp()	Compares items of two dictionaries. (Not available in Python 3)	dict1 = {'Name': 'Arya', 'Age': 10}; dict2 = {'Name': 'Ajay', 'Age': 47}; dict3 = {'Name': 'Amar', 'Age': 40}; dict4 = {'Name': 'Arya', 'Age': 10}; print "Return Value : %d" % cmp (dict1, dict2) print "Return Value : %d" % cmp (dict2, dict3) print "Return Value : %d" % cmp (dict1, dict4) **Output:** Return Value : -1 Return Value : 1 Return Value : 0

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

1.7 UNDERSTANDING PYTHON 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. Python uses the character sets as the building block to form the basic program elements such as variables, identifiers, keywords, constants, etc.

1.7.1 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.

- **Lowercase English Letters:** a to z.
- **Uppercase English Letters:** A to Z.
- **Punctuation and Symbols:** "$", "!", etc.
- **Whitespace Characters:** An actual space (" "), as well as a newline, carriage return, horizontal tab, vertical tab, and a few others.
- **Non-Printable Characters:** Characters such as backspace, "\b" that cannot be printed literally in the way that the letter A can be printed.
- **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 identified 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.

1.7.2 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:
  - An identifier can be a combination of uppercase letters, lowercase letters, underscores, and digits (0-9). Examples: Name, myClass, Emp_Salary, var_1, _Address and print_hello_world.
  - We can use underscores to separate multiple words in the identifier. For example, Emp_Salary.
  - An identifier starts with a letter which can be alphabet (either lowercase or uppercase), underscore (_).
  - Identifiers can be of any length.
  - Identifiers cannot start with digit and must not contain any space or tabs. Example includes, 2variable, 10ID.
  - We cannot use Python keywords as identifiers.
  - Special characters such as %, @, and $ are not allowed within identifiers. Example includes, $Money, @salary.
  - 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.

1.7.3 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. Following table lists keywords in Python programming:

Table 1.19: 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	

1.7.4 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.
- 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.
- 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 variables 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 = "Vijay"
>>> z = x + y
>>> print(x); print(y); print(name); print(z)
10
5
Vijay
15
```


1.7.5 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
```

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.

Table 1.20: Numerical data 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 lowercase 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          # Here N is in uppercase
>>> val1
10
>>> val2
>>> print(val2)
None
```

5. **Literal Collections:** Collections such as tuples, lists and dictionary are used in Python.
   - (i) **List:** List contains items of different data types. Lists are mutable i.e., modifiable. The values stored in list are separated by commas(,) and enclosed within a square brackets ([]). We can store different type of data in a list. Value stored in a list can be retrieved using the slice operator([] and [:]). The plus sign (+) is the list concatenation and asterisk(*) is the repetition operator.
   - (ii) **Tuple:** Tuple is used to store the sequence of immutable python objects. A tuple can be created by using () brackets and separated by commas (,).
   - (iii) **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 (,).

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'>
```

1.8 PYTHON PROGRAM FLOW CONTROL

- The program control flow is the order in which the program 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 refers 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.
- Python has three types of control structures:
  - **Sequential:** default mode.
  - **Selection:** used for decisions and branching.
  - **Repetition:** used for looping, i.e., repeating a piece of code multiple times.

1. **Sequential Control flow statements:** These are a set of statements whose execution process happens in a sequence. This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

2. **Selection Statements:** In Python, the Selection Statements are also known as **Decision Control Flow Statements** or **Branching Statements.** The selection statement allows a program to test several conditions and execute instructions based on which condition is true. Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other.

   Some Decision Control Statements are:

   - Simple if
   - if-else
   - nested if
   - if-elif-else

4. **Loop Control Flow Statement:** It is used to repeat a group(block) of programming instructions. This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (*for* loop and *while* loop). Loop Control Flow Statements are also called Repetition statements or Iteration statements.

   In Python, we generally have two loops/repetitive statements:

   - for loop
   - while loop

1.8.1 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, as shown in Fig.1.15.

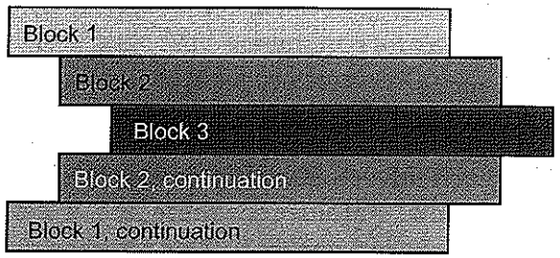


Fig. 1.15: Indentation in Python

- 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
```

1.9 CONDITIONAL BLOCKS USING if, else AND elif

- 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.

1.9.1 Simple if Statement

- The **if** statement executes a statement if a condition is true. A *simple if* only has one condition to check.

  Syntax:

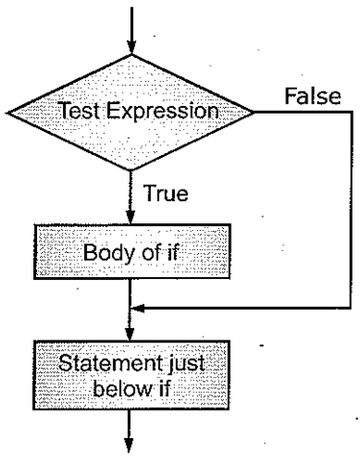
```
if condition:
    statement(s)
```

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<20):
   print("i is less than 20")
print("This statement is not in if")
```

Output:

```
i is less than 20
This statement is not in if
```

Fig. 1.16: Control Flow diagram and example of if-statement

Example: 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
```

1.9.2 if-else Statement

- The if-else statement evaluates the condition. If the test condition is `True`, we will execute the body of if, but if the condition is `False` then the body of else is executed. The if-else statement takes care of a true as well as false condition.

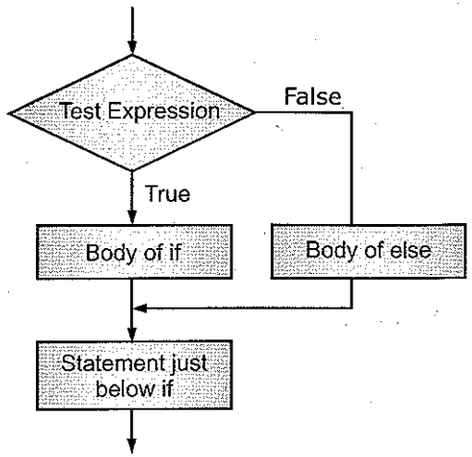
Syntax:

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

OR

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


Control Flow diagram of if-else statement:



Example:

```
i=20
if(i<10):
   print("i is less than 10")
else:
   print("i is greater than 10")
```

Output:

```
i is greater than 15
```

Fig. 1.17: Control Flow diagram and example of if-else statement

Example: 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
```

1.9.3 Nested if Statement

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

Syntax:

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

Control Flow diagram of Nested-if statement:

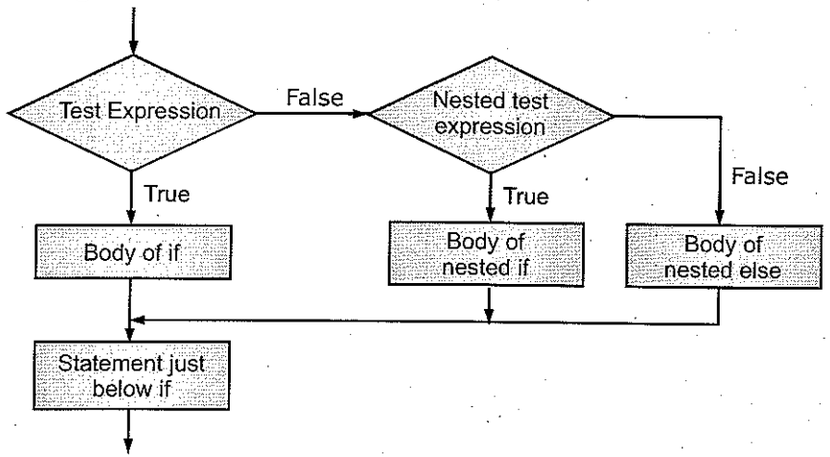


Fig. 1.18: Control Flow diagram and example of Nested-if statement

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
```

1.9.4 Multiway if-elif-else (Ladder) Statement

- 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
```


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
```

Control Flow diagram of if-elif-else statement

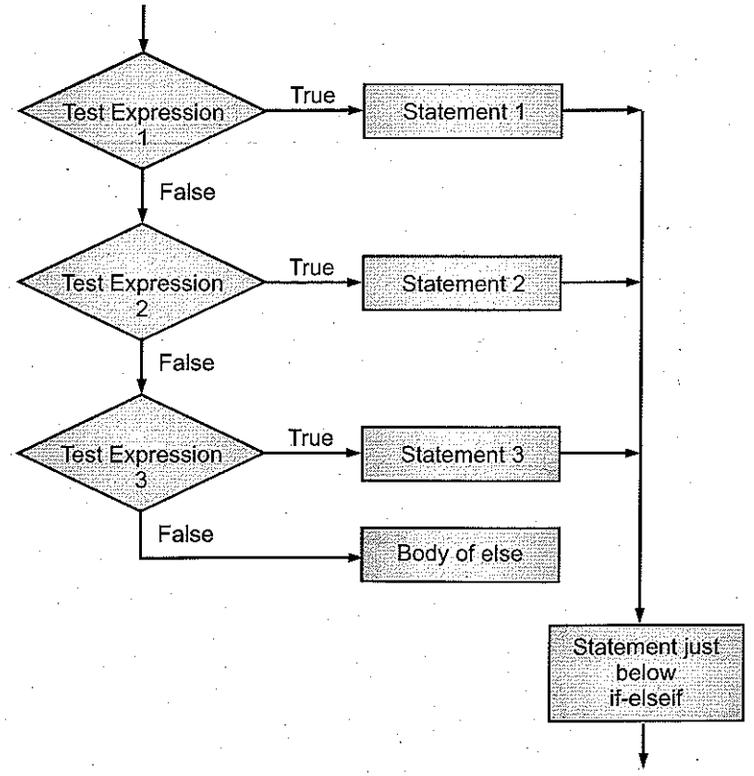


Fig. 1.19: Control Flow diagram and example of if-elif-else statement

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)
```

Output:

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

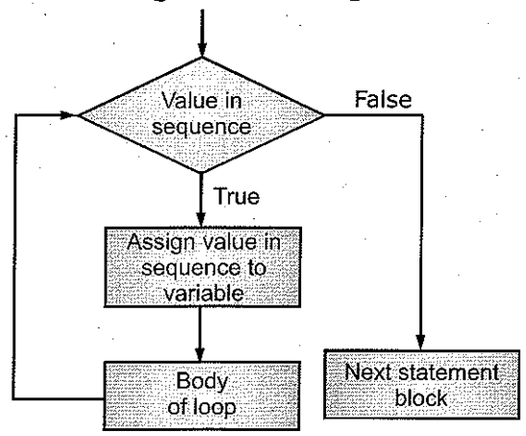
1.10 SIMPLE for LOOPS IN PYTHON

- 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 of for Loop:



Example:

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

Output:

```
10
20
30
40
50
```

Fig. 1.20: Control Flow diagram and Example of for Loop

1.11 for LOOP USING RANGES, STRING, LIST AND DICTIONARIES

- A *for* loop is used to iterate over a sequence that is either a list, tuple, dictionary or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.

1.11.1 for loop using range()

- 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(Start, 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]
```

Table 1.21: More examples of range() function

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

Example:

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

Output:

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

Example: 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
```

1.11.2 String Iteration and range()

- We can perform string manipulation using for `range()`.

Example:

```
string_name = "Python"
# Iterate over the string
for element in string_name:
 print(element, end=' ')
print("\n")

# Iterate over index
for element in range(0, len(string_name)):
 print(string_name[element])
```

Output:

```
P y t h o n

P
y
t
h
o
n
```


1.11.3 Loop using list

- List is equivalent to array in other language. The list is a type of container in data structure, which is used to store multiple data at the same time. We can use for loop which iterates numbers from list.

Example:

```
list = [2, 4, 6, 8, 10]
# variable to store the sum
sum = 0

# iterate over the list
for i in list:
 print (i)
 sum = sum+i

print("The sum is", sum)
```

Output:

```
2
4
6
8
10
The sum is 30
```

1.11.4 Loop through Dictionary

- Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair.

Example:

```
states_Capitals = {
 'Gujarat' : 'Gandhinagar',
 'Maharashtra' : 'Mumbai',
 'Rajasthan' : 'Jaipur',
 'Bihar' : 'Patna'
 }
print('List of given states:\n')
# Iterating over keys
for state in states_Capitals:
 print(state)
print('List of given capitals:\n')

# Iterating over values
for capital in states_Capitals.values():
 print(capital)
```

Output:

```
List of given states:
Gujarat
Maharashtra
Rajasthan
Bihar

List of given capitals:
Gandhinagar
Mumbai
Jaipur
Patna
```

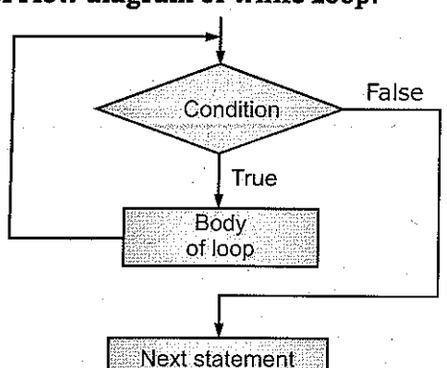
1.12 USE OF while LOOP IN PYTHON

- 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 of while Loop:



Example: for while loop

```
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
```

Fig. 1.21: Control Flow diagram and example of while statement

Example: To display Fibonacci Series.

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



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

Output:

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

1.13 LOOP MANIPULATION USING pass, continue, break AND else

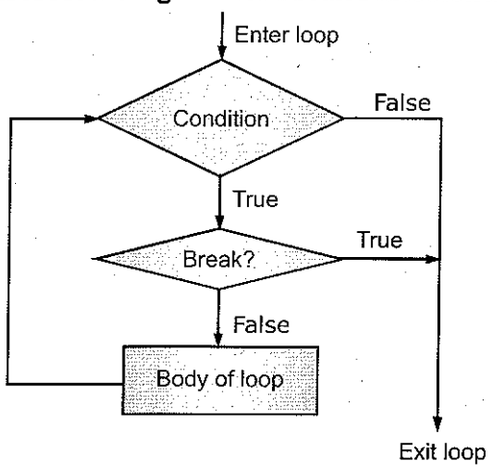
- In Python, loop statements give us a way execute the block of code repeatedly. But sometimes, we may want to exit a loop completely or skip specific part of loop when it meets a specified condition. It can be done using loop control mechanism.
- Loop control statements change execution from its normal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

1.13.1 break Statement

- The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break statement found in C.
- The break statement can be used in both while and for loops. If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

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
```

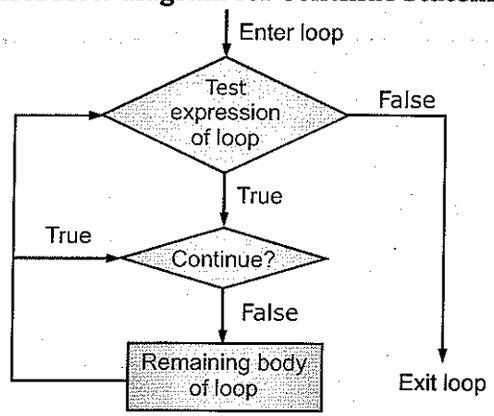
Fig. 1.22: Control Flow diagram and example of Break statement

1.13.2 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<=5:
 i=i+1
 if i==3:
 continue
 print("i= ",i)
```

Output:

```
i=1
i=2
i=4
i=5
```

Fig. 1.23: Control Flow diagram and example of Continue statement

1.13.3 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. This statement is also useful in places where your code will eventually go, but has not been written yet.

  Syntax: `pass`

Example:

```
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
```

1.14 PROGRAMMING USING PYTHON CONDITIONAL AND LOOPS BLOCK

Program 1: Program 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!
```

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

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

Output:

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

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

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

Output:

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

Program 4: Program to find the Factorial of a number.

```
number = int(input('Enter a number'))
factorial = 1
if number < 0:
        print("Factorial doesn't exist for negative numbers")
elif number == 0:
        print('The factorial of 0 is 1')
else:
        for i in range(1, number + 1):
                factorial = factorial * i
print(f"The factorial of number {number} is {factorial}")
```

Output:

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


Program 5: Program to find out the Reverse of the given number.

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

Output:

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

Program 6: Program to find sum of digit of a given number.

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

Output:

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

Program 7: 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
```

Program 8: Program to Find the GCD of Two Positive Numbers.

```
m = int(input("Enter first positive number "))
n = int(input("Enter second positive number "))
if m == 0 and n == 0:
      print("Invalid Input")
if m == 0:
      print(f"GCD is {n}")
if n == 0:
      print(f"GCD is {m}")
while m != n:
      if m > n:
            m = m - n
      if n > m:
            n = n - m
print(f"GCD of two numbers is {m}")
```

Output:

```
Enter first positive number 8
Enter second positive number 12
GCD of two numbers is 4
```

Program 9: Program to print multiplication table of the given number.

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

Output:

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


Program 10: 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-2:70
Enter marks of Subject-3:80
Total Marks= 210
Percentage= 70.0
First class
```

Program 11: 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 character for specific operation +,-,*,/: ")
result = 0
if ch == '+':
      result = num1 + num2
elif ch == '-':
      result = num1 - num2
elif ch == '*':
      result = num1 * num2
elif ch == '/':
      result = num1 / num    2
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 character for specific operation +,-,*,/: *
20 * 10 : 200
```

Program 12: Program to check whether a string is a Palindrome or not.

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

Output:

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

Program 13: Program to check whether a number is a Palindrome or not.

```
num = int(input("enter a number: "))
temp = num
rev = 0
while temp != 0:
      rev = (rev * 10) + (temp % 10)
      temp = temp // 10
```



```
if num == rev:
      print("number is palindrome")
else:
      print("number is not palindrome")
```

Output:

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

Program 14: Program to return Prime numbers from a list.

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

Output:

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

Program 15: 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
```

Summary

- Python is a high-level, interpreted scripting language developed in the late 1980s by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- A list is a collection which is ordered and changeable.
- A tuple is a collection which is ordered and unchangeable.
- Python dictionary is an unordered collection of 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.
- Set is an unordered collection of unique items.
- Indentation refers to the spaces at the beginning of a code line.
- Python decision making statements includes, if statements, if-else statements, nested-if statements, multi-way if-elif-else statements.
- While loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.
- For loops are used for sequential traversal.
- Continue Statement returns the control to the beginning of the loop.
- Break Statement brings control out of the loop.
- Pass statement are used to write empty loops.

Check Your Understanding

1. In which language Python is written?
   (a) English (b) PHP
   (c) C (d) All of the above
2. What do we use to define a block of code in Python language?
   (a) Key (b) Brackets
   (c) Indentation (d) None of these
3. Which of the following precedence order is correct in Python?
   (a) Parentheses, Exponential, Multiplication, Division, Addition, Subtraction
   (b) Multiplication, Division, Addition, Subtraction, Parentheses, Exponential
   (c) Division, Multiplication, Addition, Subtraction, Parentheses, Exponential
   (d) Exponential, Parentheses, Multiplication, Division, Addition, Subtraction
4. What is the output of this expression, 3*1**3?
   (a) 27 (b) 9
   (c) 3 (d) 1
5. What is the output of the following?

```
i = 2
while True:
 if i%3 == 0:
 break
 print(i)
 i += 2
```

   (a) 2 4 6 8 10 (b) 2 4
   (c) 2 3 (d) Error
6. What will be the output of 7^10 in Python?
   (a) 13 (b) 15
   (c) 2 (d) None
7. What will be the output of the following Python code?

```
>>>t=(1,2,4,3)
>>>t[1:3]
(1, 2)
(1, 2, 4)
(2, 4)
(2, 4, 3)
```

8. Which of the following statements is used to create an empty set?
   (a) {} (b) set()
   (c) [] (d) ()
9. What is the output of the following?

```
x = "abcdef"
i = "a"
while i in x:
 x = x[:-1]
 print(i, end = " ")
```

   (a) iiiiii (b) aaaaaa
   (c) aaaaa (d) None of the above
10. What is the maximum possible length of an identifier?
   (a) 32 characters (b) 63 characters
   (c) 79 characters (d) 31 characters

Answers

1. (c)	2. (c)	3. (a)	4. (c)	5. (b)	6. (a)	7. (c)	8. (b)	9. (b)	10. (d)

Practice Questions

Q.I Answer the following questions in short.

1. Enlist applications for Python programming.
2. What are the features of Python?
3. List any four editors used for Python programming.
4. 'Python programming language is interpreted and interactive' comment this sentence.
5. Python has developed as an open source project. Justify this statement.
6. How to run python scripts? Explain in detail.
7. Write the steps to install Python and to run Python code.
8. What is the difference between interactive mode and script mode of Python?
9. What is a lambda function?
10. What operators does python support?
11. What is the use of comparison operator?
12. Mention the features of identity operators?
13. Give the characteristics of membership operator?

Q.II Answer the following questions.

1. Explain the following features of Python programming: (i) Simple (ii) Platform independent (iii) Interactive (iv) Object Oriented.
2. Describe the internal working of Python with diagram.
3. What is variable? What are the rules and conventions for declaring variables?
4. What are the various data types available in Python programming?
5. Explain Dictionary in Python with example.
6. Define the following terms: (i) Identifier (ii) Literal (iii) Data type (iv) Tuple (v) List.
7. What is meant by control structure of a program? Explain with its types.
8. What is operator? Which operators used in Python?
9. Define the terms: (i) Loop, (ii) Program, (iii) Operator, (iv) Control flow.
10. What are the different loops available in Python?
11. Explain about different logical operators in Python with appropriate examples.
12. Explain about different relational operators in Python with examples.
13. Explain about Identity operators in Python with appropriate examples.
14. Explain about arithmetic operators in Python.
15. List different conditional statements in Python.
16. Explain if-else statement with an example.
17. Explain continue statement with an example.
18. Explain use of break statement in a loop with example.
19. Predict output and justify your answer: (i) –11%9 (ii) 7.7//7 (iii) (200-70)*10/5 (iv) 5*1**2.
20. List different operators in Python, in the order of their precedence.

Q.III Write a short note on:

1. Membership operators in Python
2. Applications of Python
3. The role of indentation in Python
4. Different nested loops available in Python
5. Role of Python in AI and Data science

Q.IV Write Programs.

1. Write Python Program to search a specific value from a given list of values using Linear search method.

2. Write Python Program to search a specific value from a given list of values using binary search method.
3. Write Python Program to Accept the base and height of a triangle and compute the area.
4. Write Python Program to compute the future value of a specified principal amount, rate of interest, and a number of years.
5. Write Python Program to create all possible permutations from a given collection of distinct numbers.

■■■

2... Python Functions, Modules & Packages

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.

2.1 FUNCTION BASICS - SCOPE, NESTED FUNCTION, NON-LOCAL STATEMENTS

- 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 referenced. Python gives us many built-in functions like print(),input(), compile(),etc. but we can also create our own functions called as user- defined functions.

2.1.1 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.

2.1.2 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(parameters):
        "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. 2.1.

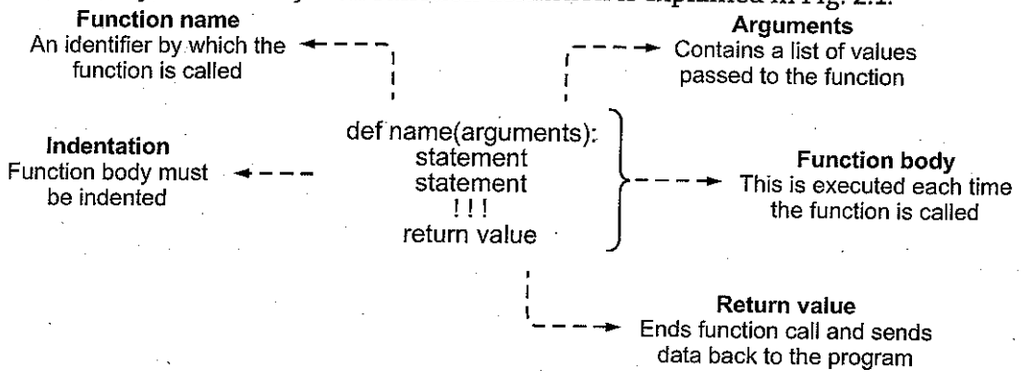


Fig. 2.1: Function Definition

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 reduce the overall size of the program.

2.1.3 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 name.

  Example: For calling a function.

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

2.1.4 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).
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)
>>> 343
```

2.1.5 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:

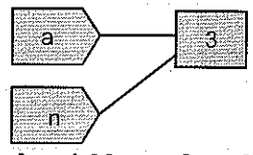


Fig. 2.2 (a): The local variable n referred to the same object

- 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:

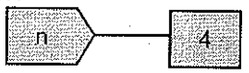


Fig. 2.2 (b): Create a new object with the value 4

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

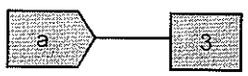


Fig. 2.2 (c)

- 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
```


- 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**.
- **Passing mutable objects:** 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.

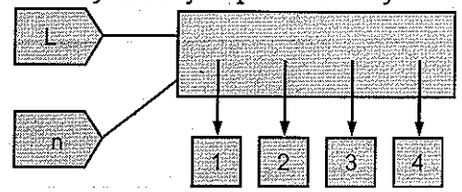


Fig. 2.2 (d): Mutable Objects

- Hence the output of the above program will be:

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

2.1.6 Nested Function

- A **Nested function** is a function defined within other function. Sometimes it is called as **inner function.**
- They are useful when performing complex task multiple times within another function to avoid loops or code duplication. A nested function can act as a closure.

  Example: Nested Function.

```
def outer(a, b):          # Outer Function
    def inner(c, d):      # Inner Function
        return c + d
    return inner(a, b)

result = outer(2, 4)
print(result)
```

Output:

```
6
```

2.1.7 Recursive Function

- Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body.
- A function is said to be a recursive if it calls itself. For example, let's say we have a function abc() and in the body of abc() there is a call to the abc().

Example: For recursive function.

```
def fact(n):
        if n == 0:
                return 1
        else:
                return n * fact(n-1)
print(fact(0))
print(fact(4))
print(fact(6))
```

Output:

```
1
24
720
```

- The factorial of 4 (denoted as 4!) is 1*2*3*4 = 24.
- Each function calls multiples 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           # returns from 4th call as number=1
4 * 3 * 2               # returns from 3rd call
4 * 6                   # returns from 2nd call
24                      # returns 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.

2.1.8 Scope of Variable

- All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular variable/identifier.
- The availability/accessibility of a variable in different parts of a program is referred to as its scope.
- There are two basic scopes of variables in Python:
  1. **Global Variables:** Global variables can be accessed throughout (outside) the program body by all functions.
  2. **Local Variables:** Local variables can be accessed only inside the function in which they are declared.

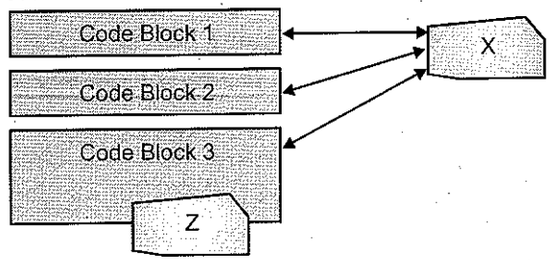


Fig. 2.3: Scopes of variables

- Fig.2.3 shows, global variable (x) can be reached and modified anywhere in the code, local variable (z) exists only in block 3.

Example: For scope of variables.

```
g=10  # global variable g
def show():
        l=20                    # local variable l
        print("local variable=",l)
        print("Global variable=",g)
show()
```

Output:

```
local variable= 20
Global variable= 10
```

Table 2.1: Difference between Local Variable and Global Variable

Local Variable	Global Variable
1. Local variables are declared inside a function.	1. Global variables are declared outside any function.
2. Accessed only by the statements, inside a function in which they are declared.	2. Accessed by any statement in the entire program.
3. Local variables are alive only for a function.	3. Global variables are alive till the end of the program.
4. A local variable is destroyed when the control of the program exit out of the block in which local variable is declared.	4. Global variable is destroyed when the entire program is terminated.

2.1.9 Nonlocal statements

- Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope. "nonlocal" keyword is used to create nonlocal variables.

Example:

```
def outerfunc():
        a = 10
        b = 20
    def innerfunc():
    # nonlocal binding
        nonlocal a
        a = 100          # a will update
        b = 200          # b will not update, because it will be considered
                         as a local variable
    # calling inner function
    innerfunc()
# printing the value of a and b
    print("a : ", a)
    print("b : ", b)

# main code calling the function i.e. outerfunc()
outerfunc()
```

Output:

```
a: 100
b: 20
```


- Here, a and b are variables of `outerfunc()` and in `innerfunc()`. We are binding variable 'a' as local, thus 'a' is not local here but 'b' will be considered as local for `innerfunc()`. Nonlocal keywords are used to create a nonlocal variable, the `innerfunc()` function is defined in the scope of another function `outerfunc()`.

2.2 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 come in category of mathematical functions and some are called type conversion functions and so on.

2.2.1 Data Type 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.

Table 2.2: Data type Conversion Functions

Function	Description	Example
`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`
`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`
`float(x)`	Converts x to a floating-point number.	`x=float('123.45')=123.45`
`complex(real[,imag])`	Creates a complex number.	`x=complex(1,2) = (1+2j)`
`str(x)`	Converts object x to a string representation.	`x=str(10) = '10'`
`repr(x)`	Converts object x to an expression string.	`x=repr(3) = 3`
`eval(str)`	Evaluates a string and returns an object.	`x=eval('1+2') = 3`
`tuple(s)`	Converts s to a tuple.	`x=tuple('123') = ('1', '2', '3')` `x=tuple([123]) = (123,)`
`list(s)`	Converts s to a list.	`x=list('123') = ['1', '2', '3']` `x=list(['12'] = ['12']`
`set(s)`	Converts s to a set.	`x=set('Python')` `= {'y', 't', 'o', 'P', 'n', 'h'}`

`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'}`

contd. ...

`chr(x)`	Converts an integer to a character.	`x=chr(65) = 'A'`
`unichr(x)`	Converts an integer to a Unicode character.	`x=unichr(65) =u'A'`
`ord(x)`	Converts a single character to its integer value.	`x=ord('A')= 65`
`hex(x)`	Converts an integer to a hexadecimal string.	`x=hex(12) = 0xc`
`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'
```

Table 2.3: Formats for Parameter Values

Format	Meaning	Example

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

>	Right aligns the result (within the available space)	`>>>x=10.23456`
		`>>> format(x,">10.2f")`
		`' 10.23'`

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

=	Places the sign to the leftmost position	`>>>x=-10.23456`
		`>>> format(x,"=10.2f")`
		`'- 10.23'`

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

contd. ...

