SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES
(AUTONOMOUS)
MCA DEPARTMENT

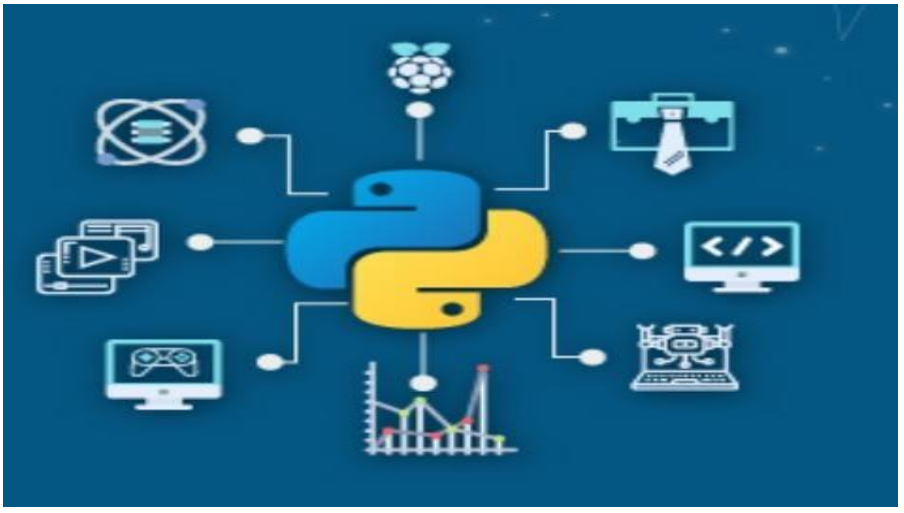
# LECTURE NOTES

**Subject Name:** PYTHON PROGRAMMING

**Year / Branch:** I MCA – II SEMESTER

**Regulation:** R24

**Prepared By:** M Rajesh, Assistant Professor



"Code is read much more often than it is written"-Guido Van Rossum

## Syllabus:

| I MCA – II SEMESTER | | | |
|---|---|---|---|
| **COURSE CODE:** | **24MCA122** | **CREDITS:** | **4** |
| **COURSE TITLE:** | **PYTHON PROGRAMMING** | **L-T-P:** | **4-0-0** |

***PREREQUISITES:*** *A Course on "C Programming or Java Programming " may be useful*

***COURSE EDUCATIONAL OBJECTIVES:***

***CEO1:*** *To impart the basics of Python and its IDEs*
***CEO2 :****To Understand the various Data Structures in Python*
***CEO3 :****To implement Object Oriented Programming through Python.*
***CEO4*** *To familiarize core libraries of Python*
***CEO5*** *To Explore streamlit for Designing GUI*

**UNIT – 1: BASICS OF PYTHON** — ***Lecture Hrs:8***

Python and its features, various IDEs of Python, variables and its scope, Input and Output statements, Comments, Operators,Operator Precedence, Selective statements and Iterative statements, Strings

**UNIT – II: FUNCTIONS AND PYTHON DATA STRUCTURES :** — ***Lecture Hrs:12***

Functions and Functional Programming , Recursive Functions , Recursive Vs Iterative Functions. **Built in Data structures** – List,tuple,sets and dictionary.

**UNIT – III: OBJECT ORIENTED CONCEPTS THROUGH PYTHON :** — ***Lecture Hrs:12***

OOPs Concepts – class, object, Encapsulation, Inheritance, polymorphism, Abstraction . Basic Programs on class-Object, polymorphism and Inheritance.**Explore Python Libraries** - Math and Random, Create your own Library

**UNIT – IV: PANDAS AND NUMPY**: — ***Lecture Hrs:12***

Introduction to pandas Data Structure, Essential functionality – Reindexing, Indexing ,selection and filtering, reshaping, summarizing and computing descriptive statistics, handling missing data, filter and query methods, grouping, reading and writing data in text format – read_csv,read_table. NumPy Basics- creating Arrays, universal functions – Basic unary and binary functions , File Input and Output with arrays – saving and loading text files, Linear Algebra – commonly used linalg functions

**UNIT – V: STREAMLIT -TO DEVELOP AN GUI** : — ***Lecture Hrs:12***

What is Streamlit, Features of Streamlit, Text and Table elements – Text Elements, Titles, Headers, Subheaders ,markdowns, tables, dataframes, Buttons and sliders – Buttons, RadioButton, Checkbox, Dropdown ,Multiselect, Progress bar, Slidder, Forms, Develop an Streamlit Application

***TEXT BOOKS***


1. *Charles Dierbach, "Introduction to Computer Science using Python: A Computational Problem-Solving Focus", Wiley India Edition, 2016.*
2. *"Core Python Programming ",Wesley J chun,2e,2012.*
3. *John V. Guttag., "Introduction to computation and programming using python: with applications to understanding data", PHI Publisher, 2016.*
4. *John Hunt, "A Beginners Guide to Python 3 Programming", Springer Publisheers,2020*

## *REFERENCE BOOKS*


1. *"Python for Data Analysis", wesMckinney,o'Reilly,2012*
2. *"Beginner's Guide to Streamlit with Python: Build Web-Based Data andMachine Learning Applications", Sujay Raghavendra,Dharwad, Karnataka, India,Apress,2023,*
3. *Allen B. Downey, "Think Python: How to Think Like a Computer Scientist", Second Edition, 2016*
4. *Charles Severance, "Python for everybody: exploring data in Python 3", Creative CommonsAttribution-Non Commercial Share Alike 3.0 Unported License, 2016.*


| ***COURSE OUTCOMES:*** *On successful completion of this course, students will be able to:* | | POs related to COs |
|---|---|---|
| **CO1** | **Learn and Execute** basic Python concepts | PO1,PO2 |
| **CO2** | **Analyze** various Python Functions and Data Structures | PO1,PO2,PO3,PO8 |
| **CO3** | **Experiment** Object Oriented Programming concepts in python | PO1,PO3, PO8 |
| **CO4** | **Explore** various core Python Libraries | PO1,PO2 ,PO3,PO4,PO8 |
| **CO5** | **Develop** an GUI using Streamlit | PO1,PO3,PO4, PO8 |

**CO-PO MAPPING** ( DETAILED; HIGH:3; MEDIUM:2; LOW:1)

| Course | POs / COs | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 |
|---|---|---|---|---|---|---|---|---|---|
| C202: Python Programming | C202.1 | 3 | 2 | - | - | - | - | - | - |
| | C202.2 | 3 | 3 | 3 | - | - | - | - | 3 |
| | C202.3 | 3 | - | 3 | - | - | - | - | 2 |
| | C202.4 | 3 | 3 | 2 | 3 | - | - | - | 3 |
| | C202.5 | 3 | - | 3 | 3 | - | - | - | 2 |
| | C202 | 3 | 2.67 | 2.75 | 3 | - | - | - | 2.5 |

# Unit–1: Basics of Python

## 1. Unit Overview

Unit introduces to the Python programming language and its key features. It explains the use of different Python IDEs for writing and executing programs. The unit covers variables and their scope, input and output statements, and the use of comments to improve code readability. It also introduces operators and operator precedence used in expressions. In addition, the unit discusses control flow statements, including selective (decision-making) and iterative (looping) statements, and concludes with the basic concept and operations of strings in Python.

## 2. Objectives of the Unit

After studying this unit, students will be able to:

1. To understand the basic concepts and features of Python programming language.
2. To learn about different Python IDEs used for writing and executing programs.
3. To understand variables, their declaration, and scope in Python.
4. To learn how to use input and output statements in Python programs.
5. To understand comments and different types of operators in Python.
6. To understand decision-making (selective statements) and looping (iterative statements) in Python.

## 3. Learning Outcomes

After completing this unit, students will be able to:

1. Understand the basic features of Python programming language.
2. Use Python IDEs to write and execute Python programs.
3. Define and use variables with proper scope in Python programs.
4. Apply input and output statements to interact with users.
5. Implement selective and iterative statements for decision making and looping.

## 4. Importance of studying this Unit:

- Builds a strong foundation in Python programming basics
- Improves logical thinking and problem-solving skills
- Helps understand variables, operators, and control statements
- Enables handling of input/output and string data

- Prepares for advanced topics and career opportunities

## 5. Key Concepts:

- **Variables & Scope** – Store data and control its access
- **Input & Output** – Interaction with user
- **Operators & Precedence** – Perform calculations in correct order
- **Selective Statements** – Decision making (if-else)
- **Iterative Statements** – Repetition using loops
- **Strings** – Handling text data

# Introduction to Python Programming Language:

- Python is a high-level, interpreted, and general-purpose programming language that was first released in 1991 by Guido van Rossum.
- It is designed to emphasize code readability, with a syntax that allows programmers to express concepts in fewer lines of code than would be possible in languages like C++ or Java.
- Python is used for a wide range of applications, including web development, data analysis, artificial intelligence, machine learning, and scientific computing.
- It is known for its simplicity, flexibility, and ease of use, making it a popular choice among developers of all skill levels.
- Python is an open-source language, which means that the source code is available to the public, and anyone can contribute to its development.
- The creator of Python, Guido van Rossum, was a big fan of Monty Python's Flying Circus, a popular British television comedy show that aired in the 1970s.

## 1. Python Features

### 1. Easy to Code

Python is a very high-level programming language, yet it is effortless to learn because Python syntax is very easy, as compared to other popular languages like C, C++, and Java.

### 2. Easy to Read

- Python code looks like simple English words.
- There is no use of semicolons or brackets, and the indentations define the code block.

### 3. Free and Open-Source

- Python is developed under an OSI-approved open source license.
- Hence, it is completely free to use, even for commercial purposes.
- It doesn't cost anything to download Python or to include it in your application.
- It can also be freely modified and re-distributed. Python can be downloaded from the official **Python website**.

### 4. Robust Standard Library

- Python has an extensive standard library available for anyone to use.
- This means that programmers don’t have to write their code for every single thing unlike other programming languages.

### 5. Interpreted

When a programming language is interpreted, it means that the source code is executed line by line, and not all at once.

### 6. Portable

- Python is portable in the sense that the same code can be used on different machines.
- Suppose you write a Python code on a Mac.
- If you want to run it on Windows or Linux later, you don't have to make any changes to it.

### 7. Object-Oriented and Procedure-Oriented

- A programming language is object-oriented if it focuses design around data and objects, rather than functions and logic.
- On the contrary, a programming language is procedure-oriented if it focuses more on functions (code that can be reused).
- One of the critical Python features is that it supports both object-oriented and procedure-oriented programming.

### 8. Expressive

- Python needs to use only a few lines of code to perform complex tasks.
- For example, to display Hello World, you simply need to type one line - print("Hello World").
- Other languages like Java or C would take up multiple lines to execute this.

### 9. Support for GUI

- One of the key aspects of any programming language is support for GUI.
- Python offers various toolkits, such as **Tkinter, wxPython and JPython**, which allows for fast and easy GUI development.

### 10. Dynamically Typed

- Many programming languages need to declare the type of the variable before using it.
- With Python, the type of the variable can be decided during runtime. This makes Python a dynamically typed language.

### 11. Simplify Complex Software Development

- Python can be used to develop both desktop and web apps and complex scientific and numerical applications.

- Python's data analysis features help to create custom big data solutions without so much time and effort.
- Python also helps to use the Python data visualization libraries and APIs to present data in a more appealing way.
- Several advanced software developers use Python to accomplish high-end AI and natural language processing tasks.

## 2.Various IDEs of Python

- IDE stands for **Integrated Development Environment**. It is a programming environment that contains a lot of things in a single package i.e. code editor, compiler, debugger.
- It has a user-friendly interface consisting of an editor and a compiler. We can write the code in the editor window and compile it using the compiler.
- Consequently, we can run it to check the output of the program on the output terminal.
- IDEs increase programmer productivity by introducing features like editing source code, building executables, and debugging.
- IDEs and code editors are tools that software developers use to write and edit code.
- IDEs, or Integrated Development Environments, are usually more feature-rich and include tools for debugging, building and deploying code.
- Code editors are generally more straightforward and focused on code editing. Many developers use IDEs and code editors, depending on the task.

### Features of IDE

1. **Code editor:**
   IDEs provide a code editor with advanced features such as syntax highlighting, code completion, and code refactoring. These features help developers write code more efficiently and accurately.
2. **Integrated debugger:**
   IDEs have a built-in debugger that allows developers to identify and fix errors in their code. This feature saves time by reducing the need for manual debugging.
3. **Version control integration:**
   IDEs can be integrated with version control systems such as Git or SVN, allowing developers to manage changes to their code and collaborate with others.
4. **Project management:**
   IDEs provide tools for managing projects, such as file organization, project templates, and code generation.

5. **Build automation:**
   IDEs can automate the build process, allowing developers to compile and run their code without having to use command-line tools.
6. **Testing tools:**
   IDEs provide testing tools that help developers test their code and identify defects early in the development cycle.
7. **Plugins and extensions:**
   IDEs often support plugins and extensions that can be used to extend their functionality, adding new features or integrating with other tools.

Overall, IDEs provide a complete environment for software development, making it easier and more efficient for developers to write, test, and debug their code.

## LIST OF IDE'S:

| | | | |
|---|---|---|---|
| **IDLE** | **PyCharm** | **Spyder** | **Thonny** |
| **Jupyter** | PyDev | Vim | GNU Emacs |
| **Wing** | Geany | Sublime Text | Eric Python |

**Python IDLE** is an integrated development environment (IDE) for Python that provides an interactive mode and a script mode for writing and running Python code.

Here are the differences between interactive mode and script mode in Python IDLE:

### 1) Interactive mode:

In interactive mode, you can type Python code directly into the IDLE shell, and the code is executed immediately after you press the Enter key. This allows you to experiment with Python code, test out ideas, and get immediate feedback.

### 2) Script mode:

In script mode, you write Python code in a file with a **.py** extension, and then run the file from within IDLE or from the command line. This mode is used for writing larger programs that require multiple lines of code and need to be saved for later use.

### Variables and its Scope

- A variable is a named identifier that represents a memory location used to store a value.

- It is a fundamental concept in programming and allows you to store and manipulate data.
- Variables are used to store different types of values, such as numbers, text, or objects.

- To create a variable in Python, you simply assign a value to a name using the assignment operator (=).
- In the above example, the variable message is created and assigned the value "Hello, World!".
- Now, you can use the variable message to access or modify the stored value.

```python
message = "Hello, World!"
```

- Python is a dynamically typed language, which means no need to explicitly declare the type of a variable.
- The type of the variable is inferred based on the value assigned to it.
- For example, in the previous code snippet, the variable message is inferred to be of type str(string)because it was assigned a string value.
- Variables can be reassigned with a different value or change its type later in the code:

```python
message = "Hello, World!"
print(message)  # Output: Hello, World!

message = "Welcome to Python"
print(message)  # Output: Welcome to Python

message = 42
print(message)  # Output: 42
```

- In this example, the variable message is initially assigned a string value, then reassigned to a different string value, and finally reassigned to an integer value.
- Variables provide a way to store and manipulate data throughout your program, making your code more flexible and dynamic.
- They play a crucial role in programming and are widely used to store retrieve information

## 4. Input and Output Statements

### input() function:

- This function prompts the user for input from the console.
- It returns the user's input as a string.
- It takes prompt message as an argument to the function.

## To get Integer Input:

```
# Prompting the user for input
age_input = input("Enter your age: ")

# Converting the input to an integer
age = int(age_input)

# Checking conditions based on user input
if age < 0:
    print("Please enter a valid age.")
elif age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

**Output**

```
Enter your age: 22
You are an adult.
```

## To get Floating Point Input:

```
# Taking input as float
# Typecasting to float
price = float(input("Price of each rose?: "))
print(price)
```

**Output**

```
Price of each rose?: 50.3050.3
```

## To get Multiple Input

To take multiple inputs from the user in a single line, the values entered by the user are split into separate variables for each value using the split() method.

```
# taking two inputs at a time
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)

# taking three inputs at a time
x, y, z = input("Enter three values: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)
```

Output

```
Enter two values: 5 10
Number of boys:  5
Number of girls:  10
Enter three values: 5 10 15
Total number of students:  5
Number of boys is :  10
Number of girls is :  15
```

## print() function:

- This function displays output to the console.
- Multiple arguments can be passed to print(), separated by commas.
- print() automatically adds a newline character at the end of the output.

```
print("Hello, World!")
```

Output

```
Hello, World!
```

## print single and multiple variables

Print statement can be used to print single and multiple variables

```python
# Single variable
s = "Bob"
print(s)

# Multiple Variables
s = "Alice"
age = 25
city = "New York"
print(s, age, city)
```

**Output**

```
Bob
Alice 25 New York
```

**String Concatenation:**

You can use the + operator to concatenate strings, but this can be less readable for complex formatting.

```python
name = "Smith"
age = 25
print("My name is " + name + " and I am " + str(age) + " years old.")
```

**Output**

My name is Smith and I am 25 years old.

**Formatted Output:**

- ✓ f-strings( Formatted string literals)
- ✓ These provide a concise and readable way to embed expressions within a string
- ✓ Prefix the string with braces {} f and enclose expressions in curly braces {}

Python

```python
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

**output:**

```
My name is Alice and I am 30 years old.
```

**Controlling output behavior:**

- `sep` argument: Specifies the separator between arguments (default is a space).
- `end` argument: Specifies what to print at the end of the line (default is a newline `\n`).

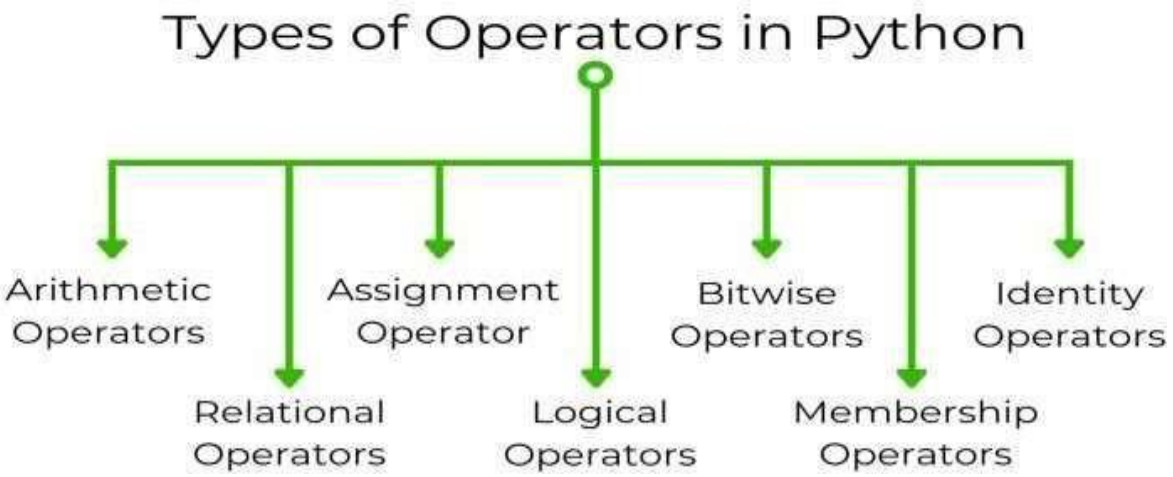
```
# end Parameter with '@'
print("Python", end='@')
print("GeeksforGeeks")


# Seprating with Comma
print('G', 'F', 'G', sep='')

# for formatting a date
print('09', '12', '2016', sep='-')

# another example
print('pratik', 'geeksforgeeks', sep='@')
```

Output

```
Python@GeeksforGeeks
GFG
09-12-2016
pratik@geeksforgeeks
```

## 5.Operator:

- Operators In General Are Used To Perform Operations On Values And Variables.
- **Operators:** These Are The Special Symbols. eg- + , * , /,
- **Operand:** It Is The Value or Variable On Which The Operator Is Applied. Eg :




## Operators in Python

| Operators | Type |
|---|---|
| +, -, *, /, % | Arithmetic operator |
| <, <=, >, >=, ==, != | Relational operator |
| AND, OR, NOT | Logical operator |
| &, \|, <<, >>, -, ^ | Bitwise operator |
| =, +=, -=, *=, %= | Assignment operator |

## Arithmetic Operators:

Python Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication and division.

## Program:

```
# Variables
a = 15
b = 4

# Addition
print("Addition:", a + b)

# Subtraction
print("Subtraction:", a - b)

# Multiplication
print("Multiplication:", a * b)

# Division
print("Division:", a / b)

# Floor Division
print("Floor Division:", a // b)

# Modulus
print("Modulus:", a % b)

# Exponentiation
print("Exponentiation:", a ** b)
```

Output

```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Floor Division: 3
Modulus: 3
Exponentiation: 50625
```

## Comparison Operators

In Python, comparison or relational operators compare the values. It either returns **True** or **False** according to the condition.

## Program

```
a = 13
b = 33

print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

Output

```
False
True
False
True
False
True
```

## Logical Operators:

Python logical operators perform **Logical AND, Logical OR, and Logical NOT** operations. It is used to combine conditional statements.

```
a = True
b = False
print(a and b)
print(a or b)
print(not a
```

Output

```
False
True
False
```

## Bitwise Operators:

Python Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Bitwise Operators in Python are as follows:

1. Bitwise NOT
2. Bitwise Shift
3. Bitwise AND
4. Bitwise XOR
5. Bitwise OR

```
a = 10
b = 4

print(a & b)
print(a | b)
print(~a)
print(a ^ b)
print(a >> 2)
print(a << 2)
```

Output

```
0
14
-11
14
2
40
```

## Identity Operators:

- In Python, **is** and **is not** are the identity operators; 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.
- **is** True if the operands are identical; **is not** True if the operands are not identical

```
a = 10
b = 20
c = a

print(a is not b)
print(a is c)
```

Output

```
True
True
```

## Membership Operators:

- In Python, **in** and **not in** are the membership operators that are used to test whether a value or variable is in a sequence.
- **in** True if value is found in the sequence; **not in** True if value is not found in the sequence.

```
x = 24
y = 20
list = [10, 20, 30, 40, 50]

if (x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")

if (y in list):
    print("y is present in given list")
else:
    print("y is NOT present in given list")
```

Output

```
x is NOT present in given list
y is present in given list
```

## Operator precedence:

| Operator | Symbol | Description |
|---|---|---|
| **Parentheses** | ( ) | Parentheses |
| **Exponent** | ** | Exponentiation |
| | ~X | Bitwise Nor |
| **Arithmetic** | *, /, //, % | Multiplication, Divide, Floor Division, Mod |
| | + , - | Addition, Subtraction |
| **Bitwise** | & | Bitwise And |
| | ^ | Bitwise Not |
| | \| | Bitwise Or |
| **Relational, Identity** | <, <=, >, >=, !=, ==, is, is not | Comparison, Identity Operators |
| **Logical Operator** | not | Not (Logical Operator) |
| | and | And (Logical Operator) |
| | or | Or (Logical Operator) |


1. Parentheses:

Python

```
result = (2 + 3) * 4  # Parentheses force addition first
print(result)  # Output: 20
```

Without parentheses:

Python

```
result = 2 + 3 * 4  # Multiplication happens before addition
print(result)  # Output: 14
```

2. Exponentiation:

Python

```
result = 2 ** 3 * 2  # Exponentiation happens before multiplication
print(result)  # Output: 16
```

3. Multiplication and Division vs. Addition and Subtraction:

```python
result = 10 + 5 * 2 - 8 / 4  # Multiplication and division before addition and
print(result)  # Output: 18.0
```

Calculation breakdown:

- `5 * 2 = 10`
- `8 / 4 = 2.0`
- `10 + 10 - 2.0 = 18.0`

4. Logical Operators:

```python
result = True and not False or False  # 'not' before 'and' before 'or'
print(result)  # Output: True
```

Calculation breakdown:

- `not False = True`
- `True and True = True`
- `True or False = True`

5. Comparison Operators:

```python
result = 5 > 3 and 2 < 4
print(result) #output: True
```

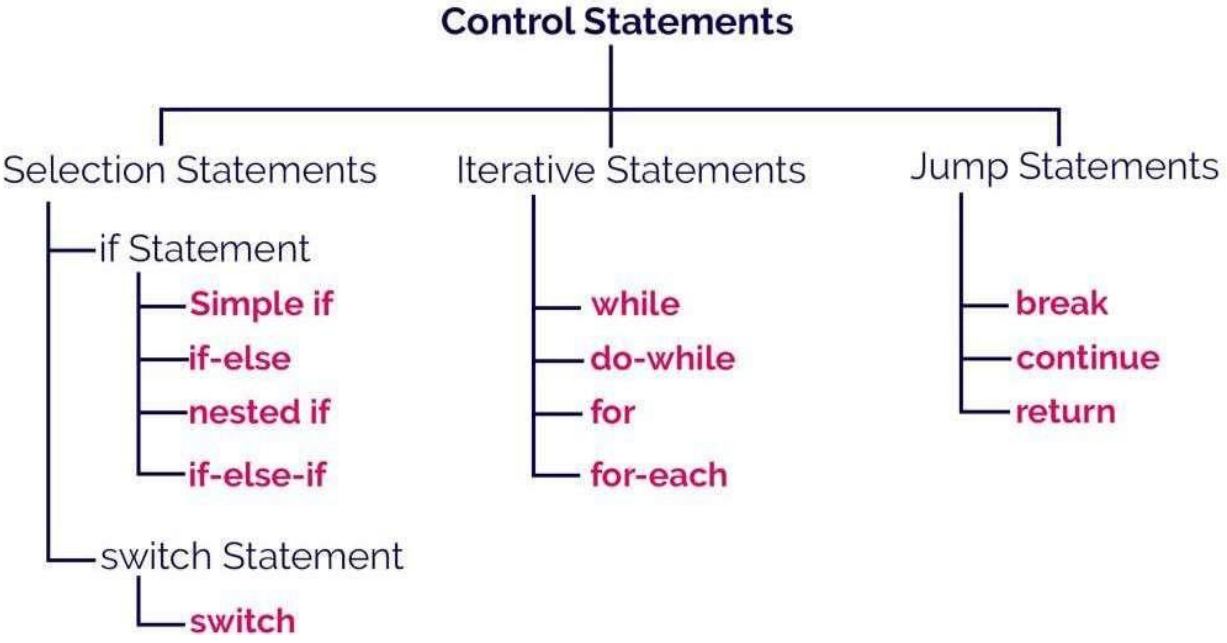
Comparisons are done before the and operator.

6. Assignment operators

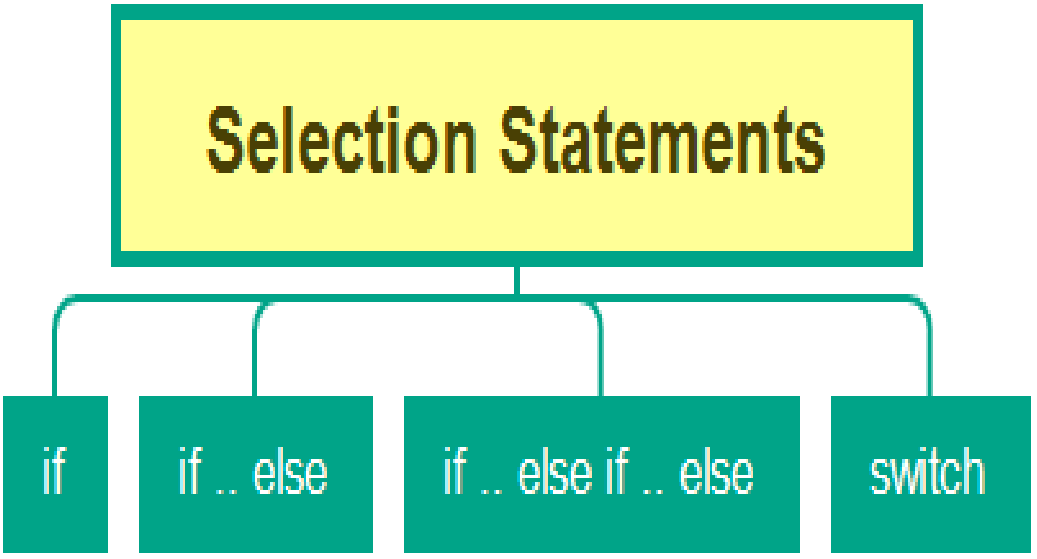
```python
x = 5
x += 3 * 2 #Multiplication happens before +=
print(x) #output: 11
```

## Selective and Iterative Statements:

In Python, conditional statements are used to control the flow of a program based on certain conditions.




## Selection Statements (or) Branching Statements:




## Python If statements

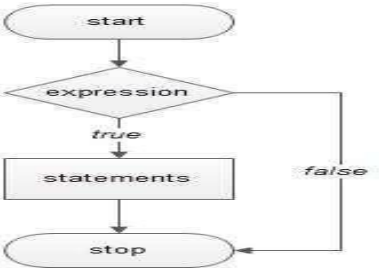
- This construct of python program consist of one if condition with one block of statements.
- When condition becomes true then executes the block given below it.

**Syntax:**

**if ( condition):**

**……..**

**Flowchart** ⇨




Find the Given Number is Even

Example-1: n=int(input("Enter n Value: "))

if (n%2==0):

print(n," is a Even Number")

