

QMP 7.1 D/F



Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)
(NAAC Accredited & ISO 9001:2015 Certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.



Department of Civil Engineering

PYTHON PROGRAMMING LAB FOR CIVIL ENGINEERS (1BCV307C)

B.E - III Semester

Lab Manual

Name : _____

USN : _____

Batch : _____ Section : _____



Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)

(NAAC Accredited & ISO 9001:2015 Certified Institution)

NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.



Department of Civil Engineering

PYTHON PROGRAMMING LAB FOR CIVIL ENGINEERS

Prepared by:

Mr. Manjunatha M Katti

Assistant Professor

Department of Civil Engineering.

Approved by:

Ramya.H.N

Assistant Professor & Head

Department of Civil Engineering.

INDEX PAGE

SL NO.	Name of the Experiment	Date			Manual marks (20)	Record Marks (Max. 10)	Signature Student	Signature Faculty
		Conduction	Repetition	Submission of Record				
Average								

Note: If the student fails to attend the regular lab, the experiment has to be completed in the same week. Then the manual/observation and record will be evaluated for 50% of maximum mark



Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)

(NAAC Accredited & ISO 9001:2015 Certified Institution)

NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.



DEPARTMENT OF CIVIL ENGINEERING.

COURSE OBJECTIVE

This course will enable students to;

1. Automate Data Analysis and Structural Design Calculations
2. Enable Engineering Data Visualization and Modeling
3. Introduce Computational Problem Solving and Algorithm Development

COURSE OUTCOME

- | | |
|----------|--|
| CO 502.1 | Apply Python programming basics to solve civil engineering-related numerical and logical problems. |
| CO 502.2 | Implement conditional statements, loops, functions, and arrays for engineering computations. |
| CO 502.3 | Analyze civil engineering data using Python libraries and interpret computational results. |
| CO 502.4 | Develop Python scripts for structural, transportation, environmental, and geotechnical engineering applications. |
| CO 502.5 | Interpret outputs from Python-based solutions to support engineering decision making. |

'Instructions to the Candidates'

1. Students should come with thorough preparation for the experiment to be conducted.
2. Students without uniform will not be permitted to attend the laboratory classes
3. Students will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiment conducted in the previous class.
4. All the calculations should be made in the observation book
5. Wherever graphs are to be drawn, A-4 size graphs only should be used and the same should be firmly attached to the practical record.
6. Practical record should be neatly maintained.
7. They should obtain the signature of the staff-in-charge in the observation book after completing each experiment.
8. Theory regarding each experiment should be written in the practical record before procedure in your own words.



Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)

(NAAC Accredited & ISO 9001:2015 Certified Institution)

NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.



DEPARTMENT OF CIVIL ENGINEERING.

REFERENCES:

1. Severance, C. R., Python for Everybody, Exploring Data Using Python 3, Open book version / equivalent publisher edition.
2. Lutz, M., Learning Python, O'Reilly, latest edition.
3. Gaddis, T., Starting Out with Python, Pearson, latest edition.
4. Necaie, R., Data Structures and Algorithms Using Python, Wiley, latest edition.

CHANNABASAVESHWARA INSTITUTE OF TECHNOLOGY



Course Name:

PYTHON PROGRAMMING LAB FOR CIVIL ENGINEERS

Course Code:

1BCV307C

LAB MANUAL

Prepared by:

Mr. MANJUNATHA M KATTI

Assistant Professor

Department of Civil Engineering



DEPARTMENT OF CIVIL ENGINEERING

Why civil engineers learn programming language

Civil engineers learn programming languages for a variety of reasons, as programming skills can significantly enhance their capabilities in various aspects of their work. Here are some key reasons why civil engineers.

Benefit from learning programming languages:

1. **Automation of Repetitive Tasks** Civil engineering involves a lot of repetitive tasks, such as calculations, data analysis, and report generation. Programming can automate these tasks, saving time and reducing the likelihood of human error.

2. **Data Analysis and Visualization** Civil engineers often deal with large datasets from surveys, sensors, and simulations. Programming languages like Python and R provide powerful tools for data analysis and visualization, helping engineers to make informed decisions based on data insights.

3. **Simulation and Modeling** Programming is essential for developing and using simulation and modeling tools. For example, finite element analysis (FEA) software often requires scripting for custom simulations. MATLAB and Python are commonly used for creating models and running simulations.

4. **Custom Software Development** Sometimes, off-the-shelf software doesn't meet all the specific needs of a project. Knowing how to program allows civil engineers to develop custom software solutions tailored to their specific requirements, improving project efficiency and outcomes.