-	Use a sign for negative values only	>>>x=-123 >>> format(x,"+") '-123'
' '	Use a leading space for positive numbers	

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

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

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

c	Converts the value into the corresponding unicode character	>>>x=10 >>> format(x,"c") '\n'
d	Decimal format	>>>x=10 >>> format(x,"d") '10'
e	Scientific format, with a lower case e	>>>x=10 >>> format(x,"e") '1.000000e+01'
E	Scientific format, with an upper case E	>>>x=10 >>> format(x,"E") '1.000000E+01'
f	Fix point number format	>>>x=10 >>> format(x,"f") '10.000000'
F	Fix point number format, uppercase	
g	General format	>>>x=10 >>> format(x,"g") '10'
G	General format (using a uppercase E for scientific notations)	

contd. ...

o	Octal format	>>>x=10 >>> format(x,"o") '12'
x	Hex format, lowercase	>>>x=10 >>> format(x,"x") 'a'
X	Hex format, uppercase	>>>x=10 >>> format(x,"X") 'A'
n	Number format	

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

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

2.2.2 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 math functions you need to import math module first.

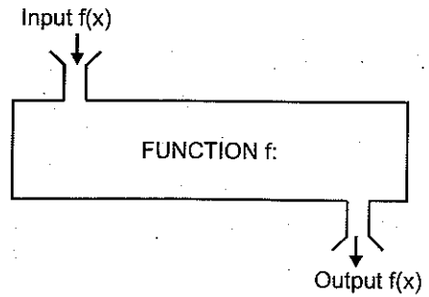


Fig. 2.4: Concept of Function

- **Built-in Functions (Mathematical):** These are the functions which don'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 built-in mathematical functions:

Table 2.4: Built-in Mathematical Functions

Functions	Description	Example
`min()`	Returns smallest value among supplied arguments.	`>>> min(20, 10, 30)` `10`
`max()`	Returns largest value among supplied arguments.	`>>> max(20, 10, 30)` `30`

`pow()`	The pow() function returns the value of x to the power of y (x^y). 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`

`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`

`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`

2.3 ARGUMENTS PASSING, ANONYMOUS FUNCTION: LAMBDA

2.3.1 Arguments Passing

- Many built-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:
  1. Required arguments.
  2. Keyword arguments.
  3. Default arguments.
  4. Variable-length arguments.

2.3.1.1 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
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").

2.3.1.2 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="Vijay",age=43)
Name: Vijay
Age: 43
>>> display(age=43,name="Vijay")
Name: Vijay
Age: 43
>>>
```

2.3.1.3 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=43):
        print("Name:",name)
        print("Age:",age)
        return
>>> display(name="Vijay")
Name: Vijay
Age: 43
>>> display(name="Vijay",age=43)
Name: Vijay
Age: 43
>>>
```

2.3.1.4 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 non-keyword 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
>>>
```

2.3.2 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
>>>
```

2.3.2.1 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
```

2.3.2.2 void Functions

- void functions are those functions which do not return any value. Programming with 'Python' 4.12 Python Functions, Modules and Packages.

Example: For void function.

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

2.3.3 Anonymous Function: Lambda

- Python Anonymous function is a function which has no name.
- Lambda functions are named after the Greek letter λ (lambda). These are also known as anonymous functions.
- These functions are called anonymous because they are not declared in the standard manner by using the def keyword.
- You can use the lambda keyword to create small anonymous functions.
- Lambda functions can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Syntax: `lambda [arg1 [,arg2,.....argn]]:expression`

Example:

`def sum ( a, b ):`
`  return a+b`
`print (sum(10, 20))`
`print (sum(20, 30))`
Output:
`30`
`50`
	We can convert this function to anonymous function
	→
		`# Function definition is here`
		`sum = lambda arg1, arg2: arg1 + arg2`
		`# Now you can call sum as a function`
		`print ("Sum : ", sum( 10, 20 ))`
		`print ("Sum : ", sum( 20, 30 ))`
		Output:
		`('Sum : ', 30)`
		`('Sum : ', 50)`

2.4 DECORATORS AND GENERATORS

2.4.1 Decorators

- A decorator is a function that accepts a function as input and returns a new function as output, allowing you to extend the behaviour of the function without explicitly modifying it.
- Generator function is like a normal function, creates its own iterator function.
- A generator is a special type of function which does not return a single value, instead it returns an iterator object with a sequence of values. In a generator function, a `yield` statement is used rather than a return statement.
- Decorators are a very powerful and useful tool in Python because it allows programmers to modify/control the behavior of function or class.
- It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.
- For example, you had a white car with basic wheel setup and a mechanic changes the color of your car to red and fits alloy wheels to it then the mechanic decorated your car, similarly a decorator in Python is used to decorate (or add functionality or feature) to your existing code.

Example:

```
#first function
def first(msg):
        print(msg)
# second function
def second(func, msg):
        func(msg)
# calling the second function with first as argument
second(first, "Hello!")
```

Output:

```
Hello!
```

- In the example, above the second function took the function first as an argument and used it, a function can also return a function. When there is a nested function (function inside a function) and the outer function returns the inner function it is known as **Closure** in Python.

Decorating functions with parameters:

Without decorator function	With decorator function

def divide(func):
  def inner(a, b):
  print("Dividing", a, "and", b)
  if b == 0:
  print("Whoops! cannot divide")
  return
  return func(a, b)
  return inner
def divide(a, b):
  print(a/b)
Output:
>>> divide(2,4)
0.5
>>> divide(2,0)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
  divide(2,0)
  File "C:\Users\Vijay Patil\AppData\Local\Programs\Python\Python39\test.py", line 12, in divide
  print(a/b)
ZeroDivisionError: division by zero
	def divide(func):
	  def inner(a, b):
	  print("Dividing", a, "and", b)
	  if b == 0:
	  print("Whoops! cannot divide")
	  return
	  return func(a, b)
	return inner
	@divide
	def divide(a, b):
	  print(a/b)
	Output:
	>>> divide(2,4)
	Dividing 2 and 4
	0.5
	>>> divide(2,0)
	Dividing 2 and 0
	Whoops! cannot divide

- We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. Python allows to use decorator in easy way with @ symbol. Sometimes it is called "pie" syntax.

2.4.2 Generators

- Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in for loop.
- A generator has parameter, which we can called and it generates a sequence of numbers. But unlike functions, which return a whole array, a generator yields one value at a time which requires less memory.
- Any Python function which contains a keyword "yield" may be called as generator.

- When you call a function, its local variables have their own scope. After the return statement is executed in a function, the local variables are destroyed and the value is returned to the caller. Later, a call to the same function creates a fresh set of local variables that have their own scope.
- A generator function does not include a return statement.

Example 1:

```
def simpleGenerator():
    yield 10
    yield 20
    yield 30
for value in simpleGenerator():
    print(value)
```

Output:

```
10
20
30
```

- Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a "for in" loop.

Example 2:

```
>>>def Generator_no(N):
    for i in range(N):
        yield i
>>> gen=Generator_no(3)
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
```

2.5 MODULE BASIC USAGE, NAMESPACES, RELOADING MODULES - MATH, RANDOM, DATETIME, etc.

- A module in Python programming allows us to logically organize the Python code. A module is a single source code file. The module in Python has the .py file extension. The name of the module will be the name of the file.
- A Python module can be defined as a Python program file which contains a Python code including Python functions, class or variables. In other words, we can say that our Python code file saved with the extension (.py) is treated as the module.

2.5.1 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 performs some task.
  2. Create a second file in the same directory called main file where we can import the module to the top of the file and call the function.
- Second file needs to be in the same directory so that Python knows where to find the module since it's not a built-in module.

Example: For creating a module. Type the following code and save it as **p1.py**.

```
def add(a, b):
      #This function adds two numbers and returns the result
      result = a + b
      return result
def sub(a, b):
      #This function subtracts two numbers and returns the result
      result = a - b
      return result
def mul(a, b):
      #This function multiplies two numbers and returns the result
      result = a * b
      return result
def div(a, b):
      #This function divides two numbers and returns 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
```


2.5.2 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
>>>
```

2.5.3 Importing Objects from Module

- Import statement in Python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. Python provides three different ways to import modules.

1. **From x import a:**

- Imports the module x, and creates references in the current namespace to all public objects defined by that module. If we run this statement, we can simply use a plain name to refer to things defined in module x. We can access attribute / methods 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
```


2.5.4 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 returns the result"
    result = a + b
    return result
def sub(a, b):
    "This function subtracts two numbers and returns the result"
    result = a - b
    return result
def mul(a, b):
    "This function multiplies two numbers and returns the result"
    result = a * b
    return result
def div(a, b):
    "This function divides two numbers and returns 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
```

2.5.5 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 is 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.

2.5.5.1 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.

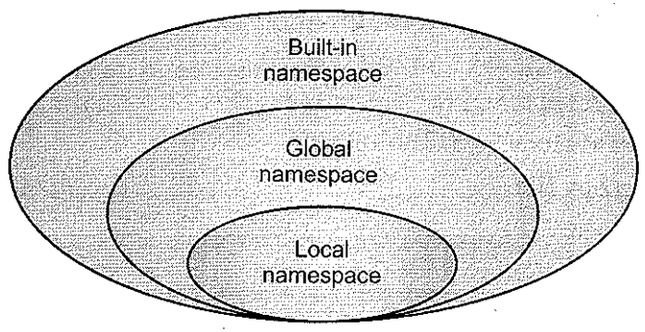


Fig. 2.5: Types of Namespaces

- 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:
  - Scope of the current function which has local names.
  - Scope of the module which has global names.
  - 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.
  - **Local Scope Variable:** All those variables which are assigned inside a function known as local scope Variable
  - **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
30
```

2.5.6 Reloading modules

- **reload()** reloads a previously imported module. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object.

  Syntax: `importlib.reload(module)`

- The argument passed in the reload() function should be the module that we are reloading. Note that this module must have been imported before in our program.

  Example 1: As `reload()` function for math module which has been imported before as you can see in the program. Python code runs fine.

```
>>> import math
>>> import importlib
>>> importlib.reload(math)
<module 'math' (built-in)>
```

Example 2: When we try to reload a module that has not been imported before, the `reload()` throws an error.

```
>>> import importlib
>>> importlib.reload(math)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    importlib.reload(math)
NameError: name 'math' is not defined
```

2.5.7 Built-in Functions (Math Module)

- The second type of functions requires 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.

Table 2.5: Built-in Mathematical functions of Math Module

Functions	Description	Example
`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`
`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. ...

`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`

`cosh()`	Returns the hyperbolic cosine of x.	`>>>print(math.cosh(3))` `10.067661995777765`
`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`
`exp()`	The method exp() returns exponential of x.	`>>> math.exp(1)` `2.718281828459045`

`fabs()`	This function will return an absolute or positive value.	`>>> math.fabs(10)`
		`10.0`
		`>>> math.fabs(-20)`
		`20.0`

`factorial()`	Returns the factorial of x.	`>>> math.factorial(5)` `120`

`fmod()`	This function returns x % y.	`>>> math.fmod(50,10)`
		`0.0`
		`>>> math.fmod(50,20)`
		`10.0`

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

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

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

2.5.8 Random Module

- Python defines a set of functions that are used to generate or manipulate random numbers through the **random module.** List of all the functions defined in random module are:

1. **Choice():** Returns a random item from a list, tuple, or string.

   Syntax: `random.choice(sequence)`

   A sequence like a list, a tuple, a range of numbers etc.

   Example:

```
import random
l1=[10,20,30,40,50]
print(random.choice(l1))
string="Python"
print(random.choice(string))
```

   Output:

```
20
t
```

2. **randrange() :** Returns a random number between the given range.

   Syntax: `randrange(start, stop, step)`

   Where,

   Start: Optional. An integer specifying at which position to start. Default 0.

   Stop: Required. An integer specifying at which position to end.

   Step: Optional. An integer specifying the incrementation. Default 1.

   Example:

```
import random

#choice() is used to generate a random number from a given list
print("A random number from list is : ", end="")
print(random.choice([10, 20, 30, 40, 50]))

# randrange() to generate in range from 20 to 50.
#The last parameter 10 is step size to skip ten numbers when
selecting.
print("A random number from range is : ", end="")
print(random.randrange(100, 200, 10))
```

   Output:

```
A random number from list is : 40
A random number from range is : 190
```

3. **shuffle():** It is used to shuffle a sequence(list). Shuffling means changing the position of the elements of the sequence.

   Syntax: `random.shuffle(sequence)`

   Example:

```
import random
list1 = [10, 20, 30, 40, 50]
print("Shuffling number list : ")
random.shuffle(list1)
print(list1)

list2 = ['A', 'B', 'C', 'D', 'E']
random.shuffle(list2)
print("\Shuffling character list: ")
print(list2)
```

   Output:

```
Shuffling number list :
[20, 30, 10, 40, 50]
Shuffling character list:
['A', 'E', 'B', 'C', 'D']
```

4. **random():** Returns random numbers between 0.0 and 1.0.

   Syntax: `random.random()`

   Example:

```
import random
print(random.random())
```

   Output:

```
0.5894451486388866
```

5. **sample():** Return a list that contains any 3 of the items from a list.

   Syntax: `random.sample(sequence, k)`

   A sequence can be any sequence: list, set, range etc. k is the size of the returned list.

   Example:

```
import random
mylist = ["Apple", "Banana", "Cherry","Mango", "Pineapple"]
print(random.sample(mylist, k=3))
```

   Output:

```
['Mango', 'Pineapple', 'Banana']
```

6. **uniform():** The `uniform()` method returns a random floating number between the two specified numbers (both included).

   Syntax: `random.uniform(a, b)`

   Example:

```
import random
print(random.uniform(20,50))
```

   Output:

```
36.30287864044642
```

2.5.9 Datetime Module

- Python datetime module deals with date, times and time intervals. Date and datetime in Python are the objects, so when you manipulate them, you are actually manipulating objects and not string or timestamps. Whenever you manipulate dates or time, you need to import datetime function.

  Example: To get current date and time

```
>>> import datetime                #imported datetime module
>>> ob=datetime.datetime.now()
>>> print(ob)
2021-02-06 15:26:37.237265
>>>
```

- One of the classes defined in the datetime module is datetime class. We then used now() method to create a datetime object containing the current local date and time.

  Example: To display current date

```
>>> import datetime
>>> ob=datetime.date.today()
>>> print(ob)
2021-02-06
```

Commonly used classes in the datetime module are:

- **date Class:** Manipulate just date (Month, day, year)
- **time Class:** Time independent of the day (Hour, minute, second, microsecond)
- **datetime Class:** Combination of time and date (Month, day, year, hour, second, microsecond)
- **timedelta Class:** A duration of time used for manipulating dates

1. **Date Class:** When an object of this class is instantiated, it represents a date in the format

   YYYY-MM-DD.

   Syntax: `class datetime.date(year, month, day)`

   Example:

```
from datetime import date
today=date.today()
print("Current date=",today)
print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

   Output:

```
Current date= 2021-02-06
Current year: 2021
Current month: 2
Current day: 6
```


2. **Time Class:** Time object represents local time, independent of any day.

   Syntax: `class datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)`

   Example:

```
from datetime import time

# time(hour = 0, minute = 0, second = 0)
a = time()
print("a =", a)

# time(hour, minute and second)
b = time(15, 30, 56)
print("b =", b)
print("Hour=:", b.hour)
print("Minute=:", b.minute)
print("Second=:", b.second)
print("Microsecond=:", b.microsecond)

# time(hour, minute and second)
c = time(hour = 15, minute = 30, second = 56)
print("c =", c)

# time(hour, minute, second, microsecond)
d = time(15, 30, 56, 234566)
print("d =", d)
```

   Output:

```
a = 00:00:00
b = 15:30:56
Hour=: 15
Minute=: 30
Second=: 56
Microsecond=: 0
c = 15:30:56
d = 15:30:56.234566
```

3. **Datetime class:** Information on both date and time is contained in this class.

   Syntax: `class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)`

Example:

```
from datetime import datetime
today=datetime.now()
print("Current date and time is",today)

#datetime(year, month, day)
a = datetime(2021, 2, 6)
print(a)

# datetime(year, month, day, hour, minute, second, microsecond)
b = datetime(2021, 2, 6, 15, 30, 56, 342380)
print(b)

print("year =", b.year)
print("month =", b.month)
print("hour =", b.hour)
print("minute =", b.minute)
```

Output:

```
Current date and time is 2021-02-06 15:55:59.444537
2021-02-06 00:00:00
2021-02-06 15:30:56.342380
year = 2021
month = 2
hour = 15
minute = 30
```

4. **Timedelta Class:** A timedelta object represents the difference between two dates or times.

   Syntax: `class datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)`

   `Returns: Date`

   Example:

```
from datetime import datetime, date
#Difference between two dates and times
t1 = date(year = 2021, month = 2, day = 6)
t2 = date(year = 2020, month = 12, day = 10)
t3 = t1 - t2
print("t3 =", t3)

t4 = datetime(year = 2021, month = 2, day = 6, hour = 7, minute = 9,
     second = 33)
t5 = datetime(year = 2020, month = 12, day = 10, hour = 5, minute =
     55, second = 13)
```



```
t6 = t4 - t5
print("t6 =", t6)

print("type of t3 =", type(t3))
print("type of t6 =", type(t6))
```

Output:

```
t3 = 58 days, 0:00:00
t6 = 58 days, 1:14:20
type of t3 = <class 'datetime.timedelta'>
type of t6 = <class 'datetime.timedelta'>
```

2.6 PACKAGE: IMPORT BASICS

- 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 organizes them by some mean which can be achieved by packages.
- A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages and so on.
- Packages allow for a hierarchical structuring of the module namespace using dot notation. Packages are a way of structuring many packages and modules which help in a well-organized hierarchy of data set, making the directories and modules easy to access.
- A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional __init__.py file (For example: Phone/__init__.py).

2.6.1 Writing Python Packages

- Creating a package is quite easy, since it makes use of the operating system's inherent hierarchical file structure as shown in Fig. Here, there is a directory named **mypkg** that contains two modules, p1.py and p2.py. The contents of the modules are:

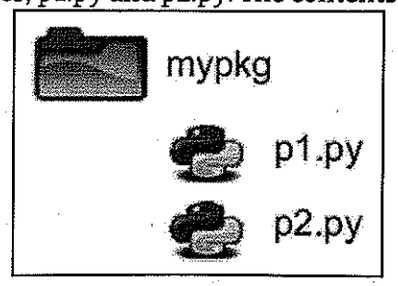


Fig. 2.6: Concept of Python Package

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
>>>
```

2.7 PYTHON NAMESPACE PACKAGES

- Namespace packages allow you to split the sub-packages and modules within a single package across multiple, separate distribution packages.
- Python scans all entries in sys.path. If matching directory with `__init__.py` is found, a normal package is loaded. If foo.py is found then it is loaded. Otherwise all matching directories in sys.path are considered part of the namespace package.

Example:

```
mynamespace/
    __init__.py
    subpackage_a/
        __init__.py
        ...
    subpackage_b/
        __init__.py
        ...
    module_b.py
setup.py
```

- We can use this package in our code as:

```
from mynamespace import subpackage_a
from mynamespace import subpackage_b
```

- We can break these sub-packages into two separate distributions. Each sub-package can now be separately installed, used, and versioned.

```
mynamespace-subpackage-a/
setup.py
mynamespace/
    subpackage_a/
        __init__.py

mynamespace-subpackage-b/
 setup.py
 mynamespace/
    subpackage_b/
        __init__.py
    module_b.py
```

2.8 USER DEFINED MODULES AND PACKAGES

2.8.1 User-defined Modules

- Any text file with the .py extension containing Python code is basically a module. Different Python objects such as functions, classes, variables, constants, etc., defined in one module can be made available to an interpreter session or another Python script by using the import statement.
- Functions defined in built-in modules need to be imported before use. On similar lines, a custom module may have one or more user-defined Python objects in it. These objects can be imported in the interpreter session or another script.

Creating user-defined module (test.py)

```
def add(x,y):
    return x + y
```

Importing a module:

```
>>>import calc
>>>calc.add(10,20)
30
```

2.8.2 User-defined Packages

- We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in Python takes the concept of the modular approach to next logical level.
- As we know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.
- Let's create a package named **MyPkg**, using the following steps:

Step 1: Create a folder MyPkg on "C:\Users\Vijay\AppData\Local\Programs\Python\Python37\".

Create modules **Message.py** and **Mathematics.py** with following code:

Message.py

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

Mathematics.py

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

Step 2: Create an empty __init__.py file in the MyPkg folder. The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

1. The Python interpreter recognizes a folder as the package if it contains __init__.py file.
2. __init__.py exposes specified resources from its modules to be imported.

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

Step 3: Create P1.py file in MyPkg folder and write following code:

```
from MyPkg import Mathematics
from MyPkg import Message
greet.SayHello("Vijay")
x=functions.power(3,2)
print("power(3,2) : ", x)
```

Output:

```
Hello Vijay
power(3,2) : 9
```

Using __init__.py File:

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

```
__init__.py
from .Mathematics import average, power
from .Message import SayHello
```

- The specified functions can now be imported in the interpreter session or another executable script.

Create test.py in the MyPkg folder and write following code:

test.py

```
from MyPkg import power, average, SayHello
SayHello()
x=power(3,2)
print("power(3,2) : ", x)
```

- Note that functions `power()` and `SayHello()` are imported from the package and not from their respective modules, as done earlier.
- The output of above script is:

```
Hello world
power(3,2) : 9
```

Summary

- A function can be defined as the organized block of reusable code, which can be called whenever required.
- In Python, a function is defined using the *def* keyword.
- The return statement is used at the end of the function and returns the result of the function.
- Types of arguments which can be passed at the time of function call: Required arguments, Keyword arguments, Default arguments, Variable-length arguments.
- Recursion means that a function calls itself.

- A lambda function can take any number of arguments, but can only have one expression.
- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.
- Generators in Python are special routine that can be used to control the iteration behaviour of a loop.
- Modules refer to a file containing Python statements and definitions.
- A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional __init__.py file.

Check Your Understanding

1. Which of the following is not a valid namespace?
   (a) Global namespace (b) Public namespace
   (c) Built-in namespace (d) Local namespace
2. What will be the output of the following Python code?
```
from math import factorial
print(math.factorial(5))
```
   (a) 120
   (b) Nothing is printed
   (c) Error, method factorial doesn't exist in math module
   (d) Error, the statement should be: `print(factorial(5))`
3. Which of the following items are present in the function header?
   (a) function name (b) function name and parameter list
   (c) Parameter list (d) return value
4. What is the output of the following code snippet?
```
def func(message, num = 1):
    print(message * num)
func('Welcome')
func('Viewers', 3)
```
   (a) Welcome - Viewers
   (b) Welcome - ViewersViewersViewers
   (c) Welcome - Viewers,Viewers,Viewers
   (d) Welcome
5. Which of the following function headers is correct?
   (a) def f(a = 1, b): (b) def f(a = 1, b, c = 2):
   (c) def f(a = 1, b = 1, c = 2): (d) def f(a = 1, b = 1, c = 2, d):
6. A package is a folder containing one or more Python modules. One of the modules in a package must be called _____.
   (a) main.py (b) init.py
   (c) __main__.py (d) __init__.py

7. What is the method inside the class in Python language?
   (a) Object (b) Function
   (c) Attribute (d) Argument
8. What is the output of the following program? z = lambda x : x * x print(z(6))
   (a) 6 (b) 36
   (c) 0 (d) Error
9. Which operator is used in Python to import modules from packages?
   (a) . (b) *
   (c) -> (d) &
10. How many keyword arguments can be passed to a function in a single function call?
    (a) Zero (b) One
    (c) Zero or more (d) One or more

Answers

1. (b)	2. (d)	3. (b)	4. (b)	5. (c)	6. (d)	7. (b)	8. (b)	9. (a)	10. (c)

Practice Questions

Q.I Answer the following questions in short.

1. Define function. Write syntax to define function. Give example of function definition.
2. Can a Python function return multiple values? If yes, how it works?
3. How function is defined and called in Python?
4. Define is module? What are the advantages of using module?
5. Mention the types of arguments in Python.
6. List some built-in modules in Python.
7. Define recursive function?

Q.II Answer the following questions.

1. Explain about void functions with suitable examples.
2. What is actual and formal parameter? Explain the difference along with example.
3. Discuss the difference between local and global variable.
4. Explain math module with its any five functions.
5. Write a function that takes single character and prints 'character is vowel' if it is vowel, 'character is not vowel' otherwise.
6. How to create a module and use it in a Python program? Explain with an example.
7. Explain the concept of namespaces with an example.
8. Write about the concept of scope of a variable in a function.
9. Write about different types of arguments in a function.

10. What type of parameter passing is used in Python? Justify your answer with sample programs.
11. Explain about the import statement in modules.
12. What are packages? Give an example of package creation in Python.
13. What is module? How many ways to import module in Python?

Q.III Write a short note on:

1. Recursive function
2. User-defined Modules
3. Fruitful functions
4. Anonymous functions
5. Packages in Python

Q.IV Write Programs.

1. Write a Python program to calculate factorial of a number using recursive function.
2. Write a python program to calculate factorial of given number using recursive function.
3. Write a python program to find reverse of a given number using user defined function.
4. Write a Python program that interchanges the first and last characters of a given string.
5. Write a Python code to check the given number is prime or not using modules.
6. Write a function that takes single character and prints "character is vowel". If it is vowel, "character is not vowel" otherwise.
7. Write a Python program to check whether the given no is Armstrong or not using user defined function.
8. Write a Python program to generate the Fibonacci series using recursion.

■■■

3... Python Object Oriented Programming

Objectives...

- To learn Object Oriented Programming Concepts in Python programming.
- To know how to create Classes and Objects in Python.
- To learn Methods, Method Overloading, Inheritance etc.
- To study about Delegation and Container.

3.1 INTRODUCTION

- Python is an Object Oriented Programming Language (OOPL) that 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 make 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.

3.2 CONCEPT OF CLASS, OBJECT AND INSTANCES, METHOD CALL

- 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 behaviour (variables and methods) that are shared by all its objects.

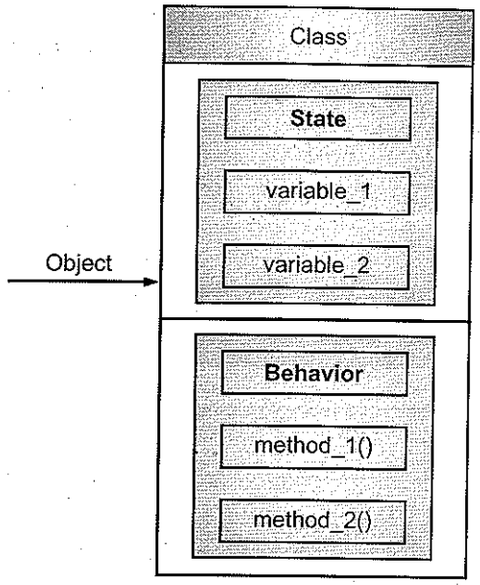


Fig. 3.1: Generic Object Diagram

3.2.1 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 as 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 at least one argument that is called **self-Parameter**. The self-parameter is a reference to the class itself and is used to access variables that belong 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.
- 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.

3.2.2 Objects and Creating Objects

- An **object** is an instance of a class that has some attributes and behaviour. Objects can be used to access the attributes of the class. The act of creating an object from a class is called **Instantiation.**

  Syntax: `obj_name=class_name()`

  Example 1:

```
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 2: 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
```

3.2.3 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.

Syntax: To access data attributes.

```
Object_name.Data_name
```

Syntax: To assign value to data attribute.

```
Object_name.Data_name=value
```

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
```


3.2.4 Method Call

- A **method** is called by its name but it is associated with an object (dependent). It is implicitly passed to an object on which it is invoked. It may or may not return any data.
- A method can operate the data (instance variables) that is contained by the corresponding class.

Syntax:

```
Class class_name:
    def method_name():
    #method body
```

Example:

```
>>> class myclass:
    def display():
        print("Hello Python")
>>> myclass.display()
```

Output:

```
Hello Python
```

Creating Instance Method:

- An instance method is a part of the individual instances. You use instance methods to manipulate the data that the class manages. As a consequence, you can't use instance methods until you instantiate an object from the class.

Syntax: To call method

```
object_name.method_name()
```

Example:

```
>>> class myclass:
        def display(self):
            print("Hello Python")
>>> ob1 = myclass()
>>> ob1.display()
```

Output:

```
Hello Python
```

3.3 CONSTRUCTOR, CLASS ATTRIBUTES AND DESTRUCTORS

3.3.1 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 confirming 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
```


Example 3: Create a Circle class and initialize 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
```

3.3.2 Types of Constructors

1. Default Constructor:

- The default constructor is simple constructor which does not accept any arguments. Its 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 parameterized constructor")
      def show(self,name):
            print("Hello",name)
s1 = Student()
s1.show("Vijay")
```

Output:

```
This is non parameterized constructor
Hello Vijay
```

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 takes 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 parameterized constructor")
                self.name = name
        def show(self):
                print("Hello",self.name)
s1 = Student("Vijay")
s1.show()
```

Output:

```
This is parameterized constructor
Hello Vijay
```

3.3.3 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.
```

3.3.4 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__)
```

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>}
```

3.4 REAL TIME USE OF CLASS IN LIVE PROJECTS

- In Real-time projects, we can use classes and object oriented concepts. So many live projects you can develop in Python, some of the list of real-time projects are:
  1. **Virtual Assistant:** A virtual assistant is an application that can understand voice commands and complete tasks for a user. Google's assistant and Amazon's Alexa are good examples of virtual assistants.
  2. **Reconnaissance Scanner:** Reconnaissance Scanner will scan any website that is available over the internet and will provide you with results in a file.
  3. **Rock Paper Scissors Game Clone:** Creating games is probably the best way to learn coding, logic, and any new programming language like Python.
  4. **URL Shortening Service like bit.ly:** If you want to do web development, creating websites and web applications then Python is a great choice. It has a lot of great frameworks like Python and Flask which makes it easy to create web applications.

5. **A Real-Time Price Alert App:** Real-time price alert app that will notify you when cryptocurrencies hit certain prices in USD (US Dollar).

3.5 INHERITANCE, SUPER CLASS AND OVERLOADING OPERATORS

- 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.

3.5.1 Inheritance

- In inheritance, objects of one class procure the properties of objects of another class. Inheritance provides 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
```

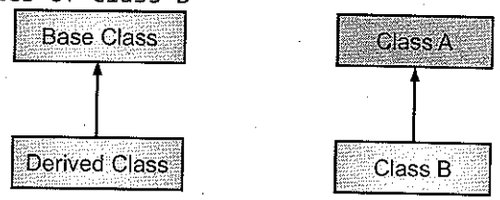


Fig. 3.2: Concept of Inheritance (Single Inheritance)

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()
```

Output:

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

3.5.2 Multilevel Inheritance

- Multilevel inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multilevel inheritance is archived in Python.
- In multilevel inheritance, 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.

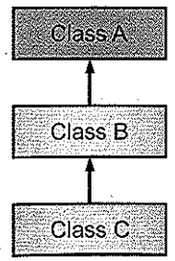


Fig. 3.3: Multilevel Inheritance

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
```

3.5.3 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 this 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.