```
1   n = int(input("Enter the n Value"))
2   if(n%2==0):
3       print(n,"is a Even Number")
4
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\Padmaja R> & "C:/Users/Padmaja R/AppData/Local/Programs/Python/Python311/python.exe" "c:/Users/Padmaja R/OneDrive/Desktop/Even.py"
Enter the n Value4
4 is a Even Number
PS C:\Users\Padmaja R>
```

## Python if - else statements:

This construct of python program consist of one if condition with two blocks. When condition becomes true then executes the block given below it. If condition evaluates result as false, it will executes the block given below else.

## Syntax:

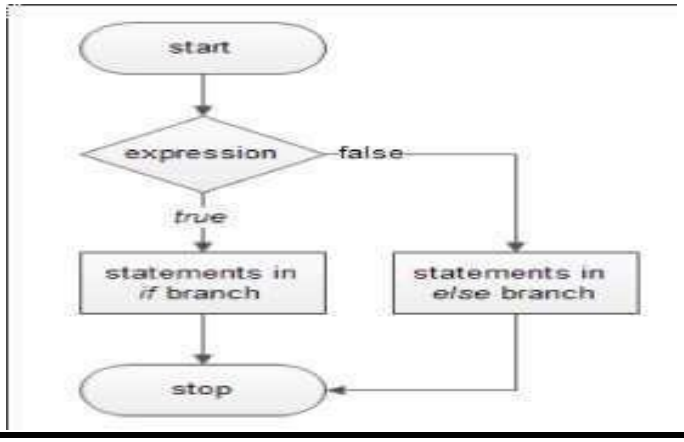
**if ( condition):**

**.....**

**else:**

**.....**

**Flow chart:**



```
1   n = int(input("Enter the n Value"))
2   if(n%2==0):
3       print(n,"is a Even Number")
4   else:
5       print(n,"is a Odd Number")
6
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Padmaja R> & "C:/Users/Padmaja R/AppData/Local/Programs/Python/Python311/python.exe" "c:/Users/Padmaja R/OneDrive/Desktop/Even.py"
Enter the n Value7
7 is a Odd Number
PS C:\Users\Padmaja R>
```

**To Check Given Number is Even or Odd**

**Example-1:**

```
n=int(input("Enter n Value: "))
if ( n%2==0):
print(n, "is Even")
else:
print(n, "is Odd")
```

**To Check Given age is eligible for voting or not**

**Example-2:**

```
Age=int(input("Enter Age: "))
if ( age>=18):
print("You are eligible for vote")
else:
print("You are not eligible for vote")
```

**Python Ladder if else statements (if-elif-else):**

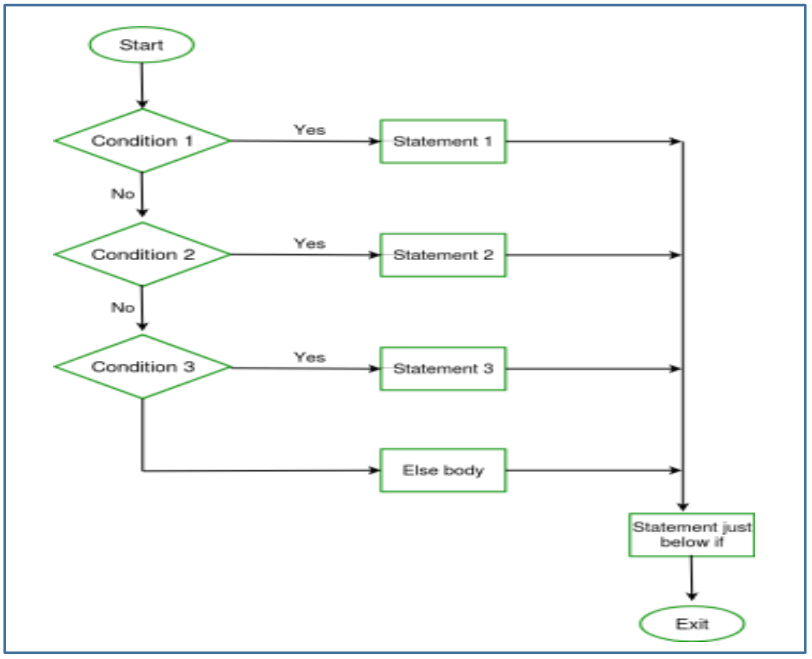
This construct of python program consist of more than one if condition. When first condition evaluates result as true then executes the block given below it. If condition evaluates result as false, it transfer the control at else part to test another condition. So, it is multi-decision making construct.

**Syntax:**

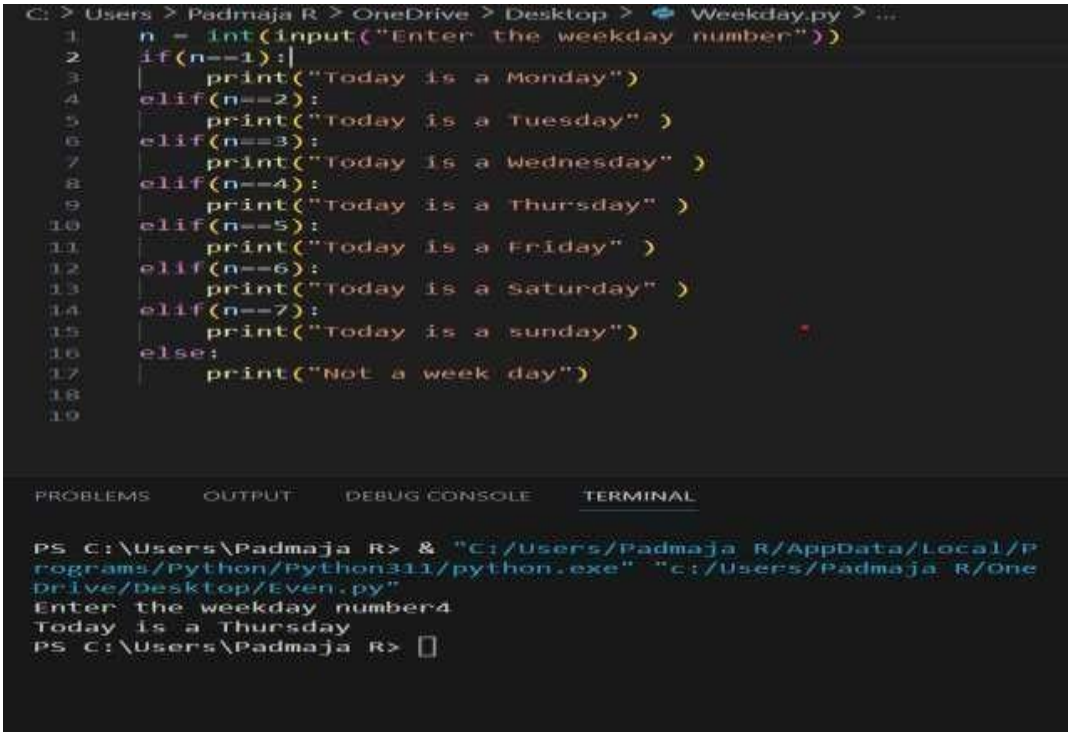
```
if ( condition-1):
        ……..
```

```
        ……..
elif (condition-2):
        ……..
        ……..
elif (condition-3):
        ……..
        ……..

 else:
        ……..
        ……..
if ( condition-1):
        ……..
        ……..
 elif (condition-2):
        ……..
        ……..
elif (condition-3):
        ……..
        ……..
 else:
        ……..
        ……..
```

**Flow Chart:**



## Program:



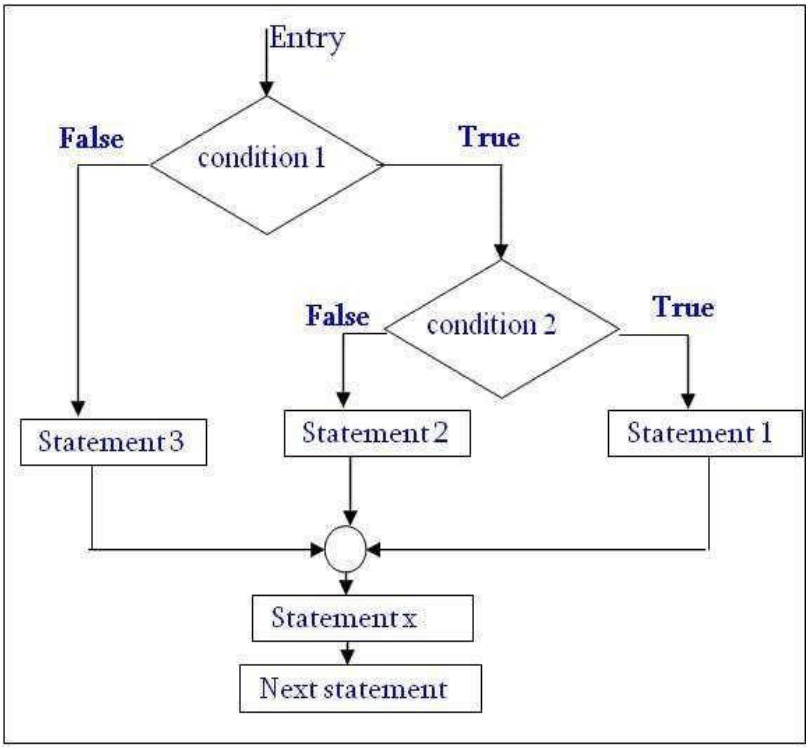

## Python Nested if statements:

It is the construct where one if condition take part inside of other if condition. This construct consist of more than one if condition. Block executes when condition becomes false and next condition evaluates when first condition became true. So, it is also multi-decision making construct.

### Syntax:

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

## Flowchart:




## Example:

**To check the given number is Negative, Zero or Positive**

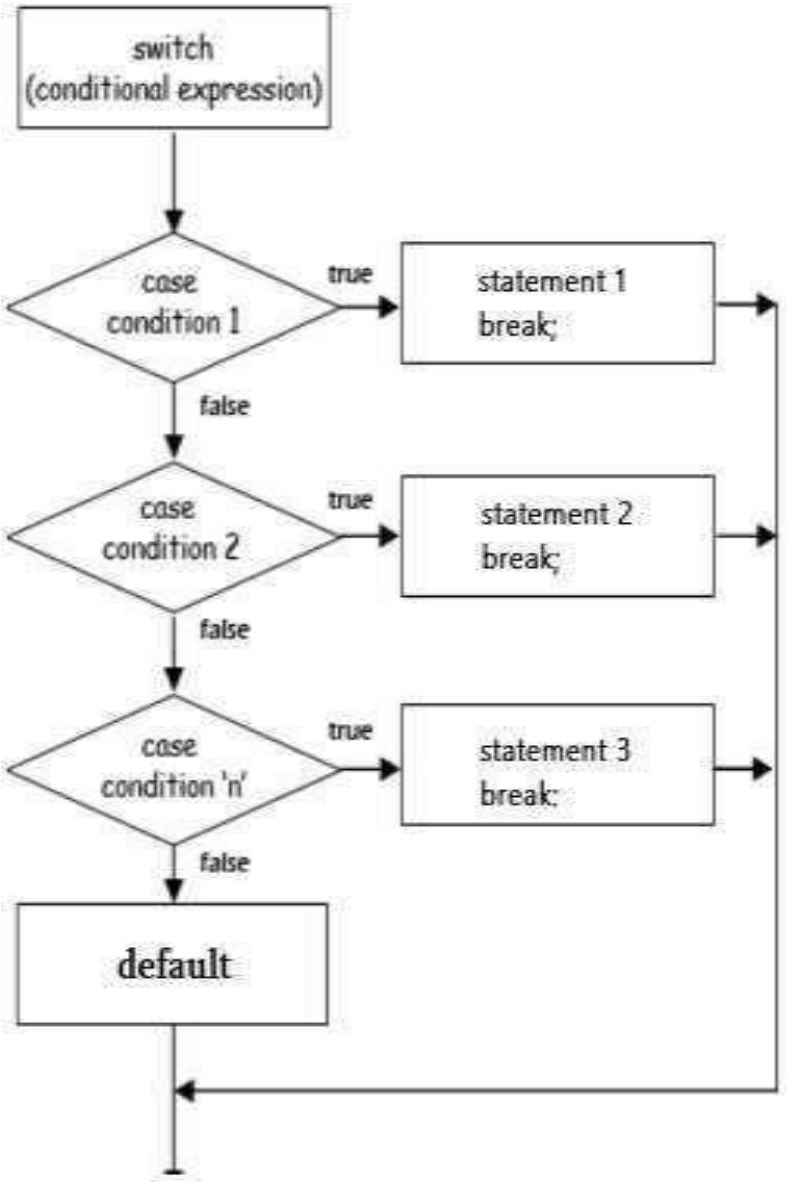
```
n = int(input("Enter a number: "))

if(n >= 0):
        if(n == 0):
                print("Number is Zero")
        else:
                 print("Number is Positive")
else:
        print("Number is Negative")
```

## Switch Case:

- Python Switch Case is a selection control statement.
- The switch expression is evaluated once. The value of the expression is compared with the values of each case; if there is a match, the associated block of code is executed.
- Then it makes a decision based on whether the condition is true or not.
- If the condition is true, it evaluates the indented expression; however, if the condition is false, the indented expression under else will be evaluated.
- When we need to run several conditions, you can place as many **elif** conditions as necessary between the **if** condition and the **else** condition.

- Switch case statements are a substitute for long **if statements** that compare a variable to several integral values.
- The switch statement is a **multiway branch statement**. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

## Flowchart:




Since Python 3.10, we can now use a new syntax to implement this type of functionality with a match case. The match case statement allows users to implement code snippets exactly to switch cases.
expression = value

Syntax:

```
match subject:
    case <pattern_1>:
        <action_1>
    case <pattern_2>:
        <action_2>
    case <pattern_3>:
        <action_3>
    case _:
        <action_wildcard>
```

```python
n = int(input("Enter the weekday number"))
match n:
    case 1:
        print("Today is Monday")
    case 2:
        print("Today is Tuesady")
    case 3:
        print("Today is Wednesday")
    case 4:
        print("Today is Thursday")
    case 5:
        print("Today is Friday")
    case 6:
        print("Today is Saturday")
    case 7:
        print("Today is Sunday")
    case _1:
        print("Not a Week day")

```

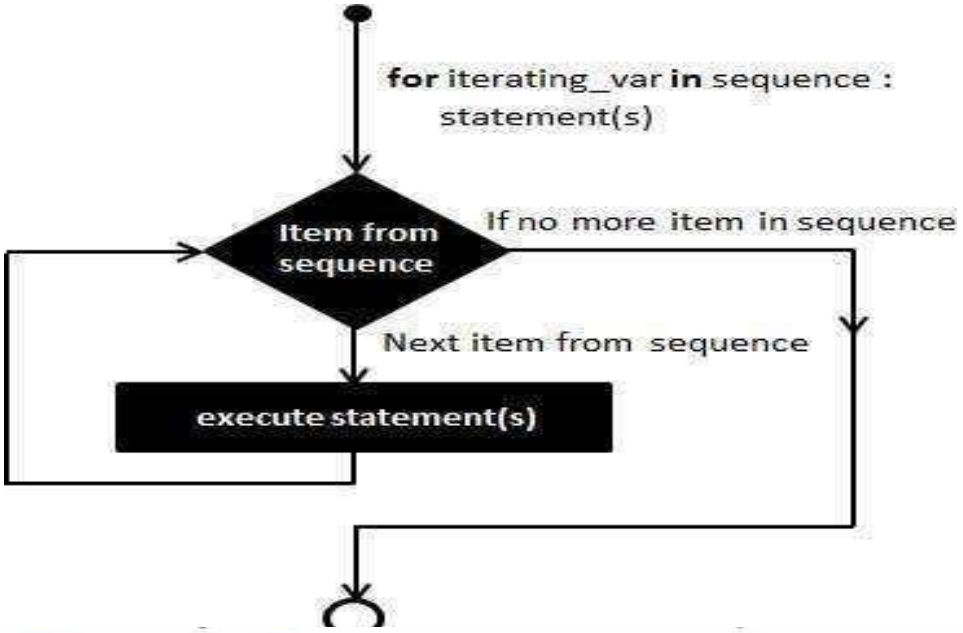
PROBLEMS 7 OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Padmaja R> & "C:/Users/Padmaja R/AppData/Local/Programs/Python/Python311/python.exe" "c:/Users/Padmaja R/OneDrive/Desktop/matchCase.py"
Enter the weekday number6
Today is Saturday
PS C:\Users\Padmaja R>
```

## For Loop:

- The for loop is used in the case where a programmer needs to execute a part of the code until the given condition is satisfied.
- The for loop is also called a pre-tested loop.
- It is best to use for loop if the number of iterations is known in advance.

## Flow chart:




```
# Iterate from 0 to 4
for i in range(5):
    print(i) # prints from 0 to 4

# Iterate from 2 to 9
for i in range(2, 10):
    print(i) # prints 2,3,4,5,6,7,8,9

# Iterate from 0 to 9, with a step of 2
for i in range(0, 10, 2):
    print(i) # prints 0,2,4,6,8
```

- **Python For Loops** are used for iterating over a sequence like **lists, tuples, strings, and ranges.**
- The range() function is commonly used with for loops to generate a sequence of numbers.
- It can take one, two, or three arguments:
- **range(stop):** Generates numbers from 0 to stop-1.
- **range(start, stop):** Generates numbers from start to stop-1.
- **range(start, stop, step):** Generates numbers from start to stop-1, incrementing by step.

**Program:**

```
# Iterate from 0 to 4
for i in range(5):
    print(i) # prints from 0 to 4

# Iterate from 2 to 9
for i in range(2, 10):
    print(i) # prints 2,3,4,5,6,7,8,9

# Iterate from 0 to 9, with a step of 2
for i in range(0, 10, 2):
    print(i) # prints 0,2,4,6,8
```

- For loop to iterate over a string and print each character on a new line.
- The loop assigns each character to the variable i and continues until all characters in the string have been processed.

```
s = "Geeks"
for i in s:
    print(i)
```

**Output**

```
G
e
e
k
s
```

## Iterating from Sequence

For val in sequence:

Statement

- Here, val accesses each item of sequence on each iteration.
- The loop continues until we reach the last item in the sequence

**Example:**

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print 'Current fruit :', fruit

print "Good bye!"
```

**Output:**

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

## Iterating by Sequence Index:

An alternative way of iterating through each item is by index offset into the sequence itself

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]

print "Good bye!"
```

When the above code is executed, it produces the following result –

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, len() is a built-in function, which provides the total number of elements in the tuple as well as the range() is another built-in function to give us the actual sequence to iterate over.

## Using else Statement with For Loop:

- Python supports to have an else statement associated with a loop statement
- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

```
digits = [0, 1, 5]

for i in digits:
    print(i)
else:
    print("No items left.")
```

Run Code >>

**Output**

```
0
1
5
No items left.
```

Here, the `for` loop prints all the items of the `digits` list. When the loop finishes, it executes the `else` block and prints `No items left.`

> **Note:** The else block will not execute if the for loop is stopped by a break statement.

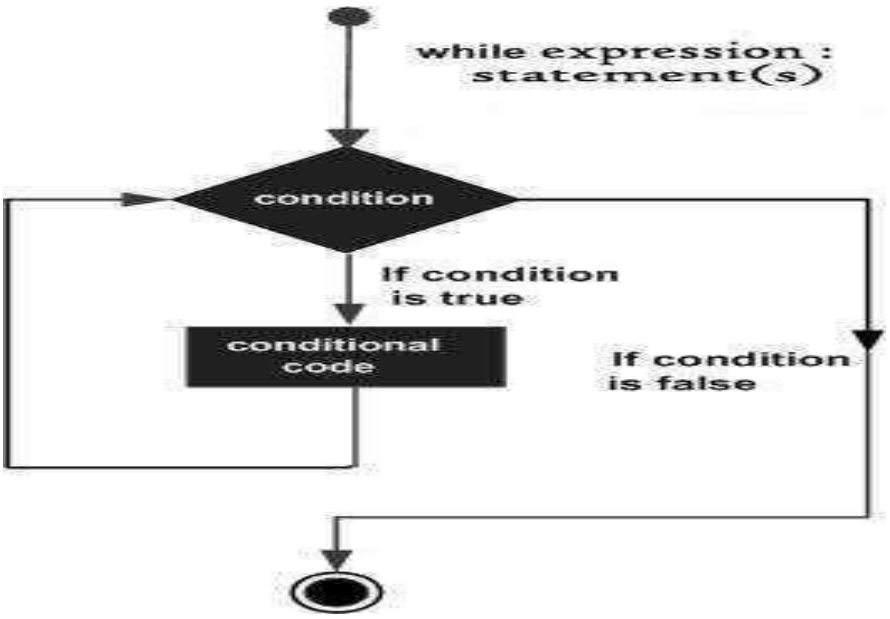
## While Loop

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

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

- Here, statement(s) may be a single statement or a block of statements.
- The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code.
- Python uses indentation as its method of grouping statements.

**Flow chart:**




**Program:**

```
i = 1
while i <= 5:
    print(i)
 i += 1
```

**Output:**

```
1
2
3
4
5
```

## Nested Loop

Python programming language allows to use one loop inside another loop.

### Syntax

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

The syntax for a **nested while loop** statement in Python programming language is as follows –

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

**Nested For Example:**

```
for i in range(2,11):
    c=0
    for j in range(1,i+1):
        if(i%j==0):
            c=c+1
    if(c==2):
        print(i,"is Prime")
```

**Output:**

```
2 is Prime
3 is Prime
5 is Prime
7 is Prime
```

**Nested While Example:**

```
i=2
while(i<=10):
    c=0
    j=1
    while(j<=i):
        if(i%j==0):
            c=c+1
        j=j+1
    if(c==2):
        print(i,"is a prime number")
    i=i+1
```

**Output:**

```
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
```

**Jumping statements:**

Jumping statements are used to control the flow of loops by skipping or terminating iterations.

Types:

**Break:** Terminates the loop immediately when the condition is met.

Program:

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

Output:

**1**
**2**

**Continue:** Skips the current iteration and continues with the next iteration of the loop.

Program:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output:

**1**
**2**
**4**
**5**

## STRING HANDLING:

- Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes.
- The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.
- Each character is encoded in the ASCII or Unicode character.
- So we can say that Python strings are also called the collection of Unicode characters.

strings in python can be created using single quotes or double quotes or even triple quotes

```
# Creating a String with single Quotes
String1 = 'Welcome to the Python World'
print("String with the use of Single Quotes: “ ,String1)


# Creating a String # with double Quotes
String1 = “Python Strings"
```

print("String with the use of Double Quotes: “,String1)

# Creating a String # with triple Quotes

String1 = '''I'm a Geek and I live in a world of "Geeks"'''

print("String with the use of Triple Quotes: “,String1)

In Python, individual characters of a String can be accessed by using the method of Indexing.


**Program:**

# **Python Program to Access characters of String**

String1 = “Python World" print("Initial String: “,String1)

# **Printing First character**

print("First character of String is: “,String1[0])

# **Printing Last character**

print("Last character of String is: “,String1[-1])

**Output:**

Initial String: Python World

First character of String is: P

Last character of String is: d

- Python Program to get a string made of the first 2 and last 2 given string. characters of a

str = "PythonWorld"

print(str[:2]+str[-2:]) \\ prints Pyld

- Python program to get a string from a given string where all occurrences of its first char have been changed to '$', except the first char itself.

```
str = "restart"
char = str[0]
str = str.replace(char,'$')
str = char +str[1:]
print(str) \\ prints resta$t
```

- Python program to get a single string from two given strings, separated by a space and swap the first two characters of each string

```
str1 = 'abc'
str2 = 'xyz'
new_str1 = str2[:2]+str1[2:]
new_str2 = str1[:2]+str2[2:]
print(new_str1+' '+new_str2)
```

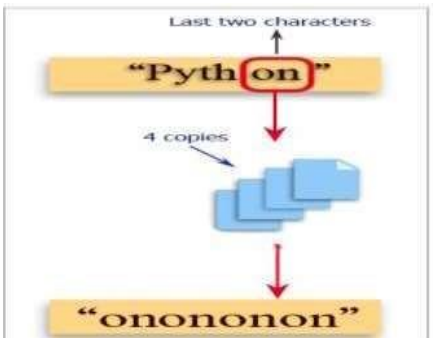
- Python program to remove the nth index character from a nonempty string.

```
str = input("Enter the String : ")
n = int(input("Enter the position of the character to be removed : "))
print(str[:n]+str[n+1:])
```

Write a Python program to change a given string to a newly string where the first and last chars have been exchanged.


```
str ='abcd'
print(str[-1:]+str[1:-1]+str[:1])
```

Write a Python function to get a string made of 4 copies of the last two characters of a specified string (length must be at least 2)




```
str='Python'
sub_str = str[-2:]
res = sub_str*4 print(res)
```

## Question Bank:

| UNIT – 1: BASICS OF PYTHON | | |
|---|---|---|
| **Python and its features, various IDEs of Python, variables and its scope, Input and Output statements, Comments, Operators, Operator Precedence, Selective statements and Iterative statements, Strings** | | |
| **PART –A** | | |
| **Q.No.** | **Questions** | **Blooms Taxonomy Level** |
| 1 | List out python Features | L1 |
| 2 | List out Python IDE's | L1 |
| 3 | Differentiate between variable and constant | L2 |
| 4 | List of Binary Operators | L1 |
| 5 | List out selective statements | L1 |
| 6 | List out Looping statements | L1 |
| 7 | List out Jumping Statements | L1 |
| 8 | What is meant by operator precedence | L2 |
| **PART –B** | | |
| 1 | Discuss Python features in detail | L2,L3 |
| 2 | What are python IDEs and explain how benefits of IDR's over text editors | L1,L2 |
| 3 | What is variable and illustrate the scope of variable with example | L2,L3 |
| 4 | Apply types of python operators with examples | L3 |
| 5 | Explain Python operator precedence in detail | L2 |
| 6 | Summarize various python selective statement with examples | L2,L3 |
| 7 | Classify various python iterative statements with examples | L2,L3 |
| 8 | Apply various python string functions and string slicing with examples | L3 |

# PYTHON PROGRAMMING
# UNIT – II
# FUNCTIONS AND PYTHON DATA STRUCTURES

# UNIT – II : FUNCTIONS AND PYTHON DATA STRUCTURES

## 1. Unit Overview:

This unit covers key Python concepts such as functions, recursion, and built-in data structures. Functions help create reusable and organized code, while recursion involves a function calling itself until a base condition is met. Iterative functions use loops like for and while and are generally more efficient. The unit also includes data structures like lists (ordered and mutable), tuples (ordered and immutable), sets (unordered with unique elements), and dictionaries (key-value pairs), which are used to store and manage data effectively

## 2. Objectives of the Unit:

By the end of this unit, students should be able to:

- To understand the concept and importance of **functions** in Python programming.
- To learn how recursive functions work and apply them to solve problems.
- To differentiate between **recursive and iterative** approaches.
- To gain knowledge of built-in data structures **like lists, tuples, sets, and dictionaries**.
- To develop skills in organizing and managing data efficiently using appropriate data structures.