5. **Interdisciplinary Collaboration** Modern civil engineering projects often involve interdisciplinary collaboration. Programming skills enable engineers to integrate with other fields such as mechanical engineering, electrical engineering, and computer science, facilitating smoother and more effective collaboration.

6. **Enhanced Problem-Solving Skills** Learning to program enhances problem-solving skills. It encourages logical thinking and a systematic approach to problem-solving, which are crucial in civil engineering for tasks like design, analysis, and troubleshooting.

7. **Increasing Demand for Digital Competence** The construction and infrastructure sectors are increasingly adopting digital technologies like Building Information Modeling (BIM), Geographic Information Systems (GIS), and Internet of Things (IoT). Programming knowledge is often essential to effectively use and integrate these technologies.

8. **Career Advancement** Programming skills can make civil engineers more competitive in the job market. They open up opportunities in areas like smart infrastructure, urban planning, and research and development, where digital and computational skills are highly valued.

9. **Efficient Project Management** Programming can aid in project management by automating scheduling, resource allocation, and progress tracking. Tools like Python and VBA can be used to develop custom project management solutions.

10. **Machine Learning and AI Applications** There is a growing interest in applying machine learning and artificial intelligence in civil engineering, such as predictive maintenance of infrastructure, optimizing construction processes, and smart city planning. Programming is fundamental to leveraging these advanced technologies.

Programming skills can greatly enhance the capabilities of civil engineers, making them more efficient, versatile, and competitive. Whether it's automating tasks, analyzing data, developing custom solutions, or integrating with advanced technologies, programming provides a valuable toolkit that can significantly benefit civil engineering professionals.

The Jupyter interface is a environment that allows users to containing live code, equations, It is widely used for data analysis, learning, and more. Here's a brief and features:

Key Components

Notebook: The core execute code. It supports multiple common.



web-based interactive computing create and share documents visualizations, and narrative text. scientific research, machine overview of its key components

component where you write and languages, but Python is the most

Cells: Notebooks are divided into cells. There are mainly two types of cells: Code Cells: Where you write and execute your code.

Markdown Cells: Where you can write formatted text using Markdown syntax.

Kernel: The computational engine that executes the code contained in the notebook. Different kernels can be used for different programming languages.

Features

- **Interactive Code Execution:** Run code interactively and see the results immediately.
- **Rich Text Formatting:** Use Markdown to add headings, lists, links, images, and LaTeX equations.
- **Data Visualization:** Integrate with libraries like Matplotlib, Seaborn, and Plotly to create visualizations.
- **Export Options:** Export notebooks to various formats like HTML, PDF, and slides.
- **Extensions and Widgets:** Enhance functionality with extensions and interactive widgets.

Basic Operations

Running Cells: Press Shift + Enter to run a cell and move to the next one, or Ctrl + Enter to run the cell and stay on it.

Adding Cells: Use the toolbar or the Insert menu to add new cells.

Saving Notebooks: Use the save button or Ctrl + S to save your work.

Example Workflow

Start a Notebook: Launch Jupyter and create a new notebook.

Write Code: Enter your code in code cells.

Execute Code: Run the cells to execute the code and see the output.

Document Your Work: Use Markdown cells to explain your code, document findings, and add context.

Visualize Data: Create plots and graphs to visualize your data.

Save and Share: Save your notebook and share it with colleagues or export it to another format.

INTRODUCTION TO PYTHON PROGRAMMING

- Python is a high-level, general-purpose, interpreted, interactive and object-oriented programming language.
- It was created by Guido van Rossum during 1985- 1990.
- Like Perl, Python source code is also available under the GNU General Public License (GPL).

i. Why the name python?



Guido van Rossum was interested on watching a comedy show, which is telecasting on the BBC channel from 1969 to 1974 the complete Monty Python's Flying Circus.

Guido Van Rossum thought he needed a name that was short, unique and slightly mysterious for his programming language, so he decided to call the language Python.

ii. Applications of Python Programming

Scientific and Numeric applications

Artificial Intelligence and Machine Learning based applications Data Science related applications

Software Development

Enterprise-level/Business Applications Education programs and training courses

Image Processing

Data analysis

Introduction to Python in Civil Engineering

Python can be an incredibly useful tool in civil engineering for various tasks such as data analysis, simulation, modeling, and automation. Here's a brief introduction to how Python can be applied in civil engineering.

Numerical Analysis and Simulation: Python's libraries such as NumPy, SciPy, and SymPy provide powerful tools for numerical analysis, solving differential equations, and symbolic mathematics. These are crucial for modeling various civil engineering phenomena like structural analysis, fluid dynamics, and geotechnical behavior.