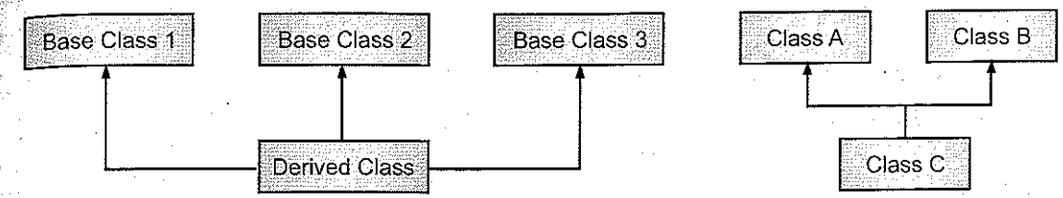


Fig. 3.4: Multiple Inheritance

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 class A and B
# add more properties to class C
```

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
```

3.5.4 Hierarchical Inheritance

- When more than one derived classes are created from a single base then it is called Hierarchical Inheritance.

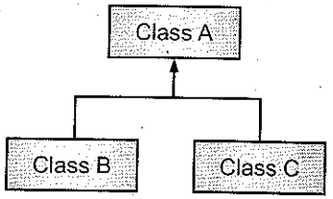


Fig. 3.5: Hierarchical Inheritance

- In following program, we have a parent (base) class name: 'Email' and two child (derived) classes named 'Gmail' and 'Yahoo'.

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) )
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
```

3.5.5 Super Class

- In single inheritance, the built-in `super()` function can be used to refer to base classes without naming them explicitly, thus making the code more maintainable. If you need to access the data attributes from the base class in addition to the data attributes being specified in the derived class's `__init__()` method, then you must explicitly call the base class `__init__()` method using `super()` yourself, since that will not happen automatically. However, if you do not need any data attributes from the base class, then no need to use `super()` function to invoke base class `__init__()` method.
- In Python, `super()` has two major use cases:
  1. Allows us to avoid using the base class name explicitly.
  2. Working with Multiple Inheritance.
- The `super()` function is used to give access to methods and properties of a parent or sibling class.
- This function returns an object that represents the parent class.

Syntax:

```
super()
super().__init__(base_class_parameter(s))
```


- Its usage is shown below:

```
class DerivedClassName(BaseClassName):
def __init__(self, derived_class_parameter(s), base_class_parameter(s))
super().__init__(base_class_parameter(s))
self.derived_class_instance_variable = derived_class_parameter
```

Example:

```
class Parent(object):
  def __init__(self, message):
    print(message, 'Parent  Class')

class Child(Parent):
  def __init__(self):
    print('Child Class')
    super().__init__('Hello')

obj1 = Child()
```

Output:

```
Child Class
Hello Parent  Class
```

3.5.6 Overloading Operators

- In object-oriented programming, there exists this concept called "Polymorphism". Polymorphism means "one action, many forms". OOP allows objects to perform a single action in different ways. One way of implementing Polymorphism is through operator overloading.
- In Python, same operators are uses/behaves differently with different types. For example, operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because the '+' operator is overloaded by *int* class and *str* class. This feature in python allows the same operator to have different meanings according to the context is called **Operator Overloading**.

Example:

```
# + operator is used to add two numbers
print(10+20)
# concatenate two strings
print("Hello " + " Python")
#Adding two objects
class A:
    def __init__(self,a):
        self.a=a

#Adding two objects
    def __add__(self,o):
        return self.a+o.a
```

```
ob1=A(11)
ob2=A(22)
ob3=A("Welcome ")
ob4=A(" Python")
print(ob1+ob2)
print(ob3+ob4)
```

Output:

```
30
Hello Python
33
Welcome Python
```

- Some special functions used for overloading the operators are shown below:

Table 3.1: Special Functions for overloading the operators

Name	Symbol	Special Function
Mathematical Operators		
Addition	+	`__add__(self, other)`
Subtraction	-	`__sub__(self, other)`
Division	/	`__truediv__(self, other)`
Floor Division	//	`__floordiv__(self, other)`
Modulus (or Remainder)	%	`__mod__(self, other)`
Power	**	`__pow__(self, other)`
Assignment Operators		
Increment	+=	`__iadd__(self, other)`
Decrement	-=	`__isub__(self, other)`
Product	*=	`__imul__(self, other)`
Division	/=	`__idiv__(self, other)`
Modulus	%=	`__imod__(self, other)`
Power	**=	`__ipow__(self, other)`
Relational Operator		
Less than	<	`__lt__(self, other)`
Greater than	>	`__gt__(self, other)`
Equal to	=	`__eq__(self, other)`
Not equal	!=	`__ne__(self, other)`
Less than equal to	<=	`__le__(self, other)`
Greater than equal to	>=	`__gt__(self, other)`

contd. ...

Binary Operators		
Right shift	>>	`__rshift__(self, other)`
Left Shift	<<	`__lshift__(self, other)`
AND	&	`__and__(self, other)`
OR	|	`__or__(self, other)`
XOR	^	`__xor__(self, other)`

3.6 STATIC AND CLASS METHODS

3.6.1 Static Method

- A static method is also a method which is bound to the class and not the object of the class.
- A static method can't access or modify class state.
- It is present in a class because it makes sense for the method to be present in class.
- A static method does not receive an implicit first argument.
- For static method, `@staticmethod` decorator is used.
- In Python, there are two ways of defining a static method:
  1. Using the `staticmethod()`
  2. Using the `@staticmethod`

1. Using `Staticmethod()`:

- The `staticmethod()` built-in function returns a static method for a given function.

Syntax: `staticmethod(function)`

Example: Static method using staticmethod()

```
class Maths:
    def add(x, y):
        return x + y
# create add method
Maths.add = staticmethod(Maths.add)
print('The sum is:', Maths.add(10, 20))
```

Output:

```
The sum is: 30
```

2. Using the @staticmethod:

- Using `@staticmethod` is a more modern approach of defining static method, in which Define a class with a method. Add the keyword `@staticmethod` above it to make it static.

Syntax:

```
@staticmethod
def func(args, ...):
#function body
Return value
```

Example:

```
class Program:
    @staticmethod
    def run():
        print("Execute Program")
Program.run()
```

Output:

```
Execute Program
```

Static-methods inside a class:

- A class can contain both static and non-static methods. If you want to call non-static methods, you'll have to create an object. The code below does not work because an object is not created:

```
class Program:
    @staticmethod
    def run():
        print("Execute Program")

    def stop(self):
        print("Stop Program")

Program.run()
Program.stop()
```

Output:

```
Execute Program
Traceback (most recent call last):
  File "C:\Users\acer\Desktop\test.py", line 10, in <module>
    Program.stop()
TypeError: stop() missing 1 required positional argument: 'self'
```

Calling static methods:

- Normal class methods and static methods can be mixed, we use both the concept of object orientation and functional programming mixed in one class. If you create an object, we can call non-static methods. But you can also call the static method without creating the object.

Example:

```
class Program:
    @staticmethod
    def run():
        print("Execute Program")

    def stop(self):
        print("Stop Program")

Program.run()
obj=Program()
obj.stop()
```

Output:

```
Execute Program
Stop Program
```


3.6.2 Class Method

- The class method in Python is a method which is bound to the class but not the object of that class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- Class method works with the class since its parameter is always the class itself.
- It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.
- For class methods, we need to specify `@classmethod` decorator.
- Class methods can be called by both class and object. These methods can be called with a class or with an object.

Syntax:

```
Class.classmethod()
Or even
Class().classmethod()
```

1. **Using classmethod():**

- `classmethod()` methods are bound to a class rather than an object. Class methods can be called by both class and object. These methods can be called with a class or with an object. `classmethod()` function is used in factory design patterns where we want to call many functions with the class name rather than the object.

Syntax:

```
classmethod(function)
```

Example:

```
class Book:
    name="Python Programming"

    def display(obj):
        print("Book Name is: ",obj.name)

Book.display=classmethod(Book.display)
Book.display()
```

Output:

```
Book Name is:  Python Programming
```

2. **Using The `@classmethod` Decorator:**

- A class method is a method which is bound to the class and not the object of the class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that would be applicable to all the instances.

Syntax:

```
class my_class:
    @classmethod
    def function_name(cls, arguments):
        #Function Body
        return value
```

Example:

```
@classmethod
class Student:
    def __init__(self, name, age):
        self.name = name  # instance attribute
        self.age = age # instance attribute
    @classmethod
    def getobject(cls):
        return cls('Vijay', 43)
```

Output:

```
>>> stu1=Student.getobject()
>>> stu1.name
'Vijay'
>>> stu1.age
43
>>>
```

@classmethod Characteristics:

- Declares a class method.
- The first parameter must be `cls` which can be used to access class attributes.
- The class method can only access the class attributes but not the instance attributes.
- The class method can be called using `ClassName.MethodName()` and also using object.
- It can return an object of the class.

3.7 ADDING AND RETRIEVING DYNAMIC ATTRIBUTES OF CLASSES

- Dynamic attributes in Python are attributes for instances that you define dynamically at runtime (after creating the instances).
- Python, everything is an object (even functions). So you can define a dynamic instance attribute for nearly anything.
- Dynamic attributes are ones that are defined after the object instance has been created. They can be patched elsewhere in the same code base or even come from external data sources. And because functions are also objects you can assign them with custom attributes too.

Example:

```
class MyClass():
    pass
def value():
    return 10
obj1=MyClass()
obj1.attribute1 = "Hello World!"
print(obj1.attribute1)
obj2=MyClass()
#Dynamic attribute of a class object
obj2.attribute2=value
#Dynamic attribute of a function
value.attribute2="Python"
print(value.attribute2)
print(obj2.attribute2()==value())
```

Output:

```
Hello World!
Python
True
```

3.8 PROGRAMMING USING OOPS

Program 1: Design a class that store the information of student and display the same.

```
class Student:
    def __init__(self, name, sex, course, result):
        self.name=name
        self.sex=sex
        self.course=course
        self.result=result
    def display(self, name, sex, course, result):
        self.name=name
        self.sex=sex
        self.course=course
        self.result=result
        print ('Name:', name)
        print ('Sex:',sex)
        print ('course:',course)
        print ('result:', result)
s1=Student()
s1.display('Vijay Patil','Male','MCA','91.23%')
```

Output:

```
Name: Vijay Patil
Sex: Male
course: MCA
result: 91.23%
```

Program 2: Write a program to add two complex numbers using classes and objects.

```
class complex:
    def __init__(self,real,imaginary):
        self.real=real
        self.imaginary=imaginary
    def __add__(self,other):
       return complex(self.real+other.real,self.imaginary+other.imaginary)
    def __sub__(self,other):
       return complex(self.real-other.real,self.imaginary-other.imaginary)
    def __mul__(self,other):
       return complex(self.real*other.real,self.imaginary*other.imaginary)
    def __str__(self):
        return f"{self.real}+i{self.imaginary}"
        return f"{self.real}-i{self.imaginary}"
        return f"{self.real}*i{self.imaginary}"

def main():
    complex_number_1=complex(4,5)
    complex_number_2=complex(2,3)
    complex_number_sum=complex_number_1+complex_number_2
    print(f"Addition of two complex numbers {complex_number_1} and
    {complex_number_2} is {complex_number_sum}")
    complex_number_sub=complex_number_1-complex_number_2
    print(f"Subtraction of two complex numbers {complex_number_1} and
    {complex_number_2} is {complex_number_sub}")
    complex_number_mul=complex_number_1*complex_number_2
    print(f"Multiplication of two complex numbers {complex_number_1} and
    {complex_number_2} is {complex_number_mul}")

if __name__=="__main__":
    main()
```

Output:

```
Addition of two complex numbers 4+i5 and 2+i3 is 6+i8
Subtraction of two complex numbers 4+i5 and 2+i3 is 2+i2
Multiplication of two complex numbers 4+i5 and 2+i3 is 8+i15
```

Program 3: Python program to create a class to print the area of a square and a rectangle. The class has two methods with the same name but different number of parameters. The method for printing area of rectangle has two parameters which are length and breadth respectively while the other method for printing area of square has one parameter which is side of square.


```
class Area:
    def print Area(self, x, y=None):
        if y is not None:
            print("Area of Rectangle=",x*y)
        else:
            print("Area of Square=",x*x);
    a=Area();
    a.printArea(10)
    a.printArea(10,20)
```

Output:

```
Area of Square= 100
Area of Rectangle= 200
```

Program 4: Python program to create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.

```
class Degree:
    def getDegree(self):
        print("I got a diploma degree")
class Undergraduate(Degree):
    def getDegree(self):
        print("I got a undergraduate degree")
class Postgraduate(Degree):
    def getDegree(self):
        print("I got a postgraduate degree")
d=Degree()
u=Undergraduate()
p=Postgraduate()
d.getDegree()
u.getDegree()
p.getDegree()
```

Output:

```
I got a diploma degree
I got a undergraduate degree
I got a postgraduate degree
```

3.9 DELEGATION AND CONTAINER

3.9.1 Delegation in Python

- Delegation is an object oriented technique (also called a Design pattern). Let's say you have an object x and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of x.

- One common use of delegation occurs with special method `__init__`. When Python creates an instance, the `__init__` methods of base classes are not automatically called, as they are in some other object-oriented languages. Thus, it is up to a subclass to perform the proper initialization of superclasses, by using delegation, if necessary.
- Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:
    def __init__(self, outfile):self._outfile = outfile
    def write(self, s): self._outfile.write(s.upper())
    def __getattr__(self, name):return getattr(self._outfile, name)
```

Here, the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self.__outfile.write()` method. All other methods are delegated to the underlying `self.__outfile` object. The delegation is accomplished via the `__getattr__` method.

3.9.2 Container in Python

- The collection Module in Python provides different types of containers. A Container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them. Some of the built-in containers are Tuple, List, Dictionary, etc.

1. Counter:

- It is used to keep the count of the elements in an iterable in the form of an unordered dictionary where the key represents the element in the iterable and value represents the count of that element in the iterable.

Syntax: `class collections.Counter([iterable-or-mapping])`

Example:

```
from collections import Counter

# With sequence of items
print(Counter(['B','B','A','B','C','A','B','B','A','C']))
 # with dictionary
print(Counter({'A':3, 'B':5, 'C':2}))
 # with keyword arguments
print(Counter(A=3, B=5, C=2))
```

Output:

```
Counter({'B': 5, 'A': 3, 'C': 2})
Counter({'B': 5, 'A': 3, 'C': 2})
Counter({'B': 5, 'A': 3, 'C': 2})
```

2. Chainmap:

- A ChainMap encapsulates many dictionaries into a single unit and returns a list of dictionaries.

Syntax: `class collections.ChainMap(dict1, dict2)`

Example:

```
from collections import ChainMap
d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}
d3 = {'e': 5, 'f': 6}
# Defining the chainmap
c = ChainMap(d1, d2, d3)
print(c)
```

Output:

```
ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6})
```

3. Deque:

- It is the optimized list for quicker append and pop operations from both sides of the container.
- Elements in deque can be inserted from both ends. To insert the elements from right `append()` method is used and to insert the elements from the left `appendleft()` method is used.
- Elements also can be removed from the deque from both the ends. To remove elements from right use `pop()` method and to remove elements from the left use `popleft()` method.

Example:

```
from collections import deque
# Declaring deque
dque=deque([10,20,30])
print(dque)
#insert element at end
dque.append(40)
dque.append(50)
print(dque)
#insert element at start
dque.appendleft(70)
print(dque)
#delete element of right side
dque.pop()
print(dque)
#delete element from left side
dque.popleft()
print(dque)
```

Output:

```
deque([10, 20, 30])
deque([10, 20, 30, 40, 50])
deque([70, 10, 20, 30, 40, 50])
deque([70, 10, 20, 30, 40])
deque([10, 20, 30, 40])
```

Summary

- 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.
- 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.
- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- The super() function is used to give access to methods and properties of a parent or sibling class.
- With operator overloading, we are able to change the meaning of a Python operator within the scope of a class.
- A static method (or static function) is a method defined as a member of an object but is accessible directly from an API object's constructor, rather than from an object instance created via the constructor.
- Dynamic attributes in Python are attributes for instances that you define dynamically at runtime (after creating the instances).
- A Container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them. Some of the built-in containers are Tuple, List, Dictionary, etc.

Check Your Understanding

1. In Python, a class is _____ for a concrete object.
   (a) A blueprint (b) A nuisance
   (c) An instance (d) A distraction
2. _____ is used to create an object.
   (a) Class (b) Constructor
   (c) User-defined functions (d) In-built functions
3. Which of the following Python code creates an empty class?
   (a) Class A: return (b) Class A: pass
   (c) Class A: (d) It is not possible to create an empty class.
4. Which of the following best describes inheritance?
   (a) Ability of a class to derive members of another class as a part of its own definition.
   (b) Means of bundling instance variables and methods in order to restrict access to certain class members.
   (c) Focuses on variables and passing of variables to functions.
   (d) Allows for implementation of elegant software that is well designed and easily modified.

5. Suppose B is a subclass of A, to invoke the __init__ method in A from B, what is the line of code you should write?
   (a) A.__init__(self) (b) B.__init__(self)
   (c) A.__init__(B) (d) B.__init__(A)
6. What type of inheritance is illustrated in the following Python code?

```
class A():
    pass
class B(A):
    pass
class C(B):
    pass
```

   (a) Multi-level inheritance (b) Multiple inheritance
   (c) Hierarchical inheritance (d) Single-level inheritance
7. Which is not an object?
   (a) String (b) List
   (c) Dictionary (d) None of the above
8. The assignment of more than one function to a particular operator is _____.
   (a) Operator over-assignment (b) Operator overriding
   (c) Operator overloading (d) Operator instance
9. Syntax of constructor in Python.
   (a) def__init__() (b) def init()
   (c) init() (d) All of these
10. What is the output of the following code?

```
class test:
 def init(self):
  print ("Hello World")
 def init(self):
  print ("Bye World")
obj=test()
```

    (a) Hello World (b) Compilation Error
    (c) Bye World (d) Ambiguity

Answers

1. (a)	2. (b)	3. (b)	4. (a)	5. (a)	6. (a)	7. (d)	8. (c)	9. (a)	10. (c)

Practice Questions

Q.I Answer the following questions in short.

1. List and explain built in class attributes with example.
2. Define Inheritance In Python Programming.

3. Define terms class, attributes and methods in Python.
4. State the use of parameter "self" in python class.
5. Differentiate between method overloading and method overriding.
6. Define destructor in Python.

Q.II Answer the following questions.

1. List the features and explain about different Object Oriented features supported by Python.
2. Design a class that store the information of student and display the same.
3. What are basic overloading methods?
4. What is method overriding? Write an example.
5. How operator overloading can be implemented in Python? Give an example.
6. Write a Python program to implement the concept of inheritance.
7. Define the following terms: Object, Class, Inheritance, Data abstraction.
8. Describe the term composition classes with example.
9. Explain customization via inheritance specializing inherited methods.
10. Explain constructor function in Python class with example.
11. What are different types of inheritance supported by Python? Explain.

Q.III Write a short note on:

1. Data abstraction
2. Class inheritance in Python
3. Destructor in Python
4. Container in Python
5. Delegation

Q.IV Write Programs.

1. Write a Python program to demonstrate parameterized constructor in base class and derived class.
2. Create a class employee with data members: name, department and salary. Create suitable methods for reading and printing employee information.
3. Write a Python program to overload + operator.
4. Create a class student with following member attributes: roll no, name, age and total marks. Create suitable methods for reading and printing member variables. Write a Python program to overload '==' operator to print the details of students having same marks.
5. Write a program to create a Python class and delete object of that class using del keyword.
6. Write a class with following criteria:
   Class name: Flower
   Objects: lily, rose, hibiscus
   Properties: price, color, smell
   Methods: get(), display()

■■■

4... Python Regular Expression

Objectives...

- To learn about pattern matching and searching in Python.
- To study pattern searching using regex in Python.
- To know about Real time parsing of data using regex.
- To learn Password, Email, URL validation using regular expression.
- To get information of Pattern finding programs using regular expression.

4.1 INTRODUCTION

- Regular expressions also called REs, or regexes, or regex patterns, are essentially a tiny, highly specialized programming language embedded inside Python, which provides a powerful way to search and manipulate strings.
- The Python module reprovides full support for Perl-like regular expressions in Python.
- Regular expressions use a sequence of characters and symbols to define a pattern of text. Such a pattern is used to locate a chunk of text in a string by matching up the pattern against the characters in the string.
- Regular expressions are a very useful technique for extracting information from text such as code, spreadsheets, documents or log files.
- Regular expressions are useful for finding phone numbers, email addresses, dates, and any other data that has a consistent format.

4.2 POWERFUL PATTERN MATCHING AND SEARCHING

- Python allows you to specify a pattern to match when searching for a substring contained in a string. This pattern is known as a **regular expression**. A regular expression can be formed by using the mix of meta-characters, special sequences, and sets. Meta-character is a character with a specified meaning.

Table 4.1: Meta-characters for Pattern Matching and Searching

Character	Description	Example
[]	It represents the set of characters. The square brackets[] are used for specifying a character class, also called a "character set," which is a set of characters that you wish to match.	
[abc]	Matches single character in the set i.e. either match a, b or c.	a[bB]c matches abc and aBc.
[^abc]	Match a single character other than a, b and c.	a[^bB]c matches a, followed by anything but b or B, followed by c.
[^a-z]	Any character not in a range of characters.	a[^a-z]c matches a, followed by anything that is not a lowercase letter, followed by c.
[a-z]	match a single character in the range a to z.	a[a-z]c matches a, followed by any non-capitalized letter, followed by c.
[a-zA-Z]	Match a single character in the range a-z or A-Z.	a[a-z]c matches a, followed by any non-capitalized letter, followed by any Capitalized letter, followed by c.
. (a period)	It matches any single character except a newline.	The pattern .n matches the substrings 'an' and 'on' in the string, but not after 'n' like not, nay, etc.
^	It represents the pattern present at the beginning of the string.	^abc matches abc in abcd, but does not match abc in dabc.
$	It represents the pattern present at the end of the string.	abc$ matches abc in dabc, but does not match abc in abcd.
*	It represents zero or more occurrences of a pattern in the string.	ab*c matches ac, abc, abbc, and so on.
+	It represents one or more occurrences of a pattern in the string.	ab+c matches abc, abbc, and so on.
?	It specifies 0 or 1 occurrence of character.	ab?c matches ac and abc.

contd. ...

|	It is used for alternation, essentially to specify 'this OR that' within a pattern.	abc|aBc matches abc or aBc.
{}	The specified number of occurrences of a pattern the string.	
{m}	The previous character can occur exact m times.	ab{3}c matches abbbc
{m,n}	The previous character can occur from m to n times.	ab{1,3}c matches abc, abbc and abbbc
()	Capture and group	(a|b|c)xz match any string that matches either a or b or c followed by xz.

- **Special Sequences:** Special sequences are the sequences containing \ followed by one of the characters.

Table 4.2: Special Characters

Character	Description	Example
\A	It returns a match if the specified characters are present at the beginning of the string.	\Athe match in string "the sun" \Athe not match in string "in the sun"
\b	It returns a match if the specified characters are present at the beginning or the end of the string. Matches a word boundary.	The pattern \bfoo matches in the string "football", but not matches in string "Afootball". And foo/b matches in the string "the foo".
\B	It returns a match if the specified characters are present at the beginning of the string but not at the end. Matches a non-word boundary.	The pattern \Bfoo matches in the string "football", but not match in string "afootball". Pattern foo/B match in string "A football", but not match in string "the foo".
\d	It returns a match if the string contains digits [0-9].	The pattern \d or [0-9] matches the character '2' in the string "A2 is the suite number."
\D	It returns a match if the string doesn't contain the digits [0-9]. Matches any non-digit character. Equivalent to [^0-9].	The pattern \D or [^0-9] matches the character 'A' in the string "A2 is the suite number."
\s	It returns a match if the string contains any white space character. Equivalent to [\n\t\f].	The pattern \s\w* matches the substring 'bar' in the string "Long bar."

contd. ...

\S	It returns a match if the string doesn't contain any white space character. Equivalent to [^ \n\t\f].	The pattern \S* matches the substring 'Long' in the string "Long bar."
\w	It returns a match if the string contains any word characters. Equivalent to [a-zA-Z0-9_].	The pattern \w matches the character 'a' in the string "apple", the character '3' in the string "3D."
\W	It returns a match if the string doesn't contain any word. Equivalent to [^A-Za-z0-9_].	The pattern \W or [^A-Za-z0-9_] matches the character '%' in the string "60%".
\Z	Returns a match if the specified characters are at the end of the string.	Python/Z matches in the string "the Python", but not matches in the string "Python is fun".

4.3 POWER OF PATTERN SEARCHING USING REGEX IN PYTHON

4.3.1 Compile Function

- With the `re.compile()` function, we can compile pattern into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions. This can be done using `re.compile()` method.

Example:

```
Import re
Pat=re.compile(r"")
```

- The purpose of the compile method is to compile the regex pattern which will be used for matching later. It's advisable to compile regex when it'll be used several times in your program. Resaving the resulting regular expression object for reuse, which re.compile does, is more efficient.

Program 1: Write a program to accept input from the user contains only letters, spaces or . (no digits). Any other character is not allowed.

```
import re
name_check = re.compile(r"[^A-Za-zs.]")
name =input ("Please, enter your name: ")
while name_check.search(name):
    print ("Please enter your name correctly!")
    name =input ("Please, enter your name: ")
```

Output:

```
Please, enter your name: 123
Please enter your name correctly!
Please, enter your name: abc123
Please enter your name correctly!
Please, enter your name: Vijay
```


Program 2: Write a program to accept input from the user contains only numbers, parentheses, spaces or hyphen (no letters). Any other character is not allowed.

```
import re
phone_check = re.compile(r"[^0-9s-]")
phone=input("Please, enter your phone: ")
while phone_check.search(phone):
    print ("Please enter your phone correctly!")
    phone=input ("Please, enter your phone: ")
```

Output:

```
Please, enter your phone: abc
Please enter your phone correctly!
Please, enter your phone: abc123
Please enter your phone correctly!
Please, enter your phone: 123456
```

- The `compile()` method returns a regular expression as a Python object which can be used for matching patterns by using its `match()`, `search()`, `sub()`, `findall()` and other methods.

Table 4.3: Methods Supported by Compiled Regular Expression Objects.

Method	Description
`match()`	This method matches the regex pattern in the string with the optional flag. It returns true if a match is found in the string otherwise it returns false.
`search()`	This method returns the match object if there is a match found in the string.
`findall()`	It returns a list that contains all the matches of a pattern in the string.
`split()`	Returns a list in which the string has been split in each match.
`sub`	Replace one or many matches in the string.

4.3.2 Match Function

- The `match()` method only find matches if they occur at the start of the string being searched. You can use the match function to match the RE pattern with the given string. The match function contains flags. Flags define the behavior of a regular expression and can contain different values.

Syntax: `re.match(pattern, string, flags)`

Table 4.4: Parameters of Match function

Parameter	Description
Pattern	It is the regular expression pattern which is to be matched.
String	It is the data string which is checked to match the pattern anywhere in the string.
Flags	It is used to change the behavior of regular expression, and it is optional.

- `re.match()` searches only from the beginning of the string and return match object if found. But if a match of substring is found somewhere in the middle of the string, it returns none. We would use `group(num)` or `groups()` function of match object to get matched expression.

Table 4.5: Match Object Methods

Match Object Method	Description
`group(num=0)`	This method returns entire match (or specific sub group **num**)
`groups()`	This method returns all matching sub groups in a tuple (empty if there weren't any).

Example:

```
>>>import re
>>>str="Hello Python Programming"
>>>result=re.match(r"hello", str)
>>>print(result)
<re.match (object; span = (0, 5), match='Hello')>
>>> print(result.group())
Hello
>>> result=re.match(r"Python",str,re.I)
>>> print(result)
None
```

Explanation:

- In this code, first of all re-module is imported. Then you will compare a string *str* with the RE pattern and the value returned from the match function will be assigned to *result*. The Match function is called using *re* then inside parenthesis the first argument is the pattern to be matched, and then you will have the given string from which pattern will be matched and also a flag value is passed. Here `re.I` is the flag value which means IGNORECASE, so it will be ignored whether the pattern and the string have different case letters (either upper case or lower case).

Regular-expression Modifiers-Flags:

- You can use Flags are to change the behavior of a regular expression. In a function, flags are optional.
- You can use flags in two different ways that is by either using the keyword flags and assigning it flag value or by directly writing the value of the flag.
- You can have more than one value of flag in the RE literal; this can be done by using bitwise OR operator '|'.

Table 4.6: Regular-expression Modifiers-Flags

Flag value	Long Syntax	Description
re.I	re.IGNORECASE	This modifier will ignore the case of strings and patterns while matching.
re.L	re.LOCALE	This modifier is used to interpret words with respect to the current locale.
re.M	re.MILTILINE	This modifier is used to make **$** to match to the end of the line and not to end of string. Similarly, **^** will match at the beginning of the line instead of at the beginning of the string.
re.S	re.DOTALL	This modifier is used to make a dot **.** to match any character. This includes a newline also.
re.U	re.UNICODE	This modifier is used to interpret the characters as Unicode character set.
re.X	re.VERBOSE	It is used to ignore the whitespaces. It will make **#** as a marker of comment.

Table 4.7: Methods supported by Match Object

Methods	Description
`group()`	If it is called without argument then it will returns the substring, which had been matched by the complete regular expression.
`groups()`	It returns all matching subgroups in a tuple.
`start()`	It returns the string index where the regular expression started matching in the string.
`end()`	It returns the string index where the regular expression ended matching in the string.
`span()`	It returns a tuple with the string index where the regular expression started matching in the string and ended matching.

4.3.3 Search Function

- This function searches for first occurrence of RE pattern within string with optional flags.