## 3. Learning Outcomes:

After completing this unit, students should will be able to:

- Understand the concept of functions and their usage
- Apply recursion to solve problems effectively
- Differentiate between recursive and iterative methods
- Use built-in data structures like lists, tuples, sets, and dictionaries
- Organize and manage data efficiently in Python programs

## 4. Importance of Studying this Unit:

- Helps in writing clean, modular, and reusable code using functions
- Improves problem-solving skills through recursion and logical thinking
- Enhances understanding of efficient programming techniques
- Provides knowledge of essential data structures for data handling
- Forms a strong foundation for advanced programming and real-world applications

## 5. Key Concepts:

- **Functions:** Reusable blocks of code for modular programming.
- **Types of Functions:** Built-in and user-defined functions.
- **Functional Programming:** Writing clean and reusable code using functions.
- **Recursive Functions:** Functions that call themselves

- **Recursive vs Iterative:** Recursion uses function calls; iteration uses loops.
- **Lists:** Ordered and mutable collections.
- **Tuples:** Ordered and immutable collections.
- **Sets:** Unordered collections with unique elements.
- **Dictionaries:** Key-value pair data structures.

## Introduction:

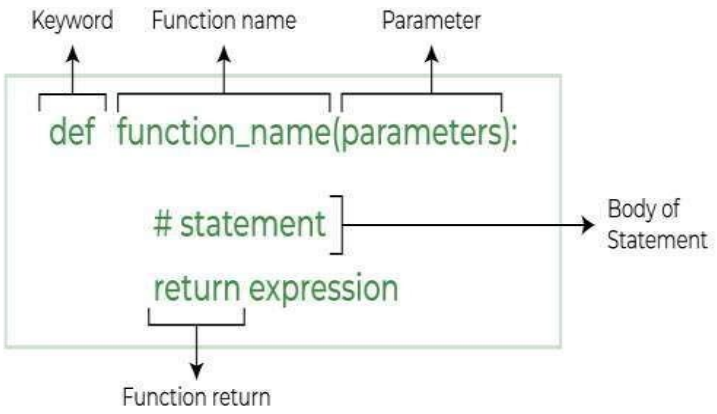
- In Python, a function is a block of reusable code that performs a specific task.
- It allows you to organize your code into modular and reusable units, making your programs more structured and easier to understand.

Benefits of Using Functions

- Increase Code Readability
- Increase Code Reusability

## Python Function Declaration:

It's the keyword used to define a function in Python.




**function_name:**

It's the name given to the function. You can choose any valid name that follows Python's naming conventions.

**parameters:**

They are optional input values that can be passed to the function for it to perform its tasks . Parameters are placed inside parentheses and separated by commas. If a function doesn't require any input, you can leave the parentheses empty.

**Function body:**

It consists of one or more statements that are indented under the function definition. These statements define the tasks the function performs.

**Return statement:**

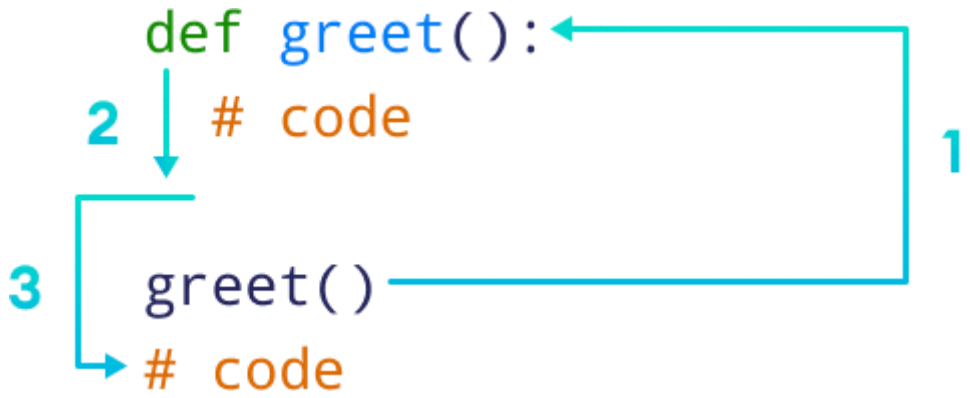
It's optional and is used to specify the value that the function should return when it is called. If a function doesn't have a return statement, it will implicitly return None

**Python Functions are classified into**

1) **Function with No Arguments**
2) **Function with Arguments**
   - Default arguments
   - Keyword arguments
   - Positional arguments
   - Arbitrary Keyword arguments
3) **Function with return statement**
4) **Function without Return statement**

## Function with No Arguments:

writing Function with No arguments




**Sample Code :**

```
1  # function definition
2  def greet():
3      print("Welcome to Python Functions")
4  # calling the function with no argument
5  greet()
6  print("outside of the function")
```

**Output:**

```
Welcome to Python Functions
outside of the function
```

## Function with Arguments:

A function with arguments is a function that accepts input values (parameters) to perform a specific task. These arguments allow the function to work with different data and produce results based on the given inputs, making the function more flexible and reusable.

## Function without Arguments:

A function without arguments does not take any input values. It performs a task and returns or displays a result without depending on external input.

Program:

```
def greet():
    print("Hello, Welcome!")
```

```
greet()
```

Output:

```
Hello, Welcome!
```

## Function with Arguments:

A function with arguments takes input values (parameters) from the user or program. It uses these inputs to perform operations and produce results, making the function more flexible and reusable.

Program:

```
def add(a, b):

  result = a + b
  print("Sum:", result)

add(5, 3)
```

**Output:**

```
Sum: 8
```

## Function with Arguments:

### Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments

Program:

```
# function with default arguments
def sum(a,b=3):
    c=a+b
    return c
# calling the function
res1 = sum(2)
print("sum of 2 numbers",res1)
res2 = sum(4,5)
print("sum of 2 numbers",res2)
```

Output:

```
sum of 2 numbers 5
sum of 2 numbers 9
```

## Keyword arguments:

Function by specifying the argument name along with its value (or) The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Program:

```
1  # function with Keyword arguments
2  def greet(name,message):
3      print("hello",name,message)
4  greet("Alice","How are you") # positional Arguments
5  greet(message = "how are u",name="jones") # Keyword Argument
```

Output:

```
hello Alice How are you
hello jones how are u
```

## Positional arguments

Positional arguments, also known as positional parameters, are a type of argument in programming that are passed to a function or method based on their position or order. In contrast to keyword arguments, which are identified by their names, positional arguments rely on their position in the
argument list.

Program:

```
1  # function with Keyword arguments
2  def greet(name,message):
3      print("hello",name,message)
4  greet("Alice","How are you") # positional Arguments
5  greet(message = "how are u",name="jones") # Keyword Argument
```

Output:

```
hello Alice How are you
hello jones how are u
```

## Arbitrary Keyword arguments:

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- To handle this kind of situation, we can use arbitrary arguments in Python.
- Arbitrary arguments allow us to pass a varying number of values during a function call

- We use an asterisk (*) before the parameter name to denote this kind of argument.
- In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols.
- There are two special symbols:
  - *args in Python (Non-Keyword Arguments)
  - **kwargs in Python (Keyword Arguments)

```
# program to find sum of multiple numbers def find_sum(*numbers):
        result = 0
        for num in numbers:
        result = result + num
        print("Sum = ", result)

# function call with 3 arguments
        find_sum(1, 2, 3)

# function call with 2 arguments
        find_sum(4, 9)
```

**Output:**

```
        Sum =6
        Sum=13
```

**Variable Length Arguments**

```
        def add_numbers(*args):
                total = 0
                for num in args:
                        total += num
                print("Sum:", total)

        add_numbers(1, 2, 3)
        add_numbers(10, 20, 30, 40)
```

**Output:**

```
        Sum: 6
        Sum: 100
```

**Variable Length Keyword Arguments**

```
        def display_info(**kwargs):
                for key, value in kwargs.items():
                print(key, ":", value)

        display_info(name="Rajesh", age=22)
        display_info(city="Nellore", country="India", pincode=524001)
```

**Output:**

```
        name : Rajesh
```

age : 22
city : Nellore
country : India
pincode : 524001

**Function without Return Statement:**

A function without a return statement is a function that performs a specific task but does not return any value to the caller. It simply executes the given statements and may display output using print statements.

**Program:**

```
1  # function without Return Statement
2  def sum(x,y):
3      print("sum of 2 numbers",x+y)
4  sum(6,8)
```

**Output:**

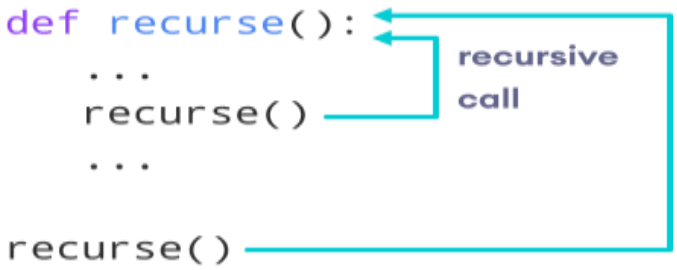
Sum of  2 numbers 14

**Function with Return Statement:**

```
7  #function with Return Statement
8  def Sum2(x,y):
9      z=x+y
10     return z
11 res = Sum2(6,8)
12 print("sum of 2 numbers",res)
```

## Recursive Functions:

A recursive function is a function that calls itself to solve a problem. It breaks the problem into smaller parts and continues until a base condition is reached.




- Factorial of a number is the product of altheintegersfrom1to that number.
- For example, the factorial of 6(denoted as 6!) is 1*2*3*4*5*6=720

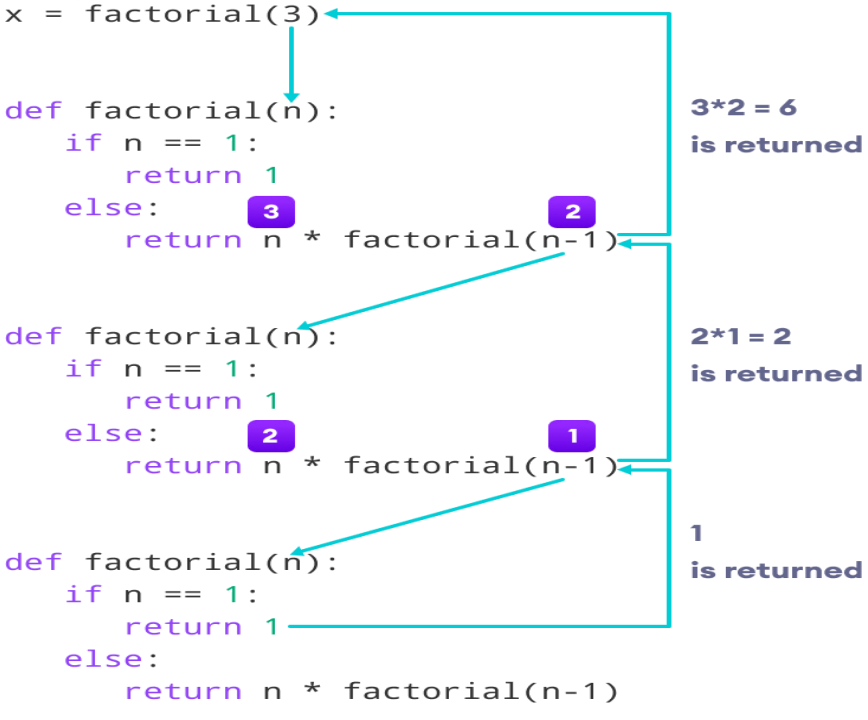
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

num = 6
result = factorial(num)
print("Factorial of", num, "is", result)
```

**Output:**

```
Factorial of 6 is 720
```

- In the above example, factorial() is a recursive function as it calls itself.
- When we call this function with a positive integer, it will recursively call itself by decreasing the number.
- Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

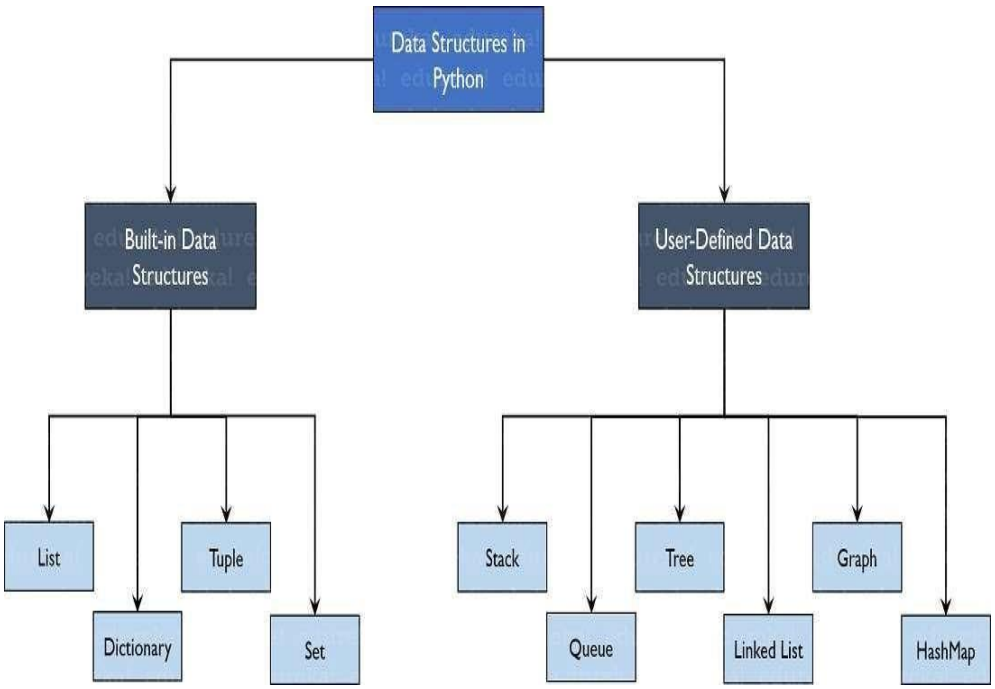
```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```



## List Structures:

- A list is a collection of elements stored in a single variable
- Lists are ordered, meaning elements have a fixed position
- Lists are mutable, so elements can be changed after creation
- Lists allow duplicate values
- Elements can be of different data types (int, string, float, etc.)
- Lists are defined using square brackets []
- Elements can be accessed using index values starting from 0

## Types of Data Structures in python:

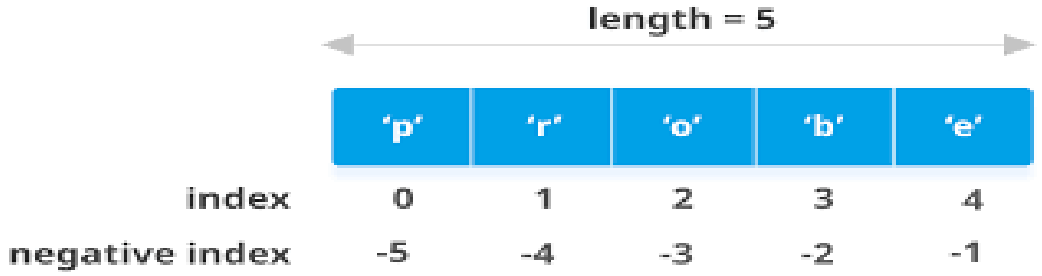



- As the name suggests, these data structures are built-in in Python, which makes programming easier and helps programmers use them to obtain solutions faster.
- Let's discuss each of them in detail.

## List

- List are one of the built-in data structures in Python
- Lists are ordered collection of data
- Lists are mutable, meaning that after creation we can modify their elements
- Lists allow duplicate elements
- Lists are used to store data of different data types in a sequential manner
- There are addresses assigned to every element of the list, which is called as index

- The index value starts from 0 and goes on until the last element called the positive index
- There is also negative indexing which starts from -1, enabling you to access elements from last to first




**List Operations**

- **Creation:** Creating a list using square brackets []
- **Accessing:** Accessing elements using index values
- **Updating:** Modifying elements in a list

- **Appending:** Adding elements at the end using append()
- **Inserting:** Adding elements at a specific position using insert()
- **Deleting:** Removing elements using remove(), pop(), or del
- **Slicing:** Accessing a range of elements using slicing [start:end]
- **Length:** Finding number of elements using len()
- **Sorting:** Arranging elements using sort()
- **Reversing:** Reversing list elements using reverse()

**1. Creating Lists:**

Creating a list means defining and initializing a collection of elements in Python using square brackets `[]`.

**Program:**

```
# Creating lists
list1 = [1, 2, 3, 4]
list2 = ["apple", "banana", "cherry"]
list3 = [10, "hello", 3.5]

print("List1:", list1)
print("List2:", list2)
print("List3:", list3)
```

**Output:**

```
List1: [1, 2, 3, 4]
List2: ['apple', 'banana', 'cherry']
List3: [10, 'hello', 3.5]
```

**2. Adding elements in List:**

Adding elements in a list means inserting new values into an existing list using methods like append(), insert(), or extend().

**Program:**

```
# Creating a list
my_list = [1, 2, 3]

# Adding element at the end
my_list.append(4)

# Inserting element at specific position
my_list.insert(1, 10)

# Adding multiple elements
my_list.extend([5, 6])

print("Updated List:", my_list)
```

**Output:**

Updated List: [1, 10, 2, 3, 4, 5, 6]

### 3. Removing Elements from List:

Removing elements from a list means deleting items from an existing list using methods like remove(), pop(), or del.

**Program:**

```
# Creating a list
my_list = [10, 20, 30, 40, 50]

# Removing a specific element
my_list.remove(30)

# Removing element by index
my_list.pop(1)

# Deleting element using del
del my_list[0]

print("Updated List:", my_list)
```

**Output:**

Updated List: [40, 50]

### 4. Modifying List Elements:

Modifying list elements means changing the value of existing items in a list using their index positions.

**Program:**

```
# Creating a list
my_list = [10, 20, 30, 40]

# Modifying elements
my_list[1] = 25
my_list[3] = 45

print("Modified List:", my_list)
```

**Output:**

Modified List: [10, 25, 30, 45]

### 5. Accessing Elements in List:

Accessing elements in a list means retrieving values from the list using index positions.

**Program:**

```
# Creating a list
my_list = [10, 20, 30, 40, 50]


# Accessing elements
print("First element:", my_list[0])
print("Third element:", my_list[2])
print("Last element:", my_list[-1])
```

**Output:**

```
First element: 10
Third element: 30
Last element: 50
```

### 6. Sorting in List

Sorting in a list means arranging the elements in a specific order, either ascending or descending.

**Program:**

```
# Creating a list
my_list = [40, 10, 30, 20]

# Sorting in ascending order
my_list.sort()
print("Ascending Order:", my_list)

# Sorting in descending order
my_list.sort(reverse=True)
```

```
print("Descending Order:", my_list)
```

**Output:**

```
Ascending Order: [10, 20, 30, 40]
Descending Order: [40, 30, 20, 10]
```

### 7. Concatenating Lists

Concatenating lists means combining two or more lists into a single list.

**Program:**

```
# Creating lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Concatenating lists
result = list1 + list2

print("Concatenated List:", result)
```

**Output:**

```
Concatenated List: [1, 2, 3, 4, 5, 6]
```

### 8. Slicing the List

Slicing a list means extracting a portion of elements from a list using a range of indices.

**Program:**

```
# Creating a list
my_list = [10, 20, 30, 40, 50]

# Slicing list
print("Elements from index 1 to 3:", my_list[1:4])
print("First three elements:", my_list[:3])
print("Last two elements:", my_list[-2:])
```

**Output:**

```
Elements from index 1 to 3: [20, 30, 40]
First three elements: [10, 20, 30]
Last two elements: [40, 50]
```

### 9. Built-in List:

Python provides several built-in functions and methods to work with lists efficiently. Some commonly used functions are:

**Program:**

```
my_list = [5, 2, 8, 2, 7]

# Using built-in functions
my_list.append(10)
my_list.insert(1, 15)
my_list.remove(2)
print("List:", my_list)
print("Count of 2:", my_list.count(2))
print("Index of 7:", my_list.index(7))
my_list.sort()
print("Sorted List:", my_list)
my_list.reverse()
print("Reversed List:", my_list)
print("Length:", len(my_list))
print("Sum:", sum(my_list))
```

**Output:**

```
List: [5, 15, 8, 2, 7, 10]
Count of 2: 1
Index of 7: 4
Sorted List: [2, 5, 7, 8, 10, 15]
Reversed List: [15, 10, 8, 7, 5, 2]
Length: 6
Sum: 47
```

## 10. Two-Dimensional List (2D List)

A two-dimensional list is a list of lists in Python, where each element of the main list is itself a list. It is commonly used to represent matrices or tabular data.

**Program:**

```
# Creating a 2D list
matrix = [
    [1, 2, 3],
```

```
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing elements
print("Element at row 1, column 2:", matrix[0][1])

# Printing the 2D list
print("2D List (Matrix):")
for row in matrix:
    print(row)
```

**Output:**

```
Element at row 1, column 2: 2
2D List (Matrix):
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

# TUPLES:

- Tuples are **ordered** collections of elements.
- Tuples are **immutable**; their elements cannot be changed after creation.
- Tuples can store **different data types** in a single tuple.
- Tuples **allow duplicate elements**.
- Tuples are defined using **parentheses ()**.
- Elements in a tuple can be **accessed using indexing** (positive or negative).
- Tuples can be **nested**, i.e., a tuple can contain other tuples.

## 1. Creating Tuples

Creating a tuple means defining and initializing an ordered collection of elements in Python using parentheses ().

**Program:**

```
# Creating tuples
tuple1 = (1, 2, 3, 4)
tuple2 = ("apple", "banana", "cherry")
tuple3 = (10, "hello", 3.5)
tuple4 = ()  # Empty tuple
tuple5 = (5,)  # Single element tuple (comma is required)
```

```
# Printing tuples
print("Tuple1:", tuple1)
print("Tuple2:", tuple2)
print("Tuple3:", tuple3)
print("Empty Tuple:", tuple4)
print("Single Element Tuple:", tuple5)
```

Output:

```
Tuple1: (1, 2, 3, 4)
Tuple2: ('apple', 'banana', 'cherry')
Tuple3: (10, 'hello', 3.5)
Empty Tuple: ()
Single Element Tuple: (5,)
```

### 2. Adding Elements to Tuples

Tuples are immutable; elements **cannot be added directly**. You can create a new tuple by concatenation.

**Program:**

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5)
tuple3 = tuple1 + tuple2  # Adding by concatenation
print("Updated Tuple:", tuple3)
```

**Output:**

```
Updated Tuple: (1, 2, 3, 4, 5)
```

### 3. Removing Elements from Tuples

Tuples are immutable; elements **cannot be removed directly**. To remove elements, you must convert the tuple to a list, modify it, then convert back to a tuple.

**Program:**

```
tuple1 = (1, 2, 3, 4)
temp_list = list(tuple1)
temp_list.remove(3)  # remove element
tuple1 = tuple(temp_list)
print("After Removal:", tuple1)
```

**Output:**

```
After Removal: (1, 2, 4)
```

### 4. Modifying Tuple Elements

Tuples are immutable; direct modification is not allowed. Use conversion to a

list, modify, then convert back.

**Program:**

```
tuple1 = (1, 2, 3)
temp_list = list(tuple1)
temp_list[1] = 20  # modify second element
tuple1 = tuple(temp_list)
print("Modified Tuple:", tuple1)
```

**Output:**

```
Modified Tuple: (1, 20, 3)
```

### 5. Accessing Elements from Tuples

Elements can be accessed using **indexing** (positive or negative) or **slicing**.

**Program:**

```
tuple1 = (10, 20, 30, 40, 50)
print("First element:", tuple1[0])
print("Last element:", tuple1[-1])
print("Slice (index 1 to 3):", tuple1[1:4])
```

**Output:**

```
First element: 10
Last element: 50
Slice (index 1 to 3): (20, 30, 40)
```

### 6. Sorting Tuple Elements

Tuples are immutable; to sort a tuple, convert it to a list, sort it, then convert back.

**Program:**

```
tuple1 = (40, 10, 30, 20)
sorted_tuple = tuple(sorted(tuple1))  # sorting
print("Sorted Tuple:", sorted_tuple)
```

**Output:**

```
Sorted Tuple: (10, 20, 30, 40)
```

### 7. Concatenating Tuples

Combining two or more tuples using the + operator.

**Program:**

```
tuple1 = (1, 2)
```

```
tuple2 = (3, 4)
tuple3 = tuple1 + tuple2
print("Concatenated Tuple:", tuple3)
```

**Output:**

```
Concatenated Tuple: (1, 2, 3, 4)
```

### 8. Slicing Tuple Elements

Extracting a portion of a tuple using **indices**.

**Program:**

```
tuple1 = (10, 20, 30, 40, 50)
print("Elements 1 to 3:", tuple1[1:4])
print("First three elements:", tuple1[:3])
print("Last two elements:", tuple1[-2:])
```

**Output:**

```
Elements 1 to 3: (20, 30, 40)
First three elements: (10, 20, 30)
Last two elements: (40, 50)
```

### 9. Built-in Tuple Functions

**Common Functions:**

- len(tuple): Length of tuple
- max(tuple): Maximum element
- min(tuple): Minimum element
- sum(tuple): Sum of numeric elements
- tuple.count(x): Count occurrences of x
- tuple.index(x): Index of first occurrence of x

**Program:**

```
tuple1 = (1, 2, 3, 2, 4)
print("Length:", len(tuple1))
print("Max:", max(tuple1))
print("Min:", min(tuple1))
print("Sum:", sum(tuple1))
print("Count of 2:", tuple1.count(2))
print("Index of 4:", tuple1.index(4))
```

**Output:**

Length: 5

Max: 4

Min: 1

Sum: 12

Count of 2: 2

Index of 4: 4

## 10. Packing and Unpacking Tuples

- **Packing:** Storing multiple values in a tuple.
- **Unpacking:** Extracting values from a tuple into individual variables.