Data Analysis and Visualization: With libraries like Pandas, matplotlib, and Seaborn, Python facilitates the analysis and visualization of data obtained from sensors, surveys, simulations, and experiments. Engineers can gain insights into complex datasets and communicate results effectively.

Finite Element Analysis (FEA): Packages like FEniCS and OpenSeesPython enable civil engineers to perform FEA for structural analysis, heat transfer, and fluid flow problems. These tools aid in understanding the behavior of structures and optimizing their design.

Geographic Information Systems (GIS): Python libraries like GeoPandas, Shapely, and Fiona support GIS functionalities, allowing engineers to work with spatial data, perform geospatial analysis, and create maps. This is useful for urban planning, transportation engineering, and environmental studies.

Machine Learning and AI: Python's ecosystem includes libraries like TensorFlow, scikit-learn, and Keras, which support machine learning and artificial intelligence applications. Civil engineers can leverage these tools for predictive modeling, anomaly detection, and optimization in areas like traffic engineering, structural health monitoring, and risk assessment.

Overall, Python offers a wide range of tools and resources that empower civil engineers to tackle complex problems, streamline workflows, and innovate in the field. Its flexibility and scalability make it a valuable asset in modern civil engineering practice

Python Data Types:

In Python, data types represent the type of data stored in variables. Here are some of the fundamental data types in Python:

Numeric Types:

int: Integer type, representing whole numbers (e.g., 5, -10, 1000).

float: Floating-point type, representing decimal numbers (e.g., 3.14, -0.001, 2.0).

Sequence Types:

string: String type, representing sequences of characters enclosed in single (") or double (") quotes (e.g., 'hello', "world").

List: Ordered collection of items, mutable and enclosed in square brackets (e.g., [1, 2, 3], ['a', 'b', 'c']).

Tuple: Similar to lists but immutable, enclosed in parentheses (e.g., (1, 2, 3), ('a', 'b', 'c')).

Mapping Type:

dictionary: Collection of key-value pairs, enclosed in curly braces, where each key is associated with a value (e.g., {'name': 'John', 'age': 30}).

Set Types:

set: Unordered collection of unique elements, enclosed in curly braces (e.g., {1, 2, 3}, {'a', 'b', 'c'}).

Boolean Type:

bool: Represents Boolean values, either True or False, used for logical operations and conditions.

These are the basic data types in Python, and each type has its own set of operations and methods that can be performed on it. Additionally, Python is dynamically typed, meaning you don't need to declare the data type explicitly; it is inferred from the value assigned to the variable.

Executing Python code can be done in several ways:

- **Integrated Development Environment (IDE):**

IDEs such as PyCharm, Visual Studio Code, and Jupyter Notebook provide a more comprehensive environment for writing and executing Python code. These tools offer features like syntax highlighting, code completion, debugging, and integrated terminals. You can create and run Python scripts within the IDE, making development and debugging more convenient.

- **Executing Python Code in Other Applications:**

Python code can also be executed within other applications that support Python scripting or integration, such as Blender for 3D modeling, QGIS for geographic information systems, and Autodesk Revit for building information modeling. These applications provide Python APIs or scripting environments where you can write and execute Python code to extend their functionality or automate tasks.

Naming Convention Rules for the variables

- It should not start with a number
- It cannot contain spaces
- It should also not contain any special chars except "_"
- It should not be named after existing Keywords

```
In [2]: 123abc = 10
File "C:\Users\dukef\AppData\Local\Temp\ipykernel_12436\3408308467.py", line 1
      123abc = 10
          ^
SyntaxError: invalid syntax

In [3]: abc123 = 10
abc123
Out[3]: 10

In [4]: ab cd = 10
File "C:\Users\dukef\AppData\Local\Temp\ipykernel_12436\642427778.py", line 1
      ab cd = 10
          ^
SyntaxError: invalid syntax
```

In Python, the int data type represents integers, which are whole numbers without any fractional component. int stands for "integer".

Integer

```
In [13]: a = 10
         b = 20
         print(a)
         print(b)

10
20

In [14]: type(a)
Out[14]: int

In [15]: type(b)
Out[15]: int

In [16]: a
         b # Last Line of code will be considered for generating a PRINT functionality
Out[16]: 20
```

In Python, the float data type represents floating-point numbers, which are numbers that have both an integer and a fractional part, separated by a decimal point. float stands for "floating point".