  Syntax: `re.search(pattern, string, flags=0)`

Table 4.8: Parameters of Search Function

Parameter	Description
pattern	It is the regular expression to be searched.
string	It is the data string which is checked to match the pattern anywhere in the string.
flags	This parameter is optional. It is used to control various aspects of matching. You can specify different flags at a time by using bitwise OR (|).

Example: Use of search function.

```
import re
>>> str="Hello Python Programming"
>>> result=re.search(r"Python",str)
>>> print(result.span())
(6, 12)
>>> print(result.group())
Python
>>> print(result.string)
Hello Python Programming
```

Program 3: Search the word 'programming' in String.

```
import re
str = "Hello Python Programming"
result = re.search(r"programming", str, re.I)
print(result.group())
```

Output:

```
Programming
```

Program 4: Search the word at the beginning of the string with ^ (search only at beginning of the string).

```
import re
str = "Hello Python Programming"
sobj = re.search(r"^programming", str, re.I)
print(sobj.group())                    #no match is found

sobj = re.search(r"^hello", str, re.I)
print(sobj.group())                    #matching: Hello
```

Program 5: Search the word at the end of the string with $ (search at the end of the string)

```
import re
str = "Hello Python Programming"
sobj = re.search(r"programming$", str, re.I)
print(sobj.group())                    #matching: Programming

sobj = re.search(r"hello$", str, re.I)
print(sobj.group())                    #no match found
```


Program 6: Search the word with length 5.

```
import re
str = "Hi Python Programming"
sobj = re.search(r"\w\w\w\w\w", str, re.I)
print(sobj.group())
```

Output:

```
Python
```

Program 7: Search the number in given string.

```
import re
str = "Hi Python Programming 10 "
sobj = re.search(r"\d\d", str)
print(sobj.group())
```

Output:

```
10
```

Program 8: Search the string within a given string.

```
import re
str = "Hello Python Programming"
sobj = re.search(r"Py\w+", str)
print(sobj.group())
```

Output:

```
Python
```

Program 9: Search Email-id from the string.

```
import re
str = "my email id is vp@gmail.com"
sobj = re.search(r"(\w+@\w+\.com)", str)
if sobj:
    print("String found="+sobj.group())
else:
    print("String not found")
```

Output:

```
String found=vp@gmail.com
```

Difference between match() and search() methods:

- Python offers two different primitive operations based on regular expressions: match and search.
- `match()` checks for a match only at the beginning of the string, while `search()` checks for a match anywhere in the string.

Example:

```
import re
str = "123python"
result1=re.match("[a-z]+",str)
result2=re.search("[a-z]+",str)
print("Match output=",result1)
print("Search output=",result2)
```

Output:

```
Match output= None
Search output= <re.Match object; span=(3, 9), match='python'>
```

4.3.4 Findall Function

- Finds all the possible matches in the entire sequence and returns them as a list of strings. This method returns a list containing a list of all matches of a pattern within the string.
- It returns the patterns in the order they are found. If there are no matches, then an empty list is returned.

Syntax: `Re.findall(pattern, string,flag)`

Example:

```
import re
str = " i have 1 cat, 2 rat, 3 mat, 4 bat, 5 hats"
print("String=",str)
result=re.findall("[crmb]at",str)
print("Space Seperator=",result)
```

Output:

```
String=  i have 1 cat, 2 rat, 3 mat, 4 bat, 5 hats
Space Seperator= ['cat', 'rat', 'mat', 'bat']
```

4.3.5 Split Function

- The split function is used to split a string. The `re.split()` method accepts a pattern argument.
- This pattern specifies the delimiter. With it, we can use any text that matches a pattern as the delimiter to separate text data.

Syntax: `str.split(separator, maxsplit)`

where,

- **separator**: The is a delimiter. The string splits at this specified separator. If is not provided, then any white space is a separator.
- **maxsplit**: It is a number, which tells us to split the string into maximum of provided number of times. If it is not provided, then there is no limit.

Example:

```
import re
str = "One Two, Three Four"
print("String=",str)
result1=str.split()
print("Space Seperator=",result1)
result2=str.split(',')
print("Comma Seperator",result2)
result3=str.split(' ',2)
print("space seperator with maxsplit as 2=",result3)
```

Output:

```
String= One Two, Three Four
Space Seperator= ['One', 'Two,', 'Three', 'Four']
Comma Seperator ['One Two', ' Three Four']
space seperator with maxsplit as 2= ['One', 'Two,', 'Three Four']
```

4.3.6 sub Function

- The `re.sub()` function in the re module can be used to replace substrings.

Syntax: `re.sub(pattern, replace, string, count)`

Example:

```
import re
str = "hello python hello programming"
print("String=",str)
result1=re.sub("[hH]ello","Welcome",str)
print("Replace all=",result1)
result2=re.sub("[hH]ello","Welcome",str,1)
print("Replace only one occurrence=",result2)
```

Output:

```
String= hello python hello programming
Replace all= Welcome python Welcome programming
Replace only one occurrence= Welcome python hello programming
```

4.4 REAL TIME PARSING OF DATA USING REGEX

- Regular expressions(regex) are essentially text patterns that you can use to automate searching through and replacing elements within strings of text. This can make cleaning and working with text-based data sets much easier, saving you the trouble of having to search through mountains of text by hand.
- Some of the projects where regex is used are:
  - **Web Scraping:** While most data used in classes and textbooks just appears ready-to-use in a clean format, in reality, the world does not play so nice. Getting data usually means getting our hands dirty, in this case pulling (also known as

scraping) data from the web. Python has great tools for doing this, namely the requests library for retrieving content from a webpage, and bs4 (BeautifulSoup) for extracting the relevant information.

- **Visualization:** This project is indicative of data science because the majority of time was spent collecting and formatting the data. However, now that we have a clean dataset, we get to make some plots! We can use both *matplotlib* and *seaborn* to visualize the data.
- **Extract Emails from text:** Extracting email address from the text file.
- **Using Regex for Text Pre-processing (NLP):** When working with text data, especially in NLP where we build models for tasks like text classification, machine translation and text summarization, we deal with a variety of text that comes from diverse sources. For instance, we can have web scraped data, or data that's manually collected, or data that's extracted from images using OCR techniques and so on!

4.5 PASSWORD, E-MAIL, URL VALIDATION USING REGULAR EXPRESSION

4.5.1 Password Validation in Python using regex

- Conditions for password validation:
  - Minimum length of password: 8 characters.
  - At least one alphabet must between [a-z].
  - At least one alphabet should be of Upper Case [A-Z].
  - At least 1 number or digit between [0-9].
  - At least 1 character from [$, @, #].

Program 10: Python Program for Password validation.

```
import re
def main():
# Pattern r'[a-z] checks for at least one lowercase between a and z
    lower_case_pattern = re.compile(r'[a-z]')
# Pattern r'[a-z]' checks for at least one lowercase between A and Z
    upper_case_pattern = re.compile(r'[A-Z]')
# Pattern r'\d' checks for at least one number between 0 and 9
    number_pattern = re.compile(r'\d')
# Pattern r'[$#@]'checks for at least one character from $, #, @
    special_character_pattern = re.compile(r'[$#@]')
    password = input("Enter a Password ")
    if len(password) < 8 :
        print("Invalid Password. Length Not Matching")
    elif not lower_case_pattern.search(password):
        print("Invalid Password. No Lower-Case Letters")
```



```
    elif not upper_case_pattern.search(password):
        print("Invalid Password. No Upper-Case Letters")
    elif not number_pattern.search(password):
        print("Invalid Password. No Numbers")
    elif not special_character_pattern.search(password):
        print("Invalid Password. No Special Characters")
    else:
        print("Valid Password")
if __name__ == "__main__":
    main()
```

Output:

```
Enter a Password abc
Invalid Password. Length Not Matching
Enter a Password abc123456
Invalid Password. No Upper-Case Letters
Enter a Password ABCD123456
Invalid Password. No Lower-Case Letters
Enter a Password abc123ABC
Invalid Password. No Special Characters
Enter a Password ABC123@abc
Valid Password
```

4.5.2 E-mail Validation in Python using regex

- An email is a string (a subset of ASCII characters) separated into two parts by @ symbol, a "personal_info" and a domain, that is personal_info@domain. re.search() method either returns None (if the pattern doesn't match), or `re.MatchObject` that contains information about the matching part of the string.

Program 11: Python program to check if the string is a valid E-mail address or not.

```
import re
regex = '^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'
def check(email):
    if (re.search(regex,email)):
        print('Valid Mail Address')
    else:
        print ('Invalid Mail Address')

if __name__ == '__main__' :
        email = input("Enter E-mail ")
        check(email)
```

Output:

```
Enter E-mail vijay@gmail
Invalid Mail Address
Enter E-mail vijay@gmail
Invalid Mail Address
Enter E-mail vijay@gmail.com
Valid Mail Address
Enter E-mail vijay.patil@gmail.com
Valid Mail Address
```

4.5.3 URL Validation in Python using regex

- We can accept a string and need to check is the string contains any URL in it. If the valid URL is present in the string, we will print Valid URL's been found or not.

Program 12: Python program to check if an URL is valid or not using Regular Expression.

```
import re
def checkurl(str):
    regex = ("((http|https)://)(www.)?"+
         "[a-zA-Z0-9@:%._\\+~#?&//=]"+
         "{2,256}\\.[a-z]"+
         "{2,6}\\b([-a-zA-Z0-9@:%"+
         "._\\+~#?&//=]*)")
    p=re.compile(regex)
    if(str==None):
        return False
    if(re.search(p,str)):
        return True
    else:
        return False
if __name__ == '__main__' :
      url = input("Enter URL ")
      if(checkurl(url)==True):
          print("Valid URL")
      else:
           print("Invalid URL")
```

Output:

```
Enter URL google.com
Invalid URL

Enter URL https://www.google.com
Valid URL
```


4.6 PATTERN FINDING PROGRAMS USING REGULAR EXPRESSION

- re.search() function will search the regular expression pattern and return the first occurrence. The Python re.search() function returns a match object when the pattern is found and "null" if the pattern is not found.

Program 13: Write a Python program that matches a word containing 'z', not at the start or end of the word.

```
import re
def text_match(text):
        patterns = '\Bz\B'
        if re.search(patterns,  text):
                return 'Match Found !'
        else:
                return('Match Not Found !')
print(text_match("We saw a Chimpanzee in the jungle."))
print(text_match("We saw a tiger in the jungle."))
```

Output:

```
Match Found !
Match Not Found !
```

Program 14: Write a Python program to find sequences of lower case letters joined with an underscore.

```
import re
def text_match(text):
        patterns = '^[a-z]+_[a-z]+$'
        if re.search(patterns, text):
                return 'Match Found!'
        else:
                return('Match Not Found!')
print(text_match("aaa_bbb"))
print(text_match("aaa_Bbb"))
print(text_match("Aaa_bbb"))
```

Output:

```
Match Found!
Match Not Found!
Match Not Found!
```

Program 15: Write a Python program to remove all leading zeros' from an IP address.

```
import re
ip1 = "192.165.094.122"
ip2 = "192.008.094.122"
string1 = re.sub('\.[0]*', '.', ip1)
string2 = re.sub('\.[0]*', '.', ip2)
print(string1)
print(string2)
```

Output:

```
192.165.94.122
192.8.94.122
```

Program 16: Write a Python program to find the substrings within a string.

```
import re
text = 'Python Program, Java Program, Perl Program'
pattern = 'Program'
for match in re.finditer(pattern, text):
    s = match.start()
    e = match.end()
    print('Found "%s" at %d:%d' % (text[s:e], s, e))
```

Output:

```
Found "Program" at 7:14
Found "Program" at 21:28
Found "Program" at 35:42
```

Program 17: Write a Python program to extract year, month and date from an URL.

```
import re
def extract_date(url):
        return re.findall(r'/(\d{4})/(\d{1,2})/(\d{1,2})/', url)
url1= "https://www.google.com/news/Market/2021/22/02/sensex/"
print(extract_date(url1))
```

Output:

```
[('2021', '22', '02')]
```

Program 18: Write a Python program to abbreviate 'Street' as 'St.' in a given string.

```
import re
street = 'Wadala Road'
print(re.sub('Road$', 'Rd.', street))
```

Output:

```
Wadala Rd.
```

Program 19: Write a Python program to find all five characters long word in a string.

```
import re
text = 'The python are clike free easy to learn and open source language.'
print(re.findall(r"\b\w{5}\b", text))
```

Output:

```
['clike', 'learn']
```

Summary

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- Meta-characters are characters that are interpreted in a special way by a RegEx engine. For example: [] . ^ $ * + ? {} () \ |
- The module re has to be imported to be able to work with regular expressions.
- Use the compile() method in re module to compile regular expression to match objects.

- The search() function searches the string for a match, and returns a Match object if there is a match.
- The split() function returns a list where the string has been split at each match.
- Match() checks for a match only at the beginning of the string.

Check Your Understanding

1. What does the function re.match do?
   (a) Matches a pattern at the start of the string.
   (b) Matches a pattern at any position in the string.
   (c) Such function does not exist.
   (d) None of the mentioned.
2. The expression a{5} will match _____ characters with the previous regular expression.
   (a) 5 or less (b) exactly 5
   (c) 5 or more (d) exactly 4
3. _____ matches the start of the string. _____ matches the end of the string.
   (a) '^', '$' (b) '$', '^'
   (c) '$', '?' (d) '?', '^'
4. What will be the output of the following Python code?
   `re.split('\W+', 'Hello, hello, hello.')`
   (a) ['Hello', 'hello', 'hello.'] (b) ['Hello, 'hello', 'hello']
   (c) ['Hello', 'hello', 'hello', '.'] (d) ['Hello', 'hello', 'hello', '']
5. Python has a built-in package called?
   (a) reg (b) regex
   (c) re (d) regx
6. Which function returns a list containing all matches?
   (a) findall (b) search
   (c) split (d) find
7. In Regex, [a-n] stands for?
   (a) Returns a match for any digit between 0 and 9.
   (b) Returns a match for any lower case character, alphabetically between a and n.
   (c) Returns a match for any two-digit numbers from 00 and 59.
   (d) Returns a match for any character EXCEPT a, r, and n.
8. Which character stands for Zero or more occurrences in regex?
   (a) * (b) #
   (c) @ (d) |
9. What will be the output of the following Python code?

```
import re
re.ASCII
```

   (a) 8 (b) 32
   (c) 64 (d) 256

10. Which of the following functions results in case insensitive matching?
    (a) re.A (b) re.U
    (c) re.I (d) re.X

Answers

1. (a)	2. (b)	3. (a)	4. (d)	5. (c)	6. (a)	7. (b)	8. (a)	9. (d)	10. (c)

Practice Questions

Q.I Answer the following questions in short.

1. Define regular expression and list out all the advantages of the regular expression.
2. Describe any ten meta-characters with examples.
3. Compare and contrast the use of match() and search() methods with an example.
4. Define matching? Explain with suitable example.
5. Describe any one regular expression with an example.
6. Explain the split function.

Q.II Answer the following questions.

1. Briefly describe the methods of regular expression.
2. Describe all the functions available in match objects.
3. What are regular expressions? How to find whether an email id entered by user is valid or not using Python 're' module.
4. Explain the function that is used to retrieve the parts of URL.

Q.III Write a short note on:

1. Findall function
2. Compile function
3. Searching
4. Regular-expression Modifiers-Flags
5. Meta-characters

Q.IV Write Programs.

1. Write a Python program to read a file and to convert a date of yyyy-mm-dd format to dd-mm-yyyy format.
2. Write a Python program to match if two words from a list of words starting with letter 'P'.
3. Write a Python program to find the occurrence and position of the substrings within a string.
4. Write Python program to retrieve string starting with 'm' and having 5 characters.

■■■

5... Python Multithreading and Exception Handling

Objectives...

- To study Exception handling in Python.
- To know how to handle and help developer with error code.
- To learn how to write programs using Exception handling.
- To know the concept of Multithreading.
- To understand the concept of threads and synchronizing the threads.
- To learn how to write programs using Multithreading.

5.1 EXCEPTION HANDLING

- When we execute a Python program, there may be a few uncertain conditions which occur, known as errors. Errors are also referred to as bugs that are incorrect or inaccurate actions that may cause the problems in the running of the program or may interrupt the execution of program.
- There are following three types 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 the program by the user.
  3. **Logical Errors:** Occurs due to wrong logic written in the program.
- An exception is also called as **runtime error** that can halt the execution of the program.
- Python provides a feature (Exception handling) for handling any unreported errors in the 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.
- Exceptions 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.
- An exception is an error that happens/occurs during execution of a program. When that error occurs, Python generates an exception that can be handled which avoids the normal flow of the program's instructions.

  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
```

- In Python programming, we can handle exceptions using *try-except* statement, *try-finally* statement and *raise* statement.
- Following table lists all the standard exceptions available in Python programming language:

Table 5.1: Standard Exceptions in Python programming

Exception	Cause of Error
ArithmeticError	Base class for all errors that occur for numeric calculation.
AssertionError	Raised in case of failure of the assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
Exception	Base class for all exceptions.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
FloatingPointError	Raised when a floating point calculation fails.
ImportError	Raised when an import statement fails
IndexError	Raised when an index is not found in a sequence.
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.

contd. ...

IndentationError	Raised when indentation is not specified properly.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+C.
KeyError	Raised when the specified key is not found in the dictionary.
LookupError	Base class for all lookup errors.
NameError	Raised when an identifier is not found in the local or global Namespace.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
OSError	Raised for operating system-related errors.
RuntimeError	Raised when a generated error does not fall into any category.
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
SyntaxError	Raised when there is an error in Python syntax.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.

5.2 AVOIDING CODE BREAK USING EXCEPTION HANDLING

- 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.
- 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 display the same message that represents information about the exception.
- For handling exceptions 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.

5.2.1 try-except statement

- 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 exception is thrown in the try block, the catch block is skipped.

- There can be one or more except blocks. Multiple except blocks with different exception names can be chained together.
- The except blocks are evaluated from top to bottom in the code, but only one except block is executed for each exception that is thrown.
- The first except block that specifies the exact exception name of the thrown exception is executed. If no except block specifies a matching exception name then an except block that does not have an exception name is selected, if one is present in the code.

Syntax:

```
try:
        certain operations here
        ......................
except Exception1:
        If there is Exception1, then execute this block.
except Exception2:
        If there is Exception2, then execute this block.
......................
else:
    If there is no exception then execute this block.
```

Example: 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
```

5.2.2 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)
```

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
```

5.2.3 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()
```

Program 1: Program to check for ZeroDivisionError Exception.

```
x=int(input("Enter first value:"))
y=int(input("Enter second value:"))
try:
        result=x/y
except ZeroDivisionError:
        print("Division by Zero")
else:
        print("Result is:",result)
finally:
        print("Execute finally clause")
```

Output 1:

```
Enter first value:5
Enter second value:0
Division by Zero
Execute finally clause
```

Output 2:

```
Enter first value:10
Enter second value:5
Result is: 2.0
Execute finally clause
```

5.2.4 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
```

5.3 SAFE GUARDING FILE OPERATION USING EXCEPTION HANDLING

- **The Python file object attributes:** When you call the Python `open()` function, it returns an object, which is the `filehandle`. Also, you should know that Python files have several linked attributes and we can make use of the `filehandle` to list the attributes of a file.

Table 5.2: Python file object Attributes

Attributes	Description
<file.closed>	For a closed file, it returns true whereas false otherwise.
<file.mode>	It returns the access mode used to open a file.
<file.name>	Returns the name of a file.
<file.softspace>	It returns a Boolean to suggest if a space char will get added before printing another value in the output of a <print> command.

Table 5.3: Built-in Exception related to files

Exception name	Description
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
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.
PermissionError	Raised when trying to run an operation without access rights.
IsADirectoryError	Raised when file operation is requested on a directory.
FileNotFoundError	Raised when a file/directory is requested but doesn't exist. Occurs when we are trying to append/read such a file.
FileExistsError:	Raised when trying to create an existing file/directory.

Example:

```
Open a file in write and binary mode.
fob = open("abc.txt", "wb")

#Display file name.
print ("File name: ", fob.name)
#Display state of the file.
print ("File state: ", fob.closed)
#Print the opening mode.
print ("Opening mode: ", fob.mode)
```

Output:

```
File name:  abc.txt
File state:  False
Opening mode:  wb
```

5.4 HANDLING AND HELPING DEVELOPER WITH ERROR CODE

Process of handling Exception:

- When an error occurs, Python interpreter creates an object called the **exception object**. This object contains information about the error like its type, file name and position in the program where the error has occurred. The object is handed over to the runtime system so that it can find an appropriate code to handle this particular exception. This process of creating an exception object and handing it over to the runtime system is called **throwing an exception**.
- It is important to note that when an exception occurs while executing a particular program statement, the control jumps to an exception handler, abandoning execution of the remaining program statements.
- The runtime system searches the entire program for a block of code, called the **exception handler** that can handle the raised exception. It first searches for the

method in which the error has occurred and the exception has been raised. If not found, then it searches the method from which this method (in which exception was raised) was called. This hierarchical search in reverse order continues till the exception handler is found. This entire list of methods is known as **call stack**.

- When a suitable handler is found in the call stack, it is executed by the runtime process. This process of executing a suitable handler is known as **catching the exception**. If the runtime system is not able to find an appropriate exception after searching all the methods in the call stack, then the program execution stops.

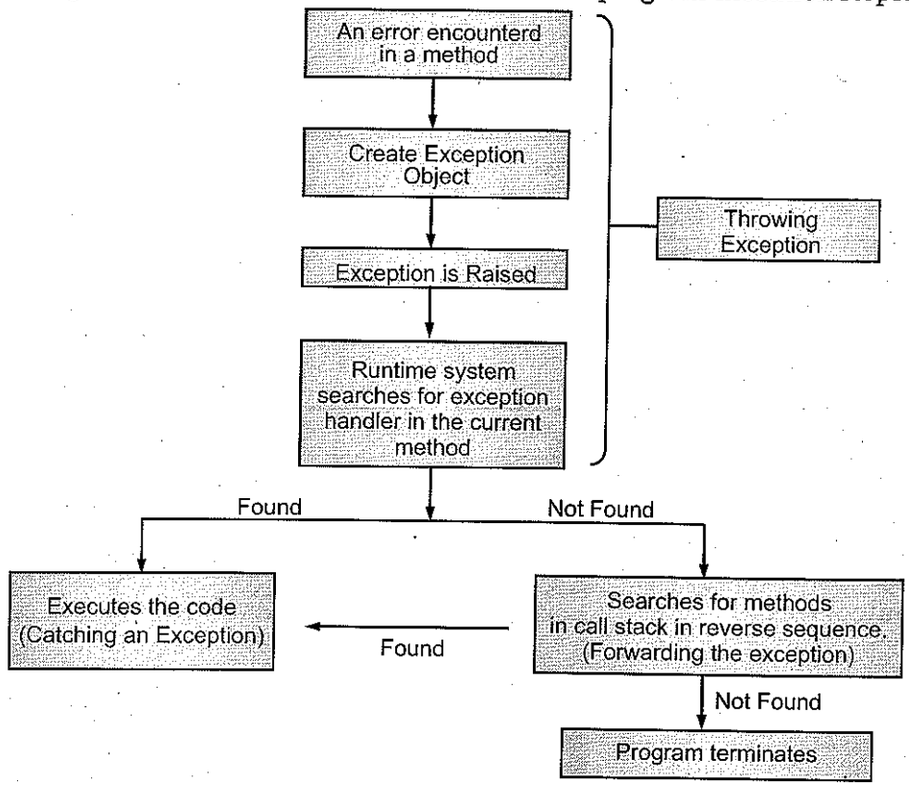


Fig. 5.1: Steps for Handling Exception

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 serve 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.

Program 2: Raise a user defined exception if age is less than 18.

```
# define Python user-defined exceptions
class Error(Exception):
"""Base class for other exceptions""" # empty class
   pass
class AgeSmallException(Error):
"""Raised when the input value is too small""" # empty class
   pass
# main program
while True:
      try:
            age = int(input("Enter your age for election: "))
            if age < 18:
                  raise AgeSmallException
            else:
                  print("You are eligible for election!")
                  break
except AgeSmallException:
      print("This value is too small, try again!")
      print()
```

Output:

```
Enter your age for election: 11
This value is too small, try again!
Enter your age for election: 15
This value is too small, try again!
Enter your age for election: 18
You are eligible for election!
```

Program 3: 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:\Python\Python37\p1.py", line 12, in
<module>
verify_password("xyz") # will raise exception
File "C:\Python\Python37\p1.py", line 6, in
verify_password
raise InvalidPassword
InvalidPassword
```

5.5 PROGRAMMING USING EXCEPTION HANDLING

Program 4: Program to accept a number from the user. Include the try block that raises a ValueError exception if the number is outside the allowed range.

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowerange")
```

Output:

```
Enter a number upto 100: 50
50 is within the allowed range
Enter a number upto 100: 200
200 is out of allowed range
```

Program 5: Program to handle multiple possible exception cases using multiple except blocks.

```
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
    print("You have divided a number by zero, which is not allowed.")
# except block handling wrong value type
except(ValueError):
print("You must enter integer value")
# genericxcept block
```


Output:

```
Enter numerator number: 8
Enter denominator number: 0
You have divided a number by zero, which is not allowed.

Enter numerator number: 6
Enter denominator number: s
You must enter integer value
```

5.6 MULTITHREADING

- A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
- A **thread** is a sequence of such instructions within a program that can be executed independently of other code.
- In software programming, a thread is the smallest unit of execution with the independent set of instructions. It is a part of the process and operates in the same context sharing program's runnable resources like memory.
- The ability of a process to execute multiple threads in a parallel way is called **Multithreading**.
- Multithreading is used to improve the performance of any program.
- Python multithreading mechanism is pretty user-friendly which you can learn quickly.
- Multiple threads can exist within one process where:
  - Each thread contains its own **register set** and **local variables (stored in stack)**.
  - All thread of a process shares **global variables (stored in heap)** and the **program code**.

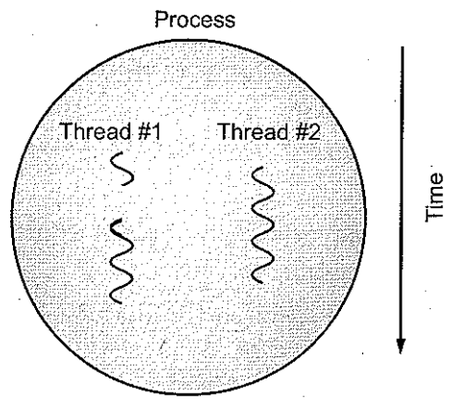


Fig. 5.2: Multithreading

Advantages of Multithreading:

- Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.

- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.
- Multithreading allows a program to remain responsive while one thread waits for input and another runs a GUI at the same time. This statement holds true for both multiprocessor and single processor systems.
- All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.
- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

5.7 UNDERSTANDING THREADS

Thread module:

- Python has two modules to implements threads:
  1. <thread> module.
  2. <threading> module.
- The <thread> module is deprecated in Python3 and renamed to <_thread> module for backward compatibility. The key difference between the two modules is that the module <thread> implements a thread as a function. On the other hand, the module <threading> offers an object-oriented approach to enable thread creation.

5.7.1 Creating thread using thread module

- We can use following method to spawn threads:

```
thread.start_new_thread ( function, args[, kwargs] )
```

- This method is a quite simple and efficient way to create threads. You can use it to run programs in both Linux and Windows. This method starts a new thread and returns its identifier. It invokes the function specified as the "function" argument with the passed list of arguments. When the <function> returns, the thread would silently exit.
- Here, second argument is "args" which is a tuple of arguments; use an empty tuple to call <function> without any arguments.
- The optional <kwargs> argument specifies the dictionary of keyword arguments.

Example 1: To create new threads.

```
import _thread
import time
# Define a function for the thread
def print_time( threadName, delay):
   count = 0
```



```
   while count < 5:
      time.sleep(delay)
      count += 1
      print ("%s: %s" %( threadName, time.ctime(time.time()) ))
# Create two threads as follows
try:
   _thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   _thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print ("Error: unable to start thread")
while 1:
   pass
```

Output:

```
Thread-1: Mon Mar  4 11:32:24 2019
Thread-2: Mon Mar  4 11:32:26 2019
Thread-1: Mon Mar  4 11:32:26 2019
Thread-1: Mon Mar  4 11:32:28 2019
Thread-2: Mon Mar  4 11:32:30 2019
Thread-1: Mon Mar  4 11:32:30 2019
Thread-1: Mon Mar  4 11:32:32 2019
Thread-2: Mon Mar  4 11:32:34 2019
Thread-2: Mon Mar  4 11:32:38 2019
Thread-2: Mon Mar  4 11:32:42 2019
```

Example 2: To implement ctime and sleep function in Python.

```
import _thread
from time import sleep, ctime
def t1():
    print("start of t1 at:",ctime())
    sleep(2)
    print("end of t1 at:",ctime())

def t2():
    print("start of t2 at:",ctime())
    sleep(4)
    print("end of t2 at:",ctime())
def main():
    print("Starting at:",ctime())
```

```
        _thread.start_new_thread(t1,())
        _thread.start_new_thread(t2,())
        sleep(6)
        print("End of main:",ctime())

    main()
```

Output:

```
Starting at: Mon Mar  4 11:37:00 2019
start of t1 at: start of t2 at: Mon Mar 4 11:37:00 2019Mon Mar 4 11:37:00
2019

end of t1 at: Mon Mar  4 11:37:03 2019
end of t2 at: Mon Mar  4 11:37:05 2019
End of main: Mon Mar  4 11:37:06 2019
```

5.7.2 Creating thread using threading module

- The latest <threading> module provides rich features and greater support for threads than the legacy <thread> module discussed in the previous section. The <threading> module is an excellent example of Python Multithreading.
- The <threading> module combines all the methods of the <thread> module and exposes few additional methods.
  1. **threading.activeCount():** It finds the total no. of active thread objects.
  2. **threading.currentThread():** You can use it to determine the number of thread objects in the caller's thread control.
  3. **threading.enumerate():** It will give you a complete list of thread objects that are currently active.
- The <threading> module also presents the <thread> class that you can try for implementing threads. It is an object-oriented variant of Python multithreading.

Table 5.4: Methods of thread class

Methods	Description
run()	It is the entry point function for any thread.
start()	The start() method triggers a thread when run method is called.
join([time])	The join() method enables a program to wait for threads to terminate.
isAlive()	The isAlive() method verifies an active thread.
getName()	The getName() method retrieves the name of a thread.
setName()	The setName() method updates the name of a thread.