**Program: 1**

```
# Packing
tuple1 = 10, 20, 30

# Unpacking
a, b, c = tuple1
print("a:", a)
print("b:", b)
print("c:", c)
```

**Output:**

a: 10

b: 20

c: 30

**Program: 2**

```
fruits = ("apple", "banana", "cherry")
(a,b,c) = fruits
print(a)
print(b)
print(c)
```

**Output:**

apple

banana

cherry

**Program: 3**

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(a, b, *c) = fruits
 print(a)
 print(b)
 print(c)
```

**Output:**

```
apple
banana
['cherry', 'strawberry', 'raspberry']
```

# SETS:

## Creating Sets

You can create a set by using the set() constructor or by using curly braces {}.

```
# Using curly braces

another_set = {1, 2, 3, 4}
print(another_set)

# Output: {1, 2, 3, 4}

# Creating an empty set
empty_set = set()
print(empty_set)  # Output: set()
```

## Adding Elements

You can add elements to a set using the add() method.

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)

Output: {1, 2, 3, 4}
```

## Removing Elements

Elements can be removed using the remove() or discard() methods. The remove() method will raise a KeyError if the element is not found, while discard() will not.

```
my_set = {1, 2, 3, 4}

 my_set.remove(4)

print(my_set)
```

```
# Output: {1, 2, 3}

my_set.discard(3)
print(my_set)
# Output: {1, 2}

# Discarding an element not in the set
my_set.discard(5)  # No error
```

## Set Operations

Python sets support several standard set operations:

**Union (| or union()):** Combines allelements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2 # or
union_set = set1.union(set2) print(union_set)
# Output: {1, 2, 3, 4, 5}
```

**Intersection (& or intersection()):** Returns elements that are common to both sets.

```
intersection_set = set1.intersection(set2)

print(intersection_set)  # Output: {3}
```

**Difference (- or difference()):** Returns elements that are in the first set but not in the second set.

```
difference_set = set1 - set2 # or
difference_set = set1.difference(set2)
print(difference_set)
# Output: {1, 2}
```

**Symmetric Difference (^ or symmetric_difference()):** Returns elements that are in either of the sets, but not in both.

```
sym_diff_set = set1.symmetric_difference(set2)

print(sym_diff_set)
```

```
# Output: {1, 2, 4, 5}
```

## Checking Membership

You can check if an element is in a set using the in keyword.

```
my_set = {1, 2, 3}
print(2 in my_set)  # Output: True
print(5 in my_set)  # Output: False
```

### Iterating Through a Set

You can iterate through the elements of a set using a for loop.

```
my_set = {1, 2, 3}
for elem in my_set:
    print(elem)
```

## Built-in Functions

Some useful built-in functions for sets include:

- len(set): Returns the number of elements in the set.
- min(set): Returns the smallest element in the set.
- max(set): Returns the largest element in the set.
- sum(set): Returns the sum of all elements in the set.

```
my_set = {1, 2, 3, 4} print(len(my_set)) # Output: 4
print(min(my_set)) # Output: 1
print(max(my_set)) # Output: 4
print(sum(my_set)) # Output: 10
```

## PROGRAM

```
# Creating sets set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print("Set1:", set1)
print("Set2:", set2)

# Adding elements to a set
set1.add(5)
print("Set1 after adding 5:", set1)
```

```
# Removing elements from a set
set1.remove(1)  # Raises KeyError if 1 is not in the set

print("Set1 after removing 1:", set1)


# Discarding elements from a set
set1.discard(2)  # Does not raise an error if 2 is not in the set

print("Set1 after discarding 2:", set1)


# Union of sets
union_set = set1 | set2
print("Union of Set1 and Set2:", union_set)
```

```
# Intersection of sets
intersection_set = set1 & set2
  print("Intersection of Set1 and Set2:", intersection_set)

  # Difference of sets
  difference_set = set1 - set2
  print("Difference of Set1 and Set2:", difference_set)


  # Symmetric difference of sets
  symmetric_difference_set = set1 ^ set2
   print("Symmetric Difference of Set1 and Set2:", symmetric_difference_set)

  # Checking membership
  print("Is 3 in Set1?", 3 in set1)
  print("Is 1 in Set1?", 1 in set1)

  # Iterating through a set
  print("Elements in Set1:") for elem
  in set1:
        print(elem)


  # Using built-in functions
  print("Length of Set1:", len(set1))
  print("Minimum in Set1:", min(set1))
  print("Maximum in Set1:", max(set1))
  print("Sum of elements in Set1:", sum(set1))
```

**OUTPUT**

```
Set1: {1, 2, 3, 4}
Set2: {3, 4, 5, 6}
Set1 after adding 5: {1, 2, 3, 4, 5}
Set1 after removing 1: {2, 3, 4, 5}
Set1 after discarding 2: {3, 4, 5}

Union of Set1 and Set2: {3, 4, 5, 6}
```

Intersection of Set1 and Set2: {3, 4, 5}

Difference of Set1 and Set2: set()

Symmetric Difference of Set1 and Set2: {6} Is 3 in Set1?
True
Is 1 in Set1? False Elements
in Set1:
3
4
5
Length of Set1: 3 Minimum in Set1: 3
Maximum in Set1: 5
Sum of elements in Set1: 12

# DICTIONARY

- In Python, a dictionary is an unordered collection of items where each item consists of a key-value pair.
- Dictionaries are mutable, which means they can be changed after creation. They are also dynamic, meaning they can grow and shrink as needed.

## Creating Dictionaries

You can create a dictionary using curly braces {} or the dict() constructor.

```
# Using curly braces
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
# Using the dict() constructor
another_dict = dict(name='Bob', age=30, city='San Francisco')

print(another_dict) # Output: {'name': 'Bob', 'age': 30, 'city': 'San Francisco'}


# Creating an empty dictionary

empty_dict = {} print(empty_dict) #
    Output: {}
```

## Accessing Values

You can access the value associated with a specific key using square brackets [] or the get() method.

```
print(my_dict['name']) # Output: Alice
# Using get() method

print(my_dict.get('age')) # Output: 25


# Using get() with a default value
print(my_dict.get('country', 'Not Found')) # Output: Not Found
```

## Adding and Updating Items

You can add new key-value pairs or update existing ones using the square brackets [].

```
# Adding a new key-value pair
    my_dict['country'] = 'USA'

print(my_dict)

Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA'}
```

```
# Updating an existing key-value pair
my_dict['age'] = 26
print(my_dict)
```

**Output:** {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}

## Removing Items

You can remove items using the del statement, the pop() method, or the popitem() method.

```
# Using pop() method

# Using del statement del
my_dict['city']

print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'country': 'USA'}


my_dict.pop('age')
print(age)  # Output: 26
print(my_dict)  # Output: {'name': 'Alice', 'country': 'USA'}


# Using popitem() method (removes the last inserted item)
item = my_dict.popitem()
print(item)  # Output: ('country', 'USA')
print(my_dict)  # Output: {'name': 'Alice'}
```

## Dictionary Methods

Python provides several methods to work with dictionaries:

- keys(): Returns a view object containing the keys.
- values(): Returns a view object containing the values.

- items(): Returns a view object containing the key-value pairs.
- update(): Updates the dictionary with elements from another dictionary or an iterable of key-value pairs.
- clear(): Removes all items from the dictionary.

```
# Using keys(), values(), and items() methods
    print(my_dict.keys())#Output:dict_keys(['name','age','city'])
    print(my_dict.values())#Output:dict_values['Alice',25,'New York']

    print(my_dict.items())# Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])


# Using update() method

my_dict.update({'age': 27, 'city': 'Los Angeles'})

print(my_dict)  # Output: {'name': 'Alice', 'age': 27, 'city': 'Los Angeles'}


# Using clear() method

my_dict.clear()

print(my_dict)  # Output: {}
```

**Iterating Through a Dictionary:**

You can iterate through a dictionary using a for loop.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Iterating through keys

for key in my_dict:
   print(key, my_dict[key])
    # Output:
   # name Alice
```

```
    # age 25
    # city New York

# Iterating through values
for value in my_dict.values():
    print(value) #
        Output:
    # Alice # 25

    # New York

# Iterating through key-value pairs
for key, value in my_dict.items():
    print(key, value)

    # Output:
    # name Alice
    # age 25
    # city New York
```

**Built-in Functions**

Some useful built-in functions for dictionaries include:

- len(dict): Returns the number of items in the dictionary.
- sorted(dict): Returns a sorted list of the dictionary's keys.

```
print(len(my_dict))  # Output: 3
print(sorted(my_dict))  # Output: ['age', 'city', 'name']
```

**PROGRAM**

```
# Creating dictionaries
dict1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}
dict2 = dict(name='Bob', age=30, city='San Francisco')
print("Dictionary 1:", dict1)
print("Dictionary 2:", dict2)

# Accessing values
print("Name in dict1:", dict1['name'])
 print("Age in dict1:", dict1.get('age'))
print("Country in dict1:", dict1.get('country', 'Not Found'))

# Adding and updating items dict1['country'] = 'USA'
print("Dict1 after adding country:", dict1) dict1['age'] = 26
print("Dict1 after updating age:", dict1)

# Removing items del dict1['city']
print("Dict1 after deleting city:", dict1) age = dict1.pop('age')
print("Popped age:", age)
print("Dict1 after popping age:", dict1) item = dict1.popitem()
print("Popped item:", item)
print("Dict1 after popping item:", dict1)
```

```
# Dictionary methods
dict1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}

print("Keys in dict1:", dict1.keys())

print("Values in dict1:", dict1.values())

print("Items in dict1:", dict1.items())


# Updating dictionary
dict1.update({'age': 27, 'city': 'Los Angeles'})

print("Dict1 after update:", dict1)


# Clearing dictionary dict1.clear()
print("Dict1 after clearing:", dict1)



# Iterating through a dictionary
dict1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}

print("Iterating through keys:")
    for key in dict1:
    print(key, dict1[key])


print("Iterating through values:")

for value in dict1.values():

        print(value)

print("Iterating through items:")
for key, value in dict1.items():
        print(key, value)
```

```
# Built-in functions
print("Length of dict1:", len(dict1))
print("Minimum key in dict1:", min(dict1))
print("Maximum key in dict1:", max(dict1))
print("Sorted keys in dict1:", sorted(dict1))
```

**OUTPUT**

```
Dictionary 1: {'name': 'Alice', 'age': 25, 'city': 'New York'}
Dictionary 2: {'name': 'Bob', 'age': 30, 'city': 'San Francisco'} Name in dict1: Alice
Age in dict1: 25
Country in dict1: Not Found
Dict1 after adding country: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA'}
Dict1 after updating age: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
Dict1 after deleting city: {'name': 'Alice', 'age': 26, 'country': 'USA'} Popped age: 26
Dict1 after popping age: {'name': 'Alice', 'country': 'USA'} Popped item:
    ('country', 'USA')
Dict1 after popping item: {'name': 'Alice'} Keys in dict1: dict_keys(['name', 'age', 'city'])
Values in dict1: dict_values(['Alice', 25, 'New York'])
Items in dict1: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')]) Dict1 after update:
    {'name': 'Alice', 'age': 27, 'city': 'Los Angeles'}
Dict1 after clearing: {} Iterating through keys:
```

name Alice age 25

city New York

Iterating through values:

Alice 25

New York

Iterating through items:

name Alice age 25

city New York Length of dict1: 3

Minimum key in dict1: age Maximum key in dict1:

name

Sorted keys in dict1: ['age', 'city', 'name']

| | | Mutable/ Immutable | Ordered/ Unordered | Allows /do not Allow Duplicates |
|---|---|---|---|---|
| LIST | Items in [] | Mutable | Ordered | Allows |
| TUPLE | Items in () | Immutable | Ordered | Allows |
| SET | Items in {} | Mutable | Unordered | Do not allow |
| DICTIONARY | Items in key-value pairs | Mutable | Unordered | Allows |

## Question Bank:

| | | |
|---|---|---|
| **UNIT – II: FUNCTIONS AND PYTHON DATA STRUCTURES :** | | |
| **Functions and Functional Programming , Recursive Functions , Recursive Vs Iterative Functions. Built in Data structures – List,tuple,sets and dictionary.** | | |
| PART –A | | |
| 1. | What is Python Function | L1 |
| 2. | List out Python function types | L1 |
| 3. | What is Recursion | L1,L2 |
| 4. | Differentiate between recursion vs iteration | L2,L4 |
| 5. | List out Python built in data structures | L1 |
| 6. | Write about the List Data structure properties | L2 |
| 7. | Differentiate between List and set | L2,L4 |
| 8. | Differentiate between List and tuple | L2,L4 |
| PART –B | | |
| 1. | Explain Functions with and without return statement with an example | L2,L3 |
| 2. | Discuss Positional and Keyword Arguments in detail | L2 |
| 3. | Briefly explain what is function , how to declare functions and types of functions | L1,L2 |
| 4. | Discuss and evaluate List and tuples | L2,L5 |
| 5. | Discuss and evaluate List and sets | L2,L5 |
| 6. | Discuss and evaluate sets and dictionaries | L2,L5 |
| 7. | What is recursion? Explain the difference between recursion and iteration with example | L2,L3 |
| 8. | Discuss about python functions, benefits and its types | L2 |

# PYTHON PROGRAMMING
# UNIT – III
# OBJECT-ORIENTED CONCEPTS THROUGH PYTHON

# UNIT – III: OBJECT-ORIENTED CONCEPTS THROUGH PYTHON

## 1. Unit Overview:

This unit introduces students to Object-Oriented Programming (OOP) concepts using Python. Students will learn how to model real-world entities using classes and objects, encapsulate data, implement inheritance and polymorphism, and use abstraction to hide complexity. The unit also covers exploring Python's built-in libraries such as math and random, and guides students on creating their own reusable Python libraries. The focus is on writing modular, maintainable, and reusable code.

## 2. Objectives of the Unit:

By the end of this unit, students should be able to:

- Understand the fundamental **principles of OOP** and their significance in programming.
- Learn to define **classes and create objects** to model real-world entities.
- Implement **encapsulation** to protect data and hide implementation details.
- Apply **inheritance** to reuse code across classes.
- Demonstrate **polymorphism** through method overriding or operator overloading.
- Understand **abstraction** and how to simplify complex systems.
- Explore **Python built-in libraries** (math, random) for common operations.
- Develop and use custom Python libraries to organize code effectively.

## 3. Learning Outcomes:

After completing this unit, students should will be able to:

- Create Python **classes and instantiate objects** to represent entities.
- Implement **encapsulation** using private/protected members.
- Use **inheritance** to derive child classes and extend functionality.
- Apply **polymorphism** to make code flexible and reusable.
- Implement **abstraction** to manage complex systems efficiently.
- Utilize **Python libraries** to perform mathematical and random operations.
- Create **modular Python libraries** for reuse in multiple projects.

## 4. Importance of Studying this Unit:

- Builds a strong foundation in modular programming and code organization.
- Essential for developing real-world applications, including web apps, AI, and games.
- Helps in software maintenance, improving readability and reducing redundancy.
- Prepares students for advanced programming topics and professional coding practices.

## 5. Key Concepts:

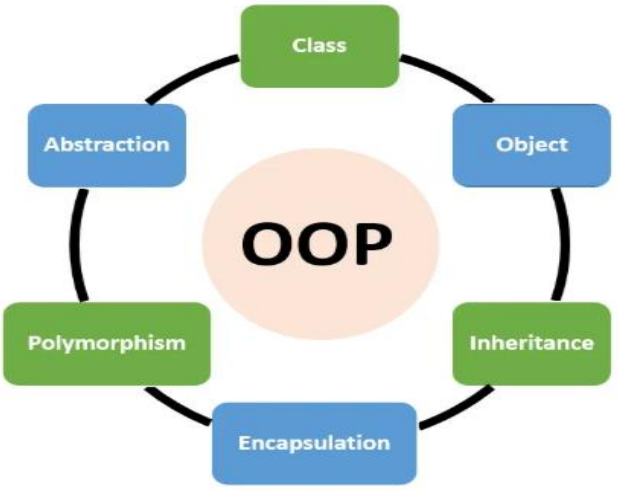
- **Class & Object** – Blueprint and instance of a class.
- **Encapsulation** – Hiding internal implementation details.
- **Inheritance** – Code reuse across parent and child classes.
- **Polymorphism** – Same method, different behavior in different contexts.
- **Abstraction** – Exposing only essential features.
- **Python Libraries** – math and random for mathematical and random operations.
- **Custom Library** – Writing reusable Python modules.

## Introduction to Object Oriented Programming:

- Like other general-purpose programming languages, Python is also an object-oriented language since its beginning.
- It allows us to develop applications using an Object-Oriented approach.
- An Object-Oriented Paradigm is
  - The method of structuring the program
  - To design the program using classes and objects
  - It is a widespread technique to solve problems by creating objects
- A Procedure-Oriented Paradigm is
  - To design programs using functions (or)
  - It is a technique to solve problems by creating functions

| Procedural Oriented Programming | Object Oriented Programming |
|---|---|
| In procedural programming, the program is divided into small parts called ***functions***. | In object-oriented programming, the program is divided into small parts called ***objects***. |
| Procedural programming follows a ***top-down approach***. | Object-oriented programming follows a ***bottom-up approach***. |
| Adding new data and functions is not easy. | Adding new data and function is easy. |
| Procedural programming does not have any proper way of hiding data so it is ***less secure***. | Object-oriented programming provides data hiding so it is ***more secure***. |
| In procedural programming, **the function** is more important than the data. | In object-oriented programming**, data is** more important than function. |
| Procedural programming is used for designing **medium-sized programs.** | Object-oriented programming is used for designing **large and complex programs.** |
| **Code reusability** absent in procedural programming, | **Code reusability** present in object-oriented programming. |
| **Examples:** C, FORTRAN, Pascal, Basic, etc. | **Examples:** C++, Java, Python, C#, etc. |

## Object Oriented Programming Concepts:




## Object:

An entity that has a state and behavior associated with it.

**Example 1**: Smith is an entity that has state and behavior

**State** = Roll No, Name, Age, Branch, etc.

**Behavior** = Writing internal exams, external exams, attendance, etc.

**Example 2**: German Shepherd is an entity that has state and behavior

**State** = Age, Breed, Color, etc.

**Behavior** = Eating, Walking, Sleeping, Barking

## Class:

- A **class** is a collection of similar objects.
- It acts as a **blueprint** for creating objects.
- Objects belonging to the same class share **common state and behavior**.

**Example 1:**

- A collection of **Smith, Jones, and King** is called the class **Student**.
- Because all these objects share similar **state** and **behavior**.

**Example 2:**

- A collection of **German Shepherd, Labrador, and Poodle** is called the class **Dog**.
- Because all these objects share similar **state** and **behavior**.

## Inheritance:

- **Inheritance** is the process by which one object or class acquires the properties of another class.
- It allows a **child class** to inherit the properties and behavior of its **parent class**.
- It supports **code reusability** and reduces duplication.

**Example:**

- A **child** inherits properties from its **parents**.

## Polymorphism:

- **Polymorphism** means "many forms".
- It refers to using a **single entity** (method, operator, or object) to perform **different actions** in different situations.
- It improves **flexibility and reusability** of code.

**Example:**

- A **mixer** can produce different outputs like grinding, blending, and mixing — same device, different functions.

## Abstraction:

- **Abstraction** means **hiding internal details** and showing only the essential features to the user.
- It helps to reduce **complexity** and improve **usability**.

**Example:**

- While driving a **car**, the driver only uses controls like steering and pedals, without knowing the internal working of the engine.

## Encapsulation

- **Encapsulation** is the process of **wrapping data and methods into a single unit (class)**.
- It is used to **protect data from direct access**.
- Data can be accessed only through **methods (functions)**.
- It helps in **data hiding and security**.

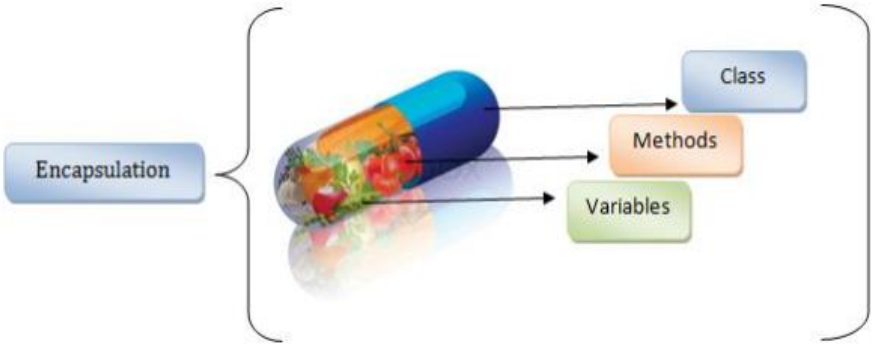
**Example:**

- A **capsule** contains medicine inside → data is hidden.
- Similarly, in a class, variables are hidden and accessed through methods.

## ENCAPSULATION:

- Encapsulation refers to the bundling of attributes and methods inside a single class.
- It prevents outer classes from accessing and changing attributes and methods of a class.
- This also helps to achieve **data hiding**.
- A class is an example of encapsulation as it encapsulates all the data that is member functions,

variables, etc.




- Consider a real-life example of encapsulation. In a company, there are different sections such as:
  - Accounts section
  - Finance section
  - Sales section
- The **finance section** handles all the financial transactions and maintains records related to finance.
- Similarly, the **sales section** handles all the sales-related activities and maintains sales records.
- Now, there may be a situation where an official from the **finance section** needs sales data for a particular month.
- In this case, he is **not allowed to directly access** the data of the sales section.
- He must first **contact an authorized person** in the sales section and request the required data.
- This is what **encapsulation** means. The data of the sales section and the employees who manage it are **wrapped under a single unit called "sales section."**
- Encapsulation also ensures **data hiding**. The data of sections like sales, finance, or accounts is **hidden from other sections**.

## Program: 1

```
# Define the Person class
class Person:

# Constructor to initialize object attributes
def __init__(self, name, age):
        self.name = name
        self.age = age

# Method to get the person's name
def get_name(self):
        return self.name

# Method to get the person's age
def get_age(self):
        return self.age

# Method to update the person's age
def update_age(self, new_age):
        self.age = new_age

# Main program
# Create Person objects (instances)
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Access object attributes using methods
print(person1.get_name(), "is", person1.get_age(), "years old.")
print(person2.get_name(), "is", person2.get_age(), "years old.")

# Update age
person1.update_age(26)
```

```
# Check the updated age
print(person1.get_name(), "is now", person1.get_age(), "years old.")
```

## Output:

```
Alice is 25 years old.
Bob is 30 years old.
Alice is now 26 years old.
```

## Program: 2

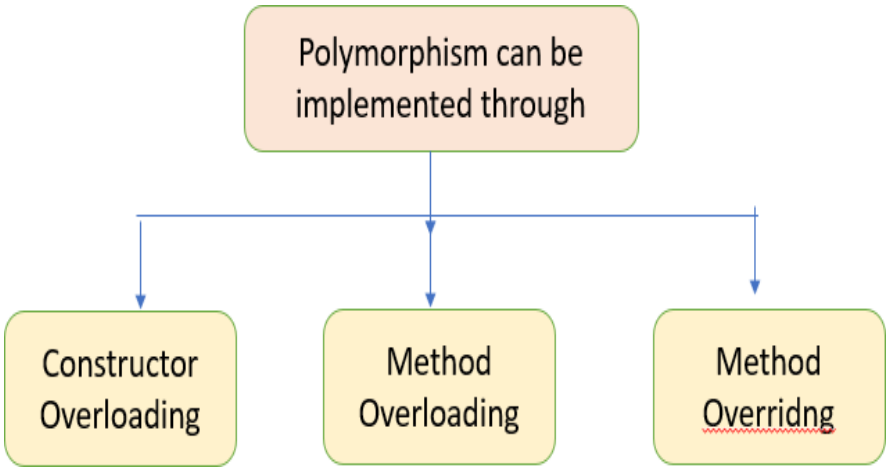
```
# Constructor to initialize length and breadth
def __init__(self, l, b):
        self.l = l
        self.b = b

# Method to calculate area
def area(self):
        return self.l * self.b


# Create objects
r1 = Rectangle(2, 3)
print('Area of first Rectangle:', r1.area())

r2 = Rectangle(4, 5)
print('Area of second Rectangle:', r2.area())
```

## Output:

```
Area of first Rectangle: 6
Area of second Rectangle: 20
```

# POLYMORPHISM:

- **Polymorphism** means **one entity can perform different operations** in different scenarios.
- It allows the same method or operator to behave differently based on context.
- It is implemented through:
  - **Method Overloading**
  - **Method Overriding**




## CONSTRUCTOR OVERLOADING:

- In Python, constructor overloading refers to the ability to create objects of a class with different sets of parameters.
- It allows objects to be initialized in multiple ways depending on the arguments provided.
- Unlike other programming languages, Python does not support explicit constructor overloading (i.e., multiple constructors with the same name but different parameters).
- Instead, Python achieves this using:
  - Default arguments
  - Optional parameters

## Program:

```
class Const:

    # Constructor with default arguments
    def __init__(self, a=0, b=0, c=0):
            self.a = a
            self.b = b
            self.c = c

    # Method to calculate sum
    def sum(self):
            return self.a + self.b + self.c
    # Create objects
```

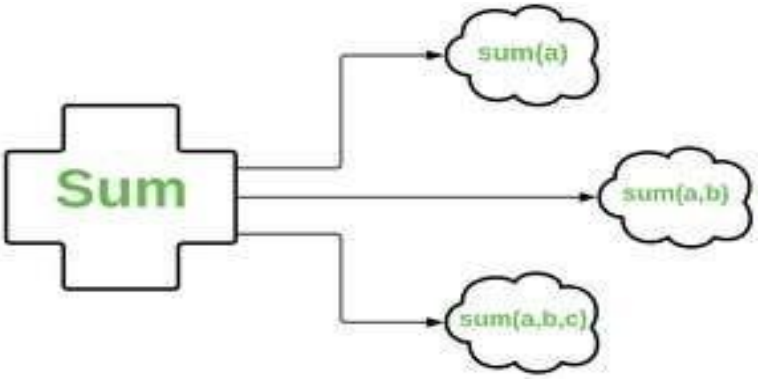
```
s1 = Const(3, 4)
sum2 = s1.sum()
print('Sum of 2 numbers:', sum2)

s2 = Const(2, 3, 4)
sum3 = s2.sum()
print('Sum of 3 numbers:', sum3)

s3 = Const()
sum0 = s3.sum()
print('Sum of 0 numbers is:', sum0)
```

## Output:

```
Sum of 2 numbers: 7
Sum of 3 numbers: 9
Sum of 0 numbers is: 0
```

# METHOD OVERLOADING:

- In Python, **method overloading** refers to the ability to define multiple methods in a class with the **same name but different parameter lists**.
- Each method performs a **different operation** based on the **number or type of arguments** passed to it.