Float

```
In [19]: x = 10.25
         y = 25.69
         print(x, y)

10.25 25.69

In [37]: type(x)
         type(y)
Out[37]: float

In [21]: type(y)
Out[21]: float

In [22]: x
Out[22]: 10.25
```

Operations on integer and float:

Operations between integers and floats in Python follow the rules of mixed-type arithmetic, where Python automatically converts integers to floats if necessary to perform the operation. Here are some examples of operations between integers and floats:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Floor Division (//)
- Modulus (%)
- Power(**)

Operations on int and float

```
In [23]: print(a, b)
```

```
10 20
```

```
In [24]: print(x, y)
```

```
10.25 25.69
```

```
In [25]: # 10 ^ 2
```

```
In [41]: a + x  
a - x
```

```
Out[41]: -0.25
```

```
In [42]: print(a + x)  
print(a - x)
```

```
20.25  
-0.25
```

```
In [26]: # Arithmetic Operations  
print("Addition", a + x)  
print("Subtraction", a - x)  
print("Multiplication", a * x)  
print("Division", a / y)  
print("Floor Division", a // 3)  
print("Floor Division", a // 6)  
print("Modulus", a % 2)  
print("Modulus", 11 % 2)  
print("Power", a ** 3) # 10 * 10 * 10
```

```
Addition 20.25  
Subtraction -0.25  
Multiplication 102.5  
Division 0.38925652004671074  
Floor Division 3  
Floor Division 1  
Modulus 0  
Modulus 1  
Power 1000
```

In Python, the string data type represents strings, which are sequences of characters. Strings are used to store textual data and are enclosed within either single quotes (') or double quotes (").

```
Strings

In [ ]: a = 10

In [43]: a = 'Python'
         b = "Python"
         print(a, b)
Python Python

In [44]: type(a)
Out[44]: str

In [45]: type(b)
Out[45]: str

In [46]: a
Out[46]: 'Python'

In [47]: b
Out[47]: 'Python'
```

Indexing - Slicing in string:

Indexing:

- Indexing allows you to access individual characters within a string by their position (index).
- In Python, indexing starts from 0 for the first character, with subsequent characters numbered sequentially.
- You can use positive indices to access characters from the beginning of the string and negative indices to access characters from the end of the string.
- To access a character at a specific index *i*, use the syntax `string[i]`.

Slicing:

- Slicing allows you to extract a substring (a portion of a string) by specifying a range of indices.
- The syntax for slicing is `string [start index: end index]`, where start index is inclusive and end index is exclusive.
- Omitting start index defaults to 0, and omitting end index defaults to the end of the string.
- You can also specify a step size using the syntax `string [start index: end index: step size]`.

```
In [49]: a
Out[49]: 'Python'

In [ ]: # Select SUBSTR(a, 3, 2)

In [50]: a[:] # indexing | Start : End
Out[50]: 'Python'

In [51]: a[0]
Out[51]: 'P'

In [52]: a[3]
Out[52]: 'h'

In [53]: a[2:] # Starting from the 2nd positioned val, I want everything
Out[53]: 'thon'

In [54]: a[:4] # Until the end position 4, give me everything
Out[54]: 'Pyth'

In [55]: a[1:5]
Out[55]: 'ytho'
```

```
In [59]: x = "Python Programming"
x
Out[59]: 'Python Programming'

In [71]: x[1:10:2]
Out[71]: 'yhnPo'

In [61]: x[::2]
Out[61]: 'Pto rgamn'

In [62]: x[::1]
Out[62]: 'Python Programming'

In [63]: a
Out[63]: 'Python'

In [64]: a[-1]
Out[64]: 'n'

In [66]: a[:-2] # Everything except the last 2 chars
Out[66]: 'Pyth'
```

Operations in Strings:

Operations on strings in Python involve a variety of tasks, including concatenation, repetition, comparison, searching, replacing, splitting, joining, and formatting.

Operations in Strings

```
In [73]: a = "Hello World, Welcome to Python Programming"
a
```

```
Out[73]: 'Hello World, Welcome to Python Programming'
```

```
In [74]: a.upper()
```

```
Out[74]: 'HELLO WORLD, WELCOME TO PYTHON PROGRAMMING'
```

```
In [75]: a.lower()
```

```
Out[75]: 'hello world, welcome to python programming'
```

```
In [76]: a.capitalize()
```

```
Out[76]: 'Hello world, welcome to python programming'
```

```
In [77]: a.title()
```

```
Out[77]: 'Hello World, Welcome To Python Programming'
```

```
In [108]: t = "Python is a PL and python is also an animal"
t
```

```
Out[108]: 'Python is a PL and python is also an animal'
```

```
In [111]: t.lower().count("python")
```

```
Out[111]: 2
```

```
In [112]: c = "Python Programming"
c.replace(" ", " ")
```

```
Out[112]: 'Python Programming'
```

```
In [113]: "Python " + "Programming"
```

```
Out[113]: 'Python Programming'
```

```
In [114]: a = "Python Programming"
a
```

```
Out[114]: 'Python Programming'
```

```
In [115]: a.index("P")
```

```
Out[115]: 0
```

```
In [116]: a.index("P", 1) # Start searching for "P" from the 1st position
```

```
Out[116]: 7
```