Steps to implement threads using the Threading Module:

1. Construct a subclass from the <Thread> class.
2. Override the <__init__(self [,args])> method to supply arguments as per requirements.
3. Next, override the <run(self [,args])> method to code the business logic of the thread.

- Once you define the new <Thread> subclass, you have to instantiate it to start a new thread. Then, invoke the <start()> method to initiate it. It will eventually call the <run()> method to execute the business logic.

Example: Implement threads using the Threading Module.

```
import time
from threading import Thread

def myfunc(i):
    print ("sleeping 5 sec from thread %d" % i)
    time.sleep(5)
    print ("finished sleeping from thread %d" % i)

for i in range(10):
    t = Thread(target=myfunc, args=(i,))
    t.start()
```

Output:

```
sleeping 5 sec from thread 0
sleeping 5 sec from thread 3
sleeping 5 sec from thread 1
sleeping 5 sec from thread 4
sleeping 5 sec from thread 5
sleeping 5 sec from thread 6
sleeping 5 sec from thread 7
sleeping 5 sec from thread 8
sleeping 5 sec from thread 9
>>>
sleeping 5 sec from thread 2
finished sleeping from thread 0
finished sleeping from thread 3
finished sleeping from thread 1
finished sleeping from thread 4
finished sleeping from thread 5
finished sleeping from thread 6
finished sleeping from thread 7
finished sleeping from thread 8
finished sleeping from thread 9
finished sleeping from thread 2
```

5.8 SYNCHRONIZING THE THREADS

- The threading module provided with Python includes a simple to implement locking mechanism that allows you to synchronize threads.
- A new lock is created by calling the `lock()` method, which returns the new lock.
- The acquire(blocking) method of new lock object is used to force the threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.
- If blocking is set to 0, the threads return immediately with a 0 value. If the lock cannot be acquired and with a 1 is the lock was acquired. If blocking is set to 1, the thread blocks and waits for the lock to be released.
- The `release()` method of the new lock object is used to release the lock when it is no longer required.

Program 6: Program for synchronizing the threads.

```
import threading
import time
class myThread (threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print("Starting " + self.name)
      # Get lock to synchronize threads
      threadLock.acquire()
      print_time(self.name, self.counter, 3)
      # Free lock to release next thread
      threadLock.release()

def print_time(threadName, delay, counter):
   while counter:
      time.sleep(delay)
      print ("%s: %s" % (threadName, time.ctime(time.time())))
      counter -= 1

threadLock = threading.Lock()
threads = []

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
```



```
# Start new Threads
thread1.start()
thread2.start()

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exiting Main Thread")
```

Output:

```
Starting Thread-1
Starting Thread-2

Thread-1: Mon Mar  8 08:48:12 2021
Thread-1: Mon Mar  8 08:48:13 2021
Thread-1: Mon Mar  8 08:48:14 2021
Thread-2: Mon Mar  8 08:48:16 2021
Thread-2: Mon Mar  8 08:48:18 2021
Thread-2: Mon Mar  8 08:48:20 2021
Exiting Main Thread
```

5.9 PROGRAMMING USING MULTITHREADING

Program 7: Write a program to illustrate creation of thread with different state.

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
  format = "%(asctime)s: %(message)s"
  logging.basicConfig(format=format,level=logging.INFO,datefmt="%H:%M:%S")
  logging.info("Main    : before creating thread")
  x = threading.Thread(target=thread_function, args=(1,))
```

```
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

Output:

```
19:23:31: Main    : before creating thread
19:23:31: Main    : before running thread
19:23:31: Thread 1: starting
19:23:31: Main    : wait for the thread to finish
19:23:31: Main    : all done
19:23:33: Thread 1: finishing
```

Program 8: Write multithread program, where one thread prints square numbers and another thread prints cube of numbers.

```
import threading
def print_cube(num):
    print("Cube: {}".format(num * num * num))

def print_square(num):
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()

    # both threads completely executed
    print("Completed!")
```

Output:

```
Square: 100
Cube: 1000
Completed!
```


Program 9: Develop Python program to run three threads.

```
import time
import _thread

def thread_test(name, wait):
   i = 0
   while i <= 3:
      time.sleep(wait)
      print("Running %s\n" %name)
      i = i + 1

   print("%s has finished execution" %name)

if __name__ == "__main__":

    _thread.start_new_thread(thread_test, ("First Thread", 1))
    _thread.start_new_thread(thread_test, ("Second Thread", 2))
    _thread.start_new_thread(thread_test, ("Third Thread", 3))
```

Output:

```
Running First Thread
Running Second Thread
Running First Thread
Running Third Thread
Running First Thread
Running Second Thread
Running First Thread
First Thread has finished execution
Running Second Thread
Running Third Thread
Running Second Thread
Second Thread has finished execution
Running Third Thread
Running Third Thread
Third Thread has finished execution
```

Summary

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- A try statement can have more than one except clause, to specify handlers for different exceptions.

- `else` keyword is used to define a block of code to be executed if no errors were raised.
- The finally block always executes after normal termination of try block or after try block terminates due to some exception.
- The raise statement allows the programmer to force a specific exception to occur.
- Multithreading refers to concurrently executing multiple threads by rapidly switching the control of the CPU between threads (called context switching).
- A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

Check Your Understanding

1. How many except statements can a try-except block have?
   (a) zero (b) one
   (c) more than one (d) more than zero
2. When is the finally block executed?
   (a) When there is no exception.
   (b) When there is an exception.
   (c) Only if some condition that has been specified is satisfied.
   (d) Always
3. What will be the output of the following Python code?

```
def foo():
    try:
        return 1
    finally:
        return 2
k = foo()
print(k)
```

   (a) 1
   (b) 2
   (c) 3
   (d) Error, there is more than one return statement in a single try-finally block.
4. What will be the output of the following Python code?

```
#generator
def f(x):
    yield x+1
g=f(8)
print(next(g))
```

   (a) 8 (b) 9
   (c) 7 (d) Error

5. Which of the following is not an exception handling keyword in Python?
   (a) try (b) except
   (c) accept (d) finally
6. In _____ an object of type Thread in the namespace System. Threading represents and controls one thread.
   (a) .PY (b) .SAP
   (c) .NET (d) .EXE
7. Command to make thread sleep?
   (a) Thread.Sleep (b) Thread_Sleep
   (c) ThreadSleep (d) Thread_sleep
8. Multithreading is a mechanism for splitting up a program into several parallel activities called _____.
   (a) Methods (b) Objects
   (c) Classes (d) Threads
9. Scheduler switches threads in _____.
   (a) Multilevel queue scheduling (b) Priority Scheduling
   (c) Round robin fashion (d) Multilevel feedback queue scheduling
10. What is the switching speed?
    (a) 60 times per second (b) 50 times per second
    (c) 55 times per second (d) 66 times per second

Answers

1. (d)	2. (d)	3. (b)	4. (b)	5. (c)	6. (c)	7. (a)	8. (c)	9. (c)	10. (b)

Practice Questions

Q.I Answer the following questions in short.

1. Define term Exception.
2. List and explain any five exceptions in Python.
3. List out keywords used in exception handling.
4. How to handle multiple exceptions with single except clause?
5. How to create, raise and handle user defined exceptions in Python?

Q.II Answer the following questions.

1. Explain exception handling with example using try, except, raise keywords.
2. Explain try-except blocks for exception handling in Python.
3. How Python handles the exception? Explain with an example program.
4. Differentiate between an error and an exception.
5. Which are the different ways of creation of threads? Explain each with an example.
6. Comparison between else block and finally block in exception handling? Explain with an example program.

Q.III Write a short note on:

1. Exception handling
2. Multithreading
3. Threads
4. User defined exception
5. Process of handling exception.

Q.IV Write Programs:

1. Write a Python program which will throw exception if the value entered by user is less than zero.
2. Write a Python program to accept an integer number and use try/except to catch the exception if a floating point number is entered.
3. Write a Python program to implement exception handling with at least 2 to 3 exceptions.

■■■

6... Python File Operation

Objectives...

- To learn Python file handling operations such as reading config files and writing log files.
- To understand read functions and write functions.
- To know the manipulation of file pointer using seek method.
- To learn programming using file operations.

6.1 INTRODUCTION

- A file is a collection of related data that acts as a container of storage as data permanently. 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.
- Python supports two types of files: text files and binary files.
  1. **A text file** is a file that stores information in the term of a sequence of lines and a line is s sequence of characters (textual information). The line is terminated by EOL (End of Line) character. A text file is a file containing characters, structured as individual lines of text. In addition to printable characters, text files also contain the nonprinting newline character, \n, to denote the end of each text line. The newline character causes the screen cursor to move to the beginning of the next screen line. Thus, text files can be directly viewed and created using a text editor.

Common extensions for text file:

- **Source files:** c, cpp, py, java, php, sh, cs, js ...
- **Document file:** txt, ps, rtf ....
- **Web standards:** html, xml, css, json ...

2. **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. Such files can only be read and written via a computer program. 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.

   Common extensions for binary files:

   - **Image files:** jpg, png, gif, bmp, tiff, psd ...
   - **Videos:** mp4, mkv, avi, mov, mpeg ...
   - **Documents:** pdf, doc, xls, ppt, docx ...
   - **Audio:** mp3, aac, wav, wma ...
   - **Executables:** exe, dll, class ...

6.1.1 Different Modes of File Opening

- 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 the 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 an optional parameter and the default file access mode is read (r).
- **buffering:** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If 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 the current working directory, we can just provide the filename, just like in the following examples:

Example:

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


- If the file resides in a directory other than PWD, we have to provide the full path with the file name.

Example:

```
>>> 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.

Different Modes of Opening File:

- 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 the following table:

Table 6.1: Modes of Opening File

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
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.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
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.
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.
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.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

contd. ...

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.
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.
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.
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.
t	Opens in text mode (default).
b	Opens in binary mode.
+	Opens a file for updating (reading and writing).

Example:

```
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
```

6.1.2 Accessing File Contents using Standard Library Function

- Once a file is opened and you have one file object; you can get various information related to that file. Here is a list of all attributes related to file object:

Table 6.2: Attributes related to file object

Attribute	Description	Example

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

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

`file.name`	Returns name of the file.	`>>> f.name` `'abc.txt'`
`file.softspace`	Returns false if space explicitly required with print, true otherwise.	

Example:

```
>>> f=open("file.txt","r")
>>> print(f.name)
file.txt
>>> print(f.closed)
False
>>> print(f.mode)
r
>>> f.close()
>>> print(f.closed)
True
```

6.1.3 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 READING CONFIG FILES IN PYTHON

- Configuration files are well suited to specify configuration data to your program. Within each **config** file, values are grouped into different sections (e.g., "installation", "debug" and "server").
- Config files help creating the initial settings for any project, they help avoiding the hardcoded data.
- Use the **ConfigParser** module to manage user-editable configuration files for an application. The configuration files are organized into sections, and each section can contain name-value pairs for configuration data.
- The ConfigParser module from Python's standard library defines functionality for reading and writing configuration files as used by Microsoft Windows OS. Such files usually have .INI extension.

- **Creating config file in Python:** In Python, by using ConfigParser module we can create config files (.ini format).

```
from configparser import ConfigParser
#Get the configparser object
config_object = ConfigParser()
#Assume we need 2 sections in the config file, let's call them USERINFO
and SERVERCONFIG
config_object["USERINFO"] = {
    "admin": "Vijay Patil",
    "loginid": "vijaypatil",
    "password": "vijay123"
}
config_object["SERVERCONFIG"] = {
    "host": "google.com",
    "port": "8080",
    "ipaddr": "8.8.8.8"
}
#Write the above sections to config.ini file
with open('config.ini', 'w') as conf:
    config_object.write(conf)
```

The config.ini file was created and it looks like:

```
[USERINFO]
admin = Vijay Patil
loginid = vijaypatil
password = vijay123

[SERVERCONFIG]
host = google.com
port = 8080
ipaddr = 8.8.8.8
```

- We can read the configuration data so that you can use it by "keyname" to avoid hardcoded data:

Example:

```
from configparser import ConfigParser
#Read config.ini file
config_object = ConfigParser()
config_object.read("config.ini")
#Get the password
userinfo = config_object["USERINFO"]
print("Password is {}".format(userinfo["password"]))
```

Output:

```
Password is vijay123
```


6.3 WRITING LOG FILES IN PYTHON

- If you want to print python logs in a file rather than on the console then we can do so using the `basicConfig()` method by providing filename and `filemode` as a parameter.
- You can change the format of logs, log level or any other attribute of the `LogRecord` along with setting the filename to store logs in a file along with the mode. Logging is a means of tracking events that happen when some software runs.
- Logging is important for software developing, debugging and running. If you don't have any logging record and your program crashes, there are very little chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time. With logging, you can leave a trail of breadcrumbs so that if something goes wrong, we can determine the cause of the problem.

Built-in levels of the log message:

- **Debug:** These are used to give detailed information, typically of interest only when diagnosing problems.
- **Info:** These are used to confirm that things are working as expected.
- **Warning**: These are used as an indication that something unexpected happened, or indicative of some problem in the near future.
- **Error:** This tells that due to a more serious problem, the software has not been able to perform some function.
- **Critical:** This tells serious error, indicating that the program itself may be unable to continue running

Steps to create log files in Python:

1. First of all, simply import the logging module just by writing import logging.
2. The second step is to create and configure the logger. To configure logger to store logs in a file, it is mandatory to pass the name of the file in which you want to record the events.
3. In the third step, the format of the logger can also be set. Note that by default, the file works in append mode but we can change that to write mode if required.
4. You can also set the level of the logger.

Following are creating a log file name: std.log with some message

```
#importing module
import logging
 #Create and configure logger
logging.basicConfig(filename="std.log",
                    format='%(asctime)s %(message)s',
                    filemode='w')
 #Creating an object
logger=logging.getLogger()
```

```
 #Setting the threshold of logger to DEBUG
logger.setLevel(logging.DEBUG)
 #Test messages
logger.debug("This is debug Message")
logger.info("Just an information Message")
logger.warning("It is a Warning Message")
logger.error("This is an Error Message")
logger.critical("Critical Message: Internet is down")
```

Output: If we will open the file:std.log, then the messages will be written as follows:

```
2021-03-17 19:20:19,484 This is debug Message
2021-03-17 19:20:19,485 Just an information Message
2021-03-17 19:20:19,485 It is a Warning Message
2021-03-17 19:20:19,486 This is an Error Message
2021-03-17 19:20:19,486 Critical Message: Internet is down
```

6.4 UNDERSTANDING read FUNCTIONS, read(), readline() AND readlines()

- 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 the following three methods available for reading data purpose:

1. read([n]) Method:

- The `read()` method just outputs the entire file if the 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.

Syntax: `fileObject.read(size);`

Parameters:

size: This is the number of bytes to be read from the file.

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 an empty string on further reading.

Syntax: `file Object.readline(size)`

Parameter:

size: (optional) Here, you can specify the number, an integer value to readline(). It will get the string of that size. By default, the value of size is -1, and hence the entire string is returned.

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. The method readlines() reads until EOF using readline() and returns a list containing the lines. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totaling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read. An empty string is returned only when EOF is encountered immediately.

Syntax: `fileObject.readlines(sizehint);`

Parameter:

sizehint: This is the number of bytes to be read from the file.

Example: For readlines() method.

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

Output:

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

6.4.1 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: `raw_input()` and `input()`.

1. **raw_input():** This function reads only one line from the standard input and returns it as a String.

   [**Note:** This function is withdrawn in Python 3.]

2. **input():** The input function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

   Syntax: input([prompt])

   Parameter:

   Prompt: The prompt string is printed on the console and the control is given to the user to enter the value.

   Example:

```
str=input("Enter any Number:")
print("Number you entered are: ",str)
```

   Output:

```
Enter any Number:10
Number you entered are:  10
```

6.5 UNDERSTANDING WRITE

- Python file method `write()` writes a string *str* to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the `flush()` or `close()` method is called.

Syntax: `file Object.write(str)`

Parameter:

str: The string to be written in the file.

Example 1:

```
i=open("demo.txt","w")
i.write("Hello Python")
```

Example 2: Program to copy content of one file into another file.

```
with open('first.txt','r') as firstfile, open('second.txt','a') as
secondfile:

    # read content from first file
    for line in firstfile:

            # append content to second file
            secondfile.write(line)
```


Table 6.3: Python write file Methods

Method	Description
`write(s)`	To write string *s* into a file and return the number of characters written.
`writelines(lines)`	To write a list of lines into a file.
`writable()`	It returns true if the file stream can be written to.

6.6 FUNCTIONS write() AND writelines()

- 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:
  1. write(string) (for text)
  2. write(byte_string) (for binary)
  3. writelines (list).

6.6.1 write(string) Method

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

Example:

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

Output:

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

6.6.2 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
```

6.7 MANIPULATING FILE POINTER USING SEEK

- 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()` is used to move the file pointer to the particular position.

Example 1: For file position.

```
f=open("sample.txt","r")
print(f.tell())
print(f.read())
print(f.tell())
print(f.read()) # print blank line
print(f.seek(0))
print(f.read())
```

Output:

```
0
first line
second line
third line
37
0
first line
second line
third line
>>>
```

Example 2:

```
fruits=["Orange\n","Banana\n","Apple\n"]
f=open("sample.txt",mode="w+",encoding="utf-8")
for fru in fruits:
   f.write(fru)
print("Tell the byte at which the file cursor is:",f.tell())
f.seek(0)
for line in f:
   print(line)
```

Output:

```
Tell the byte at which the file cursor is: 23
Orange
Banana
Apple
```

6.8 PROGRAMMING USING FILE OPERATIONS

Program 1: Program to create a simple file and write some content in it.

```
print("Enter 'x' for exit.");
filename = input("Enter file name to create and write content: ");
if filename == 'x':
   exit();
else:
   c = open(filename, "w");
print("\n The file,",filename," created successfully!");
print("Enter sentences to write on the file: ");
sent1 = input();
c.write(sent1);
c.close();
print("\n Content successfully placed inside the file!!");
```

Output:

```
Enter 'x' for exit.
Enter file name to create and write content: file1
The file, file1 created successfully!
Enter sentences to write on the file:
Good morning
Content successfully placed inside the file!!
```

Program 2: Program to open a file in write mode and append some content at the end of 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("\n Content appended to file!!");
```


Output:

```
Enter 'x' for exit.
Enter file name to append content: file1
Enter sentences to append on the file:
Good afternoon
Content appended to file!!
```

Program 3: Program to open a file in read mode and print number of occurrences of characters 'a'.

```
fname = input("Enter file name: ")
l=input("Enter letter to be searched:")
k = 0
with open(fname, 'r') as f:
for line in f:
   words = line.split()
   for i in words:
      for letter in i:
         if(letter==l):
               k=k+1
print("Occurrences of the letter:")
print(k)
```

Output:

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

Summary

- A file is a collection of related data that acts as a container of storage as data permanently.
- Python supports two basic file types, namely text files and binary files.
- Before you can read or write a file, you have to open it using Python's built-in open() function.
- The close() method of a file object flushes any unwritten information and closes the file object, once the file is closed no more writing can be done.
- The file object provides a set of access methods to make use of files easier to access. read() and
  write() methods to read and write files.
- The read(), readline(), and readlines() methods are used to read data from a file.

- readline( ) includes the newline character at the end of the string it returns. To read the contents of the file one line at a time, call readline( ) method repeatedly.
- read( ) method can also read the rest of the file all at once. This method returns any text in the file that you have not read yet.
- readline( ) method returns as a string the next line of a text file, including the end-of-line character, \n.
- The write() and writes() methods are used to write data to a file.
- The write( ) method writes any string to an open file.

Check Your Understanding

1. Which of the following command is used to open a file "c:\temp.txt" in read-mode only?
   (a) infile = open("c:\temp.txt", "r")
   (b) infile = open("c:\\temp.txt", "r")
   (c) infile = open(file = "c:\temp.txt", "r+")
   (d) infile = open(file = "c:\\temp.txt", "r+")
2. Which of the following commands can be used to read "n" number of characters from a file using the file object <file>?
   (a) file.read(n) (b) n = file.read()
   (c) file.readline(n) (d) file.readlines()
3. Which of the following commands can be used to read the entire contents of a file as a string using the file object <tmpfile>?
   (a) tmpfile.read(n) (b) tmpfile.read()
   (c) tmpfile.readline() (d) tmpfile.readlines()
4. Which of the following commands can be used to read the next line in a file using the file object <tmpfile>?
   (a) tmpfile.read(n) (b) tmpfile.read()
   (c) tmpfile.readline() (d) tmpfile.readlines()
5. Which of the following commands can be used to read the remaining lines in a file using the file object <tmpfile>?
   (a) tmpfile.read(n) (b) tmpfile.read()
   (c) tmpfile.readline() (d) tmpfile.readlines()
6. Which of the following functions do you use to write data in the binary format?
   (a) write (b) output
   (c) dump (d) send
7. What are the two built-in functions to read a line of text from standard input, which is by default the keyboard?
   (a) Raw_input (b) Input
   (c) Read (d) Scanner

8. To read two characters from a file object infile, we use _____.
   (a) infile.read(2) (b) infile.read()
   (c) infile.readline() (d) infile.readlin
9. To open a file c:\scores.txt for appending data, we use _____.
   (a) outfile = open("c:\\scores.txt", "a")
   (b) outfile = open("c:\\scores.txt", "rw")
   (c) outfile = open(file = "c:\scores.txt", "w")
   (d) outfile = open(file = "c:\\scores.txt", "w")
10. What is the correct syntax of open() function?
    (a) file = open(file_name [, access_mode][, buffering])
    (b) file object = open(file_name [, access_mode][, buffering])
    (c) file object = open(file_name)
    (d) None of the above

Answers

1. (b)	2. (a)	3. (b)	4. (c)	5. (d)	6. (c)	7. (a)	8. (a)	9. (a)	10. (b)

Practice Questions

Q.I Answer the following questions in short.

1. Define file and explain the different types of files.
2. Write the symbols used in text file mode for the following operations: Read Only, Write only, Read and Write, Write and Read.
3. What is the difference between readline() and readlines().
4. Write the syntax of fopen()?
5. What are various modes of file object? Explain any five modes.

Q.II Answer the following questions.

1. Explain the different file mode operations with examples.
2. Describe with an example how to read and write to a text file.
3. Explain open() and close() methods for opening and closing a file.
4. Explain any 3 methods associated with files in Python.
5. Explain how to create log files in Python.

Q.III Write a short note on :

1. Text file
2. Binary file
3. Write() function
4. Config files
5. Built-in levels of the log message

Q.IV Write Programs.

1. Write a Python program to get the file size of a plain text file.
2. Write a Python program to read first 10 characters from a file named "data.txt".
3. Write a Python program to read entire content from the file named "test.txt".
4. Write a Python program to read contents of "first.txt" file and write same content in "second.txt" file.
5. 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.
6. Write a Python program to read a text file and print number of lines, words and characters.
7. Write a Python program that reads a text file and changes the file by capitalizing each character of file.

■■■

7... Python Database Interaction

Objectives...

- To learn about NoSQL Databases.
- To introduce of MongoDB with Python.
- To know about Collections and Documents of MongoDB.
- To study basic CRUD operations with MongoDB.

7.1 INTRODUCTION TO NoSQL

- **NoSQL** Database is a non-relational Data Management System that does not require a fixed schema. It avoids joins and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with enormous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.
- **NoSQL database** stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozzi introduced the NoSQL concept in 1998.
- Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data.

7.1.1 Why NoSQL?

- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.
- To resolve this problem, we could "scale up" our systems by upgrading our existing hardware. This process is expensive.
- The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as "scaling out."
- NoSQL database is non-relational, so it scales out better than relational databases as they are designed with web applications in mind.

7.1.2 Brief History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database.
- 2000- Graph database Neo4j is launched.
- 2004- Google BigTable is launched.
- 2005- CouchDB is launched.
- 2007- The research paper on Amazon Dynamo is released.
- 2008- Facebooks open sources the Cassandra project.
- 2009- The term NoSQL was reintroduced.

7.1.3 Features of NoSQL

1. Non-relational:

- NoSQL databases never follow the relational model.
- Never provide tables with flat fixed-column records.
- Work with self-contained aggregates or BLOBs.
- Doesn't require object-relational mapping and data normalization.
- No complex features like query languages, query planners, referential integrity joins, ACID.

2. Schema-free:

- NoSQL databases are either schema-free or have relaxed schemas.
- Do not require any sort of definition of the schema of the data.
- Offers heterogeneous structures of data in the same domain.

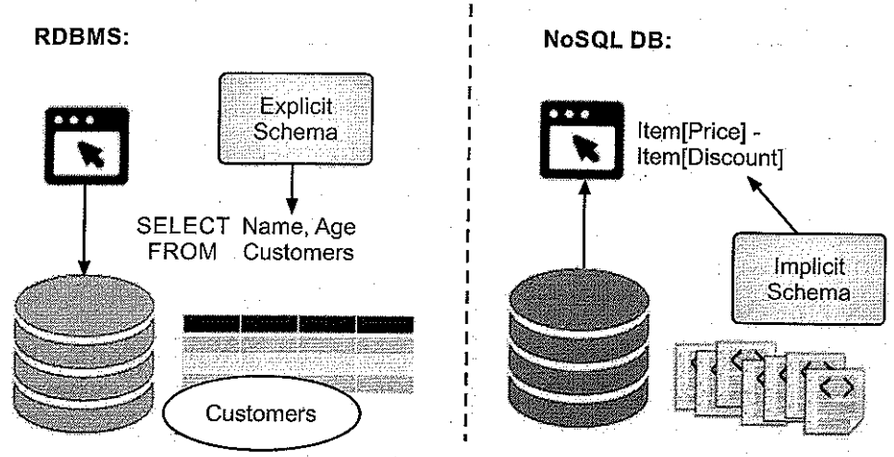


Fig. 7.1: Schema-Free NoSQL

3. Simple API:

- Offers easy to use interfaces for storage and querying data provided.
- APIs allow low-level data manipulation & selection methods.
- Text-based protocols are mostly used with HTTP REST with JSON.
- Mostly used no standard based NoSQL query language.
- Web-enabled databases running as internet-facing services.

4. Distributed:

- Multiple NoSQL databases can be executed in a distributed fashion.
- Offers auto-scaling and fail-over capabilities.
- Often ACID concept can be sacrificed for scalability and throughput.
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication.
- Only providing eventual consistency.
- Shared Nothing Architecture. This enables less coordination and higher distribution.

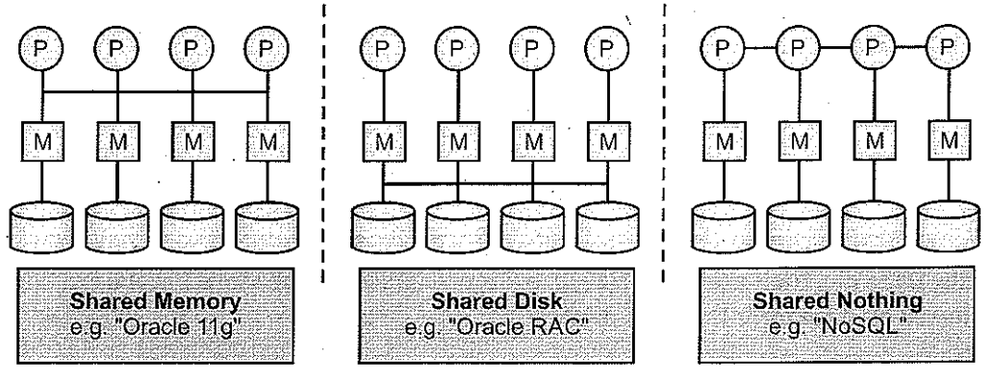


Fig. 7.2: NoSQL -- Shared Nothing Architecture

7.1.4 Types of NoSQL Databases

- NoSQL Databases are mainly categorized into four types:
  1. Key-value Pair Based
  2. Column-oriented Graph
  3. Graphs based
  4. Document-oriented
- Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

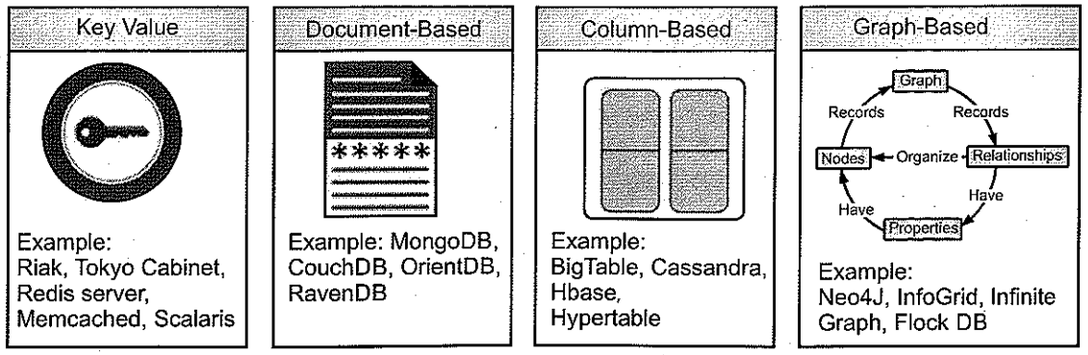


Fig. 7.3: Types of NoSQL Databases

1. Key-Value Pair Based:

- Data is stored in key-value pairs. It is designed in such a way to handle lots of data and heavy load.
- Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB (Binary Large Objects), string, etc.

Table 7.1: Key-value Pairs

Key	Value
Name	Rahul Patil
Age	20
Occupation	Manager
Height	200 cm

- It is one of the most basic NoSQL database examples. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key-value stores help the developer to store schema-less data. They work best for shopping cart contents.
- Redis, Dynamo, Riak are some NoSQL examples of key-value store Databases. They are all based on Amazon's Dynamo paper.

2. Column-based:

- Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.
- They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.
- Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs.
- HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column-based databases.

3. Document-oriented:

- Document-oriented NoSQL DB stores and retrieves data as a key-value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.

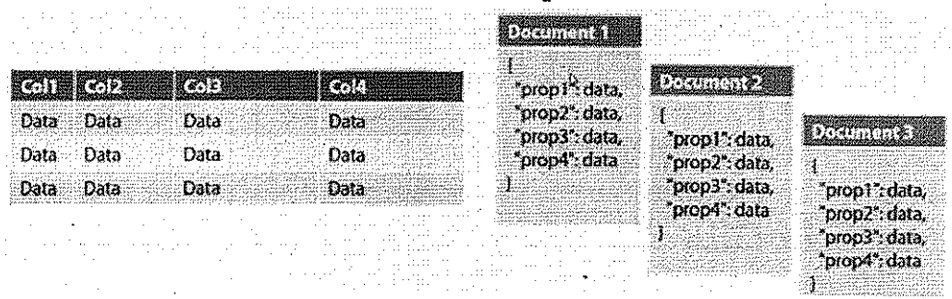


Fig. 7.4: Relational database vs. Document-oriented database

- In this diagram, on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which makes it flexible.
- The document type is mostly used for CMS systems, blogging platforms, real-time analytics and E-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.
- Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

4. Graph-based:

- A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.

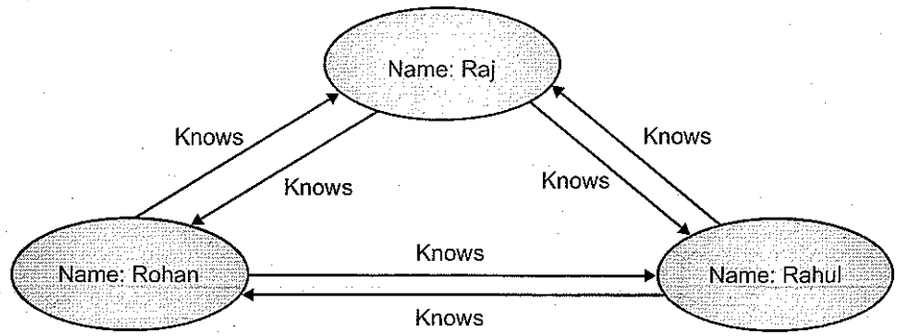


Fig. 7.5: Graph-based Database

- Compared to a relational database where tables are loosely connected, a Graph-based database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.
- Graph-based database mostly used for social networks, logistics, spatial data.
- Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

7.2 ADVANTAGES OF NoSQL

- **Scalability**: NoSQL database can be scaled up easily and with minimum effort and hence it's well suited for today's every increasing database need. NoSQL database have scalable architecture so it can efficiently manage data and can scale up to many machines instead of costly machines that are required while scaling using SQL DBMS.
- With dynamic schema, if we want to change the length of the column, or add new column we don't need to change the whole table data instead the new data will be stored with the new structure without affecting the previous data/structure. In NoSQL database, we can insert data without predefined schema.

- **Replication** provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects the database from the loss of single server.
- To use replication with sharding, deploy each shared as a replica set.
- **Sharding** is the process of storing data records across multiple machines and is MongoDB's approach to meeting the demand of data growth.
- Many NoSQL databases have integrated caching mechanisms; hence frequently used data are stored in system memory as much as possible and discarding the need for a separate caching layer.

7.3 SQL vs. NoSQL

Table 7.2: Difference between SQL and NoSQL Databases

Parameters	SQL	NoSQL
Type	Relational	Non-Relational
Data	Structured data stored in tables	Un-structured stored in JSON files but the graph database does supports relationship.
Schema	Static	Dynamic
Scalability	Vertical	Horizontal
Language	Structured Query Language	Un-Structured Query Language
Joins	Helpful to design complex queries.	No joins, don't have the powerful interface to prepare complex query.
OLTP	Recommended and best suited for OLTP system.	Less likely to be considered for OLTP system.
Support	Great support	Community dependent, they are expanding the support model.
Integrated caching	Supports in-line memory	Supports integrated caching.
Flexibility	Rigid schema bound to relationship	Non-rigid schema and flexible.
Transaction	ACID	CAP theorem
Auto Elasticity	Requires downtime in most cases	Automatic, No outage requires.

- When choosing a database, you should consider its strengths and weaknesses carefully. You also need to consider how the database fits into your specific scenario and your application's requirements. Sometimes the right solution is to use a combination of SQL and NoSQL databases to handle different aspects of a broader system.

Some common examples of SQL databases include:

- SQLite
- MySQL
- Oracle
- PostgreSQL
- Microsoft SQL Server

NoSQL database examples include:

- DynamoDB
- Cassandra
- Redis
- CouchDB
- RethinkDB
- RavenDB
- MongoDB

- In recent years, SQL and NoSQL databases have even begun to merge. For example, database systems, such as PostgreSQL, MySQL, and Microsoft SQL Server now support storing and querying JSON data, much like NoSQL databases. With this, you can now achieve many of the same results with both technologies. But you still don't get many of the NoSQL features, such as horizontal scaling and the user-friendly interface.

7.4 INTRODUCTION TO MongoDB WITH PYTHON

- MongoDB is a document-oriented database classified as NoSQL. It is become popular throughout the industry in recent years and integrates extremely well with Python. Unlike traditional SQL RDBMSs, MongoDB uses collections of documents instead of tables of rows to organize and store data.
- MongoDB stores data in schemaless and flexible JSON-like documents. Here, schemaless means that you can have documents with a different set of fields in the same collection, without the need for satisfying a rigid table schema.
- You can change the structure of your documents and data over time, which results in a flexible system that allows you to quickly adapt to requirement changes without the need for a complex process of data migration. However, the trade-off in changing the structure of new documents is that exiting documents become inconsistent with the updated schema. So, this is a topic that needs to be managed with care.
- MongoDB is written in C++ and actively developed by MongoDB Inc. It runs on all major platforms, such as MAC OS, Windows, Solaris, and most Linux distributions. In general, there are three main development goals behind the MongoDB database:
  1. Scale well.
  2. Store rich data structures.
  3. Provide a sophisticated query mechanism.
- MongoDB is a distributed database, so high availability, horizontal scaling, and geographic distribution are built into the system. It stores data in flexible JSON-like documents. You can model these documents to map the objects in your applications, which makes it possible to work with your data effectively.

- MongoDB provides a powerful query language that supports ad hoc queries, indexing, aggregation, geospatial search, text search, and a lot more. This presents you with a powerful tool kit to access and work with your data. Finally, MongoDB is freely available and has great Python support.

7.4.1 Features of MongoDB

- So far, you've learned what MongoDB is and what its main goals are. In this section, you'll learn about some of MongoDB's more important features. As for the database management side, MongoDB offers the following features:
  - **Query support:** You can use many standard query types, such as matching (==), comparison (<, >), and regular expressions.
  - **Data accommodation:** You can store virtually any kind of data, be it structured, partially structured, or even polymorphic.
  - **Scalability:** It handles more queries just by adding more machines to the server cluster.
  - **Flexibility and agility:** You can develop applications with it quickly.
  - **Document orientation and schemalessness:** You can store all the information regarding a data model in a single document.
  - **Adjustable schema:** You can change the schema of the database on the fly, which reduces the time needed to provide new features or fix existing problems.
  - **Relational database functionalities:** You can perform actions common to relational databases, like indexing.
- As for the operations side, MongoDB provides a few tools and features that you won't find in other database systems:
  - **Scalability:** Whether you need a stand-alone server or complete clusters of independent servers, you can scale MongoDB to whatever size you need it to be.
  - **Load-balancing support:** MongoDB will automatically move data across various shards.
  - **Automatic failover support:** If your primary server goes down, then a new primary will be up and running automatically.
  - **Management tools:** You can track your machines using the cloud-based MongoDB Management Service (MMS).
  - **Memory efficiency:** Thanks to the memory-mapped files, MongoDB is often more efficient than relational databases.

7.4.2 PyMongo

- PyMongo is a Python distribution which provides tools to work with MongoDB; it is the most preferred way to communicate with MongoDB database from Python.
- Python needs a MongoDB driver to access the MongoDB database.
- In this topic we will use the MongoDB driver "PyMongo".
- We recommend that you use PIP to install "PyMongo".
- PIP is most likely already installed in your Python environment.

7.4.2.1 Installation

- To install **PyMongo,** first of all make sure you have installed python3 (along with PIP) and MongoDB properly. Then execute the following command:

```
pip install pymongo
```

- After installation successful, we now verify whether installation is successful for that we open the new document and write the following code and save as 'firstprogram.py'.

```
import pymongo
```

- If you have installed PyMongo properly, if you execute the firstprogram.py as shown below, you should not get any issues.

```
C:\>Users\acer\Desktop\firstprogram.py
C:\>_
```

7.5 EXPLORING COLLECTIONS AND DOCUMENTS

- MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections. A database stores one or more collections of documents.
- A MongoDB database is a physical container for collections of documents. Each database gets its own set of files on the file system. These files are managed by the MongoDB server, which can handle several databases.

7.5.1 Databases

- To create a database in MongoDB, start by creating a **MongoClient** object then specify a connection URL with the correct IP address and the name of the database you want to create.
- MongoDB will create the database if it does not exist, and make a connection to it.

Example: In the following example, we will create the database called "mydb"

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = client["mydb"]
```

- In MongoDB, a database is not created until it gets content.
- MongoDB waits until you have created a collection (table), with at least one document (record) before it creates the database (and collection).

Check if Database Exists:

- In MongoDB, a database is not created until it gets content. So if we are creating a database first time, we have to first create collection and documents before we check if the database actually exists.
- After creating collections and documents we can check whether the database is already exist, by doing the following operation:
  - **By listing the all the database in the system:**

```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017/')
print(client.list_database_names())
```

When we execute the above code we will get the list of databases which are available like,

```
C:\>Users\acer\Desktop\firstprogram.py
['admin','config','local','mydatabase',',\mydb']
C:\>_
```

- **Or we can check by specific database name such as:**

```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017/')
dblist = client.list_database_names()
if "mydb" in dblist:
print("The database exists.")
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
The database exists.
C:\>_
```

7.5.2 Collections

- A collection in MongoDB is the same as a table in SQL databases.

Creating Collections:

- To create a collection in MongoDB, use database object and specify the name of the collection you want to create.
- MongoDB will create the collection if it does not exist.

Example: In the following example we will create the collection called "employee"

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
```

- In MongoDB, collection is not created until it gets content.
- MongoDB waits until you have inserted a document before it actually creates the collection.

Check if Collection Exists:

- In MongoDB, a collection is not created until it gets content, so if this is your first time creating a collection, we have to first create documents before we check if the collection exists.
- After creating documents we can check whether the collection is already exist, by doing the following operation:
  - **By listing the all the collections in the system:**

```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017/')
db = client['mydb']
mycol = db["employee"]
print(db.list_collection_names())
```


Output:

```
C:\>Users\acer\Desktop\firstprogram.py
['example', 'employee']
C:\>_
```

- **Or we can check by specific collection name such as:**

```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017/')
db = client['mydb']
collist = db.list_collection_names()
if "employee" in collist:
print("The collection exists.")
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
The collection exists.
C:\>_
```

7.5.3 Document

- MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON.

Document Structure:

- MongoDB documents are composed of field-and-value pairs and have the following structure:

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

7.6 PERFORMING BASIC CRUD OPERATIONS WITH MongoDB AND PYTHON

- CRUD Operations are:
  1. C-Create
  2. R-Read
  3. U-Update
  4. D-Delete
- Now we will go through each operation one by one.

7.6.1 C-Create

- Mongo store the data in the form of JSON objects. So every record for a collection in mongo is called a **document**. If the collection does not currently exist, insert operations will create the collection. We can insert the documents into collection in 2 ways:
  1. `insert_one()`
  2. `insert_many()`

1. insert_one():

- To insert a record, or document as it is called in MongoDB, into a collection, we use the `insert_one()` method.
- The first parameter of the `insert_one()` method is a dictionary containing the name(s) and value(s) of each field in the document you want to insert.

Example: In the following example, we insert record in "employee" collection.

```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017/')
db = client['mydb']
mycol = db["employee"]
mydict = { "name": "Rahul", "address": "Mumbai" }
x = mycol.insert_one(mydict)
print(x)
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
<mymongo.results.InsertOneResult object at 0x000002BD08F6CCC0>
C:\>_
```

- The `insert_one()` method returns a `InsertOneResult` object, which has a property, `inserted_id`, that holds the id of the inserted document.

2. insert_many() method:

- To insert multiple documents into a collection in MongoDB, we use the `insert_many()` method.
- The first parameter of the `insert_many()` method is a list containing dictionaries with the data you want to insert:

Example:

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
mylist = [
  { "name": "Rahul", "address": "Mumbai"},
  { "name": "Meena", "address": "Pune"},
  { "name": "Raj", "address": "Delhi"},
  { "name": "Siya", "address": "Nagpur"}
  ]
x = mycol.insert_many(mylist)
#print list of the _id values of the inserted documents:
print(x.inserted_ids)
```


Output:

```
C:\>Users\acer\Desktop\firstprogram.py
[ObjectId('605485a3d1a59c082be40db3'),
ObjectId('605485a3d1a59c082be40db4'),
ObjectId('605485a3d1a59c082be40db5'),
ObjectId('605485a3d1a59c082be40db6')]
C:\>_
```

- The `insert_many()` method returns a `InsertManyResult` object which has a property, `inserted_ids`, that holds the ids of the inserted documents.

7.6.2 R-Read

- We can retrieve the documents from a collection using 2 methods.
  1. `find()`
  2. `find_one()`

1. find():

- To select data from a table in MongoDB, we can also use the `find()` method.
- The `find()` method returns all occurrences in the selection.
- The first parameter of the `find()` method is a query object. In this example, we use an empty query object, which selects all documents in the collection.

Example: In following example it returns all the documents in the "employee" collection:

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
for x in mycol.find():
print(x)
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
{'_id':ObjectId('605485a3d1a59c082be40db3'),'name':'Rahul','address':'Mumbai'}
{'_id':ObjectId('605485a3d1a59c082be40db4'),'name':'Meena','address':'Pune'}
{'_id':ObjectId('605485a3d1a59c082be40db5'),'name':'Raj','address':'Delhi'}
{'_id':ObjectId('605485a3d1a59c082be40db6'),'name':'Siya','address':'Nagpur'}
C:\>_
```

2. find_one():

- To select data from a collection in MongoDB, we can use the `find_one()` method.
- The `find_one()` method returns the first occurrence in the selection.

Example: In following example, we get the first document in the employee collection:

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
for x in mycol.find():
print(x)
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
{'_id': ObjectId('605485a3d1a59c082be40db3'),'name':'Rahul','address':'Mumbai'}
C:\>_
```

7.6.3 U-Update()

- We can update the documents from the collection with the following methods:
  1. update_one()
  2. update_many()

1. update_one():

- You can update a record, or document as it is called in MongoDB, by using the update_one() method.
- The first parameter of the update_one() method is a query object defining which document to update.
- If the query finds more than one record, only the first occurrence is updated.
- The second parameter is an object defining the new values of the document.

Example: In the following example we change the address from *Mumbai* to *Navi Mumbai*.

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
myquery = { "address": "Mumbai" }
newvalues = { "$set": { "address": "Navi Mumbai" } }
mycol.update_one(myquery, newvalues)
#print "employee" after the update:
for x in mycol.find():
print(x)
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
{'_id':ObjectId('605485a3d1a59c082be40db3'),'name':'Rahul','address':'
                                                        Navi Mumbai'}
{'_id':ObjectId('605485a3d1a59c082be40db4'),'name':'Meena','address':'Pune'}
{'_id':ObjectId('605485a3d1a59c082be40db5'),'name':'Raj','address':'Delhi'}
{'_id':ObjectId('605485a3d1a59c082be40db6'),'name':'Siya','address':'Nagpur'}
C:\>_
```


2. update_many():

- To update *all* documents that meets the criteria of the query, use the update_many() method.

Example: In the following example, we update all the documents where address starts with the letter 'N'.

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
myquery = { "address": { "$regex": "^N" } }
newvalues = { "$set": { "name": "Manisha" } }
x = mycol.update_many(myquery, newvalues)
print(x.modified_count, "documents updated.")
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
2 documents updated.
C:\>_
```

7.6.4 D-Delete

- We can delete the documents in the collection using the following methods:
  1. delete_one()
  2. delete_many()

1. delete_one():

- To delete one document, we use the delete_one() method.
- The first parameter of the delete_one() method is a query object defining which document to delete.
- If the query finds the more than one document, only first occurrence is deleted.

Example: In the following example, we delete the document with the address "Nagpur".

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
myquery = { "address": "Nagpur" }
mycol.delete_one(myquery)
for x in mycol.find():
print(x)
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
C:\>Users\acer\Desktop\firstprogram.py
{'_id':ObjectId('605485a3d1a59c082be40db3'),'name':'Manisha','address':'
                                                        Navi Mumbai'}
{'_id':ObjectId('605485a3d1a59c082be40db4'),'name':'Meena',
                                                    'address':'Pune'}
{'_id':ObjectId('605485a3d1a59c082be40db5'),'name':'Raj',
                                                    'address':'Delhi'}

C:\>_
```

2. delete_many():

- To delete more than one document, use the `delete_many()` method.
- The first parameter of the `delete_many()` method is a query object defining which documents to delete.

Example: In the following example, we delete all the documents where address starts with the letter "N".

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
mycol = db["employee"]
myquery = { "address": {"$regex": "^N"} }
x = mycol.delete_many(myquery)
print(x.deleted_count, " documents deleted.")
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
1 documents updated.
C:\>_
```

3. Delete all the documents in the collection:

- To delete all documents in a collection, pass an empty query object to the `delete_many()` method:

Example:

```
import pymongo
client = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = client["mydb"]
mycol = mydb["employee"]
x = mycol.delete_many({})
print(x.deleted_count, " documents deleted.")
```

Output:

```
C:\>Users\acer\Desktop\firstprogram.py
2 documents deleted.
C:\>_
```

Summary

- NoSQL is a non-relational DBMS that does not require a fixed schema, avoids joins, and is easy to scale.
- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data.
- In the year 1998 Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database.

- NoSQL databases never follow the relational model it is either schema-free or has relaxed schemas.
- Four types of NoSQL Database are: 1. Key-value Pair Based 2. Column-oriented Graph 3. Graph- based 4. Document-oriented.
- Data plays a crucial role in application development. It plays such a crucial role that there are people called Database Administrators that specialize in just organizing the data.
- The traditional database used to be, and often still is, a relational database.
- MongoDB is more flexible than a relational database, and as such provides the developer with a database that can easily be changed during the development phase.
- The increased popularity of MongoDB combined with the popularity of Python has resulted in a Python library called PyMongo, which provides an API for communicating with MongoDB.

Check Your Understanding

1. _____ is not the NoSQL database.
   (a) SQL Server (b) MongoDB
   (c) Cassandra (d) None of the mentioned
2. Point out the correct statement.
   (a) Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
   (b) MongoDB has official drivers for a variety of popular programming languages and development environments.
   (c) When compared to relational databases, NoSQL databases are more scalable and provide superior performance.
   (d) All of the mentioned
3. Which of the following are the simplest NoSQL databases?
   (a) Key-value (b) Wide-column
   (c) Document (d) All of the mentioned
4. MongoDB stores all documents in _____.
   (a) tables (b) collections
   (c) rows (d) all of the mentioned
5. Collection is a group of MongoDB _____.
   (a) Database (b) Document
   (c) Field (d) None of the above
6. A collection and a document in MongoDB is equivalent to which of the SQL concepts respectively?
   (a) Table and Row (b) Table and Column
   (c) Column and Row (d) Database and Table

7. Types of NoSQL databases _____.
   (a) Document Databases (b) Kay-Value Stores
   (c) Graph oriented databases (d) All the above
8. NoSQL can be referred to as _____.
   (a) No SQL (b) Not Only SQL
   (c) Only SQL (d) None of the above
9. Which of the following format supported by MongoDB?
   (a) SQL (b) BSON
   (c) XML (d) All of the above
10. Which of the following companies developed NoSQL database Apache Cassandra?
   (a) Facebook (b) Twitter
   (c) LinkedIn (d) MySpace

Answers

1. (a)	2. (d)	3. (a)	4. (b)	5. (b)	6. (a)	7. (d)	8. (b)	9. (b)	10. (a)

Practice Questions

Q.I Answer the following questions in short.

1. Which are the types of NoSQL databases?
2. Write examples of key-value store Databases.
3. Name the fields where Column-based NoSQL databases are widely used.
4. Which are popular graph-based databases?
5. What is a Sharding?

Q.II Answer the following questions.

1. Explain NoSQL database.
2. List and explain the advantages of NoSQL database.
3. Differentiate between SQL and NoSQL.
4. Explain the features of MongoDB.
5. Explain installation steps of PyMongo.
6. Explain Collection, database and documents in PyMongo.
7. Explain the CRUD operations of MongoDB with Python.

Q.III Write a short note on:

1. Graph-based databases
2. Column-based NoSQL databases
3. Features of MongoDB
4. C-Create CRUD operation
5. Types of NoSQL Databases

■■■

8... Python for Data Analysis

Objectives...

- To learn the data analysis concepts and data visualization.
- To understand the NumPy Python library.
- To study data analysis tools in Pandas library.
- To get information about Matplotlib Python package.

8.1 NumPy

- NumPy means Numerical Python consists of many Python libraries used for scientific computing in Python.
- NumPy is a general-purpose array-processing package. It provides a high-performance multi-dimensional array object and tools for working with these arrays.
- NumPy defines a specific data structure that is an N-dimensional array defined as **ndarray**.
- Python NumPy arrays provide tools for integrating C, C++, etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multidimensional container for generic data.

Installation of NumPy:

- To install Python **NumPy** on Windows-10; go to your command prompt and type "`pip install numpy`". Once the installation is completed, go to your IDE and simply import it by typing: "`import numpy as np`".

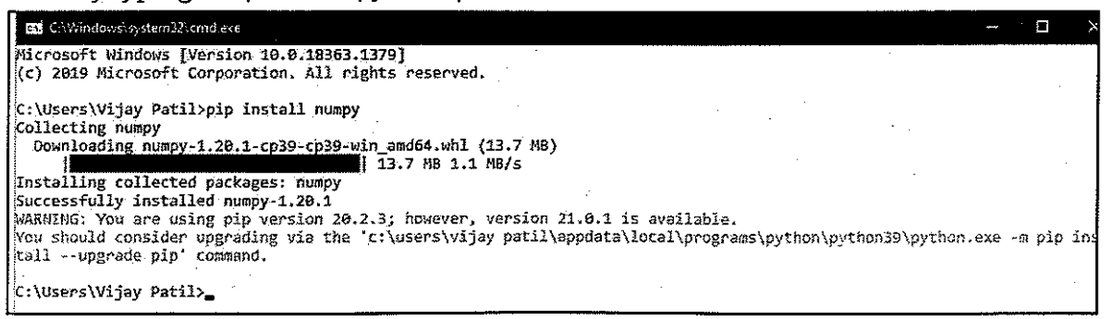


Fig. 8.1: Installation of NumPy

- To install NumPy on Ubuntu, use apt utility as:

```
sudo apt install python-pip
```

- If you need Pip for Python 3, use the command:

```
sudo apt install python3-pip
```

To verify Installation:

- Use show command to verify whether NumPy is the part of python package:

```
pip show numpy OR pip3 show numpy
```

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.18363.1379]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Vijay Patil>pip show numpy
Name: numpy
Version: 1.20.1
Summary: NumPy is the fundamental package for array computing with Python.
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD
Location: c:\users\vijay patil\appdata\local\programs\python\python39\lib\site-packages
Requires:
Required-by:

C:\Users\Vijay Patil>
```

Fig. 8.2: Verify installation

Import NumPy package:

- After installing NumPy you can import the package and set an alias for it.

```
import numpy as np
```

To upgrade NumPy:

- To upgrade latest version of NumPy using pip command as:

```
pip install -upgrade numpy OR pip3 install -upgrade numpy
```

8.2 INTRODUCTION TO 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 as 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 is 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. 8.3 shows two examples of arrays with its axis (or dimensions) and a length: (a) a one-dimensional array and (b) a two-dimensional array.

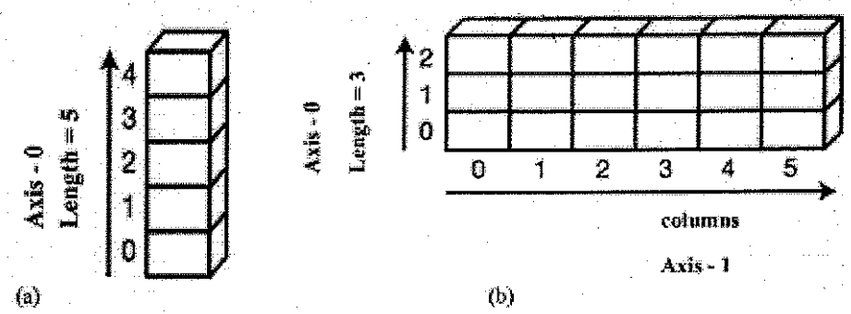


Fig. 8.3: Dimensions of NumPy Array

- 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 is three and Axis-1 has six elements, so its length is six.
- Execute the 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.

8.3 CREATING ARRAYS, USING ARRAYS AND SCALARS

8.3.1 Creating Arrays

- You can create a NumPy array from a regular Python list or tuple using the `np.array()` function.
- Arrays in NumPy can be created by multiple ways, with various numbers of Ranks, defining the size of the Array.

- Arrays can also be created with the use of various data types such as lists, tuples, etc.

1. One Dimensional Array:

```
>>> import numpy as np
>>> a=np.array([10,20,30])
>>> print(a)

[10 20 30]
>>> type(a)
<class 'numpy.ndarray'>
```

2. Two Dimensional Array:

```
>>> import numpy as np
>>> a=np.array([(10,20,30),(40,50,60)])
>>> print(a)

[[10 20 30]
 [40 50 60]]
>>>
```

8.3.2 NumPy Array Creation Functions

- NumPy provides several built-in functions to create and work with arrays from scratch.

Table 8.1: Built-in functions of NumPy

Function	Description	Example

`np.zeros()`	Creates an array of zeros.	`>>> arr=np.zeros([2,2],dtype=int)`
		`>>> print(arr)`
		`[[0 0]`
		` [0 0]]`

`np.ones()`	Creates an array of ones.	`>>> arr=np.ones([2,2],dtype=int)`
		`>>> print(arr)`
		`[[1 1]`
		` [1 1]]`

`np.empty()`	Creates an empty array.	`>>> arr=np.empty([2,2],dtype=int)`
		`>>> print(arr)`
		`[[0 0]`
		` [0 0]]`

contd. ...

`np.full()`	Creates a full array.	`>>> arr=np.full([2,2],3)`
		`>>> print(arr)`
		`[[3 3]`
		` [3 3]]`

`np.eye()`	Creates an identity matrix.	`>>> arr=np.eye(2,2)`
		`>>> print(arr)`
		`[[1. 0.]`
		` [0. 1.]]`

`np.random.random()`	Creates an array with random values.	`>>> arr=np.random.random((2,2))`
		`>>> print(arr)`
		`[[0.64936831 0.34385873]`
		` [0.45237516 0.01164925]]`

- **np.arange():** Returns an array with evenly spaced elements as per the interval. The interval mentioned is half opened i.e. [Start, Stop].

Syntax: `np.arange([start,] stop[, step,][, dtype])`

Where,

- **start:** [optional] start of interval range. By default start = 0.
- **stop:** end of interval range.
- **step:** [optional] step size of interval. By default step size = 1,

For any output out, this is the distance between two adjacent values, out[i+1] - out[i].

- **dtype:** type of output array.

Example:

```
>>> print(np.arange(5,30,5))
[ 5 10 15 20 25]
>>> print(np.arange(1,5))
[1 2 3 4]
>>> print(np.arange(0,2,0.3))
[0.  0.3  0.6  0.9  1.2  1.5  1.8]
```

8.3.3 Using Arrays and Scalars

- A scalar is one variable - for example, an integer. It can take different values at different times, but at any one time it only has one single value. Array scalars have the same attributes and methods as *ndarrays*.

Table 8.2: Built-in scalar types

Array scalar type	Related Python type
int_	int
float_	float
complex_	complex
bytes_	bytes
str_	str
bool_	bool
datetime64	datetime.datetime
timedelta64	datetime.timedelta

Scalar Addition:

- Scalars can be added and subtracted from arrays and arrays can be added and subtracted from each other.

  Example 1:

```
>>> import numpy as np
>>> a=np.array([10,20,30])
>>> b=a+2
>>> print(b)
[12 22 32]

>>> a=np.array([10,20,30])
>>> b=np.array([40,50,60])
>>> c=a+b
>>> print(c)
[50 70 90]
```

Multiplication:

- To multiply all the elements of a NumPy array with a scalar, the * operator in the NumPy package can be used.

  Example 2: Multiplication using of * operator.

```
>>> import numpy
>>> arr=numpy.array([10,20,30])
>>> newarr=arr*2
>>> print(newarr)
[20 40 60]
```

  In the above code, we first initialize a NumPy array using the `numpy.array()` function and then compute the product of that array with a scalar using the * operator.

- We can multiply a NumPy array with a scalar using the `numpy.multiply()` function in Python.

- The `numpy.multiply()` function gives us the product of two arrays. `numpy.multiply()` returns an array which is the product of two arrays given in the arguments of the function.

  Example 3: Mulitiplication using `numpy.multiply()` function.

```
>>> import numpy
>>> arr=numpy.array([10,20,30])
>>> newarr=numpy.multiply(arr,2)
>>> print(newarr)
[20 40 60]
```

  In the above code, we first initialize a NumPy array using `numpy.array()` function and then compute the product of that array with a scalar using the `numpy.multiply()` function.

8.3.4 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 as the alias array.
- Basic attributes of the `ndarray` class are as follows:

Table 8.3: Basic attributes of the `ndarray` class

Attributes	Description
shape	A tuple that specifies the number of elements for each dimension of the array. Gives the dimensions of the array. For an array with n rows and m columns, shape will be a tuple of integers (n, m).
size	The total number elements in the array.
ndim	Determines the number of axes or dimensions in the array.
nbytes	Number of bytes used to store the data.
dtype	Determines the datatype of elements stored in array.
data	Gives the buffer containing the actual elements of the array

Example 4: For array objects

```
>>> 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
```

8.3.5 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 operations as well as Unary and Binary operations. In case of +=, –=, *= operators, the existing array is modified.
- **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.
- **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 5: 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
```


8.3.5.1 Arithmetic Operations

Table 8.4: Arithmetic Operations

Operation	Description	Example
add()	The add() function sums the content of two arrays, and return the results in a new array.	import numpy as np a1 = np.array([10, 20, 30]) a2 = np.array([40, 50, 60]) a3 = np.add(a1, a2) print(a3) **Output:** [50 70 90]
subtract()	The subtract() function subtracts the values from one array with the values from another array, and return the results in a new array.	import numpy as np a1 = np.array([10, 20, 30]) a2 = np.array([40, 50, 60]) a3 = np.subtract(a2, a1) print(a3) **Output:** [30 30 30]
multiply()	The multiply() function multiplies the values from one array with the values from another array, and return the results in a new array.	import numpy as np a1 = np.array([10, 20, 30]) a2 = np.array([40, 50, 60]) a3 = np.multiply(a1, a2) print(a3) **Output:** [400 1000 1800]
divide()	The divide() function divides the values from one array with the values from another array, and return the results in a new array.	import numpy as np a1 = np.array([10, 20, 30]) a2 = np.array([40, 50, 60]) a3 = np.divide(a2, a1) print(a3) **Output:** [4. 2.5 2.]
power()	The power() function rises the values from the first array to the power of the values of the second array, and return the results in a new array.	import numpy as np a1 = np.array([10, 20, 30]) a2 = np.array([2, 2, 2]) a3 = np.power(a1, a2) print(a3) **Output:** [100 400 900]
mod()	mod() and the remainder() functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.	import numpy as np a1 = np.array([10, 20, 30]) a2 = np.array([4, 6, 8]) a3 = np.mod(a1, a2) print(a3) **Output:** [2 2 6]

8.3.5.2 Array Multiplication

- NumPy array can be multiplied by each other using matrix multiplication. These matrix multiplication methods include element-wise multiplication, the dot product, and the cross product.