## Program:

```
class Demo:

    # Method with default arguments
    def add(self, a=None, b=None, c=None):
```

```
        # If all three arguments are provided

        if a is not None and b is not None and c is not None:
            return a + b + c

        # If two arguments are provided
        elif a is not None and b is not None:
            return a + b

        # If only one argument is provided
        elif a is not None:
            return a

        else:
            return 0

    # Create object
    obj = Demo()

    # Different method calls
    print("Sum of 1 number:", obj.add(5))
    print("Sum of 2 numbers:", obj.add(5, 10))
    print("Sum of 3 numbers:", obj.add(5, 10, 15))
    print("Sum of no numbers:", obj.add())
```

## Output:

```
Sum of 1 number: 5
Sum of 2 numbers: 15
Sum of 3 numbers: 30
Sum of no numbers: 0
```

## INHERITANCE:

- **Inheritance** is a fundamental concept in object-oriented programming (OOP) that allows a class (called the **child** or **derived class**) to inherit properties and behaviors from another class (called the **parent** or **base class**).

- The **child class** can extend or modify the functionality of the parent class while reusing its existing features.
- In Python, **inheritance is supported**, and it plays a significant role in building **complex and organized code structures**.

## Types of Inheritance:

- In Python, there are several types of inheritance that allow classes to inherit properties and behaviors from other classes in different ways.
- The common types of inheritance in Python are:

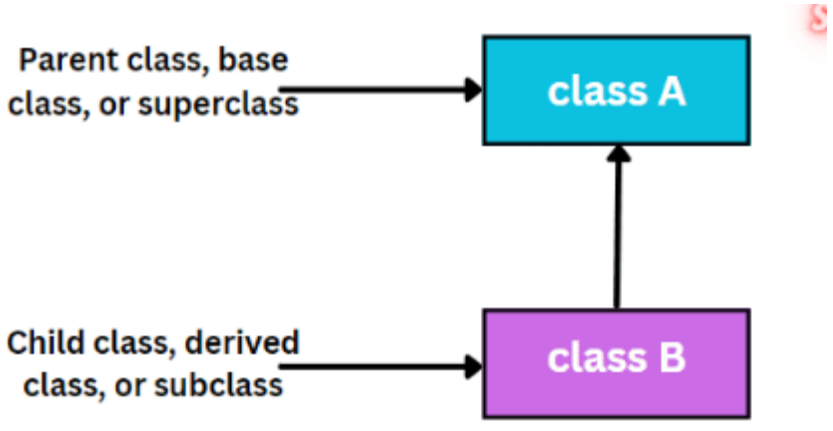
## Single Inheritance

- **Single inheritance** involves a class inheriting from only **one base class**.
- It is the **most common type of inheritance** and represents a **simple hierarchy of classes**.

**Example:**

```
class ParentClass:
    pass


class ChildClass(ParentClass):
    pass
```

## Diagram:




Single Inheritance in Python

## Program:

```
# Base class
class Animal:

    def    init    (self, name):
```

```
        self.name = name

    def make_sound(self):
        pass


# Derived class inheriting from Animal
class Dog(Animal):

    def __init__(self, name, breed):
        # Call the constructor of the base class
        super().__init__(name)
        self.breed = breed

    def make_sound(self):
        return "Woof!"


# Create object
dog = Dog("Buddy", "Golden Retriever")

# Access attributes and methods
print(dog.name, "is a", dog.breed, "and says", dog.make_sound())
```

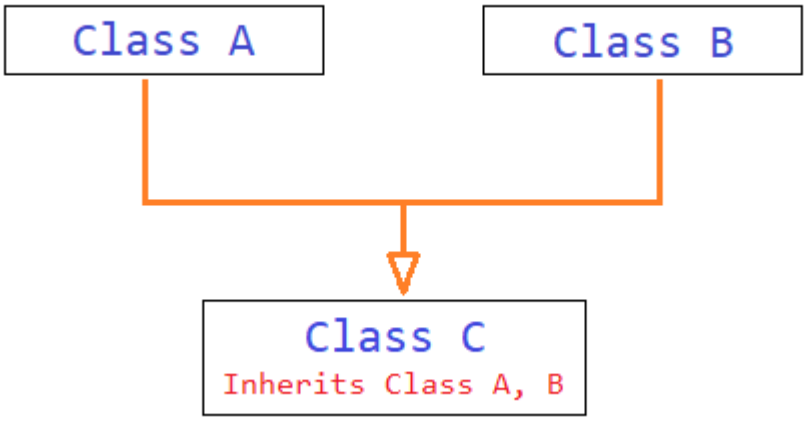
## Output:

Buddy is a Golden Retriever and says Woof!

## Multiple Inheritance

- **Multiple inheritance** involves a class inheriting from **more than one base class**.
- The derived class can access the **properties and methods of all parent classes**.
- It helps in combining features from multiple classes into a single class.

## Diagram:




## Program:

```
# Base class 1
class Father:
    def skills(self):
```

```
        print("Father's skills: Driving")

    # Base class 2
    class Mother:
        def talents(self):
            print("Mother's talents: Cooking")

    # Derived class
    class Child(Father, Mother):
        def abilities(self):
            print("Child has multiple abilities")

    # Create object
    c = Child()

    # Access methods from both parent classes
    c.skills()
    c.talents()
    c.abilities()
```

## Output:

```
Father's skills: Driving
Mother's talents: Cooking
Child has multiple abilities
```

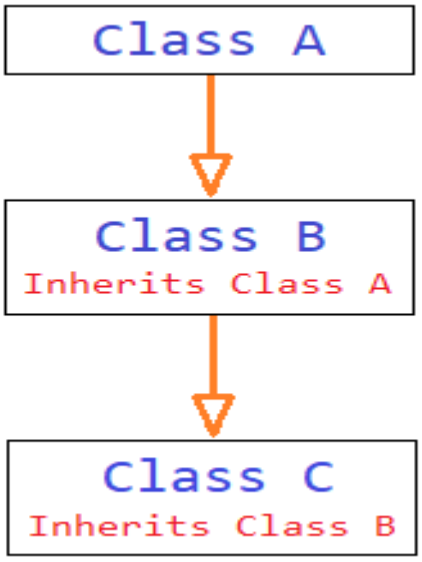
## Multilevel Inheritance

- **Multilevel inheritance** involves a class inheriting from another class, which in turn inherits from another class.
- It forms a **chain of inheritance** (Grandparent → Parent → Child).
- Each derived class can use the features of all its **ancestor classes**.

## Example:

```
class GrandParent:
    pass

class Parent(GrandParent):
    pass

class Child(Parent):
    pass
```

## Diagram:




## Program:

```
# Base class
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        return "Hi my name is " + self.name + " and I am " + str(self.age) + " years old."

# Derived class
class Employee(Person):

    def __init__(self, name, age, employee_id):
        super().__init__(name, age)
        self.employee_id = employee_id

    def work(self):
        return "I am an employee working with ID " + self.employee_id

# Further derived class
class Manager(Employee):

    def __init__(self, name, age, employee_id, department):
        super().__init__(name, age, employee_id)
        self.department = department
```

```
    def manage_team(self):
        return "I am managing a team in the department of " + self.department

# Create object
manager = Manager("John Doe", 35, "12345", "Sales")

# Access methods
print(manager.introduce())
print(manager.work())
print(manager.manage_team())
```

## Output:

Hi my name is John Doe and I am 35 years old.

I am an employee working with ID 12345

I am managing a team in the department of Sales

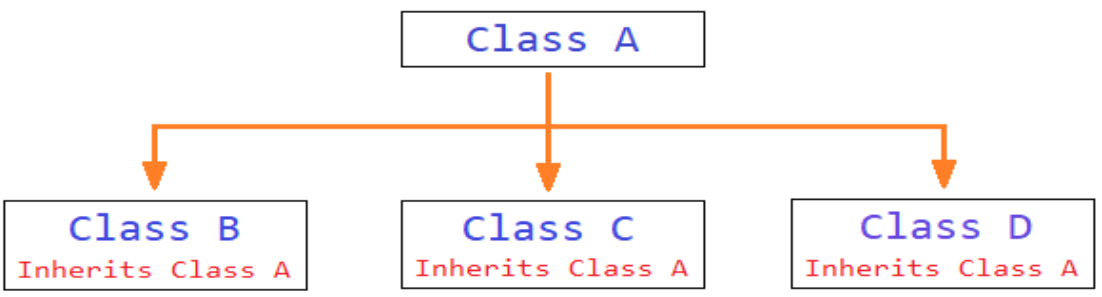
## Hierarchical Inheritance:

- **Hierarchical inheritance** involves multiple classes inheriting from the **same base class**.
- It creates a **hierarchy of classes** derived from a common base class.

### Example:

```
class ParentClass:
pass

class ChildClass1(ParentClass):
pass

class ChildClass2(ParentClass):
pass
```

## Diagram:




## Program:

```
class Animal:

    def __init__(self, name):
```

```
        self.name = name

    def make_sound(self):
        pass

    def move(self):
        print(f"{self.name} is moving.")


class Dog(Animal):

    def make_sound(self):
        print("Woof!")


class Cat(Animal):

    def make_sound(self):
        print("Meow!")


# Creating objects of the derived classes
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Calling the methods
dog.make_sound()   # Output: Woof!
dog.move()         # Output: Buddy is moving.

cat.make_sound()   # Output: Meow!
cat.move()         # Output: Whiskers is moving.
```

## Hybrid Inheritance

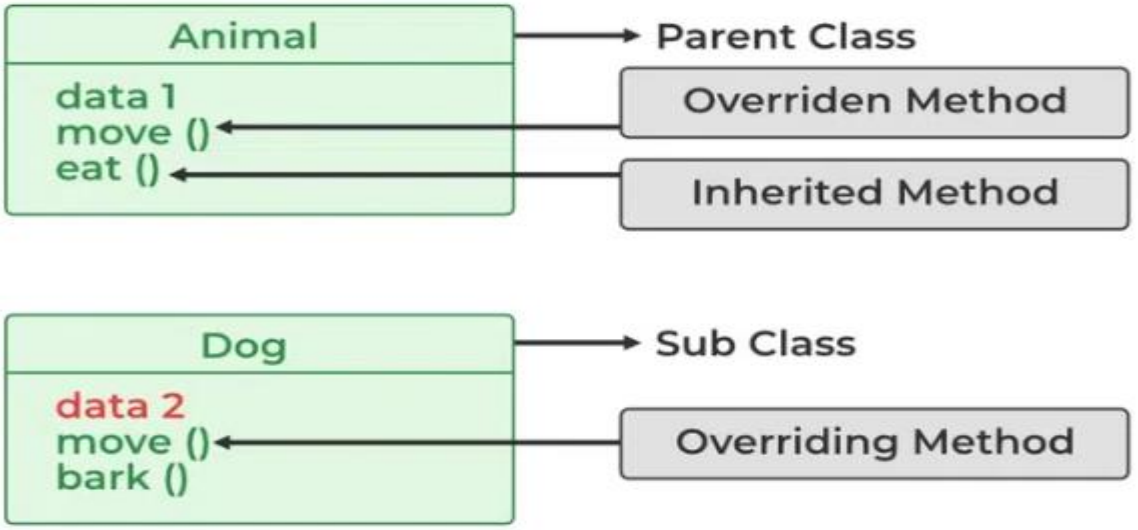
- **Hybrid inheritance** is a combination of **multiple inheritance** and **multilevel inheritance**.
- It combines different types of inheritance to form a more complex structure.

**Example:**

```
class GrandparentClass:
pass

class ParentClass1(GrandparentClass):
pass

class ParentClass2(GrandparentClass):
pass

class ChildClass(ParentClass1, ParentClass2):
pass
```

## METHOD OVERRIDING

- When a method in a **subclass** has the same name, same parameters (signature), and same return type (or subtype) as a method in its **superclass**, then the method in the subclass is said to **override** the method in the superclass.
- The version of the method that is executed depends on the **object used to call it**.
- If an object of the **parent class** is used, the method in the **parent class** will be executed.
- If an object of the **child class** is used, the method in the **child class** will be executed.
- Method overriding allows you to change or extend the behavior of the inherited method to suit the needs of the subclass.




## Program:

```
class Animal:

    def makeSound(self):
        print("Animal makes a sound")


class Dog(Animal):   # Override

    def makeSound(self):
        print("Dog barks")


# Create object
d = Dog()

# Call method
d.makeSound()
```

## Output:

Dog barks

## Python Libraries: Math and Random

- Python provides many **built-in libraries (modules)** to perform tasks easily.
- A **library/module** is a collection of **predefined functions and methods**.
- These libraries help to **reduce code complexity and development time**.
- Two commonly used libraries are:
    - **math module** → used for mathematical calculations
    - **random module** → used for generating random values

### 1. Math Library

- The **math module** provides functions to perform **advanced mathematical operations**.
- It is useful in scientific and engineering applications.
- To use it, we must import it:

```
import math
```

**Important Functions in Math Module**

| Function | Description |
|---|---|
| math.sqrt(x) | Returns square root of x |
| math.pow(x, y) | Returns x raised to power y |
| math.factorial(x) | Returns factorial of x |
| math.ceil(x) | Rounds value up to nearest integer |
| math.floor(x) | Rounds value down |
| math.log(x) | Returns natural logarithm |
| math.sin(x) | Sine value |
| math.cos(x) | Cosine value |
| math.tan(x) | Tangent value |

**Example:**

```
import math
print("Square root:", math.sqrt(16))
print("Power:", math.pow(2, 3))
print("Factorial:", math.factorial(5))
print("Ceil:", math.ceil(4.2))
print("Floor:", math.floor(4.8))
```

**Output**

```
Square root: 4.0
Power: 8.0
Factorial: 120
Ceil: 5
Floor: 4
```

**2. Random Library**

- The **random module** is used to generate **random numbers and values**.
- It is widely used in games, simulations, and testing.
- To use it:

```
import random
```

## Important Functions in Random Module

| Function | Description |
|---|---|
| random.random() | Returns random float (0 to 1) |
| random.randint(a, b) | Returns random integer between a and b |
| random.randrange(a, b) | Returns random number from range |
| random.choice(list) | Returns random element from list |
| random.shuffle(list) | Shuffles list elements |
| random.uniform(a, b) | Returns random float between a and b |

**Example :**

```
import random
print("Random float:", random.random())
print("Random integer:", random.randint(1, 10))
```

```
list1 = [10, 20, 30, 40, 50]
print("Random choice:", random.choice(list1))

random.shuffle(list1)
print("Shuffled list:", list1)
```

**Output:**

```
Random float: 0.45
Random integer: 7
Random choice: 30
Shuffled list: [20, 50, 10, 40, 30]
```

## OBJECT ORIENTED DESIGN USING UML:

- **Object-Oriented Design (OOD)** is a process of designing software systems using the principles of **object-oriented programming**.
- **Unified Modeling Language (UML)** is a standard visual representation used to model and visualize the various aspects of object-oriented systems.
- UML provides a set of **diagrams** to describe different perspectives of a software system, helping in **understanding, communication, and design** of complex systems.

## Types of UML Diagrams

### 1. Class Diagram

- Represents the **static structure** of the system.
- Shows **classes, attributes, methods**, and relationships between classes.
- Used to visualize the **class hierarchy and associations** between classes.

### 2. Object Diagram

- Represents a **snapshot of instances (objects)** at a particular time.
- Shows **objects and links** between them along with their current data values.

### 3. Use Case Diagram

- Represents the **functionalities or use cases** of the system from the **user's perspective**.
- Shows **actors** (users or external systems) and their interactions with the system through use cases.

### 4. Sequence Diagram

- Represents the **interactions between objects** in a particular scenario or use case.
- Shows the **flow of messages** between objects over time, indicating the **order of execution**.

### 5. State Diagram

- Represents the **states and transitions** of a single object or a class.

- Shows how an object **changes its state** in response to events.

### 6. Package Diagram

- Represents the **organization and dependencies** between packages or modules in a system.

### 7. Component Diagram

- Represents the **physical components** and their relationships in the system.

- Used to visualize the **architecture and distribution** of the system.

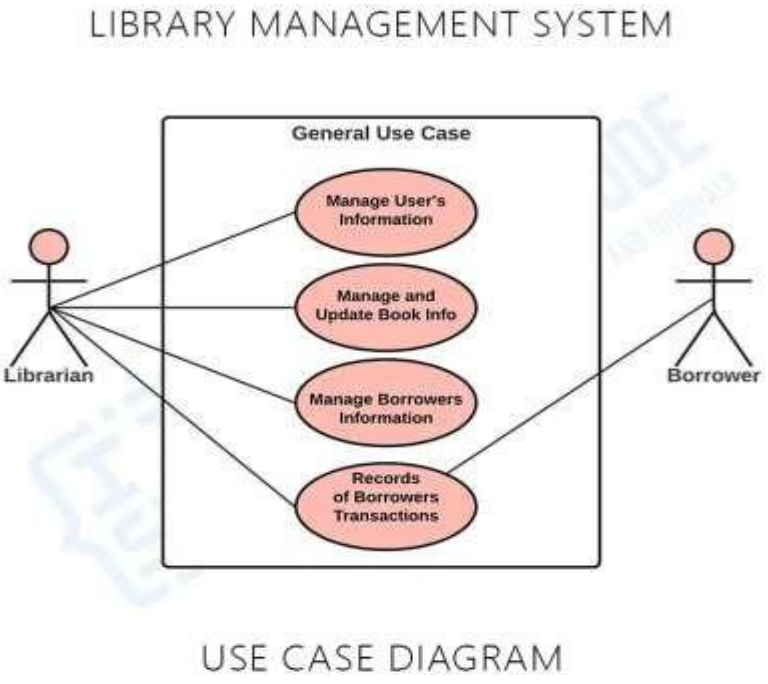
### 8. Deployment Diagram

- Represents the **physical deployment** of software components on hardware nodes.
- Shows how software artifacts are **distributed across different machines**.

## UML Diagrams for Library Management Systems

### Use Case Diagram

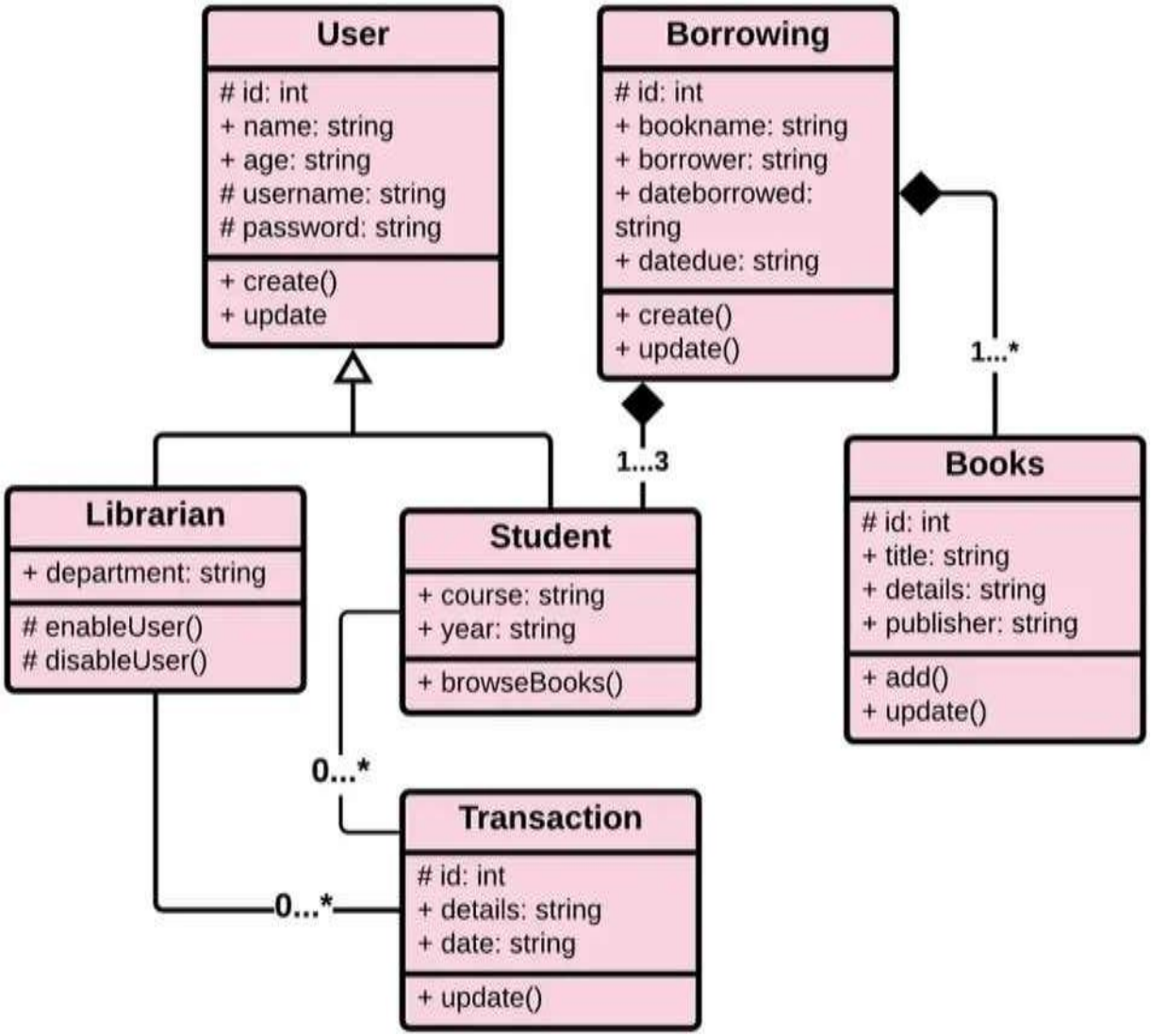
Use Case Diagram of Library Management System contains the main use cases and users in the system.



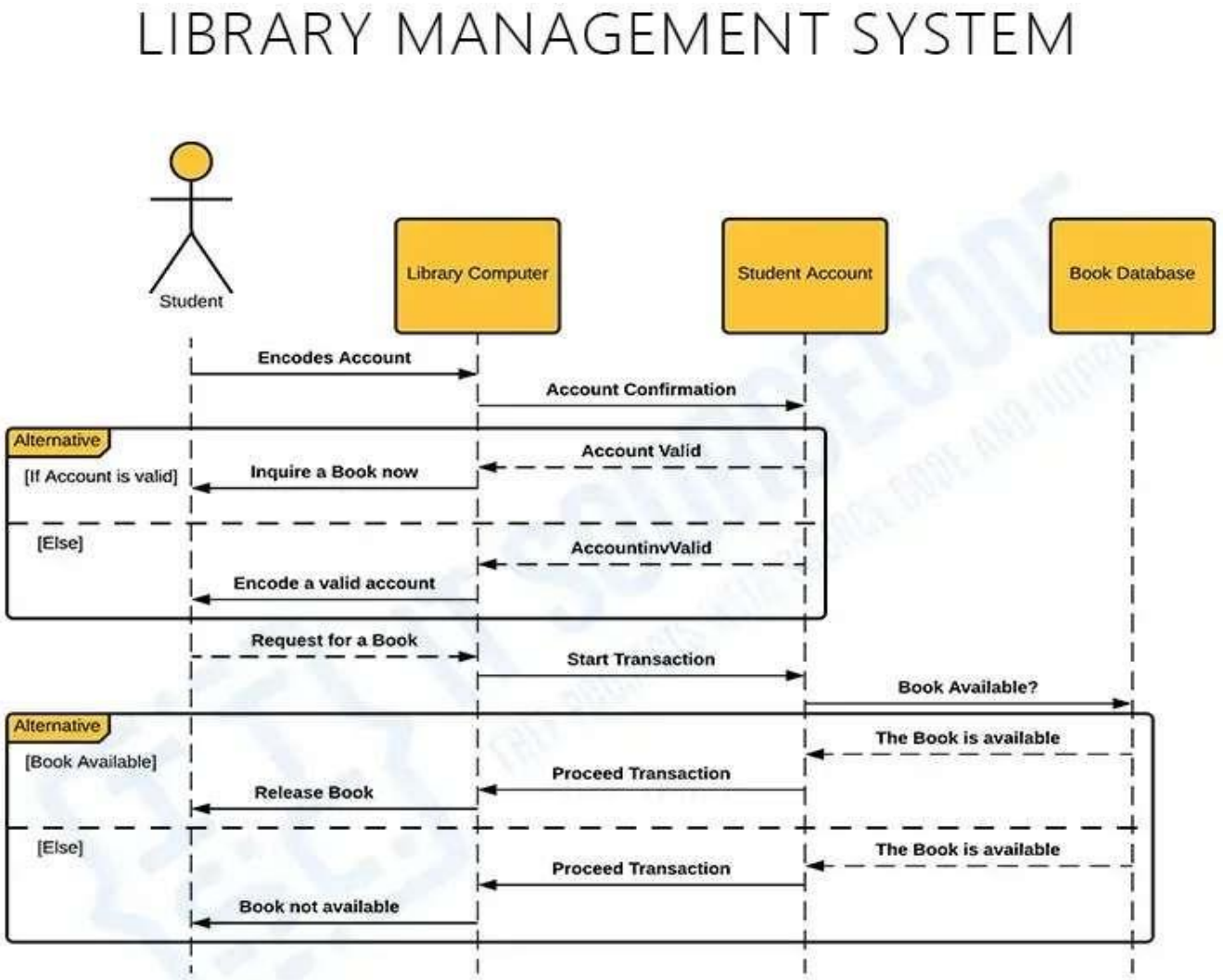

USE CASE DIAGRAM

## Class Diagram

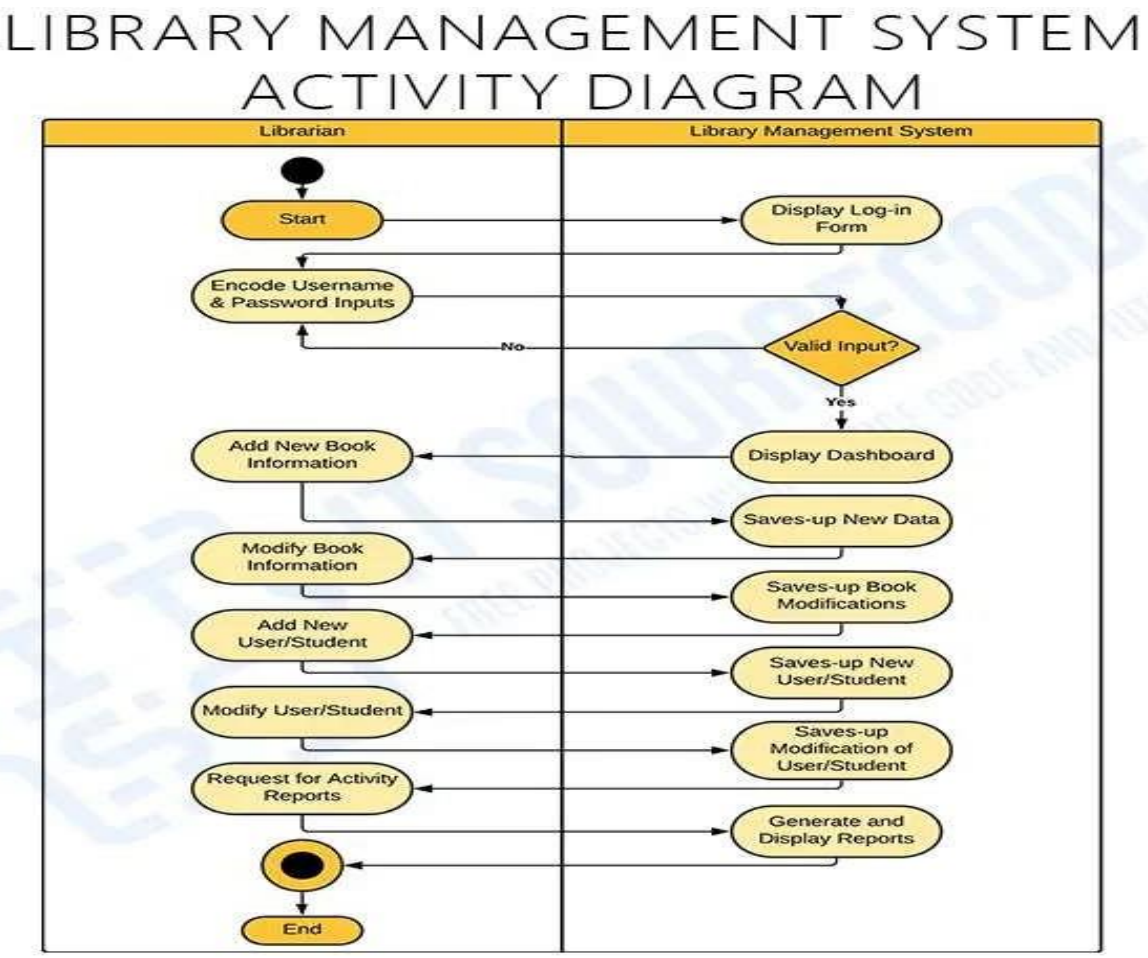
It is used to visualize the class hierarchy and associations between classes of a system