A Boolean data type is a fundamental data type in computer science and programming. It represents two possible values: true or false.

```
Boolean

In [1]: a = True
        b = False

In [2]: a
Out[2]: True

In [3]: b
Out[3]: False

In [4]: type(a)
Out[4]: bool
```

Data Structures in Python

In Python, data structures are essential for organizing and storing data efficiently. Python provides several built-in data structures, each with its own unique characteristics and use cases. Below are some of the most commonly used data structures in Python.

- List - []
 - Allows Duplicates
 - Heterogeneous
 - Ordered
 - Mutable

```
In [5]: a = [1, 2, 3, 1, 2, 3, 'Python', 12.23, True]
        a
Out[5]: [1, 2, 3, 1, 2, 3, 'Python', 12.23, True]

In [6]: a[0]
Out[6]: 1

In [7]: a.append("Java")
        a
Out[7]: [1, 2, 3, 1, 2, 3, 'Python', 12.23, True, 'Java']

In [8]: type(a)
```

Range Function

Range Function

```
In [18]: range(10) # End Value - Starts from 0 and excludes the last value
```

```
Out[18]: range(0, 10)
```

```
In [19]: list(range(10))
```

```
Out[19]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [20]: list(range(1, 10)) # Start Value: End Value( Excl.)
```

```
Out[20]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [22]: list(range(2, 11))
```

```
Out[22]: [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [21]: list(range(2, 11, 2)) # Start Value: End Value( Excl.), Intervals
```

```
Out[21]: [2, 4, 6, 8, 10]
```

- Tuple - ()

Allows Duplicates

Heterogeneous

Ordered

Immutable

```
In [96]: a
```

```
Out[96]: 12
```

```
In [1]: 1,2,3,1,2,3, "Python", 'Java', 1.23, True
```

```
Out[1]: (1, 2, 3, 1, 2, 3, 'Python', 'Java', 1.23, True)
```

```
In [3]: a = (1,2,3,1,2,3, "Python", 'Java', 1.23, True)
```

```
a
```

```
Out[3]: (1, 2, 3, 1, 2, 3, 'Python', 'Java', 1.23, True)
```

```
In [100]: a[0]
```

```
Out[100]: 1
```

```
In [101]: a[-1]
```

```
Out[101]: True
```

```
In [4]: type(a)
```

```
Out[4]: tuple
```

A tuple is an immutable, ordered collection of elements in Python. Tuples can store multiple items in a single variable and are similar to lists, but unlike lists, they cannot be modified after creation. This immutability makes tuples useful for storing data that should not change throughout the program.

- Dictionary - {Key : Value}

Overrides Duplicates
Heterogeneous
Unordered
Mutable

```
In [1]: d = {"Subject" : "Python", "No.of Students" : 45, "Hands-on" : True, 1 : 100.25 , "Subject" : "Java"}
d
```

```
Out[1]: {'Subject': 'Java', 'No.of Students': 45, 'Hands-on': True, 1: 100.25}
```

```
In [2]: type(d)
```

```
Out[2]: dict
```

```
In [3]: d[0]
```

```
-----
KeyError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8596\1089268471.py in <module>
----> 1 d[0]

KeyError: 0
```

A dictionary in Python is an unordered, mutable collection of key-value pairs. It is one of the most powerful and flexible data structures in Python, allowing for efficient data retrieval. Here's an in-depth look at the dictionary data structure, including its characteristics, methods, and common usage scenarios.

- Set - {}

Does not Allow Duplicates
Heterogeneous
Unordered
Values are immutable but as a DS it is Mutable

```
In [45]: s = {1,2,1,2,True, False, "Python", "String", 1.24}
s
```

```
Out[45]: {1, 1.24, 2, False, 'Python', 'String'}
```

```
In [46]: s[0]
```

```
-----
TypeError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8596\243613605.py in <module>
----> 1 s[0]

TypeError: 'set' object is not subscriptable
```

```
In [47]: type(s)
```

```
Out[47]: set
```

Sets in Python are a powerful and flexible data structure used to store unique elements. They are particularly useful for operations involving mathematical set theory, such as union, intersection, and difference. Here's an in-depth look at sets in Python, including their characteristics, methods, and common use cases.