Elementwise Multiplication:

- The standard multiplication sign in Python * produces element-wise multiplication on NumPy arrays.

```
>>> a=np.array([1,2,3])
>>> b=np.array([4,5,6])
>>> a*b
array([ 4, 10, 18])
```

Dot Product:

```
>>>a = np.array([1, 2, 3])
>>>b = np.array([4, 5, 6])
>>>np.dot(a,b)
32
```

Cross-Product:

```
>>>a = np.array([1, 2, 3])
>>>b = np.array([4, 5, 6])
>>>np.cross(a, b)
array([-3,  6, -3])
```

8.4 INDEXING ARRAYS, ARRAY TRANSPOSITION

8.4.1 Indexing Arrays

- Array indexing is the same as accessing an array element. You can access an array element by referring to its index number. The index of a value in an array is that value's location within the array. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example 6: For indexing arrays

```
>>>import numpy as np
>>> arr=np.array([10,20,30,40,50])
>>> print(arr)
[10 20 30 40 50]
>>> print(arr[0])
10
>>> print(arr[4])
50
>>> print(arr[2]+arr[3])
70
>>> arr[1]=60
>>> print(arr)
[10 60 30 40 50]
```


Accessing Two-dimensional Array:

- To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Example 7: To access two-dimensional array.

```
>>>import numpy as np
>>> arr1=np.array([[1,2,3,4],[5,6,7,8]])
>>> print(arr1)
[[1 2 3 4]
 [5 6 7 8]]
>>> print('3rd element from 2nd row: ',arr1[1,2])
3rd element from 2nd row:  7
#Use negative indexing to access an array from the end.
>>> print('3rd element from 1st row: ',arr1[0,-2])
3rd element from 1st row:  3
```

8.4.2 Array Transposition

- The transpose of a matrix is obtained by moving the rows data to the column and columns data to the rows. If we have an array of shape (X, Y) then the transpose of the array will have the shape (Y, X). One-dimensional array only returns an array equivalent to the original array.

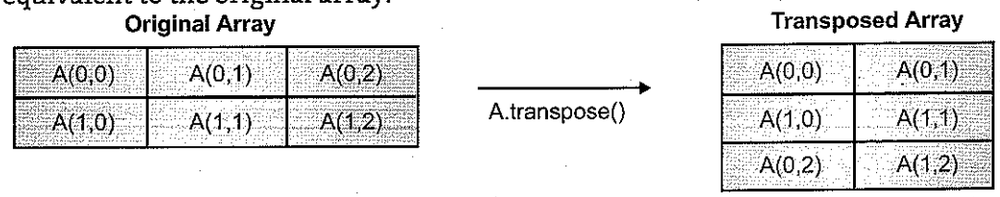


Fig. 8.4: Array Transposition

Syntax: `numpy.transpose(arr, axis=None)`

Where,

arr: It is an ndarray.

axis: If we didn't specify the axis, then by default, it reverses the dimensions otherwise permute the axis according to the given values.

Example 8: For array transposition.

```
import numpy as np
arr1=np.array([[1,2,3],[4,5,6]])
print(f'Original Array:\n {arr1}')
arr2=arr1.transpose()
print(f'Transposed Array:\n {arr2}')
```

Output:

```
Original Array:
 [[1 2 3]
 [4 5 6]]
```

```
Transposed Array:
 [[1 4]
 [2 5]
 [3 6]]
```

Transpose of an Array Like Object:

- The `transpose()` function works with an array like object too, such as a nested list.

```
arr1 = [[1, 2, 3], [4, 5, 6]]
arr2 = np.transpose(arr1)
```

NumPy transpose with axis:

- We can not only transpose a 2D array (matrix) but also rearrange the axes of a multidimensional array in any order.

Example 9: To transpose array with axis.

```
import numpy as np
arr1=np.array([[1,2],[3,4],[5,6]])
print(f'Original Array:\n {arr1}')
arr2=np.transpose(arr1,(1,0))
print(f'Transposed Array with axis:\n {arr2}')
```

Output:

```
Original Array:
 [[1 2]
 [3 4]
 [5 6]]
Transposed Array with axis:
 [[1 3 5]
 [2 4 6]]
```

Reposition elements using numpy.transpose():

- We can change the position of the array elements in the function.

```
>>> import numpy as np
>>> arr1=np.ones((10,20,30,40))
>>> arr2=np.transpose(arr1,(0,2,1,3)).shape
>>> arr2
(10, 30, 20, 40)
>>> arr3=np.transpose(arr1,(0,3,1,2)).shape
>>> arr3
(10, 40, 20, 30)
```

8.5 UNIVERSAL ARRAY FUNCTION

- Universal functions in Numpy are simple mathematical functions that cover a wide variety of operations.
- These functions include standard trigonometric functions, functions for arithmetic operations, handling complex numbers, statistical functions, etc.

8.5.1 Trigonometric Functions

Table 8.5: Trigonometric Functions

Function	Description
sin, cos, tan	Computes sine, cosine and tangent of angles
arcsin, arccos, arctan	Calculates inverse sine, cosine and tangent
hypot	Calculates hypotenuse of given right triangle
sinh, cosh, tanh	Computes hyperbolic sine, cosine and tangent
deg2rad	Converts degree into radians
rad2deg	Converts radians into degree

Example 10: To use of trigonometric functions.

```
import numpy as np
angles=np.array([0,30,45,60,90,180])
radians=np.deg2rad(angles)
print('Sine of angles in the array:')
sine_value = np.sin(radians)
print(np.sin(radians))
print('Cosine of angles in the array:')
cos_value = np.cos(radians)
print(np.cos(radians))
print('Tangent of angles in the array:')
tan_value = np.tan(radians)
print(np.tan(radians))
```

Output:

```
Sine of angles in the array:
[0.00000000e+00 5.00000000e-01 7.07106781e-01 8.66025404e-01
 1.00000000e+00 1.22464680e-16]
Cosine of angles in the array:
[ 1.00000000e+00  8.66025404e-01  7.07106781e-01  5.00000000e-01
  6.12323400e-17 -1.00000000e+00]
Tangent of angles in the array:
[ 0.00000000e+00  5.77350269e-01  1.00000000e+00  1.73205081e+00
  1.63312394e+16 -1.22464680e-16]
```

8.5.2 Statistical Functions

- These functions are used to calculate mean, median, variance, minimum of array elements.

Table 8.6: Statistical Functions

Function	Description
`min, max`	These functions return the minimum and the maximum from the elements in the given array along the specified axis.
`percentile(a, p, axis)`	Calculates p^{th} percentile of array or along specified axis. Percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall.
`mean`	Computes mean of data along specified axis
`std`	Standard deviation is the square root of the average of squared deviations from mean. `std = sqrt(mean(abs(x - x.mean())**2))`
`average`	Computes average of data along specified axis
`var`	Variance is the average of squared deviations, i.e., `mean(abs(x-x.mean())**2)`. In other words, the standard deviation is the square root of variance.

Example 11: For statistical functions.

```
import numpy as np
number=np.array([10,20,30,40,50])
print('Minimum and Maximum numbers: ')
print(np.min(number),np.max(number))
print('number which is below 70%: ')
print(np.percentile(number,70))
print('Mean value of the numbers: ')
print(np.mean(number))
print('Standard deviation of numbers: ')
print(np.std(number))
print('Average values of numbers: ')
print(np.average(number))
print('Variance of numbers: ')
print(np.var(number))
```

Output:

```
Minimum and Maximum numbers:
10 50
number which is below 70%:
38.0
Mean value of the numbers:
30.0
```



```
Standard deviation of numbers:
14.142135623730951
Average values of numbers:
30.0
Variance of numbers:
200.0
```

8.5.3 Bit-twiddling Functions

- These functions accept integer values as input arguments and perform bitwise operations on binary representations of those integers.

Table 8.7: Bit-twiddling Functions

Function	Description
`bitwise_and`	Performs bitwise AND operation between the corresponding array elements.
`bitwise_or`	Performs bitwise OR operation between the corresponding array elements.
`bitwise_xor`	Performs bitwise XOR operation on two array elements
`invert`	Perform bitwise NOT the operation of the array elements.
`left_shift`	Shift the bits of the binary representation of the elements to the left.
`right_shift`	Shift the bits of the binary representation of the elements to the right.

Example 12: For Bit-twiddling Functions.

```
import numpy as np
a=10
b=12
print("binary representation of a:",bin(a))
print("binary representation of b:",bin(b))
print("Bitwise-and of a and b: ",np.bitwise_and(a,b))
print("Bitwise-or of a and b: ",np.bitwise_or(a,b))
c=22
print("binary representation of a:",bin(c))
print("Inversion of number: ",np.invert(c))
print("left shift of 10 by 2 bits",np.left_shift(10, 2))
print("Right shift of 10 by 2 bits",np.right_shift(10, 2))
```

Output:

```
binary representation of a: 0b1010
binary representation of b: 0b1100
Bitwise-and of a and b:  8
```

```
Bitwise-or of a and b:  14
binary representation of a: 0b10110
Inversion of number:  -23
left shift of 10 by 2 bits 40
Right shift of 10 by 2 bits 2.
```

8.5.4 String Functions

- Strings are arrays of bytes representing Unicode characters. Following table shows various functions that operate on strings:

Table 8.8: String Functions

Function	Description
add()	It is used to concatenate the corresponding array elements (strings).
multiply()	It returns the multiple copies of the specified string.
center()	It returns the copy of the string where the original string is centered with the left and right padding filled with the specified number of fill characters.
capitalize()	It returns a copy of the original string in which the first letter of the original string is converted to the Upper Case.
title()	It returned string with the first letter of each word of the string is converted into the upper case.
lower()	It returns a copy of the string in which all the letters are converted into the lower case.
upper()	It returns a copy of the string in which all the letters are converted into the upper case.
split()	It returns a list of words in the string.
splitlines()	It returns the list of lines in the string, breaking at line boundaries.
strip()	Returns a copy of the string with the leading and trailing white spaces removed.
join()	It returns a string which is the concatenation of all the strings specified in the given sequence.
replace()	It returns a copy of the string by replacing all occurrences of a particular substring with the specified one.
decode()	It is used to decode the specified string element-wise using the specified codec.
encode()	It is used to encode the decoded string element-wise.

Example 13: For string functions.

```
import numpy as np
str1='hello '
str2='PYTHON '
str3='Welcome To Python Programming'
print('Concatenate two string: ',np.char.add(['hello '],['python']))
print('Capitalize string-1: ',str1.capitalize())
print("multiply a string-2 by 3 times:",np.char.multiply(str2,3))
print('Padding string with left and right with :
',np.char.center(str2,20,'*'))
print('Capitalizing String-1: ',np.char.capitalize(str1))
print('Convert string-2 with lowercase: ',np.char.lower(str2))
print('Convert string-1 with uppercase: ',np.char.upper(str1))
print("Splitting the String word by word..")
print(np.char.split(str3),sep = " ")
print(np.char.splitlines("Welcome\nTo\nPython\nProgramming"))
print('Removing leading and trailing white spaces from string')
print(np.char.strip('          Python          '))
print('joining two string: ',np.char.join(':','HM'))
print('Replacing string: ',np.char.replace(str3, "Python","Java"))
```

Output:

```
Concatenate two string:  ['hello python']
Capitalize string-1:  Hello
multiply a string-2 by 3 times: PYTHON PYTHON PYTHON
Padding string with left and right with :  ******PYTHON *******
Capitalizing String-1:  Hello
Convert string-2 with lowercase:  python
Convert string-1 with uppercase:  HELLO
Splitting the String word by word..
['Welcome', 'To', 'Python', 'Programming']
['Welcome', 'To', 'Python', 'Programming']
Removing leading and trailing white spaces from string
Python
joining two string:  H:M
Replacing string:  Welcome To Java Programming
```

8.6 ARRAY INPUT AND OUTPUT

Array Inputs from User:

- We can take number/string as a input from user using input() functions. input() function used to accept the list elements from a user in the format of a string separated by space.

Example 14: To show array Input and Output.

```
import numpy as np
list=[]
n=int(input("Enter number of elements: "))
for i in range(0,n):
    num=int(input())
    list.append(num)
print(list)
```

Output:

```
Enter number of elements: 5
10
20
30
40
50
[10, 20, 30, 40, 50]
```

Accessing Python Array Elements:

- We use indices to access elements of an array.

Example 15: To access Python array elements.

```
import numpy as np
list=[]
n=int(input("Enter number of elements: "))
for i in range(0,n):
    num=int(input())
    list.append(num)
print('index-values')
for i in range(0,n):
    print(i,list[i])
```

Output:

```
index-values
0 10
1 20
2 30
```

Changing and Adding Elements into Array:

- Arrays are mutable; their elements can be changed in a similar way as lists.

Example 16: To change and add Elements into array.

```
import array as arr
a=arr.array('i',[10,20,30])
print('Original Array')
print(a)
```



```
print('Changing values of array')
a[1]=50
print(a)
print('Adding element into array')
a.extend([40,60])
print(a)
```

Output:

```
Original Array
array('i', [10, 20, 30])
Changing values of array
array('i', [10, 50, 30])
Adding element into array
array('i', [10, 50, 30, 40, 60])
```

8.7 PANDAS

- **Pandas** is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
- 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.

Data structures supported by Pandas:

- Pandas deals with the following three data structures:

Table 8.9: Data structures supported by Pandas

Data Structure	Dimensions	Description
Series	1	1D labelled homogeneous array, size immutable.
Data Frames	2	General 2D labelled, size-mutable tabular structure with potentially heterogeneously typed columns
Panel	3	General 3D labelled, size-mutable array.

8.8 WHAT ARE PANDAS? WHERE IT IS USED?

- Pandas is an open-source, BSD-licensed Python library providing high-performance data manipulation, easy-to-use data structures and data analysis tools for the Python programming language.
- Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, statistics, analytics, etc.
- The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.

Install Pandas using pip:

- Pandas can be installed using PIP by following command:

```
pip install pandas
```

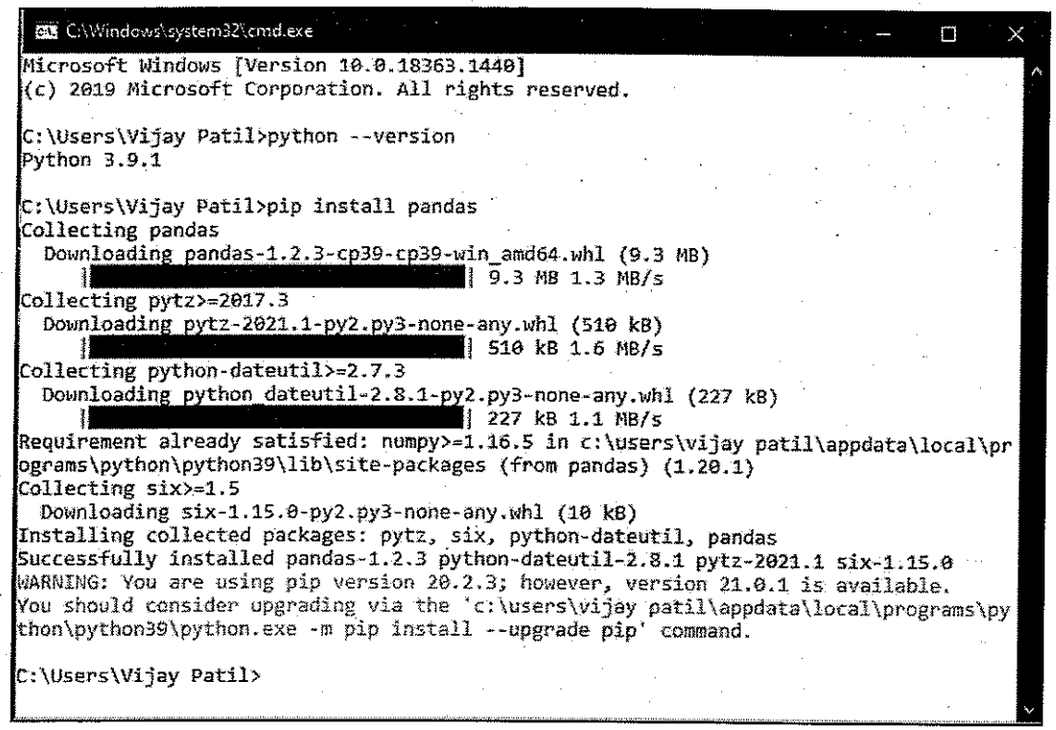


Fig. 8.5: Installation of Pandas using pip

To check version of pandas:

```
>>> import pandas as pd
>>> print(pd.__version__)
1.2.3
```

Features of Pandas:

- The Pandas library provides a really fast and efficient way to manage and explore data. Series and DataFrames help us to represent and manipulate data efficiently.
- Organization and labelling of data are perfectly taken care of by the intelligent methods of alignment and indexing, which can be found within Pandas.
- It is easy to perform data cleaning with missing values handling with help of some functions provided by Pandas.
- Pandas provides a wide array of built-in tools for the purpose of reading and writing data.
- Supports multiple file formats.
- Merging and joining of datasets is possible with help of pandas.
- Pandas provides features like moving window statistics and frequency conversion which is helpful for data scientists in time series analysis and relevant concepts.
- Pandas has an in-built ability to help you plot your data and see the various kinds of graphs formed.

- Pandas has ability of Grouping. It can separate data and grouping it according to the criteria we want. We can also find unique data in each feature.
- Pandas allows you to implement a mathematical operation on the data.

8.9 SERIES IN PANDAS, PANDAS DATAFRAMES, INDEX OBJECTS, RELNDEX

8.9.1 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 following 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 17: 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)
```

Output:

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

Example 18: 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
```

Output:

```
a  10
b  20
c  30
d  40
e  50
dtype: int64
>>>
```

Example 19: Create series from Dictionary

```
import numpy as np
import pandas as pd
dic ={'a':1, 'b':2, 'c':3, 'd':4,'e':5}
# Creating series of Dictionary type
sd=pd.Series(dic)
print(sd)
```

Output:

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

Example 20: Create series from Ndarray.

```
import numpy as np
import pandas as pd
data=[[10,20,30],[40,50,60]]
sr=pd.Series(data)
print(sr)
```

Output:

```
0    [10, 20, 30]
1    [40, 50, 60]
dtype: object
```


8.9.2 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	Umesh	Pune
13	Santosh	Mumbai

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

- It takes following arguments:
  - **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.
  - **index:** This argument holds the index value of each element in the DataFrame. The default index is `np.arange(n)`.
  - **columns:** The default values for columns is `np.arange(n)`.
  - **dtype:** This is the data type of the data passed in the method.
  - **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 21: Using Data is List

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

Output:

```
0  1
1  2
2  3
3  4
4  5
5  6
```

Example 22: Using Data as series.

```
import pandas as pd
s1=pd.Series([1,2,3])
s2=pd.Series([70.60,80.70,60.50])
s3=pd.Series(['Vijay','Santosh','Umesh'])
```

```
data={'RollID':s1,'Marks':s2,'Name':s3}
dseries=pd.DataFrame(data)
print(dseries)
```

Output:

```
    RollID    Marks    Name
0      1      70.6     Vijay
1      2      80.7     Santosh
2      3      60.5     Umesh
```

Example 23: Using Data frame is Dictionary

```
>>> import pandas as pd
>>>   dict={"Name":["Vijay","Santosh","Umesh","Pravin"],
"Age":[44,40,35,30]}
>>> df=pd.DataFrame(dict)
>>> print(df)
```

Output:

```
        Name        Age
0       Vijay       44
1       Santosh     40
2       Umesh       35
3       Pravin      30
```

Example 24: Using Data is 2D-numpy ndarray

```
import pandas as pd
d1 =[[1, 3, 5], [7, 9, 11]]
d2 =[[2, 4, 6], [8, 10, 12]]
Data ={'first': d1, 'second': d2}
df2d = pd.DataFrame(Data)
print(df2d)
```

Output:

```
        first          second
0     [1, 3, 5]      [2, 4, 6]
1     [7, 9, 11]     [8, 10, 12]
```

8.9.3 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.

  Syntax: `pandas.Panel(data, item, major_axis, minor_axis, dtype, copy)`
- It takes the following arguments:
  - **data:** The data can be of any form like ndarray, list, dict, map, DataFrame.
  - **item:** axis 0, each item corresponds to a DataFrame contained inside.

  - **major_axis:** axis 1, it is the index (rows) of each of DataFrames.
  - **minor_axis:** axis 2, it is the columns of each DataFrames.
  - **dtype:** The data type of each column.
  - **copy:** It takes a Boolean value to specify whether or not to copy the data. The default value is false.

Example 25: Use of Panel data structure.

```
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 26: Create dataframe from list.

```
import pandas as pd
import numpy 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
```

8.9.4 Index object

- Pandas Index is an immutable ndarray implementing an ordered, sliceable set. It is the basic object which stores the axis labels for all Pandas objects.

  Syntax: `index.values`
- It returns an array which substitutes the index object of the array.

Example 27: Use of index object.

```
import pandas as pd
idx=pd.Index(['Mumbai','Thane','Pune','Nagpur','Jalgaon'])
print(idx)
result=idx.values
print(result)
```

Output:

```
Index(['Mumbai', 'Thane', 'Pune', 'Nagpur', 'Jalgaon'], dtype='object')
['Mumbai' 'Thane' 'Pune' 'Nagpur' 'Jalgaon']
```

- The `Index.values` attribute has successfully returned an array representing the data of the given Index object.

8.9.4.1 Set_index Method

- Pandas `set_index()` is an inbuilt Pandas work that is used to set the List, Series or DataFrame as a record of a Data Frame. It sets the index in the DataFrame with the available columns. This command can basically replace or expand the existing index columns.

Setting Multiple Index in Pandas:

- Multiple Indexes or multiline indexes are nothing but the tabulation of rows and columns in multiple lines. Here the indexes can be added, removed, and replaced.

Example 28: For setting multiple indexes.

```
import pandas as pd
df=pd.DataFrame({'Fruits':['Mango','Orange','Banana'],'Price':[80,30,10]
})
print(df)
df=df.set_index(['Fruits','Price'])
print(df)
```

Output:

```
    Fruits    Price
0   Mango     80
1   Orange    30
2   Banana    10
Empty DataFrame
Columns: []
Index: [(Mango, 80), (Orange, 30), (Banana, 10)]
```


8.9.4.2 reindex

- We can reset index by using reset_index() method.

Example 29: For reset index.

```
import pandas as pd
df=pd.DataFrame({'Fruits':['Mango','Orange','Banana'],'Price':[80,30,10]
})
print(df)
df=df.set_index(['Fruits','Price'])
print(df)
df=df.reset_index(['Fruits','Price'])
print(df)
```

Output:

```
    Fruits    Price
0   Mango     80
1   Orange    30
2   Banana    10
Empty DataFrame
Columns: []
Index: [(Mango, 80), (Orange, 30), (Banana, 10)]
    Fruits    Price
0   Mango     80
1   Orange    30
2   Banana    10
```

8.10 DROP ENTRY, SELECTING ENTRIES

8.10.1 drop()

- The drop function allows the removal of rows and columns from your DataFrame. Rows can be removed using index label or column name using this method. To use it to remove columns, specify axis=1.

  Syntax: `DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')`

Parameters:

- **labels:** String or list of strings referring row or column name.
- **axis:** int or string value, 0 'index' for Rows and 1 'columns' for Columns.
- **index or columns:** Single label or list. index or columns are an alternative to axis and cannot be used together.
- **level:** Used to specify level in case data frame is having multiple level index.
- **inplace:** Makes changes in original DataFrame if True.
- **errors:** Ignores error if any value from the list doesn't exists and drops rest of the values when errors = 'ignore'.

  Return type: Dataframe with dropped values.

Example 30: Delete rows using drop().

```
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details,columns=['Name','Age','City'],
                index=['a','b','c','d'])
print(df)
#deleting a single row by row index label
# delete multiple rows by df.drop(['b','c'])
new_df=df.drop('c')
print(new_df)
```

Output:

```
    Name      Age   City
a   Vijay     40    Thane
b   Santosh   30    Mumbai
c   Umesh     35    Pune
d   Pravin    25    Jalgaon

    Name      Age   City
a   Vijay     40    Thane
b   Santosh   30    Mumbai
d   Pravin    25    Jalgaon
```

Example 31: Delete column by name using drop().

```
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details,columns=['Name','Age','City'],
                index=['a','b','c','d'])
print(df)
#delete column
new_df=df.drop('City',axis=1)
print(new_df)
```


Output:

```
    Name      Age   City
a   Vijay     40    Thane
b   Santosh   30    Mumbai
c   Umesh     35    Pune
d   Pravin    25    Jalgaon

    Name      Age
a   Vijay     40
b   Santosh   30
c   Umesh     35
d   Pravin    25
```

Example 32: Delete column by column number.

```
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details,columns=['Name','Age','City'],
                index=['a','b','c','d'])
print(df)
#delete column
new_df=df.drop(columns=df.columns[1])
print(new_df)
```

Output:

```
    Name      Age   City
a   Vijay     40    Thane
b   Santosh   30    Mumbai
c   Umesh     35    Pune
d   Pravin    25    Jalgaon

    Name      City
a   Vijay     Thane
b   Santosh   Mumbai
c   Umesh     Pune
d   Pravin    Jalgaon
```

8.10.2 Delete column using del()

- You can delete a column by using `del df['column name']`.

Example 33: Delete column using del().

```
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details,columns=['Name','Age','City'],
                index=['a','b','c','d'])
print(df)
#delete column using del
del df["Name"]
print(df)
```

Output:

```
    Name     Age   City
a   Vijay    40    Thane
b   Santosh  30    Mumbai
c   Umesh    35    Pune
d   Pravin   25    Jalgaon

    Age   City
a   40    Thane
b   30    Mumbai
c   35    Pune
d   25    Jalgaon
```

8.10.3 Delete column using pop()

- `pop()` function would also drop the column. Unlike the other two methods, this function would return the column.

Example 34: Delete column using `pop()`.

```
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details,columns=['Name','Age','City'],
                index=['a','b','c','d'])
print(df)
#delete column using del
df.pop("Age")
print(df)
```


Output:

```
    Name     Age   City
a   Vijay    40    Thane
b   Santosh  30    Mumbai
c   Umesh    35    Pune
d   Pravin   25    Jalgaon

    Name     City
a   Vijay    Thane
b   Santosh  Mumbai
c   Umesh    Pune
d   Pravin   Jalgaon
```

8.10.4 Add rows in DataFrame

- We can add a single row using DataFrame.loc. We can add the row at the last in our dataframe. We can get the number of rows using `len(DataFrame.index)` for determining the position at which we need to add the new row. We can also add a new row using the `DataFrame.append()` function.

Example 35: Add rows in Dataframe.

```
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
print(df)
df.loc[len(df.index)]=['Ajay',50,'Mumbai']
print(df)
#add row using DataFrame.append() function
df1={'Name':'Yogita','Age':40,'City':'Mumbai'}
df=df.append(df1,ignore_index=True)
print(df)
```

Output:

```
    Name     Age   City
0   Vijay    40    Thane
1   Santosh  30    Mumbai
2   Umesh    35    Pune
3   Pravin   25    Jalgaon
```

```
    Name      Age   City
0   Vijay     40    Thane
1   Santosh   30    Mumbai
2   Umesh     35    Pune
3   Pravin    25    Jalgaon
4   Ajay      50    Mumbai

    Name      Age   City
0   Vijay     40    Thane
1   Santosh   30    Mumbai
2   Umesh     35    Pune
3   Pravin    25    Jalgaon
4   Ajay      50    Mumbai
5   Yogita    40    Mumbai
```

8.10.5 Concat()

- We can also add multiple rows using the `pandas.concat()` by creating a new dataframe of all the rows that we need to add and then appending this dataframe to the original dataframe.

Example 36: Add rows using concat().

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df1=pd.DataFrame(details)
print(df1)

details={'Name':['Yogita','Sunil'],
     'Age':[30,40],
     'City':['Thane','Pune']}
df2=pd.DataFrame(details)
print(df2)

df3=pd.concat([df1,df2],ignore_index=True)
df3.reset_index()
print(df3)
```


Output:

```
    Name      Age   City
0   Vijay     40    Thane
1   Santosh   30    Mumbai
2   Umesh     35    Pune
3   Pravin    25    Jalgaon

    Name      Age   City
0   Yogita    30    Thane
1   Sunil     40    Pune

    Name      Age    City
0   Vijay     40    Thane
1   Santosh   30    Mumbai
2   Umesh     35    Pune
3   Pravin    25    Jalgaon
4   Yogita    30    Thane
5   Sunil     40    Pune
```

8.10.6 Different ways to add column in DataFrame

Example 37: Add column using list.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
print(df)

Dept=['CO','IF','EJ','CO']
df['Dept']=Dept
print(df)
```

Output:

```
    Name      Age   City
0   Vijay     40    Thane
1   Santosh   30    Mumbai
2   Umesh     35    Pune
3   Pravin    25    Jalgaon

    Name      Age   City      Dept
0   Vijay     40    Thane     CO
1   Santosh   30    Mumbai    IF
2   Umesh     35    Pune      EJ
3   Pravin    25    Jalgaon   CO
```

Example 38: Add new column using insert()

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
print(df)
df.insert(2,"Dept",['CO','IF','EJ','CO'],True)
print(df)
```

Output:

```
   Name      Age   City
0  Vijay     40    Thane
1  Santosh   30    Mumbai
2  Umesh     35    Pune
3  Pravin    25    Jalgaon

   Name      Age   Dept  City
0  Vijay     40    CO    Thane
1  Santosh   30    IF    Mumbai
2  Umesh     35    EJ    Pune
3  Pravin    25    CO    Jalgaon
```

Example 39: Add new column using assign()

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
print(df)
df1=df.assign(dept=['CO','IF','EJ','CO'])
print(df1)
```

Output:

```
   Name      Age   City
0  Vijay     40    Thane
1  Santosh   30    Mumbai
2  Umesh     35    Pune
3  Pravin    25    Jalgaon
```



```
   Name      Age   City      Dept
0  Vijay     40    Thane     CO
1  Santosh   30    Mumbai    IF
2  Umesh     35    Pune      EJ
3  Pravin    25    Jalgaon   CO
```

8.10.7 Selecting Entries

- We can select rows from the DataFrame using `df.loc`.

  Syntax: `df.loc[df['Column_name']condition]`

  Examples:

  1. To display rows where Age is equal or greater than 30: `df.loc[df['Age'] ==30]`
  2. We may use the & symbol to apply multiple conditions: `df.loc[(df['Name'] == 'Vijay') & (df['City'] == 'Thane')]`
  3. To select data either or: `df.loc[(df['City'] == 'Thane') | (df['City'] == 'Pune')]`
  4. Not equal to: `df.loc[df['Age'] !=40]`

Example 40: Selecting single column using column_name.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
df1=df[['Name','City']]
print(df1)
```

Output:

```
   Name      City
0  Vijay     Thane
1  Santosh   Mumbai
2  Umesh     Pune
3  Pravin    Jalgaon
```

Example 41: Selecting rows using condition.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
```

```
df=pd.DataFrame(details)
print(df)

sel=df.loc[df['Age']>=35]
print(sel)
```

Output:

```
    Name      Age   City
0   Vijay     40    Thane
1   Santosh   30    Mumbai
2   Umesh     35    Pune
3   Pravin    25    Jalgaon

    Name      Age   City
0   Vijay     40    Thane
2   Umesh     35    Pune
```

Example 42: To select multiple values using 'or (|)'.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
sel=df.loc[(df['City']=='Thane') | (df['City']=='Pune')]
print(sel)
```

Output:

```
   Name   Age City
0  Vijay  40  Thane
2  Umesh  35  Pune
```

8.11 DATA ALIGNMENT, RANK AND SORT

8.11.1 Data Alignment

- Pandas DataFrame represents data in a tabular format. We can align data in the column as left, right or center. Align the two dataframes have the same row and/or column configuration. We use align when we would like to synchronize a dataframe with another dataframe or a dataframe with a Series or in other words we want to map one dataframe into another dataframe or series using different join methods like outer, inner, left and right.