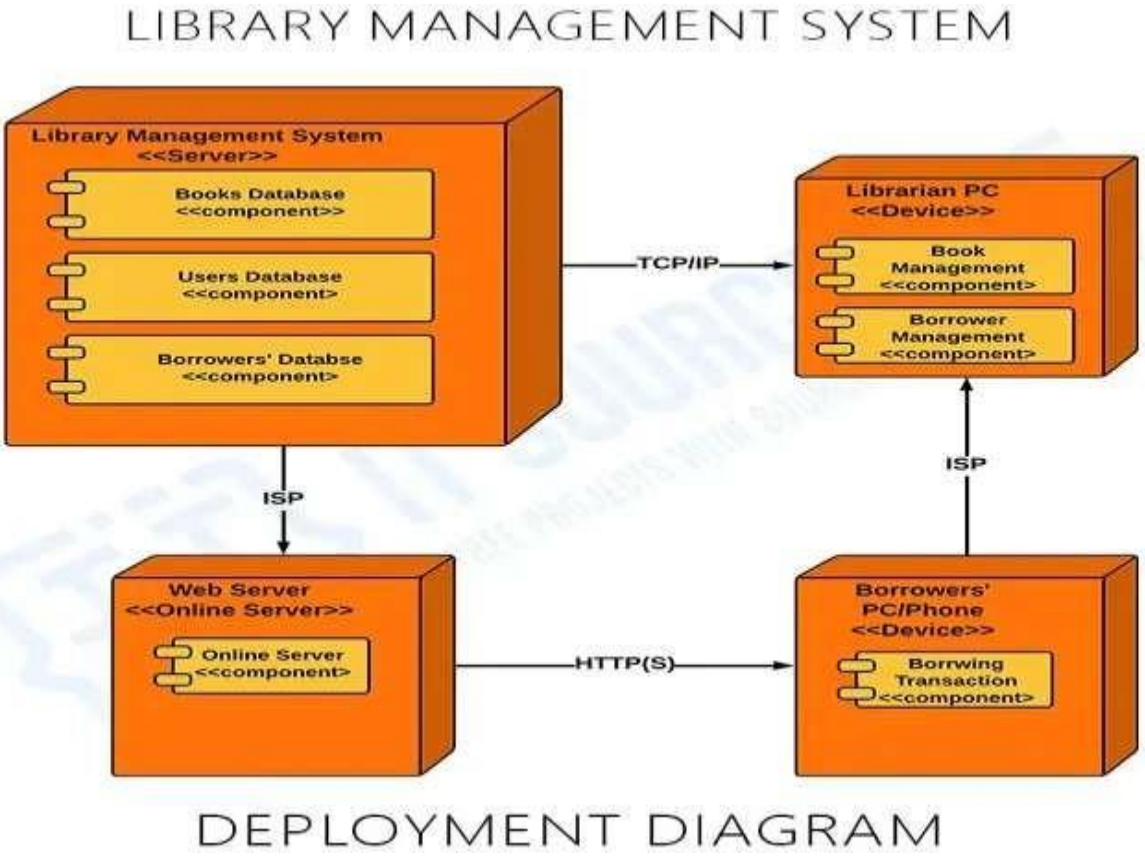
Sequence or Interaction Diagram: Represents the interactions between objects in a particular scenario or use case.



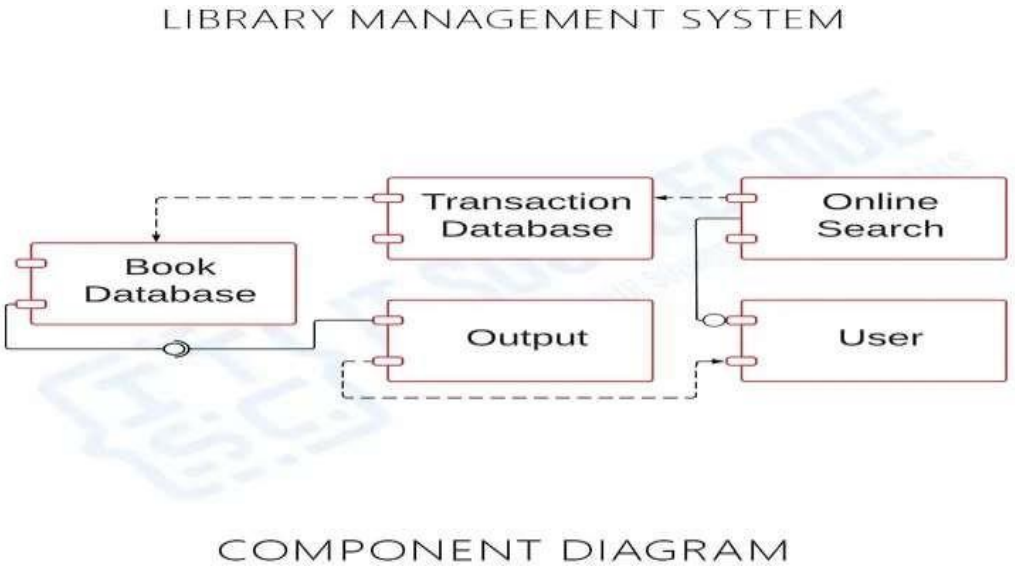
**Activity Diagram:** Activity Diagram to illustrate the activities between user and the system




**Deployment Diagram:** Represents the physical deployment of software components on hardware nodes.



**Component Diagram:** Component Diagram is used to visualize the architecture and distribution of the system.




# CASE STUDY

## COMPUTATIONAL PROBLEM SOLVING:

- Computational problem solving is the process of using computational tools, algorithms, and techniques to solve problems efficiently and effectively.
- It involves breaking down complex problems into smaller, more manageable tasks and then employing computational methods to analyze and find solutions.
- This approach is widely used in various fields, including computer science, engineering, mathematics, physics, and many other disciplines.
- The steps involved in computational problem solving typically include:

### 1) Problem Understanding:

Clearly defining the problem and understanding its requirements, constraints, and objectives. It is essential to have a thorough understanding of the problem's context before attempting to solve it computationally.

### 2) Algorithm Design:

Designing a step-by-step procedure or algorithm that outlines the solution to the problem. The algorithm should be precise, unambiguous, and capable of handling various input scenarios.

### 3) Data Representation:

Identifying the data required for the problem and determining the appropriate data structures to

represent and manipulate that data. Choosing the right data structures can significantly impact the efficiency of the solution.

### 4) Coding:

Translating the algorithm into a specific programming language and writing the code that implements the solution. This step involves using the syntax and features of the chosen programming language effectively.

### 5) Testing:

Thoroughly testing the code with different test cases to ensure that it produces the correct output for various inputs. Testing helps identify and fix any errors or bugs in the implementation.

### 6) Optimization:

Analyzing the algorithm and code to find ways to improve efficiency and reduce computational complexity. Optimization may involve reducing time and space complexity to make the solution more scalable and faster.

### 7) Execution and Evaluation:

Running the code on real data or inputs to obtain the solution and evaluating the results to determine its accuracy and effectiveness in solving the problem.

### 8) Iteration:

If necessary, revisiting the problem-solving process to make improvements or adjustments based on feedback or new requirements.

## Question Bank:

| UNIT – III: OBJECT ORIENTED CONCEPTS THROUGH PYTHON | | |
|---|---|---|
| **PART-A** | | |
| 1. | List out OOP's Concept | L1 |
| 2. | Difference between class an object | L2 |
| 3. | Explain polymorphism with an real time example | L2,L3 |
| 4. | Explain abstraction with an real time example | L2,L3 |
| 5. | List out inheritance types | L1 |
| 6. | What is encapsulation | L1,L2 |
| 7. | List out math library functions | L1 |
| 8. | list out random library functions | L1 |
| **PART-B** | | |
| 1. | Explain OOP'S concept in detail | L2 |
| 2. | What is polymorphism and how it is implemented in python | L2,L3 |
| 3 | Explain briefly about python inheritance | L2 |
| 4. | Write a basic class-object program in python | L3 |
| 5. | Illustrate Math and Random python Libraries | L3 |

# PYTHON PROGRAMMING
# UNIT – IV
# PANDAS AND NUMPY

# UNIT IV: PANDAS AND NUMPY

## 1. Unit Overview:

This unit focuses on data handling and numerical computation using Python libraries pandas and NumPy. Students will learn to work with structured data, perform operations for data cleaning, reshaping, filtering, grouping, and analyzing datasets. NumPy introduces efficient array computations, universal functions, and linear algebra operations. The unit equips students to handle real-world datasets and perform high-speed numerical calculations.

## 2. Objectives of the Unit:

By the end of this unit, students should be able to:

- Understand pandas data structures (Series, DataFrame) for data storage and manipulation.
- Perform indexing, selection, and filtering for data analysis.
- Reshape data using pivoting, stacking, and melting operations.
- Handle missing or null data effectively.
- Group and summarize data using aggregation functions.
- Read and write datasets from files (CSV, text).
- Understand NumPy arrays and their advantages over Python lists.
- Apply universal functions for vectorized operations.
- Perform linear algebra operations using linalg.

## 3. Learning Outcomes:

After completing this unit, students should will be able to:

- Create and manipulate Series and DataFrames in pandas.
- Filter and query datasets to extract useful information.
- Reshape data to suit analysis requirements.
- Handle missing data through filling or dropping values.
- Perform grouping and aggregation to summarize data.
- Read from and write data to external files for persistence.
- Use NumPy arrays and perform efficient numerical operations.
- Apply linear algebra functions to solve matrix problems.

## 4. Importance of Studying this Unit:

- Fundamental for data analysis, machine learning, and scientific computing.
- Enables handling of large datasets efficiently.
- Prepares students for real-world data processing tasks.
- Provides the foundation for advanced analytics and AI applications.

## 5. Key Concepts:

- **Pandas Series & DataFrame** – Core data structures.
- **Indexing & Selection** – Access rows/columns.
- **Filtering & Querying** – Extract based on conditions.
- **Reshaping Data** – Pivot, stack, melt.
- **Handling Missing Data** – Fill/drop nulls.
- **Grouping Data** – Aggregate by categories.
- **Reading/Writing Data** – read_csv, read_table.
- **NumPy Arrays** – Efficient numeric computation.
- **Universal Functions (ufuncs)** – Element-wise operations.
- **Linear Algebra** – Matrix operations using linalg

## 1. Introduction to pandas Data Structure

- ✓ Pandas and NumPy are essential Python libraries for data analysis and scientific computing.
- ✓ Pandas generally provide two data structures for manipulating data. They are:
  1) Series
  2) DataFrame

| SERIES | DATA FRAME |
|---|---|
| A Series is a one-dimensional labeled array that can hold any data type. Think of it like a single column in a spreadsheet with an index. | DataFrames are two-dimensional labeled data structures with columns potentially of different types. They resemble tables with rows and columns and are the most common pandas object |
| For example:<br>**pd.series(data, index)** creates a Series with data values and custom indexes | For example:<br>**pd.DataFrame(data, columns)** constructs a DataFrame with specified column names. |
| Pandas Series can be created from lists, dictionaries, scalar values, etc. | Pandas Series can be created from lists, dictionaries, scalar values, etc. |
| import pandas as pd<br>s = pd.series([10, 20, 30], index=['a', 'b', 'c']) | import pandas as pd<br>df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30]}) |
| Index : a b c<br>Data: 10 20 30 | Name age<br>Alice 25<br>Bob 30 |

To Read a CSV file

```
import pandas as pd
df = pd.read_csv("emp.csv")
print(df)
```

```
    eno    ename             desg    sal  deptno
0  e0001    smith    sales Person  20000      10
1  e0002    Jones         Manager  45000      20
2  e0003     King         Analyst  30000      10
3  e0004  krishna  Data Engineer  40000      10
4  e0005     khan         Analyst  35000      20
```

## 2. Essential functionality –

### 2.1 Indexing and Re-Indexing

- ✓ **Indexing in Pandas** refers to selecting specific rows and columns from a DataFrame.
- ✓ **also known as Subset Selection.**
- ✓ The three main types of indexing in Pandas are:

**2.1.1) DastaFrame[]:** Known as the indexing operator, used for basic selection.

**2.1.2) DataFrame.loc[]:** Label-based indexing for selecting data by row/column labels.

**2.1.3) DataFrame.iloc[]:** Position-based indexing for selecting data by row/column integer positions.

#### 2.1.1) DataFrame[]:

- ✓ Known as the indexing operator, used for basic selection.
- ✓ The most straightforward way to index data in Pandas is by using the [] operator.
- ✓ This method can be used to select individual columns or multiple columns

**a) To retrieve All Rows and Specific Column/Columns using [] operator**

```
[110] print(df['eno'])
```

```
0    e0001
1    e0002
2    e0003
3    e0004
4    e0005
Name: eno, dtype: object
```

```
print(df[["eno","ename"]])
```

```
    eno    ename
0  e0001    smith
1  e0002    Jones
2  e0003     King
3  e0004  krishna
4  e0005     khan
```

**b) To retrieve Specific Row/Rows and All Columns using [] operator**

```
# specific row and all columns
print(df[df['ename']=='smith'])
```

```
    eno  ename  age          desg    sal  deptno DESIGNATION  Dependents
0  e0001  smith   34  sales Person  20000      10         NaN           2
```

```
[118] # specific rows and all columns
      print(df[df['sal']>=40000])
```

```
    eno    ename  age           desg    sal  deptno DESIGNATION  Dependents
1  e0002    Jones   25        Manager  45000      20         NaN           1
3  e0004  krishna   35  Data Engineer  40000      10         NaN           2
```

## c) To retrieve Specific Rows and Specific Columns using [] operator

```
# specific rows and specific columns
print(df.loc[df['sal'] >= 40000, ['eno', 'sal']])
```

```
    eno    sal
1  e0002  45000
3  e0004  40000
```

### DataFrame.loc and DataFrame.iloc

```
[1] import pandas as pd
    df = pd.read_csv("emp.csv")
    print(df)
```

```
    eno    ename           desg    sal  deptno
0  e0001    smith   sales Person  20000      10
1  e0002    Jones        Manager  45000      20
2  e0003     King        Analyst  30000      10
3  e0004  krishna  Data Engineer  40000      10
4  e0005     khan        Analyst  35000      20
```

```
df.set_index("ename",inplace=True)
print(df)
```

```
           eno           desg    sal  deptno
ename
smith    e0001   sales Person  20000      10
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

| loc | Iloc |
|---|---|
| Label-based indexing for selecting data by row/column | Position-based indexing for selecting data by row/column |
| **To retrieve Specific Row** | |
| `print(df.loc['krishna'])`<br>`eno e0004`<br>`desg Data Engineer`<br>`sal 40000`<br>`deptno 10`<br>`Name: krishna, dtype: object` | `print(df.iloc[3])`<br>`eno e0004`<br>`desg Data Engineer`<br>`sal 40000`<br>`deptno 10`<br>`Name: krishna, dtype: object` |
| **To retrieve Specific Rows** | |
| `print(df.loc[['smith','Jones']])`<br>`eno desg sal deptno`<br>`ename`<br>`smith e0001 sales Person 20000 10`<br>`Jones e0002 Manager 45000 20` | `print(df.iloc[0:2])`<br>`eno desg sal dept`<br>`ename`<br>`smith e0001 sales Person 20000`<br>`Jones e0002 Manager 45000` |
| **To retrieve Specific Column** | |
| `print(df.loc[:,['desg']])`<br>`desg`<br>`ename`<br>`smith sales Person`<br>`Jones Manager`<br>`King Analyst`<br>`krishna Data Engineer`<br>`khan Analyst` | `print(df.iloc[:,[1]])`<br>`desg`<br>`ename`<br>`smith sales Person`<br>`Jones Manager`<br>`King Analyst`<br>`krishna Data Engineer`<br>`khan Analyst` |

**To retrieve Specific Columns**

```
[11] print(df.loc[:,['eno','sal']])
```

| ename | eno | sal |
|---|---|---|
| smith | e0001 | 20000 |
| Jones | e0002 | 45000 |
| King | e0003 | 30000 |
| krishna | e0004 | 40000 |
| khan | e0005 | 35000 |

```
[12] print(df.iloc[:,[0,2]])
```

| ename | eno | sal |
|---|---|---|
| smith | e0001 | 20000 |
| Jones | e0002 | 45000 |
| King | e0003 | 30000 |
| krishna | e0004 | 40000 |
| khan | e0005 | 35000 |

**To retrieve Specific Rows and Specific Columns**

```
print(df.loc[['smith','Jones'],['eno','sal']])
```

| ename | eno | sal |
|---|---|---|
| smith | e0001 | 20000 |
| Jones | e0002 | 45000 |

```
print(df.iloc[[0,1],[0,2]])
```

| ename | eno | sal |
|---|---|---|
| smith | e0001 | 20000 |
| Jones | e0002 | 45000 |

**To retrieve All Rows and All Columns**

```
print(df.loc[:,:])
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| smith | e0001 | sales Person | 20000 | 10 |
| Jones | e0002 | Manager | 45000 | 20 |
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |
| khan | e0005 | Analyst | 35000 | 20 |

```
print(df.iloc[:,:])
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| smith | e0001 | sales Person | 20000 | 10 |
| Jones | e0002 | Manager | 45000 | 20 |
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |
| khan | e0005 | Analyst | 35000 | 20 |

## 2.2) Selection and filtering

Filtering and selection are fundamental operations when working with data in Pandas. They allow you to extract specific subsets of data that meet certain conditions

**Single Condition Filtering with Comparison Operators**

```
# To get employees whose salary is greater than or equal to 30000
sal_ft =df['sal'] >= 30000
print (df[sal_ft])
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| Jones | e0002 | Manager | 45000 | 20 |
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |
| khan | e0005 | Analyst | 35000 | 20 |

**Combining Multiple Conditions with Logical Operators**

```
# To get employees whose salary is greater than or equal to 30000 and deptno = 10
sal_ft = (df['sal'] >= 30000) & (df['deptno']==10)
print (df[sal_ft])
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |

**Using .query() Method**

```
print(df.query('sal >= 3000'))
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| smith | e0001 | sales Person | 20000 | 10 |
| Jones | e0002 | Manager | 45000 | 20 |
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |
| khan | e0005 | Analyst | 35000 | 20 |

```
print(df.query('sal >= 3000 & deptno==10'))
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| smith | e0001 | sales Person | 20000 | 10 |
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |

**Filtering with String Conditions**

```
res = df['desg']=='Analyst'
print(df[res])
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| King | e0003 | Analyst | 30000 | 10 |
| khan | e0005 | Analyst | 35000 | 20 |

**Filtering with isin() Method**

```
desg_ft = df['desg'].isin(['Analyst', 'Data Engineer'])
print(df[desg_ft])
```

| ename | eno | desg | sal | deptno |
|---|---|---|---|---|
| King | e0003 | Analyst | 30000 | 10 |
| krishna | e0004 | Data Engineer | 40000 | 10 |
| khan | e0005 | Analyst | 35000 | 20 |

## 2.3) reshaping

- ✓ Reshaping in Pandas is a fundamental set of operations that allows to change the structure (the number of rows and columns) of the DataFrame or Series.
- ✓ This is often necessary to prepare data for analysis, visualization, or integration with other datasets.

### Reshape DataFrame in Pandas

- ✓ Below are the two methods that are used to reshape the layout of data in Pandas:
  1) Using Pandas stack() method
  2) Using unstack() method

### 1) Reshape the Layout of Tables in Pandas Using stack() method

- ✓ Pivots a DataFrame from a "wide" format to a "long" format by stacking columns into rows, creating a hierarchical index (MultiIndex) on the rows.
- ✓ Use **stack()** to go from wide to long when you want column labels to become part of a hierarchical row index.

Original Data Frame

```
print(df)
```

```
           eno           desg    sal  deptno
ename
smith    e0001   sales Person  20000      10
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

Stacked Dataframe

```
stacked_df = df.stack()
print(stacked_df)
```

```
0  ename             smith
   eno              e0001
   desg       sales Person
   sal              20000
   deptno              10
1  ename             Jones
   eno              e0002
   desg           Manager
   sal              45000
   deptno              20
2  ename              King
   eno              e0003
   desg           Analyst
   sal              30000
   deptno              10
3  ename            krishna
   eno              e0004
   desg      Data Engineer
   sal              40000
   deptno              10
4  ename              khan
   eno              e0005
   desg           Analyst
   sal              35000
   deptno              20
```

**2) Reshape a Pandas DataFrame Using unstack() method**

- ✓ The inverse of stack(). It pivots a DataFrame or Series with a MultiIndex (typically created by stack()) from a "long" format back to a "wide" format, with the inner level of the row index becoming the new column labels.
- ✓ Use **unstack()** to go from long (with a MultiIndex) back to wide, making an inner level of the row index the new columns.

```
unstacked_df = stacked_df.unstack()
print(unstacked_df)
```

```
     ename    eno           desg    sal deptno
0    smith  e0001   sales Person  20000     10
1    Jones  e0002        Manager  45000     20
2     King  e0003        Analyst  30000     10
3  krishna  e0004  Data Engineer  40000     10
4     khan  e0005        Analyst  35000     20
```

## 2.4) summarizing and computing descriptive statistics,

- ✓ Python Pandas provides a rich set of tools and methods for **summarizing and aggregating the data** within DataFrames and Series.
- ✓ These operations allows to get ***concise overviews, calculate descriptive statistics, and group data for more insightful summaries.***
- ✓ ***Two methods of computing summary statistics using Pandas are***
  1) Using describe() for Descriptive Statistics
  2) Using Individual Aggregation Functions:

### 1) Using describe() for Descriptive Statistics

.describe(): This is the most common and powerful function for getting a quick statistical summary of your numerical columns. It provides:

count: Number of non-missing values.
mean: Average.
std: Standard deviation.
min: Minimum value.
25% (Q1): First quartile.
50% (median or Q2): Second quartile.
75% (Q3): Third quartile.
max: Maximum value.

```
print(df['deptno'].describe())
```

```
count     5.000000
mean     14.000000
std       5.477226
min      10.000000
25%      10.000000
50%      10.000000
75%      20.000000
max      20.000000
Name: deptno, dtype: float64
```

### 2) Using Individual Aggregation Functions:

`.mean()`: Calculate the mean.
`.median()`: Calculate the median.
`.std()`: Calculate the standard deviation.
`.min()`: Get the minimum value.
`.max()`: Get the maximum value.
`.count()`: Count non-missing values.