Operators in Python

Operators in Python are special symbols or keywords that are used to perform operations on variables and values. Python supports a variety of operators, each serving a different purpose, from arithmetic calculations to logical comparisons. Here is a comprehensive overview of the different types of operators in Python.

1) Arithmetic Operators

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (remainder)	`a % b`
`**`	Exponentiation	`a ** b`
`//`	Floor division	`a // b`

Arithmetic Operators

```
In [69]: a, b = 10, 20  
         print(a, b)
```

```
10 20
```

```
In [74]: print("Addition", float(a + b))  
         print("Subtraction", a - b)  
         print("Multiplication", a * b)  
         print("Division", a / b)  
         print("Floor Division", a // b)  
         print("Modulus", a % 2)  
         print("Power", a ** 2)
```

```
Addition 30.0  
Subtraction -10  
Multiplication 200  
Division 0.5  
Floor Division 0  
Modulus 0  
Power 100
```

2) Comparison Operators

Comparison operators are used to compare two values and return a Boolean value (True or False).

Operator	Description	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater than or equal to	`a >= b`
`<=`	Less than or equal to	`a <= b`

Comparison Operators

```
In [72]: print(a)
         print(b)
```

```
10
20
```

```
In [73]: print("Equals to", a == b)
         print("Not Equals to", a != b)
         print("Greater than", a > b)
         print("Greater than or Equals to", a >= b)
         print("Lesser than", a < b)
         print("Lessest than or Equals to", a <= b)
```

```
Equals to False
Not Equals to True
Greater than False
Greater than or Equals to False
Lesser than True
Lessest than or Equals to True
```

3) Logical Operators or Boolean Operators

Logical operators are used to combine conditional statements

Operator	Description	Example
`and`	Returns True if both statements are true	`a and b`
`or`	Returns True if at least one statement is true	`a or b`
`not`	Reverses the result, returns False if the result is true	`not a`

Boolean Operators

```
In [75]: a = True
        b = False
        print(a, b)
True False

In [76]: print(a and b) #True AND False
False

In [77]: print(a or b) #True OR False
True

In [78]: print(not a)
False

In [79]: print(not b)
True
```

4) Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Description	Example
`=`	Assign	`a = b`
`+=`	Add and assign	`a += b`
`-=`	Subtract and assign	`a -= b`
`*=`	Multiply and assign	`a *= b`
`/=`	Divide and assign	`a /= b`
`%=`	Modulus and assign	`a %= b`
`//=`	Floor divide and assign	`a //= b`

```
In [80]: a = 10
        b = 20
        c = a + b
        print(c)
30

In [81]: c = c + 10
        c
Out[81]: 40

In [82]: c += 25 # C = C + 25
        c
Out[82]: 65

In [83]: c -= 10
        c
Out[83]: 55

In [84]: c /= 5
        c
Out[84]: 11.0

In [85]: c **= 5
        c
Out[85]: 161051.0
```

5) Membership Operators

Membership operators are used to test if a sequence is presented in an object.

Operator	Description	Example
<code>`in`</code>	Returns True if a sequence with the specified value is present in the object	<code>`a in b`</code>
<code>`not in`</code>	Returns True if a sequence with the specified value is not present in the object	<code>`a not in b`</code>

```
In [87]: lista = list(range(10, 20))
        lista
Out[87]: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

In [88]: 15 in lista
Out[88]: True

In [89]: 15 not in lista
Out[89]: False

In [90]: a = list("ABCDEFGHIJKL")
        a
Out[90]: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']

In [91]: "K" in a
Out[91]: True

In [92]: "X" not in a
Out[92]: True
```

2. Input and output statements in Python programming

Input Statements

In Python, input and output (I/O) operations allow a program to interact with the user or with other programs and files. Here's a detailed overview of how to handle input and output in Python.

InputStatements the primary function for getting input from the user in Python is input (). This function reads a line from input (usually from the user) and returns it as a string.

Input	
In [1]:	<pre>name = input("Please enter your name: ") print(name)</pre> Please enter your name: KATTI KATTI
In [113]:	<pre>type(name)</pre>
Out[113]:	str
In [114]:	<pre>age = input("Please enter your Age: ") age</pre> Please enter your Age: 25
Out[114]:	'25'
In [115]:	<pre>type(age)</pre>
Out[115]:	str
In [116]:	<pre>age = int(input("Please enter your Age: ")) age</pre> Please enter your Age: 25
Out[116]:	25
In [117]:	<pre>type(age)</pre>
Out[117]:	int

Output Statements

The primary function for getting input from the user in Python is input (). This function reads a line from input (usually from the user) and returns it as a string.