Syntax:

```
DataFrame.align(other, join='outer', axis=None, level=None,
copy=True, fill_value=None, method=None, limit=None,
fill_axis=0, broadcast_axis=None)
```


Parameters:

- **other**: DataFrame or Series
- **join**:{'outer', 'inner', 'left', 'right'}, default 'outer'
- **axis**: allowed axis of the other object, default None. Align on index (0), columns (1), or both (None).
- **level**: int or level name, default None.
- **copy**: bool, default True.
- **fill_value**: scalar, default np.NaN
- **method:** {'backfill', 'bfill', 'pad', 'ffill', None}, default None.
  Method to use for filling holes in reindexed Series:
  - **pad / ffill:** propagate last valid observation forward to next valid.
  - **backfill / bfill:** use NEXT valid observation to fill gap.
- **limit:** int, default None.
- **fill_axis**: {0 or 'index', 1 or 'columns'}, default 0.
- **broadcast_axis:** {0 or 'index', 1 or 'columns'}, default None.

Returns: (left, right) (DataFrame, type of other). Aligned objects.

8.11.2 Sorting by sort_values()

- Pandas sort_values() can sort the data frame in Ascending or Descending order.

Example 43: Sort data in ascending order.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
print("Original Data")
print(df)
print('Sort by column')
df1=df.sort_values(by=['Age'])
print(df1)
```

Output:

```
Original Data
   Name      Age   City
0  Vijay     40    Thane
1  Santosh   30    Mumbai
2  Umesh     35    Pune
3  Pravin    25    Jalgaon
```

```
Sort by column
   Name      Age   City
3  Pravin    25    Jalgaon
1  Santosh   30    Mumbai
2  Umesh     35    Pune
0  Vijay     40    Thane
```

Example 44: Sort data in descending order.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,35,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
print("Original Data")
print(df)
print('Sort by column')
df1=df.sort_values(by=['Age'],ascending=False)
print(df1)
```

Output:

```
Original Data
   Name      Age   City
0  Vijay     40    Thane
1  Santosh   30    Mumbai
2  Umesh     35    Pune
3  Pravin    25    Jalgaon
Sort by column
   Name      Age   City
0  Vijay     40    Thane
2  Umesh     35    Pune
1  Santosh   30    Mumbai
3  Pravin    25    Jalgaon
```

8.11.3 Ranking

- Compute numerical data ranks (1 through n) along axis.

Syntax: `DataFrame.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)`

Parameters:

- **axis:** {0 or 'index', 1 or 'columns'}, default 0.
- **method:** {'average', 'min', 'max', 'first', 'dense'}, default 'average'.

How to rank the group of records that have the same value (i.e. ties):

- **average:** average rank of the group
- **min:** lowest rank in the group
- **max:** highest rank in the group
- **first:** ranks assigned in order they appear in the array
- **dense:** like 'min', but rank always increases by 1 between groups.

- **numeric_only:** bool, optional, For DataFrame objects, rank only numeric columns if set to True.
- **na_option:** {'keep', 'top', 'bottom'}, default 'keep'.

  How to rank NaN values:
  - **keep:** assign NaN rank to NaN values
  - **top:** assign smallest rank to NaN values if ascending
  - **bottom:** assign highest rank to NaN values if ascending.
- **ascending:** bool, default True.
- **pct:** bool, default False, Whether or not to display the returned rankings in percentile form.

Returns: same type as caller.

- Return a Series or DataFrame with data ranks as values.

Example 45: Compute numerical data ranks.

```
import numpy as np
import pandas as pd
#dictionary of student details
details={'Name':['Vijay','Santosh','Umesh','Pravin'],
         'Age':[40,30,15,25],
         'City':['Thane','Mumbai','Pune','Jalgaon'],
         }
df=pd.DataFrame(details)
df['Age_ranked']=df['Age'].rank(ascending=0)
print(df)
```

Output:

```
   Name      Age   City      Age_ranked
0  Vijay     40    Thane     1.0
1  Santosh   30    Mumbai    2.0
2  Umesh     15    Pune      4.0
3  Pravin    25    Jalgaon   3.0
```

8.12 SUMMARY STATICS, MISSING DATA, INDEX HIERARCHY

8.12.1 Pandas Statistics

- Python statistics libraries are comprehensive, popular, and widely used tools that will assist you in working with data. We can use numeric quantities to describe and summarize datasets, calculate descriptive statistics and visualize datasets. Following table shows functions under Descriptive Statistics in Python pandas:

Table 8.10: Functions under Descriptive Statistics in Python pandas

Function	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of Values
median()	Median of Values
mode()	Mode of values
std()	Standard Deviation of the Values
min()	Minimum Value
max()	Maximum Value
abs()	Absolute Value
prod()	Product of Values
cumsum()	Cumulative Sum
cumprod()	Cumulative Product

Example 46: Use of functions under descriptive statistics in Python pandas.

```
import pandas as pd
import numpy as np
#create dictionary
data={'Name':['Vijay','Santosh','Umesh','Pravin','Yogita'],
        'Age':[40,30,20,25,30],
        'Marks':[80,70,60,55,70],
        }
df=pd.DataFrame(data)
print(df)
print('Sum of marks:\n',df.sum())
print('sum of column:\n',df.sum(1))
print('Mean:\n',df.mean())
```


Output:

```
   Name      Age   Marks
0  Vijay     40    80
1  Santosh   30    70
2  Umesh     20    60
3  Pravin    25    55
4  Yogita    30    70
Sum of marks:
   Name     VijaySantoshUmeshPravinYogita
   Age      145
   Marks    335
dtype: object
sum of column:
   0   120
   1   100
   2    80
   3    80
   4   100
dtype: int64
Mean:
   Age      29.0
   Marks    67.0
   dtype: float64
```

8.12.2 Summarizing Data

- The describe() function computes a summary of statistics pertaining to the DataFrame columns.

Example 47: Summarize data.

```
import pandas as pd
import numpy as np
#create dictionary
data={'Name':['Vijay','Santosh','Umesh','Pravin','Yogita'],
        'Age':[40,30,20,25,30],
        'Marks':[80,70,60,55,70],
        }
df=pd.DataFrame(data)
print(df)
print('Summarize data:')
print(df.describe())
```

Output:

```
    Name      Age   Marks
0   Vijay     40    80
1   Santosh   30    70
2   Umesh     20    60
3   Pravin    25    55
4   Yogita    30    70
Summarize data:
            Age          Marks
count    5.000000     5.000000
mean    29.000000    67.000000
std      7.416198     9.746794
min     20.000000    55.000000
25%     25.000000    60.000000
50%     30.000000    70.000000
75%     30.000000    70.000000
max     40.000000    80.000000
```

- 'include' is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.
  - **Object:** Summarizes String columns.
  - **Number:** Summarizes Numeric columns.
  - **All:** Summarizes all columns together (Should not pass it as a list value).

Example:

```
print (df.describe(include=['object']))
print (df.describe(include=['all']))
```

8.12.3 Missing Data

- As data comes in many shapes and forms, Pandas aim to be flexible with regard to handling missing data. While NaN (Not a Number) is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, Boolean, and general object.
- In many cases, the Python None will arise and we wish to also consider that "missing" or "not available" or "NA". Functions for detecting, removing, and replacing null values in Pandas DataFrame are:

Table 8.11: Functions to handle Missing data

Function	Description
isnull()	Returns value is NaN or not.
notnull()	Returns not null values.
dropna()	Allows the user to analyze and drop Rows/Columns with Null values.
fillna()	Allow user to replace NaN values with some value of their own.
replace()	Replace a string, regex, list, dictionary, series, number etc. from a dataframe.
interpolate()	Used to fill missing values in the dataframe or series using linear method.

Example 48: Use of functions to handle missing data.

```
import pandas as pd
import numpy as np
#create dictionary
df=pd.DataFrame(np.random.randn(5,3),index=['a','c','e','f','h'],
columns=['Col1','Col2','Col3'])
df=df.reindex(['a','b','c','d','e','f','g','h'])
print(df)
print('Checking for Missing Values')
print(df['Col1'].isnull())
print('Checking for Not Null Values')
print(df['Col1'].notnull())
print('NA: replace with zeros')
print(df['Col1'].fillna(0))
print('Drop rows of NA values')
print(df.dropna())
print('Replace NA values with -99')
print(df.replace(to_replace=np.nan,value=-99))
print('Interpolate Values')
print(df.interpolate(method='linear', limit_direction='forward'))
```

8.12.4 Index Hierarchy

- The Python and NumPy indexing operators "[ ]" and attribute operator "." provides quick and easy access to Pandas data structures across a wide range of use cases.
- Pandas now supports three types of Multi-axes indexing:

Table 8.12: Methods of Multi-axes Indexing

Method	Description
.loc()	Label Based
.iloc()	Integer Based
.ix()	Both Label and Integer Based

Example 49: Use of methods of Multi-axes indexing.

```
import pandas as pd
import numpy as np
#create dictionary
df=pd.DataFrame(np.random.randn(4,2),index=['a','c','c','d'],
columns=['A','B'])
print(df)
print(df.loc[:,'A'])
print(df.iloc[:2])
```

- The MultiIndex object is the hierarchical analogue of the standard Index object which typically stores the axis labels in Pandas objects. The Pandas MultiIndex type contains multiple levels of indexing and multiple labels for each data point which encode these levels.

Example 50: Use of MultiIndex object.

```
import pandas as pd
import numpy as np
index=[('A',1),('A',2),('B',1),('B',2)]
s=pd.Series([1.0,2.0,3.0,4.0],index=index)
print(s)
index=pd.MultiIndex.from_tuples(index)
print(index)
s=s.reindex(index)
print(s)
```

Output:

```
(A, 1)    1.0
(A, 2)    2.0
(B, 1)    3.0
(B, 2)    4.0
dtype: float64
MultiIndex([('A', 1),
            ('A', 2),
            ('B', 1),
            ('B', 2)],
           )
A  1    1.0
   2    2.0
B  1    3.0
   2    4.0
dtype: float64
```


8.13 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.

```
pip install matplotlib OR  conda install matplotlib
```

OR

```
python -mpip install -U matplotlib
```

- **Importing Matplotlib:**

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

OR

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

Advantages of using Matplotlib to visualize data:

1. Matplotlib is fast and efficient because it provides a multi-platform data visualization tool built on the NumPy and sidepy framework.
2. It possesses the ability to work well with many operating systems and graphic backend.
3. It possesses high-quality graphics and plots to print and view for a range of graphs such as histograms, bar charts, pie charts, scatter plots and heat maps.
4. With Jupyter notebook integration, the developers have been free to spend their time implementing features rather than struggling with compatibility.
5. It has large community support and cross-platform support as it is an open source tool.
6. It has full control over graph or plot styles such as line properties, thoughts, and access properties.

8.14 PYTHON FOR DATA VISUALIZATION

- Data visualization is the technique to present the data in a pictorial or graphical format. Data visualization is the discipline of trying to understand data by placing it in a visual context so that patterns, trends and correlations that might not otherwise be detected can be exposed.
- Python provides popular plotting libraries are:
  - **Matplotlib:** low level, provides lots of freedom.
  - **Pandas Visualization:** easy to use interface, built on Matplotlib.
  - **Seaborn:** high-level interface, great default styles.
  - **ggplot:** based on R's ggplot2, uses Grammar of Graphics.
  - **Plotly:** can create interactive plots.

Advantages of data visualization are as follows:

1. It simplifies the complex quantitative information.
2. It helps analyze and explore big data easily.
3. It identifies the areas that need attention or improvement.
4. It identifies the relationship between data points and variables.
5. It explores new patterns and reveals hidden patterns in the data.

Major considerations for Data Visualization:

- **Clarity** ensures that the data set is complete and relevant. This enables the data scientist to use the new patterns yield from the data in the relevant places.
- **Accuracy** ensures using appropriate graphical representation to convey the right message.
- **Efficiency** uses efficient visualization technique which highlights all the data points.

Basic factors that one would need to be aware of before visualizing the data:

- **Visual Effect** includes the usage of appropriate shapes, colors, and sizes to represent the analyzed data.
- The **Coordinate System** helps to organize the data points within the provided coordinates.
- The **Data Types and Scale** choose the type of data such as numeric or categorical.
- The **Informative Interpretation** helps create visuals in an effective and easily interpreted ill manner using labels, title legends, and pointers.

8.15 INTRODUCTION TO Matplotlib

- Matplotlib is one of the most popular Python packages used for data visualization. Matplotlib comes with a wide variety of plots. Plots help to understand trends, patterns, and to make correlations. It provides an object-oriented API that helps in embedding plots in applications using Python GUI toolkits such as PyQt, WxPythonotTkinter. It can be used in Python and IPython shells, Jupyter notebook and web application servers also.

8.15.1 Architecture of Matplotlib

- The architecture of Matplotlib is logically structured into three layers, which are placed at three different levels. The communication is unidirectional, that is, each layer can communicate with the underlying layer, while the lower layers cannot communicate with the top ones.
- The three layers are as follows:
  1. Scripting
  2. Artist
  3. Backend

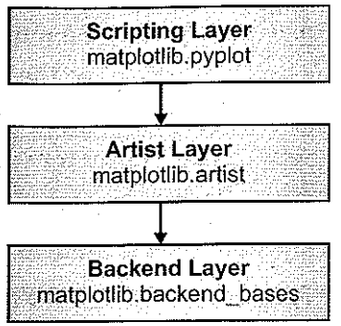


Fig. 8.6: Architecture of Matplotlib

1. **Backend Layer:** In the diagram of the Matplotlib architecture, the layer that works at the lowest level is the *Backend* layer. This layer contains the Matplotlib APIs, a set of classes that play the role of implementation of the graphic elements at a low level.
   - **FigureCanvas:** Defines the area on which the figure is drawn.
   - **Renderer:** It is the tool to draw on FigureCanvas.
   - **Event:** Handles user inputs such as keyboard strokes and mouse clicks.

   It is similar to how we do drawing on paper. Consider you want to draw a painting. You get a blank paper (FigureCanvas) and a brush (Renderer). You ask your friend what to draw (Event).
2. **Artist Layer:** As an intermediate layer, we have a layer called Artist. All the elements that make up a chart, such as the title, axis labels, markers, etc., are instances of the Artist object. Each of these instances plays its role within a hierarchical structure.

   There are two types of artist objects: Primitive and Composite.
   1. **Primitive:** The primitive artists are individual objects that constitute the basic elements to form a graphical representation in a plot, for example, a Line2D, or as a geometric figure such as a Rectangle or Circle, or even pieces of text.
   2. **Composite:** The composite artists are those graphic elements present in a chart that is composed of several base elements, namely, the Primitive artists. Composite artists are for example, the Axis, Ticks, Axes, and Figures.
      - **Figure** is the object with the highest level in the hierarchy. It corresponds to the entire graphical representation and generally can contain many Axes.
      - **Axes** is generally what you mean as plot or chart. Each Axis object belongs to only one Figure, and is characterized by two Artist Axis (three in the three-dimensional case). Other objects, such as the title, the x label, and the y label, belong to this composite artist.

- **Axis** objects that take into account the numerical values to be represented on Axes, define the limits and manage the ticks (the mark on the axes) and tick labels (the label text represented on each tick). The position of the tick is adjusted by an object called a **Locator** while the formatting tick label is regulated by an object called a **Formatter**.

3. **Scripting Layer:** This layer consists of an interface called **pyplot**. For purposes of calculation, and in particular for the analysis and visualization of data, the scripting layer is best. Scripting layer is the **matplotlib.pyplot** interface. Thus, when we create plots using "plt" after the following command, scripting layer is what we play with:

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

8.15.2 Pyplot

- The Pyplot module is a collection of command-style functions that allow you to use Matplotlib much like Matlab. Each Pyplot function will operate or make some changes to the Figure object, for example, the creation of the Figure itself, the creation of a plotting area, representation of a line, decoration of the plot with a label, etc.

8.15.3 Types of Charts

1. Line Chart:

- In Matplotlib, we can create a line chart by calling the plot method. We can also plot multiple columns in one graph, by looping through the columns we want and plotting each column on the same axis.

Steps to draw chart:

1. Import the required libraries.
2. Define or import the required data set.
3. Set the plant parameters.
4. Display the created plant.

Example 51: Draw a line chart.

```
>>> 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:

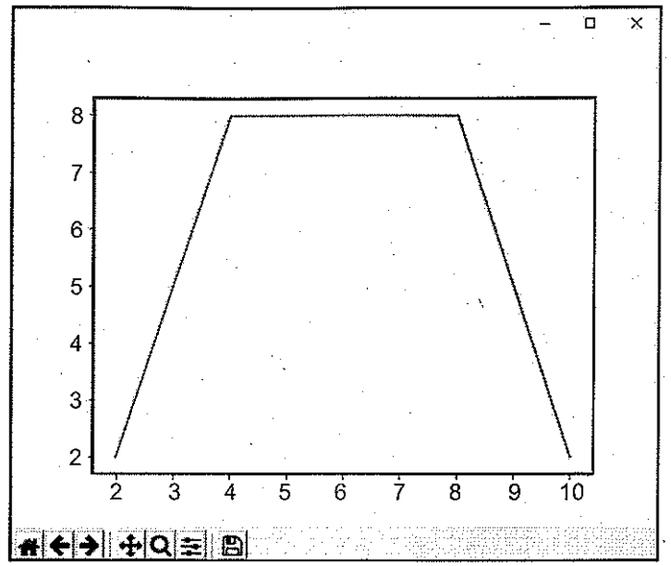


Fig. 8.7: Line Chart

2. 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 52: Draw bar graph.

```
>>> from matplotlib import pyplot as plt
>>> x=[2,4,8,10]
>>> y=[2,8,8,2]
>>> plt.xlabel('X-Axis')
Text(0.5, 0, 'X-Axis')
>>> plt.ylabel('Y-Axis')
Text(0, 0.5, 'Y-Axis')
>>> plt.title('Graph')
Text(0.5, 1.0, 'Graph')
>>> plt.bar(x,y,label="Graph",color='r',width=.5)
<BarContainer object of 4 artists>
>>> plt.show()
```

Output:

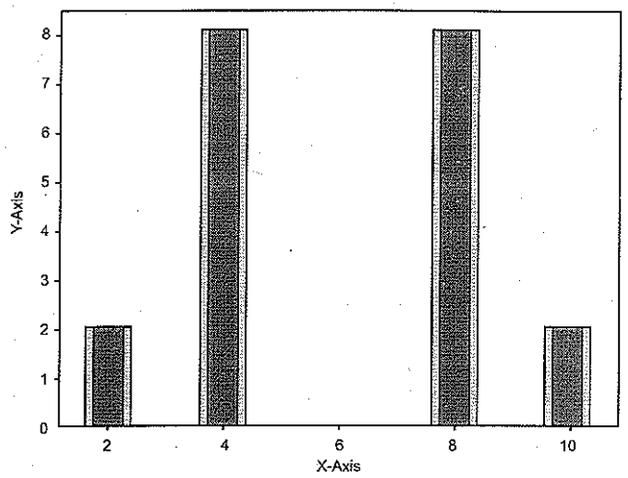


Fig. 8.8: Bar Graph

3. Histogram:

- Histograms are used to show a distribution whereas a bar chart is used to compare different entities. Histograms are useful when you have arrays or a very long list. We can create a Histogram using the **hist** method. If we pass it categorical data like the points column from the wine-review dataset it will automatically calculate how often each class occurs.

Example 53: Draw histogram.

```
>>> from matplotlib import pyplot as plt
>>> y=[2,8,8,2]
>>> plt.hist(y)
(array([2., 0., 0., 0., 0., 0., 0., 0., 0., 2.]), array([2. , 2.6, 3.2,
3.8, 4.4, 5. , 5.6, 6.2, 6.8, 7.4, 8. ]), <a list of 10 Patch objects>)
>>> plt.show()
```

Output:

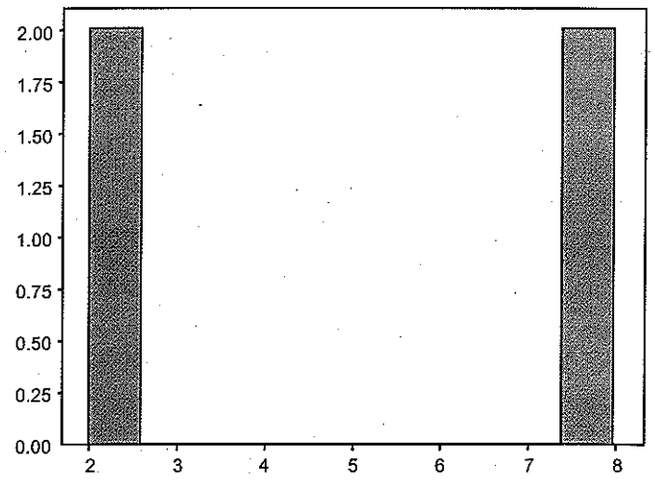


Fig. 8.9: Histogram

4. 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 54: Draw 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()
```

Output:

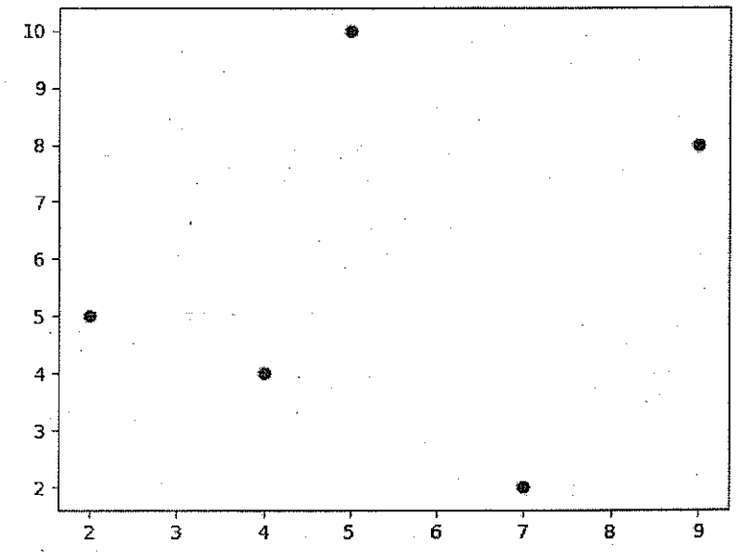


Fig. 8.10: Scatter Plot

8.16 VISUALIZATION TOOLS

- Data visualization tools provide data visualization designers with an easier way to create visual representations of large data sets.
- These data visualizations can then be used for a variety of purposes: dashboards, annual reports, sales and marketing materials, investor slide decks, and virtually anywhere else information needs to be interpreted immediately.
- Following are some Data visualization tools:

1. Plotly:

- Plotly s a web-based toolkit to form data visualisations. Plotly can also be accessed from a Python Notebook and has a great API. With unique functionalities such as

contour plots, dendrograms, and 3D charts, 3D-Plots, Map-based visualization. It has visualizations like scatter plots, line charts, bar charts, error bars, box plots, histograms, multiple axes, subplots and many others.

Plotly is highly compatible with Jupyter Notebook and Web Browsers. This means whatever interactive plots you create can easily be shared in the same manner with your teammates or end-users. Plotly's plots can also support animation capabilities as well.

2. Seaborn:

- Seaborn is a library based on Matplotlib. It provides a much terser API for creating KDE-based visualizations. It provides a high-level interface for drawing attractive and informative statistical graphics. It is tightly integrated with PyData stack, including support for NumPy and Pandas data structures. Seaborn aims to make visualization a central part of exploring and understanding data. Its dataset-oriented plotting functions operate on data frames and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots.

3. ggplot:

- ggplot is a Python implementation of the Grammar of Graphics of R programming language. It is a system for declaratively creating graphics, based on the Grammar of Graphics and is tightly integrated with Pandas. Ggplot can create data visualizations such as bar charts, pie charts, histograms, scatterplots, error charts, etc. using high-level API. Once it is told how to map the variables to aesthetics and what primitives to use, it takes care of all the details. The ggplot's method is not designed for creating highly customized graphics.

4. Altair:

- The Python library of Altair is a declarative statistical visualization library and has a simple API, is friendly and consistent and built on top of the powerful Vega-Lite visualization grammar. This elegant simplicity produces beautiful and effective visualizations with a minimal amount of code. The source of Altair is available on GitHub.

5. Bokeh:

- Bokeh is a data visualization library that provides detailed graphics with a high level of interactivity across various datasets, whether they are large or small. Bokeh is based on The Grammar of Graphics like ggplot but it is native to Python while ggplot is based on ggplot2 from R.
- Bokeh has 3 levels that can be used for creating visualizations. The first level focuses only on creating the data plots quickly; the second level controls the basic building blocks of the plot while the third level provides full autonomy for creating the charts with no pre-set defaults. This level is suited to the data analysts and IT professionals that are well versed in the technical side of creating data visualizations.

Summary

- NumPy means Numerical Python. It consists of many Python libraries used for Scientific computing in Python.
- You can create a NumPy array from a regular Python list or tuple using the np.array() function.

- 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, arrays allow a wide range of operations which can be performed on a particular array or a combination of Arrays.
- Universal functions in Numpy are simple mathematical functions that cover a wide variety of operations.
- input() function used to accept the list elements from a user in the format of a string separated by space.
- The Pandas library provides a really fast and efficient way to manage and explore data. Series and DataFrames help us to represent and manipulate data efficiently.
- Python statistics libraries are comprehensive, popular, and widely used tools that will assist in working with data.
- Data visualization is the technique to present the data in a pictorial or graphical format.
- Matplotlib is a data visualization library and 2-D plotting library of Python.
- The Pyplot module is a collection of command-style functions that allow you to use Matplotlib much like Matlab.
- Visualization Tools : Plotly, Seaborn, ggplot, Altair, Bokeh.

Check Your Understanding

1. The most important object defined in NumPy is an n-dimensional array type called?
   (a) ndarray (b) narray
   (c) nd_array (d) darray
2. What will be output for the following code?

```
import numpy as np
dt = dt = np.dtype('i4')
print (dt)
```

   (a) int32 (b) int64
   (c) int128 (d) int16
3. How we can find the type of numpy array in Python?
   (a) dtype (b) type
   (c) typei (d) itype
4. Pandas key data structure is called _____.
   (a) Keyframe (b) DataFrame
   (c) Statistics (d) Econometrics
5. A panel is a _____ container of data.
   (a) 1D (b) 2D
   (c) 3D (d) Infinite
6. Which among the following options can be used to create a DataFrame in Pandas?
   (a) A scalar value (b) An ndarray
   (c) A python dict (d) All of the above

7. Recommended way to load Matplotlib library is _____.
   (a) import matplotlib.pyplot as plt
   (b) import matplotlib.pyplot
   (c) import matplotlib as plt
   (d) import matplotlib
8. Which one of these is not a valid line style in Matplotlib?
   (a) '-' (b) '--'
   (c) '-.' (d) '<'
9. The plot method on Series and DataFrame is just a simple wrapper around _____.
   (a) gplt.plot() (b) plt.plot()
   (c) plt.plotgraph() (d) none of the mentioned
10. np.arange(2,8,1) will produce output as _____.
   (a) 2,3,4,5,6,7,8 (b) 2,3,4,5,6,7,8,9
   (c) 2,3,4,5,6,7 (d) 2,4,6,8

Answers

1. (a)	2. (a)	3. (a)	4. (b)	5. (c)	6. (d)	7. (a)	8. (d)	9. (b)	10. (c)

Practice Questions

Q.I Answer the following questions in short.

1. Mention different types of data structure in Pandas.
2. Define series in Pandas.
3. Write steps to create 1D and 2D array in NumPy.
4. What is Pyplot? Is it a Python Library?
5. What are Pandas? Where it is used?

Q.II Answer the following questions.

1. How to delete Indices, Rows or Columns from a Pandas DataFrame?
2. Describe features of NumPy.
3. Explain universal function used in NumPy.
4. Explain Statistical function used in NumPy.
5. Explain string functions used in Numpy.
6. Define terms: Series, DataFrame, Panel, Index Object.
7. Write steps to add and delete rows from DataFrame.
8. How to select entries from DataFrame?
9. Describe Missing data in Pandas.

Q.III Write a short note on:

1. NumPy
2. DataFrame in Pandas
3. Data visualization tools
4. Universal Array function
5. Matplotlib

Q.IV Write Program.

1. Write a Python program to find sum of elements in each row of a two-dimensional array of shape (3, 3).

■■■