```
     ename    eno           desg    sal deptno
0    smith  e0001   sales Person  20000     10
1    Jones  e0002        Manager  45000     20
2     King  e0003        Analyst  30000     10
3  krishna  e0004  Data Engineer  40000     10
4     khan  e0005        Analyst  35000     20
```

```
print("sum of salaries",df['sal'].sum())
print("Average of Salaries",df['sal'].mean())
print("Max of Salaries",df['sal'].max())
print("Min of Salaries",df['sal'].min())
print("Number of Employees",df['eno'].count())
```

```
sum of salaries 170000
Average of Salaries 34000.0
Max of Salaries 45000
Min of Salaries 20000
Number of Employees 5
```

## 2.5) Handling missing data,

- ✓ In Pandas, missing values, often represented as NaN (Not a Number), can cause problems during data processing and analysis.
- ✓ These gaps in data can lead to incorrect analysis and misleading conclusions.
- ✓ Pandas provides several functions and methods to
  1) ***identify, and***
  2) ***handle***
  3)

### 1) *Identifying Missing Values:*

- ✓ **isna() or isnull():** These functions detect missing values and return a DataFrame or Series of boolean values. **True** indicates a missing value (NaN), and **False** indicates a non-missing value.

**To find Missing Values in Each Column using isnull() and isna()**

```
df1 = pd.read_csv('stu.csv')
print(df1)
```

```
   sno           sname branch  mark1  mark2  mark3
0  100            Ivan    MCA   79.0   66.0   77.0
1  101           Korth    NaN   65.0   45.0   56.0
2  102   James Gosling    MBA    NaN    NaN   87.0
3  103     Chris Bates    MCA   66.0   54.0    NaN
4  104    Nageswar Rao    NaN   89.0    NaN    NaN
5  105  Bala Guruswamy    MCA    NaN   55.0   66.0
```

```
[35] print(df1.isnull())
```

```
     sno  sname  branch  mark1  mark2  mark3
0  False  False   False  False  False  False
1  False  False    True  False  False  False
2  False  False   False   True   True  False
3  False  False   False  False  False   True
4  False  False    True  False   True   True
5  False  False   False   True  False  False
```

```
print(df1.isna())
```

```
     sno  sname  branch  mark1  mark2  mark3
0  False  False   False  False  False  False
1  False  False    True  False  False  False
2  False  False   False   True   True  False
3  False  False   False  False  False   True
4  False  False    True  False   True   True
5  False  False   False   True  False  False
```

**To find Number of Missing Values in Each Column**

```
#to get number of missing values in each column
print(df1.isnull().sum())
```

```
sno       0
sname     0
branch    2
mark1     2
mark2     2
mark3     2
dtype: int64
```

## 2) Handling Missing Values:

### 2.1) Dropping Missing Values:

- ***dropna(): This function removes rows or columns that contain missing values.***
  - *axis=0 (default): Drops rows containing at least one missing value.*
  - *axis=1: Drops columns containing at least one missing value.*
  - *how='any' (default): Drop row/column if any missing values are present.*
  - *how='all': Drop row/column if all values are missing.*

```
[40] print(df1)
```

```
   sno           sname branch  mark1  mark2  mark3
0  100            Ivan    MCA   79.0   66.0   77.0
1  101           Korth    NaN   65.0   45.0   56.0
2  102   James Gosling    MBA    NaN    NaN   87.0
3  103     Chris Bates    MCA   66.0   54.0    NaN
4  104    Nageswar Rao    NaN   89.0    NaN    NaN
5  105  Bala Guruswamy    MCA    NaN   55.0   66.0
```

```
[46] #Dropping Rows with At Least One Null Value
     print(df1.dropna())
```

```
   sno sname branch  mark1  mark2  mark3
0  100  Ivan    MCA   79.0   66.0   77.0
```

```
[42] # drop rows with all NaN values
     print(df1.dropna(axis=0))
```

```
   sno sname branch  mark1  mark2  mark3
0  100  Ivan    MCA   79.0   66.0   77.0
```

```
[45] #drop columns with all NaN values
     print(df1.dropna(axis=1))
```

```
   sno           sname
0  100            Ivan
1  101           Korth
2  102   James Gosling
3  103     Chris Bates
4  104    Nageswar Rao
5  105  Bala Guruswamy
```

```
#Dropping Rows with All Null Values
print(df1.dropna(how='all'))
```

```
   sno           sname branch  mark1  mark2  mark3
0  100            Ivan    MCA   79.0   66.0   77.0
1  101           Korth    NaN   65.0   45.0   56.0
2  102   James Gosling    MBA    NaN    NaN   87.0
3  103     Chris Bates    MCA   66.0   54.0    NaN
4  104    Nageswar Rao    NaN   89.0    NaN    NaN
5  105  Bala Guruswamy    MCA    NaN   55.0   66.0
```

```
[48] #Dropping Rows/Columns with ANY Null Values
     print(df1.dropna(how='any'))
```

## *2.2) Filling Missing Values*

2.2.1) using **fillna()**
2.2.2) using **replace()**

### 2.2.1) Using fillna()

**2.2.1.1) fillna(value)**
**2.2.1.2) fillna(method="pad|ffill")**
**2.2.1.3) fillna(df['col'].mean()|median()|mode())**

***2.2.1.1) fillna(value):*** *Replaces all missing values with a specified scalar value.*

***Original data***

```
# Importing pandas and numpy
import pandas as pd
import numpy as np

# Sample DataFrame with missing values
d = {'First Score': [100, 90, np.nan, 95],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score': [np.nan, 40, 80, 98]}
df = pd.DataFrame(d)
print(df)
```

```
   First Score  Second Score  Third Score
0        100.0          30.0          NaN
1         90.0          45.0         40.0
2          NaN          56.0         80.0
3         95.0           NaN         98.0
```

***Fillna(value)***

```
print(df.fillna(10000))
```

```
   First Score  Second Score  Third Score
0        100.0          30.0      10000.0
1         90.0          45.0         40.0
2      10000.0          56.0         80.0
3         95.0       10000.0         98.0
```

### *2.2.1.2) fillna(method = 'pad|ffill'*

- ✓ **fillna(method='ffill' or 'pad'):** *Forward fill - propagates the last valid observation forward to the next missing value.*
- ✓ **fillna(method='bfill' or 'backfill'):** *Backward fill - propagates the next valid observation backward to the previous missing value.*
- ✓ **ffill will not fill leading NaN values (as there's no previous valid value).**
- ✓ **bfill or pad will not fill trailing NaN values (as there's no subsequent valid value).**

| | First Score | Second Score | Third Score |
|---|---|---|---|
| 0 | 100.0 | 30.0 | NaN |
| 1 | 90.0 | 45.0 | 40.0 |
| 2 | NaN | 56.0 | 80.0 |
| 3 | 95.0 | NaN | 98.0 |

Example: Fill with Previous Value (Forward Fill)

```
df.fillna(method='pad')  # Forward fill
```

Output

| | First Score | Second Score | Third Score |
|---|---|---|---|
| 0 | 100.0 | 30.0 | NaN |
| 1 | 90.0 | 45.0 | 40.0 |
| 2 | 90.0 | 56.0 | 80.0 |
| 3 | 95.0 | 56.0 | 98.0 |

Example: Fill with Next Value (Backward Fill)

```
df.fillna(method='bfill')  # Backward fill
```

Output

| | First Score | Second Score | Third Score |
|---|---|---|---|
| 0 | 100.0 | 30.0 | 40.0 |
| 1 | 90.0 | 45.0 | 40.0 |
| 2 | 95.0 | 56.0 | 80.0 |
| 3 | 95.0 | NaN | 98.0 |

*2.2.1.3)* ***Imputation using statistical measures: You can fill missing values with the mean, median, or mode of the respective column.***

```
print(df)
```

```
   First Score  Second Score  Third Score
0        100.0          30.0          NaN
1         90.0          45.0         40.0
2          NaN          56.0         80.0
3         95.0           NaN         98.0
```

```
[59] print(df['First Score'].fillna(df['First Score'].mean()))
```

```
0    100.0
1     90.0
2     95.0
3     95.0
Name: First Score, dtype: float64
```

```
[61] print(df['Second Score'].fillna(df['Second Score'].median()))
```

```
0    30.0
1    45.0
2    56.0
3    45.0
Name: Second Score, dtype: float64
```

```
print(df['Third Score'].fillna(df['Third Score'].mode()))
```

```
0    40.0
1    40.0
2    80.0
3    98.0
```

**

## 2.6) filter and query methods,

- ✓ ***Use df.query() when you want a more readable, string-based way to filter rows based on column values.***
- ✓ ***Use df.filter() when you need to select columns or rows based on the names or patterns of their labels.***

```
df.filter(items=None, like=None, regex=None, axis=None)
```

**Parameters:**

- **items:** A list-like of labels to keep. These should be in the `axis` specified.
- **like:** Keep labels where the string `like` is found in the label.
- **regex:** Keep labels matching the regular expression `regex`.
- **axis:** The axis to filter on.
  - `0` or `'index'` (default): Filters rows based on index labels.
  - `1` or `'columns'`: Filters columns based on column labels.

**Examples:**

```
#filter method
res = df2.filter(items=['eno','sal'])
print(res)
```

```
    eno    sal
0  e0001  20000
1  e0002  45000
2  e0003  30000
3  e0004  40000
4  e0005  35000
```

```
df2.set_index('ename',inplace=True)
res=df2.filter(regex='g$', axis=0)
print(res)
```

```
         eno     desg    sal  deptno
ename
King   e0003  Analyst  30000      10
```

```
res=df2.filter(like='n', axis=0)
print(res)
```

```
           eno            desg    sal  deptno
ename
Jones    e0002         Manager  45000      20
King     e0003         Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005         Analyst  35000      20
```

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
        'Age': [25, 30, 22, 35, 28],
        'City': ['New York', 'London', 'Paris', 'London', 'New York'],
        'Score_Math': [80, 90, 75, 85, 92],
        'Score_Science': [70, 85, 80, 90, 78]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

print("\n--- df.query() Examples (Filtering Rows by Column Values)
---")
```

```
# 1. Basic condition on a single column
query_age_over_25 = df.query('Age > 25')
print("\nRows where Age > 25:\n", query_age_over_25)

# 2. Multiple conditions using 'and'
query_london_age_over_25 = df.query('City == "London" and Age > 25')
print("\nRows where City is 'London' and Age > 25:\n",
query_london_age_over_25)

# 3. Using 'or'
query_score_math_over_90_or_age_under_23 = df.query('Score_Math > 90 or
Age < 23')
print("\nRows where Score_Math > 90 or Age < 23:\n",
query_score_math_over_90_or_age_under_23)

# 4. Using a Python variable in the query
min_score = 80
query_score_math_above_variable = df.query('Score_Math > @min_score')
print(f"\nRows where Score_Math > {min_score}:\n",
query_score_math_above_variable)

# 5. Using 'in'
cities_of_interest = ['New York', 'Paris']
query_city_in_list = df.query('City in @cities_of_interest')
print(f"\nRows where City is in {cities_of_interest}:\n",
query_city_in_list)

print("\n--- df.filter() Examples (Filtering Columns or Index Labels)
---")

# Set 'Name' as index for row filtering examples
df_indexed = df.set_index('Name')
print("\nDataFrame with 'Name' as index:\n", df_indexed)

# 1. Filtering columns using 'items'
filter_cols_items = df.filter(items=['Name', 'Age'])
print("\nColumns 'Name' and 'Age':\n", filter_cols_items)

# 2. Filtering columns using 'like'
filter_cols_like_score = df.filter(like='Score_')
print("\nColumns containing 'Score_':\n", filter_cols_like_score)

# 3. Filtering columns using 'regex'
filter_cols_regex_ends_th = df.filter(regex='th$', axis='columns')
print("\nColumns ending with 'th':\n", filter_cols_regex_ends_th)

# 4. Filtering index labels using 'items'
```

```
filter_rows_items = df_indexed.filter(items=['Alice', 'Charlie'],
axis='index')
print("\nRows with index 'Alice' and 'Charlie':\n", filter_rows_items)

# 5. Filtering index labels using 'like'
filter_rows_like_a = df_indexed.filter(like='a', axis='index')
print("\nRows with index containing 'a':\n", filter_rows_like_a)

# 6. Filtering index labels using 'regex'
filter_rows_regex_starts_with_D = df_indexed.filter(regex='^D',
axis='index')
print("\nRows with index starting with 'D':\n",
filter_rows_regex_starts_with_D)
```

## 2.7) Grouping

- ✓ ***groupby() as a way to split your DataFrame into smaller chunks or groups based on the unique values found in one or more specific columns.***
- ✓ ***Once groups are created , then apply calculations or transformations to each group independently and finally combine the results back into a structured format.***

- ✓ **`groupby()` follows a "Split-Apply-Combine" strategy:**

1) **Split:** The DataFrame is divided into multiple groups based on the values in the specified column(s). For example, if you group by a 'City' column, you'll get separate groups for each unique city (e.g., one group for 'London', one for 'Paris', etc.).
2) **Apply:** You then apply a function to each of these individual groups. Common operations include:
    a. **Aggregation:** Calculating summary statistics like mean(), sum(), count(), min(), max(), etc., for each group.
3) ***Combine: The results of the applied function on each group are then combined back into a new DataFrame or Series.***

```
[9] # Department number wise total salary
    res = df2.groupby('deptno')['sal'].sum()
    print(res)
```

```
deptno
10    90000
20    80000
Name: sal, dtype: int64
```

```
[10] # deptno wise minimum salary
     res = df2.groupby('deptno')['sal'].min()
     print(res)
```

```
deptno
10    20000
20    35000
Name: sal, dtype: int64
```

```
[11] #deptno wise Maximum salary
     res = df2.groupby('deptno')['sal'].max()
     print(res)
```

```
deptno
10    40000
20    45000
Name: sal, dtype: int64
```

```
# designation wise number of employees
res1= df2.groupby('desg')['eno'].count()
print(res1)
```

```
desg
Analyst          2
Data Engineer    1
Manager          1
sales Person     1
Name: eno, dtype: int64
```

## 3) reading and writing data in text format –

### 3.1) read_csv,

### 3.2) read_table.

***3.1) read_csv :***

- ✓ ***CSV files are the Comma Separated Files. It allows users to load tabular data into a DataFrame, which is a powerful structure for data manipulation and analysis.***
- ✓ ***To access data from the CSV file, we require a function read_csv() from Pandas that retrieves data in the form of the data frame***
- ✓ ***Syntax: pd.read_csv(filepath, sep=' ,' , header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)***

| | |
|---|---|
| filename | The path to the CSV file |
| sep | Default(' , ') |
| header | ✓ default 'infer'<br>✓ Specifies which row(s) to use as the column names<br>✓ 0: Use the first row as the header.<br>✓ 1: Use the second row as the header (0-indexed).<br>✓ None: The file has no header row, and Pandas will assign default integer column names. |
| index_col | ✓ default None<br>✓ Specifies which column(s) to use as the row index of the DataFrame |
| usecols | ✓ default None<br>✓ A list of column names or column numbers (0-indexed) to read from the file. |
| skiprows | ✓ default None<br>✓ Number of rows to skip at the beginning of the file (int). |
| nrows | ✓ default None<br>✓ Number of rows of the file to read |

```
    ename    eno           desg    sal deptno
0    smith  e0001   sales Person  20000     10
1    Jones  e0002        Manager  45000     20
2     King  e0003        Analyst  30000     10
3  krishna  e0004  Data Engineer  40000     10
4     khan  e0005        Analyst  35000     20
```

```
df3 = pd.read_csv('emp.csv',usecols=[0,1,2],skiprows=[2,3])
print(df3)
```

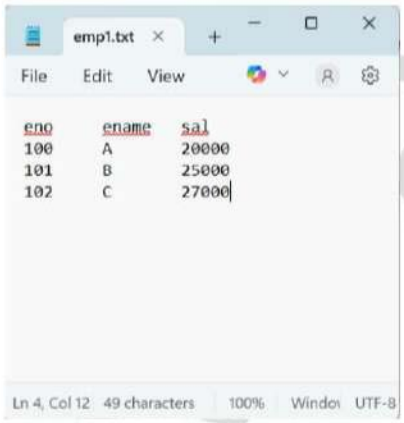
```
    eno    ename           desg
0  e0001    smith   sales Person
1  e0004  krishna  Data Engineer
2  e0005     khan        Analyst
```

## 3.2) read_table.

***The primary purpose of pd.read_table() is to read data from a file (or a URL, file-like object, etc.) where values are separated by a delimiter other than a comma (although it can also handle comma-separated files).***

✓ ***Syntax: pd.read_csv(filepath, sep=' ,' , header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)***

| | |
|---|---|
| filename | The path to the CSV file |
| sep | Default(' \t') |
| header | ✓ default 'infer'<br>✓ Specifies which row(s) to use as the column names<br>✓ 0: Use the first row as the header.<br>✓ 1: Use the second row as the header (0-indexed).<br>✓ None: The file has no header row, and Pandas will assign default integer column names. |
| index_col | ✓ default None<br>✓ Specifies which column(s) to use as the row index of the DataFrame |
| usecols | ✓ default None<br>✓ A list of column names or column numbers (0-indexed) to read from the file. |
| skiprows | ✓ default None<br>✓ Number of rows to skip at the beginning of the file (int). |
| nrows | ✓ default None<br>✓ Number of rows of the file to read |




```
df4 = pd.read_table('emp1.txt',sep='\t')
print(df4)
```

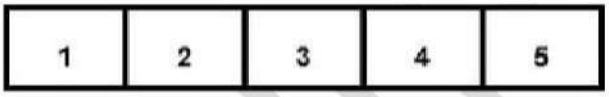
```
   eno ename    sal
0  100     A  20000
1  101     B  25000
2  102     C  27000
```

| Feature | pd.read_csv() | pd.read_table() |
|---|---|---|
| Primary Use | Comma-separated files (.csv) | General delimited files |
| Default sep | , (comma) | \t (tab) |
| Flexibility | Less flexible regarding separator | More flexible (explicitly set sep ) |
| Convenience | More convenient for CSV files | Requires specifying sep for non-tab delimited |

# 4.NUMPY

- ✓ *NumPy stands for Numerical Python.*
- ✓ *It is a Python library used for working with an array.*
- ✓ *In Python, we use the list for the array but it's slow to process.*
- ✓ *NumPy array is a powerful N-dimensional array object and is used in linear algebra, Fourier transform, and random number capabilities.*
- ✓ *It provides an array object much faster than traditional Python lists.*
- ✓ *Types of Array:*
    1. *One Dimensional Array*
    2. *Multi-Dimensional Array*

1. *One Dimensional Array:*
   - ✓ *A one-dimensional array is a type of linear array.*




*One Dimensional Array*

Example:

```
# importing numpy module
import numpy as np

# creating list
list = [1, 2, 3, 4]

# creating numpy array
sample_array = np.array(list)

print("List in python : ", list)

print("Numpy Array in python :",
      sample_array)
```

Output:

```
List in python :  [1, 2, 3, 4]
Numpy Array in python :  [1 2 3 4]
```

Example:

```
# importing numpy module
import numpy as np

# creating list
list_1 = [1, 2, 3, 4]
list_2 = [5, 6, 7, 8]
list_3 = [9, 10, 11, 12]

# creating numpy array
sample_array = np.array([list_1,
                         list_2,
                         list_3])

print("Numpy multi dimensional array in python\n",
      sample_array)
```

Output:

```
Numpy multi dimensional array in python
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

numpy.array(object, dtype=None, copy=True, order='K', ndmin=0)

This is the full syntax for the np.array() function in NumPy. Let's break down each parameter:

- **object (required):**
  - This is the **array-like object** you want to convert into a NumPy array. It can be a:
    - Python list or nested list
    - Python tuple or nested tuple
    - Another NumPy array
    - Any object that exposes the array interface
    - Any object whose __array__ method returns an array
- **dtype (optional):**
  - The desired data type of the resulting array. If None (the default), NumPy will try to infer the data type from the object.
  - You can explicitly specify a NumPy data type (e.g., np.int32, np.float64, np.str_).
- **copy (optional, default=True):**
  - If True, a new copy of the object is made.
  - If False, NumPy will try to return a view of the original object if possible. Be careful with this, as modifications to the view might affect the original object.
- **order (optional, default='K'):**
  - Specifies the memory layout of the array:
    - 'C': C-like row-major order.
    - 'F': Fortran-like column-major order.
    - 'A': Allow any order (NumPy might choose based on input).

- 'K': Keep order (try to match the input's order as closely as possible).

- **ndmin (optional, default=0):**
  - Specifies the minimum number of dimensions that the resulting array should have. NumPy will add leading axes (dimensions of size 1) to meet this requirement.

In simpler terms, you'll often use np.array() with just the object you want to convert and sometimes specify the dtype. The other parameters are for more advanced control over how the array is created.

| Function Name | Purpose | Basic Syntax | Example |
|---|---|---|---|
| np.array() | Creates an array from an existing list, tuple, or array-like object. | np.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0) | np.array([1, 2, 3]) |
| np.zeros() | Creates an array filled with zeros. | np.zeros(shape, dtype=float, order='C') | np.zeros((2, 3)) (creates a 2x3 array of zeros) |
| np.ones() | Creates an array filled with ones. | np.ones(shape, dtype=float, order='C') | np.ones(5) (creates an array of five ones) |
| np.empty() | Creates an array without initializing entries (can contain garbage values). | np.empty(shape, dtype=float, order='C') | np.empty((2, 2)) (creates a 2x2 uninitialized array) |
| np.full() | Creates an array filled with a specified scalar value. | np.full(shape, fill_value, dtype=None, order='C') | np.full((3,), 7) (creates an array of three sevens) |
| np.arange() | Creates an array with evenly spaced values within a given interval. | np.arange([start,] stop, [step,] dtype=None, *, like=None) | np.arange(0, 10, 2) (creates [0, 2, 4, 6, 8]) |
| np.identity() | Creates a square identity array (similar to np.eye() for square matrices). | np.identity(n, dtype=None, *, like=None) | np.identity(4) (creates a 4x4 identity matrix) |

```
[2] # Creating single Dimensional Array from List using Numpy
    import numpy as np
    l1=[45,23,67,54]
    a1= np.array(l1)
    print("Single Dimensio Array with Numpy ",a1)
```

```
Single Dimensio Array with Numpy  [45 23 67 54]
```

```
[4] # Creating Two Dimensional Array from Lists using Numpy
    l1=[1,2,3]
    l2=['hai','hello','hi']
    l3=[2.4,5.6,3.4]
    a2 = np.array([l1,l2,l3])
    print(a2)
```

```
[['1' '2' '3']
 ['hai' 'hello' 'hi']
 ['2.4' '5.6' '3.4']]
```

```
[7] # Creates an array filled with zeros.
    zarr= np.zeros((2,3))
    print(zarr)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

```
[8] # Creates an array filled with ones.
    oarr= np.ones((2,3))
    print(oarr)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
[14] #Creates an array without initializing entries (can contain garbage values).
     earr = np.empty((2,2))
     print(earr)
```

```
[[2.22e-322 1.14e-322]
 [3.31e-322 2.67e-322]]
```

```
[16] # Creates an array filled with a specified scalar value.
     res = np.full((3),7)
     print(res)
```

```
[7 7 7]
```

```
[18] # Creates an array with evenly spaced values within a given interval.
     res1 = np.arange(0,10,2)
     print(res1)
```

```
[0 2 4 6 8]
```

**Universal Functions – Basic Unary Functions**

| Function | Description | Example |
|---|---|---|
| np.abs() | Element-wise absolute value. | np.abs(np.array([-1, -2, 3])) -> [1, 2, 3] |
| np.fabs() | Element-wise absolute value (for floating-point types). | np.fabs(np.array([-1.5, -2.8, 3.1])) -> [1.5, 2.8, 3.1] |
| np.sqrt() | Element-wise square root. | np.sqrt(np.array([4, 9, 16])) -> [2., 3., 4.] |
| np.exp() | Element-wise exponential (e^x). | np.exp(np.array([0, 1, 2])) -> [1. , 2.71828183, 7.3890561 ] |
| np.ceil() | Element-wise ceiling (smallest integer >= x). | np.ceil(np.array([1.2, 2.7, -1.5])) -> [ 2., 3., -1.] |
| np.floor() | Element-wise floor (largest integer <= x). | np.floor(np.array([1.2, 2.7, -1.5])) -> [ 1., 2., -2.] |
| np.round() | Element-wise rounding to the nearest integer. | np.round(np.array([1.4, 1.6, -1.4, -1.6])) -> [ 1., 2., -1., -2.] |
| np.rint() | Element-wise rounding to the nearest integer. | np.rint(np.array([1.4, 1.6, -1.4, -1.6])) -> [ 1., 2., -1., -2.] |
| np.sign() | Element-wise sign (+1 for positive, -1 for negative, 0 for zero). | np.sign(np.array([-2, 0, 3])) -> [-1, 0, 1] |
| np.cos() | Element-wise cosine. | np.cos(np.array([0, np.pi])) -> [ 1. , -1.] |
| np.sin() | Element-wise sine. | np.sin(np.array([0, np.pi/2])) -> [ 0. , 1.] |
| np.tan() | Element-wise tangent. | np.tan(np.array([0, np.pi/4])) -> [ 0. , 1.] |
| np.isnan() | Element-wise check for NaN (Not a Number). | np.isnan(np.array([1, np.nan, 3])) -> [False, True, False] |
| np.isfinite() | Element-wise check for finite numbers (not NaN or inf). | np.isfinite(np.array([1, np.inf, np.nan])) -> [ True, False, False] |
| np.isinf() | Element-wise check for infinity. | np.isinf(np.array([1, np.inf, np.nan])) -> [False, True, False] |

**Binary functions ,**

| Function | Description | Example |
|---|---|---|
| np.add() | Element-wise addition. | np.add(np.array([1, 2]), np.array([3, 4])) -> [4, 6] |
| np.subtract() | Element-wise subtraction (second array from the first). | np.subtract(np.array([5, 3]), np.array([2, 1])) -> [3, 2] |
| np.multiply() | Element-wise multiplication. | np.multiply(np.array([2, 3]), np.array([4, 5])) -> [ 8, 15] |
| np.divide() | Element-wise division (first array by the second). | np.divide(np.array([10, 6]), np.array([2, 3])) -> [5., 2.] |
| np.floor_divide() | Element-wise floor division. | np.floor_divide(np.array([10, 7]), np.array([3, 3])) -> [3, 2] |
| np.power() | Element-wise exponentiation (first array to the power of the second). | np.power(np.array([2, 3]), np.array([3, 2])) -> [8, 9] |
| np.mod() | Element-wise modulo (remainder of division). | np.mod(np.array([7, 9]), np.array([3, 4])) -> [1, 1] |
| np.equal() | Element-wise equality comparison. | np.equal(np.array([1, 2]), np.array([1, 3])) -> [ True, False] |
| np.not_equal() | Element-wise inequality comparison. | np.not_equal(np.array([1, 2]), np.array([1, 3])) -> [False, True] |
| np.less() | Element-wise less than comparison. | np.less(np.array([1, 2]), np.array([2, 1])) -> [ True, False] |
| np.greater() | Element-wise greater than comparison. | np.greater(np.array([1, 2]), np.array([0, 3])) -> [ True, False] |
| np.less_equal() | Element-wise less than or equal to comparison. | np.less_equal(np.array([1, 2]), np.array([2, 2])) -> [ True, True] |
| np.greater_equal() | Element-wise greater than or equal to comparison. | np.greater_equal(np.array([1, 2]), np.array([0, 2])) -> [ True, True] |
| np.logical_and() | Element-wise logical AND. | np.logical_and(np.array([True, False]), np.array([True, True])) -> [ True, False] |
| np.logical_or() | Element-wise logical OR. | np.logical_or(np.array([True, False]), np.array([False, True])) -> [ True, True] |
| np.logical_not() | Element-wise logical NOT (unary, but related). | np.logical_not(np.array([True, False])) -> [False, True] |

## 6) File Input and Output with arrays

When working with numerical data using the NumPy library in Python, two fundamental tasks frequently arise:

Saving Arrays: The need to store NumPy arrays, whether created through computation or manipulation, to disk. This persistence allows for later use, sharing of data, and the preservation of work across different sessions or systems.

Loading Arrays: The requirement to read data stored in files back into NumPy arrays for analysis and processing. This enables the utilization of previously saved datasets or data originating from external sources.

Building complete data analysis workflows with NumPy crucially requires understanding the input and output capabilities, as they allow to persist the data and load it back seamlessly for further computation and analysis.

### Saving NumPy Arrays:

1) **`np.save(file, arr)`**: Saves a single NumPy array to a binary `.npy` file.
2) **`np.savez(file, *args, **kwds)`**: Saves one or more NumPy arrays to a single uncompressed `.npz` archive.
3) **`np.savez_compressed(file, *args, **kwds)`**: Saves one or more NumPy arrays to a single compressed `.npz` archive.
4) **`np.savetxt(fname, X, fmt='%.18e', delimiter=' ', ...)`**: Saves a NumPy array to a plain text file.

### Loading NumPy Arrays:

1) **`np.load(file, mmap_mode=None, allow_pickle=True, ...)`**: Loads arrays from `.npy` or `.npz` files. Returns a single array for `.npy` or a dictionary-like object for `.npz`.
2) **`np.loadtxt(fname, dtype=float, delimiter=None, skiprows=0, ...)`**: Loads data from a plain text file into a NumPy array.