Output	
In [140]:	<pre>print("Hello World")</pre> Hello World
In [141]:	<pre>print(15 - 89)</pre> -74
In [143]:	<pre>print(a)</pre> ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
In [144]:	<pre>print(a) print(b) print(c)</pre> ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L'] 20 161051.0
In [145]:	<pre>a b c</pre> Out[145]: 161051.0
In [146]:	<pre>print("Python","Programming",a)</pre> Python Programming ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
In [147]:	<pre>print("Python","Programming",a, sep = "----")</pre> Python----Programming----['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']

Formatting Output

There are several ways to format strings in Python for output.

Format

```
In [3]: name = input("Enter your Name: ")  
age = eval(input("Age Please: "))  
sal = eval(input("Salary Please: "))
```

```
Enter your Name: KATTI  
Age Please: 33  
Salary Please: 50000
```

```
In [4]: print(name, "is earning", sal, "dollars per year at the age of", age)
```

```
KATTI is earning 50000 dollars per year at the age of 33
```

```
In [5]: print("{} is earning {} dollars per year at the age of {}".format(name, sal, age))
```

```
KATTI is earning 50000 dollars per year at the age of 33
```

```
In [6]: print(f"{name} is earning {sal} dollars per year at the age of {age}")
```

```
KATTI is earning 50000 dollars per year at the age of 33
```

Conditionals

Conditionals are used to perform different actions based on different conditions. The most common conditional statements are if, elif, and else.

```
if condition:  
    # block of code executed if condition is true  
elif another_condition:  
    # block of code executed if another_condition is true  
else:  
    # block of code executed if no previous conditions are true
```

```
In [ ]: # if <logical_test>:  
#     perform any operations if it is true  
# else:  
#     perform any operations if it is False
```

```
In [9]: age = int(input("Please enter your age: "))
```

```
if age >= 18:  
    print("You are eligible to Vote")
```

Please enter your age: 14

```
In [1]: age = int(input("Please enter your age: "))
```

```
if age >= 18:  
    print("You are eligible to Vote")  
else:  
    print("You are not eligible to vote until you turn 18")
```

Please enter your age: 25

You are eligible to Vote

```
In [2]: age = int(input("Please enter your age: "))
```

```
if age >= 18:  
    print("You are eligible to Vote")  
else:  
    print("You are not eligible to vote until you turn 18")
```

Please enter your age: 17

You are not eligible to vote until you turn 18

```
In [7]: num = eval(input("Supply any Number: "))
```

```
if num > 0:  
    print("Positive Number")  
elif num < 0:  
    print("Negative Number")  
else:  
    print("It is a Zero")
```

Supply any Number: 0

It is a Zero

Loops:

i) For loop

The for loop in Python is used for iterating over a sequence (such as a list, tuple, dictionary, set, or string). Loops over all the values in a variable or run an operation Specific number of times.

```
In [18]: print("Python Programming")
```

```
Python Programming
```

```
In [19]: list(range(3))
```

```
Out[19]: [0, 1, 2]
```

```
In [20]: for i in range(3):  
         print(i, ". Python Programming", sep = "")
```

```
0. Python Programming  
1. Python Programming  
2. Python Programming
```

```
In [22]: for i in range(3):
```

```
    num = eval(input("Supply any Number: "))
```

```
    if num > 0:  
        print("Positive Number")  
    elif num < 0:  
        print("Negative Number")  
    else:  
        print("It is a Zero")
```

```
Supply any Number: 55
```

```
Positive Number
```

```
Supply any Number: -98
```

```
Negative Number
```

```
Supply any Number: 0
```

```
It is a Zero
```

Guessing Game

```
In [1]: import random  
secret_number = random.randint(10, 100)
```

```
In [ ]: # random.randint(10, 100)
```

```
In [41]: for i in range(5):  
         guess = int(input("Guess a number between 10 to 100: \n"))  
  
         if guess == secret_number:  
             print("Yay! You're a Genius")  
         elif guess > secret_number:  
             print("Guess Lesser")  
         else:  
             print("Guess Higher")
```

```
Guess a number between 10 to 100:
```

```
99
```

```
Guess Lesser
```

```
Guess a number between 10 to 100:
```

```
88
```

```
Guess Lesser
```

```
Guess a number between 10 to 100:
```

```
55
```

```
Guess Higher
```

```
Guess a number between 10 to 100:
```

```
66
```

```
Guess Higher
```

```
Guess a number between 10 to 100:
```

```
75
```

```
Guess Lesser
```