#### Saving NumPy Arrays

1) **`np.save(file, arr)`**: Saves a single NumPy array to a binary `.npy` file.

```
import numpy as np

# Create a NumPy array
my_array = np.array([[1, 2, 3], [4, 5, 6]])

# Save the array to a .npy file
np.save('my_array.npy', my_array)

print("Array saved to my_array.npy")
```

```
Array saved to my_array.npy
```

2) **`np.savez(file, *args, **kwds)`**: Saves one or more NumPy arrays to a single uncompressed `.npz` archive.

```
import numpy as np

array1 = np.array([10, 20, 30])
array2 = np.array([[0.1, 0.2], [0.3, 0.4]])

# Save multiple arrays to a .npz file
np.savez('multiple_arrays.npz', arr1=array1, arr2=array2)

print("Multiple arrays saved to multiple_arrays.npz")
```

```
Multiple arrays saved to multiple_arrays.npz
```

3) **`np.savez_compressed(file, *args, **kwds)`: Saves one or more NumPy arrays to a single compressed `.npz` archive.**

```
import numpy as np

large_array = np.random.rand(1000, 1000)

# Save a compressed archive
np.savez_compressed('compressed_array.npz', data=large_array)

print("Large array saved to compressed_array.npz (compressed)")
```

```
Large array saved to compressed_array.npz (compressed)
```

4) **`np.savetxt(fname, X, fmt='%.18e', delimiter=' ', ...)`**: Saves a NumPy array to a plain text file.

```
import numpy as np

text_array = np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])

# Save to a text file
np.savetxt('text_array.txt', text_array, delimiter=',', fmt='%.2f', header='Column1,Column2,Column3')

print("Array saved to text_array.txt")
```

```
Array saved to text_array.txt
```

**Loading NumPy Arrays:**

1) **`np.load(file, mmap_mode=None, allow_pickle=True, ...)`**: Loads arrays from `.npy` or `.npz` files. Returns a single array for `.npy` or a dictionary-like object for `.npz`.

```
import numpy as np

# Load the single array from .npy file
loaded_array = np.load('my_array.npy')
print("\nLoaded array from .npy:\n", loaded_array)

# Load multiple arrays from .npz file
loaded_multiple = np.load('multiple_arrays.npz')
print("\nLoaded arrays from .npz:")
print("Array 1:", loaded_multiple['arr1'])
print("Array 2:", loaded_multiple['arr2'])
loaded_multiple.close() # It's good practice to close .npz files
```

```
Loaded array from .npy:
 [[1 2 3]
 [4 5 6]]

Loaded arrays from .npz:
Array 1: [10 20 30]
Array 2: [[0.1 0.2]
 [0.3 0.4]]
```

2) **`np.loadtxt(fname, dtype=float, delimiter=None, skiprows=0, ...)`**: Loads data from a plain text file into a NumPy array.

```
import numpy as np

# Load from the text file
loaded_text_array = np.loadtxt('text_array.txt', delimiter=',', skiprows=1) # Skip the header row
print("\nLoaded array from text file:\n", loaded_text_array)
```

```
Loaded array from text file:
 [[1.1 2.2 3.3]
 [4.4 5.5 6.6]]
```

- For **saving and loading single NumPy arrays with full precision and type information**, use `np.save()` and `np.load()`.
- For **saving and loading multiple NumPy arrays in a single file**, use `np.savez()` and `np.load()`. Consider `np.savez_compressed()` for large datasets to save disk space.
- For **saving data in a human-readable format or for interoperability with other tools**, use `np.savetxt()` and `np.loadtxt()`. Be mindful of potential loss of type and shape information.

## 7) Linear Algebra –

### 7.1) commonly used linalg functions

**Transpose:**
You can transpose a matrix using the .T attribute or the np.transpose() function.

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
print("Original Matrix A:\n", A)
print("Transpose of A (A.T):\n", A.T)
print("Transpose of A (np.transpose(A)):\n", np.transpose(A))
```

```
Original Matrix A:
 [[1 2]
 [3 4]]
Transpose of A (A.T):
 [[1 3]
 [2 4]]
Transpose of A (np.transpose(A)):
 [[1 3]
 [2 4]]
```

**Matrix Multiplication:**
You can perform matrix multiplication using the @ operator (Python 3.5+) or the np.matmul() function. For element-wise multiplication, use the * operator.

```
B = np.array([[5, 6], [7, 8]])
print("Matrix B:\n", B)
print("Matrix Multiplication (A @ B):\n", A @ B)
print("Matrix Multiplication (np.matmul(A, B)):\n", np.matmul(A, B))
print("Element-wise Multiplication (A * B):\n", A * B)
```

```
Matrix B:
 [[5 6]
 [7 8]]
Matrix Multiplication (A @ B):
 [[19 22]
 [43 50]]
Matrix Multiplication (np.matmul(A, B)):
 [[19 22]
 [43 50]]
Element-wise Multiplication (A * B):
 [[ 5 12]
 [21 32]]
```

**Determinant:** Calculate the determinant of a square matrix using np.linalg.det().

```
det_A = np.linalg.det(A)
print("Determinant of A:", det_A)
```

```
Determinant of A: -2.0000000000000
```

**Rank:** Determine the rank of a matrix using np.linalg.matrix_rank(). The rank represents the number of linearly independent rows or columns.

```
C = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
rank_C = np.linalg.matrix_rank(C)
print("Rank of C:", rank_C)
```

```
Rank of C: 2
```

**Trace:** Calculate the sum of the diagonal elements of a square matrix using np.trace().

```
trace_A = np.trace(A)
print("Trace of A:", trace_A)
```

```
Trace of A: 5
```

**Matrix Inversion:**
Calculate the inverse of a square, non-singular matrix using np.linalg.inv().

```
inv_A = np.linalg.inv(A)
print("Inverse of A:\n", inv_A)
```

```
Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

| | | |
|---|---|---|
| **Solving Linear Systems:**<br>To solve a system of linear equations represented as Ax=b using np.linalg.solve(). Here, A is the coefficient matrix, b is the constant vector, and x is the vector of unknowns. | ```a = np.array([[2, 1], [1, 3]])<br>b = np.array([4, 5])<br>x = np.linalg.solve(a, b)<br>print("Solution for x in 2x + y = 4 and x + 3y = 5:", x)``` | ```Solution for x in 2x + y = 4 and x``` |
| **Eigenvalues and Eigenvectors:**<br>Find the eigenvalues and eigenvectors of a square matrix using np.linalg.eig(). The function returns a tuple: the first element is an array of eigenvalues, and the second element is a matrix where each column is the eigenvector corresponding to the eigenvalue at the same index. | ```eigenvalues, eigenvectors = np.linalg.eig(A)<br>print("Eigenvalues of A:", eigenvalues)<br>print("Eigenvectors of A:\n", eigenvectors)``` | ```Eigenvalues of A: [-0.37228132  5.3<br>Eigenvectors of A:<br>[[-0.82456484 -0.41597356]<br>[ 0.56576746 -0.90937671]]``` |

# Question Bank:

| UNIT – IV: PANDAS AND NUMPY | | |
|---|---|---|
| **PART –A** | | |
| 1. | What is a Pandas Series? | L1 |
| 2. | Explain the purpose of reindexing in Pandas. | L2 |
| 3. | How do you handle missing data in Pandas? | L1,L2 |
| 4. | What is a NumPy array? | L1 |
| 5. | Name two functions from the NumPy linalg module. | L1 |
| 6. | What is the purpose of the groupby() function in Pandas? | L2 |
| 7. | List out the universal functions of Numpy | L1 |
| 8. | Write about query() function | L2 |
| **PART –B** | | |
| 1. | Discuss pandas and its essential features like indexing, selection and filtering | L3 |
| 2. | Evaluate summarizing and computing descriptive statistics, handling missing data | L5 |
| 3. | Explain about reading and writing data in text format | L2 |
| 4. | Discuss about Numpy in detail | L2 |
| 5. | Explain how to create arrays and universal functions like unary and binary functions | L3 |
| 6. | Write about File Input and output with arrays | L2 |
| 7. | Illustrate linalg functions | L3 |
| 8. | Explain Numpy in detail | L3 |

# PYTHON PROGRAMMING
# UNIT – V
# STREAMLIT – TO DEVELOP A GUI

# UNIT V: STREAMLIT – TO DEVELOP A GUI

## 1. Unit Overview:

This unit introduces Streamlit, a Python library for building interactive web applications. Students learn to create GUIs to display data, use interactive widgets like buttons, sliders, checkboxes, and dropdowns, and develop forms and progress indicators. The focus is on making Python applications user-friendly and visually interactive.

## 2. Objectives of the Unit:

By the end of this unit, students should be able to:

- Understand the features of Streamlit and its role in GUI development.
- Display textual data, tables, and DataFrames effectively.
- Implement interactive widgets like buttons, checkboxes, sliders, and dropdowns.
- Create forms for user input collection.
- Display progress bars and other interactive elements.
- Develop complete interactive applications combining all components.

## 3. Learning Outcomes:

After completing this unit, students should will be able to:

- After completing this unit, students will be able to:
- Build web applications using Streamlit.
- Display data using text elements, tables, and DataFrames.
- Add interactivity using buttons, radio buttons, checkboxes, sliders, and dropdowns.
- Create forms to collect user inputs efficiently.
- Build functional applications combining multiple widgets and elements.

## 4. Importance of Studying this Unit:

- Allows rapid development of interactive Python applications.
- Helps visualize data and enhance user experience.
- Useful for data science, ML prototyping, and dashboards.
- Prepares students for real-world interactive applications and reporting.

## 5. Key Concepts:

- **Streamlit Basics** – Library for web apps.
- **Text & Table Elements** – Titles, headers, markdown, tables, dataframes.
- **Interactive Widgets** – Buttons, radio, checkboxes, dropdowns, multiselect.
- **Sliders & Progress Bars** – Get user input and show progress.
- **Forms** – Collect multiple inputs.
- **Develop Applications** – Combine all elements to build functional GUI app

# Unit - V

1) What is Streamlit,
2) Features of Streamlit,
3) Text and Table elements –
    a. Text Elements, Titles, Headers, Subheaders ,markdowns,
    b. Tables,
    c. Dataframes,
4) Buttons and sliders –
    a. Buttons,
    b. RadioButton,
    c. Checkbox,
    d. Dropdown ,
    e. Multiselect,
    f. Progress bar,
    g. Slidder,
5) Forms,
6) Develop an Streamlit Application

## 1) What is Streamlit,

**What is Streamlit?**

- ✓ **Streamlit** is an open-source Python library designed to enable data scientists and machine learning engineers to rapidly create and deploy interactive web applications for data visualization, machine learning, and analytics with minimal coding effort.
- ✓ Streamlit is an open-source Python library that allows data scientists and machine learning engineers to quickly build and share interactive web applications for their work.
- ✓ It focuses on simplicity and leverages the power of Python, eliminating the need for extensive front-end development knowledge.

**Key Features of Streamlit:**

- ✓ **Simplicity:** Intuitive API; write Python, get a web app.
- ✓ **Python-Native:** Leverages familiar Python syntax and data structures.
- ✓ **Real-time Updates:** Automatic app refresh on script changes.
- ✓ **Interactive Widgets:** Buttons, sliders, dropdowns, etc., for user input.
- ✓ **Automatic UI Rendering:** Smart rendering of DataFrames, plots, and other data.
- ✓ **Caching (@st.cache_data, @st.cache_resource):** Optimizes performance by memoizing function results.
- ✓ **Layout and Organization:** Columns, expanders, tabs for structuring apps.
- ✓ **Theming:** Customizable appearance.
- ✓ **Community & Ecosystem:** Active community and growing number of components.
- ✓ **Deployment:** Easy deployment on various platforms.

**Text and Table Elements**

These elements are used to display textual information and tabular data.

## a. Text Elements, Titles, Headers, Subheaders ,markdowns,

- ✓ **st.title(body):** Displays the main title of your application.
- ✓ **st.header(body):** Displays a section header, typically smaller than the title.
- ✓ **st.subheader(body):** Displays a subsection header, smaller than the header.
- ✓ **st.text(body):** Displays fixed-width, preformatted text. Useful for displaying code or raw output.
- ✓ **st.markdown(body):** Displays text formatted using Markdown syntax. This allows for rich text formatting (bold, italics, lists, links, etc.).
- ✓ **Example**

```
1  import streamlit as st
2  st.title("My Streamlit App")
3  st.header("Data Exploration")
4  st.subheader("Raw Data")
5  st.text("This is some fixed-width text.")
6  st.markdown("**This text is bold.** *This text is italic.* [Link to Streamlit](https://streamlit.io)")
```

- ✓

✓ Output

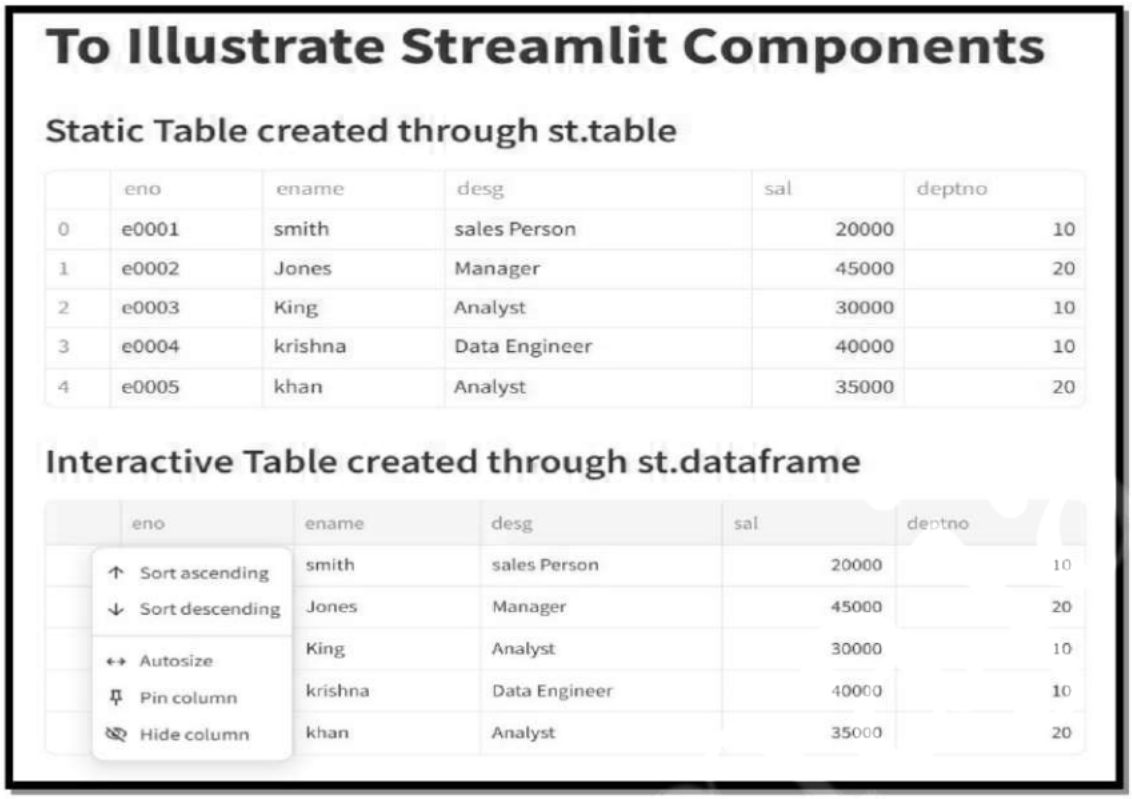

b. Tables,

- ✓ **st.table(data=None):** Displays a static table.
- ✓ data (*Pandas.DataFrame, dict, list*): The data to display in the table. Streamlit infers column names and data types.
- ✓ It's useful for displaying small, fixed datasets or summaries.
- ✓ Streamlit handles the rendering of the table.
- ✓ can pass a Pandas DataFrame or a dictionary-like object.

c. **st.dataframe(data=None, height=None, width=None):** Displays an interactive table (Pandas DataFrame). Offers sorting, searching, and resizing.

- ✓ data (*Pandas.DataFrame*): The DataFrame to display.
- ✓ height (*int* or *None*, default None): Fixed height of the dataframe in pixels.
- ✓ width (*int* or *None*, default None): Fixed width of the dataframe in pixels.
- ✓ Example

```
1  import streamlit as st
2  import pandas as pd
3  st.title("To Illustrate Streamlit Components")
4  df = pd.read_csv('emp.csv')
5  st.subheader("Static Table created through st.table")
6  st.table(df)
7  st.subheader("Interactive Table created through st.dataframe")
8  st.dataframe(df)
```




## 4) Buttons and sliders

### a. Buttons – Creates a Clickable Button

**st.button(label, key=None, on_click=None, args=None, kwargs=None, disabled=False).**

- ✓ label (*str*): The text displayed on the button.
- ✓ key (*str* or *None*, default None): A unique key for the widget, used for state management.
- ✓ on_click (*callable* or *None*, default None): A function to call when the button is clicked.
- ✓ args (*tuple* or *None*, default None): Positional arguments to pass to the on_click function.
- ✓ kwargs (*dict* or *None*, default None): Keyword arguments to pass to the on_click function.
- ✓ disabled (*bool*, default False): Whether the button is disabled.

```
if st.button("Click Me"):
    st.write("Button was clicked!")
```

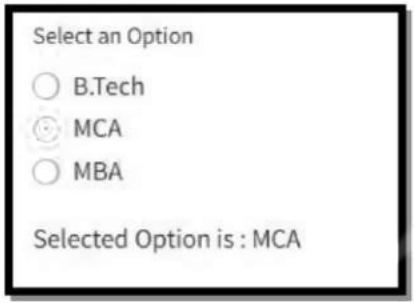

;;

## b. Radio Button - Creates a set of Mutually Exclusive Options

**st.radio(label, options, index=0, key=None, horizontal=False, disabled=False, on_change=None, args=None, kwargs=None):**

- ✓ label (*str*): The label displayed above the radio buttons.
- ✓ options (*sequence*): A sequence of options to display.
- ✓ index (*int*, default 0): The index of the option to select by default.
- ✓ key, disabled, on_change, args, kwargs: Same as st.button.
- ✓ horizontal (*bool*, default False): Whether to display options horizontally.

```
branch = st.radio("Select an Option",["B.Tech","MCA","MBA"])
st.write(f"Selected Option is : {branch}")
```




## c. Check Box - Creates a Single CheckBox

**st.checkbox(label, value=False, key=None, disabled=False, on_change=None, args=None, kwargs=None):** .

- ✓ label (*str*): The label displayed next to the checkbox.
- ✓ value (*bool*, default False): The initial state of the checkbox (checked or unchecked).
- ✓ key, disabled, on_change, args, kwargs: Same as st.button.

```
checked = st.checkbox("I agree")
if checked:
    st.write("You agreed!")
```


## d. Drop Down - Creates a dropdown list for single selection

**st.selectbox(label, options, index=0, key=None, disabled=False, on_change=None, args=None, kwargs=None):**

- ✓ label (*str*): The label displayed above the select box.
- ✓ options (*sequence*): A sequence of options to display in the dropdown.

- ✓ index (*int*, default 0): The index of the option to select by default.
- ✓ key, disabled, on_change, args, kwargs: Same as st.button.

```
selected_item = st.selectbox("Choose B.Tech Branch:", ["CSE", "EEE", "ECE","AIML","DS"])
st.write(f"You chose: {selected_item}")
```


## e. multiselect - Creates a dropdown list for multiple selections

**st.multiselect(label, options, default=None, key=None, disabled=False, on_change=None, args=None, kwargs=None):**

- ✓ label (*str*): The label displayed above the multiselect box.
- ✓ options (*sequence*): A sequence of options to display.
- ✓ default (*sequence* or *None*, default None): A sequence of options to select by default.
- ✓ key, disabled, on_change, args, kwargs: Same as st.button.

```
selected_multiple = st.multiselect("Choose Your Multiple Hobbies:", ["Reading", "Painting", "Dancing", "Sports"], default=["Reading"])
st.write(f"You selected: {selected_multiple}")
```


## f. Progress Bar - Displays a Progress Bar

**st.progress(value, text='').**

- ✓ value (*float* or *int*): The progress value, between 0 and 1 (or an integer representing percentage).
- ✓ text (*str*): Optional text to display alongside the progress bar.

```
import time
my_bar = st.progress(0)
for i in range(101):
    time.sleep(0.01)
    my_bar.progress(i / 100)
st.write("Progress complete!")
```


## g. Slider: Creates a slider for selecting a numerical value

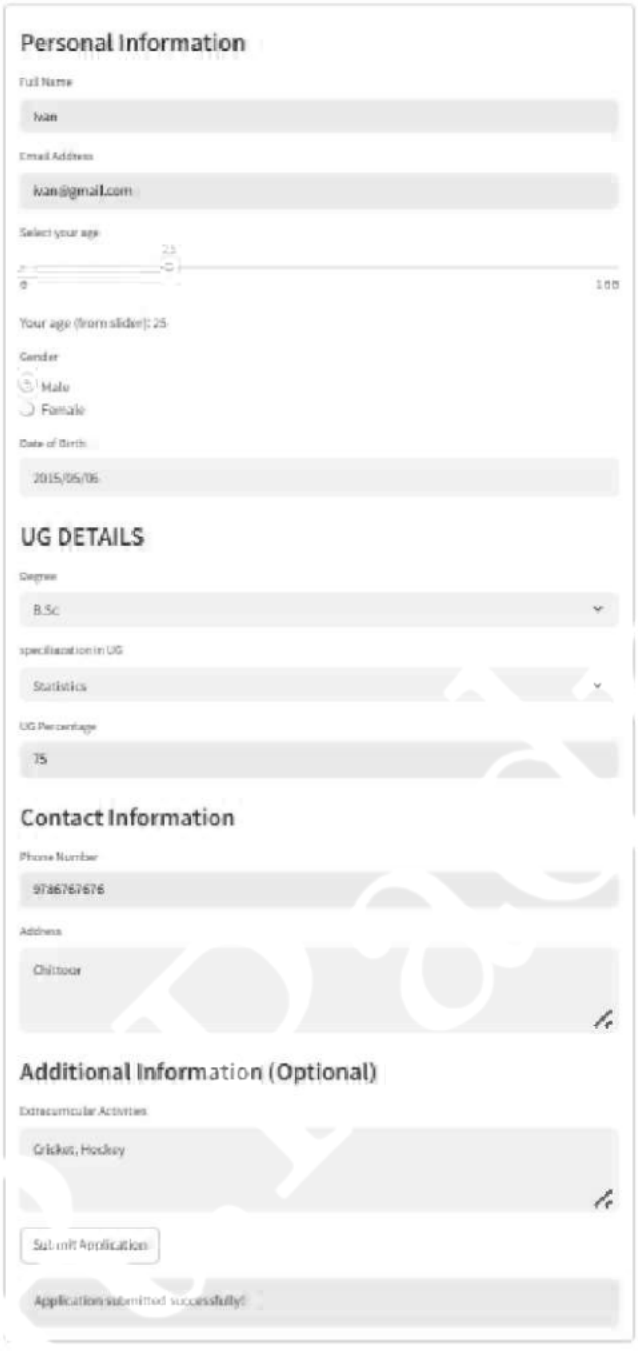
- **st.slider(label, min_value=None, max_value=None, value=None, step=None, format=None, key=None, disabled=False, on_change=None, args=None, kwargs=None):**
  - ✓ label (*str*): The label displayed above the slider.
  - ✓ min_value (*int*, *float*, *datetime.date*, *datetime.datetime* or *None*, default None): The minimum value of the slider.
  - ✓ max_value (*int*, *float*, *datetime.date*, *datetime.datetime* or *None*, default None): The maximum value of the slider.
  - ✓ value (*int*, *float*, *datetime.date*, *datetime.datetime*, *tuple* of two such values, or *None*, default None): The initial value(s) of the slider. Can be a single value or a tuple for a range slider.
  - ✓ step (*int*, *float*, *datetime.timedelta* or *None*, default None): The step size of the slider.
  - ✓ format (*str* or *None*, default None): A format string to display the slider value (e.g., "%.2f").
  - ✓ key, disabled, on_change, args, kwargs: Same as st.button.

```
age = st.slider("Select your age:", 0, 100, 25)
st.write(f"Your age is: {age}")

price_range = st.slider("Select a price range:", 0.0, 100.0, (25.0, 75.0))
st.write(f"Selected price range: {price_range}")
```


## 5) Forms

- **with st.form(key='my_form'): ...:** Creates a form that groups widgets.
  - key (*str*): A unique key for the form.
- **st.form_submit_button(label='Submit', on_click=None, args=None, kwargs=None, help=None, disabled=False):** Creates a submit button within a form.
  - ✓ label (*str*): The text displayed on the submit button.
  - ✓ on_click, args, kwargs, disabled: Same as st.button.
  - ✓ help (*str* or *None*, default None): A tooltip displayed when hovering over the button.

```
import streamlit as st
import pandas as pd
import numpy as np

with st.form(key='student_form'):
    st.subheader("Personal Information")
    name = st.text_input("Full Name", placeholder="Enter your full name")
    email = st.text_input("Email Address", placeholder="Enter your email")
    age_slider = st.slider("Select your age", 0, 100, 25)
    st.write(f"Your age (from slider): {age_slider}")
    gender = st.radio("Gender", ["Male", "Female"])
    date_of_birth = st.date_input("Date of Birth")
    st.subheader("UG DETAILS")
    degree = st.selectbox("Degree", ["B.Sc", "BCA", "BCOM", "Other"])
    specialization = st.selectbox("speciliazation in UG", ["Statistics", "Electronics", "Computer Science"])
    per = st.text_input("UG Percentage", placeholder="Enter your UG Percentage")
    st.subheader("Contact Information")
    phone_number = st.text_input("Phone Number", placeholder="Enter your phone number")
    address = st.text_area("Address", placeholder="Enter your current address")
    st.subheader("Additional Information (Optional)")
    extracurricular_activities = st.text_area("Extracurricular Activities", placeholder="List any extracurricular activities")
    submit_button = st.form_submit_button("Submit Application")
    if submit_button:
        st.success("Application submitted successfully!")
        st.balloons()
```


Personal Information
Full Name
Ivan
Email Address
ivan@gmail.com
Select your age
25
0
100
Your age (from slider): 25
Gender
Male
Female
Date of Birth
2015/05/06
UG DETAILS
Degree
B.Sc
specialization in UG
Statistics
UG Percentage
75
Contact Information
Phone Number
9786767676
Address
Chittoor
Additional Information (Optional)
Extracurricular Activities
Cricket, Hockey
Submit Application
Application submitted successfully!

# Question Bank:

| UNIT – V: STREAMLIT -TO DEVELOP AN GUI | | |
|---|---|---|
| **PART –A** | | |
| 1. | What is Streamlit? | L1 |
| 2. | Name three core features of Streamlit. | L1 |
| 3. | How do you display a Pandas DataFrame in Streamlit? | L1 |
| 4. | List few elements that we can create in streamlit | L1 |
| 5. | How do you create a button in Streamlit? | L1 |
| 6. | What is the purpose of importing the streamlit library as "st" | L2 |
| 7. | How do you run a Streamlit application from the command line? | L1 |
| 8. | What is the purpose of st.multiselect() | L2 |
| **PART –B** | | |
| 1. | Explain the core features of Streamlit and discuss how these features contribute to rapid development of data-driven web applications. | L2 |
| 2. | illustrate the different types of input widgets available in Streamlit (buttons, sliders, dropdowns, etc.) | L3 |
| 3. | Explain the process of developing a complete Streamlit application, from initial setup to deployment. | L2 |
| 4. | Describe the various text and table elements available in Streamlit | L2 |
| 5. | Discuss Streamlit Buttons and sliders with example | L2 |
| 6. | Explain streamlit multiselect, progress bar,slidder and forms | L2 |
| 7. | Illustrate basic streamlit components | L3 |
| 8. | Illustrate Header, subheader,markdowns, tables and dataframes | L3 |