Break statement:

The break statement in Python is used to exit a loop prematurely. When the break statement is executed inside a loop, the loop is immediately terminated

Break Statement

```
In [48]: for i in range(5):
        guess = int(input("\nGuess a number between 10 to 100: "))

        if guess == secret_number:
            print("Yay! You're a Genius")
            break
        elif guess > secret_number:
            print("Guess Lesser")
        else:
            print("Guess Higher")

    else:
        print("\nGood Try, the secret number is", secret_number)
```

```
Guess a number between 10 to 100: 75
Guess Higher
```

```
Guess a number between 10 to 100: 76
Yay! You're a Genius
```

ii) While loop

The while loop in Python is used to repeatedly execute a block of code as long as a specified condition is true. The condition is checked before the block of code is executed, and if the condition is False, the loop will terminate.

While Loop

```
In [62]: n = 0

        while n < 5:
            print("Python")
            n += 1
```

```
Python
Python
Python
Python
Python
```

```
In [63]: n
Out[63]: 5
```

```
In [64]: exit = "No"

        while exit == "No":
            num = eval(input("Supply any Number: "))

            if num > 0:
                print("Positive Number")
            elif num < 0:
                print("Negative Number")
            else:
                print("It is a Zero")

            exit = input("Do you want to Quit? ")
```

```
Supply any Number: 55
Positive Number
Do you want to Quit? No
Supply any Number: -98
Negative Number
Do you want to Quit? No
Supply any Number: 0
It is a Zero
Do you want to Quit? Yes
```

```
In [65]: n = 0

while n != 5:

    num = eval(input("Supply any Number: "))

    if num > 0:
        print("Positive Number")
    elif num < 0:
        print("Negative Number")
    else:
        print("It is a Zero")

    n += 1
```

```
Supply any Number: 55
Positive Number
Supply any Number: 65
Positive Number
Supply any Number: 98
Positive Number
Supply any Number: -78
Negative Number
Supply any Number: 0
It is a Zero
```

```
In [67]: n = 0

while n < 5:

    guess = int(input("Guess a number between 10 to 100: \n"))

    if guess == secret_number:
        print("Yay! You're a Genius")
    elif guess > secret_number:
        print("Guess Lesser")
    else:
        print("Guess Higher")

    n += 1
```

```
Guess a number between 10 to 100:
55
Guess Lesser
Guess a number between 10 to 100:
25
Guess Higher
Guess a number between 10 to 100:
35
Yay! You're a Genius
Guess a number between 10 to 100:
56
Guess Lesser
Guess a number between 10 to 100:
78
Guess Lesser
```

Functions - UDF - User Defined Functions

User Defined Functions (UDFs) are custom functions created by users to perform specific tasks that are not provided by built-in functions in a programming language or environment. UDFs can be written in various programming languages and are often used to extend the functionality of software systems.

Key Concepts of UDFs:

Definition and Syntax:

A UDF is defined by a user using a specific syntax that typically includes a function name, parameters, and a body that contains the logic of the function. The syntax varies depending on the programming language.

Parameters:

UDFs can accept parameters (also called arguments) which allow users to pass values to the function. Parameters can be optional or required.

Return Value:

UDFs usually return a value after execution, which can be used in further computations or processes. Some UDFs might not return any value (i.e., they return void).

Scope:

UDFs can have different scopes: local (available only within a certain block of code) or global (available throughout the program).

Reusability:

UDFs are reusable pieces of code that can be called multiple times within a program, reducing redundancy.

Functions without any Arguments

```
In [6]: def welcome_message():  
        name = input("Enter your Name: ")  
        print(f"Welcome to CIT, {name}")  
        print("Here, Have a Good Tea!!")
```

```
In [7]: welcome_message()  
  
Enter your Name: KATTI  
Welcome to CIT, KATTI  
Here, Have a Good Tea!!
```

```
In [8]: welcome_message()  
  
Enter your Name: MANJUNATH  
Welcome to CIT, MANJUNATH  
Here, Have a Good Tea!!
```

Functions without arguments are useful when the task they perform does not require any external data or when the necessary data is available within the function itself. These functions are simple to define and use, making them a fundamental part of programming in Python.

Functions with Mandatory Arguments

```
In [9]: def tea_maker(name, tea):
```

```
    if tea == "Yes":
        print(f"Here, Have a tea {name}")
    else:
        print(f"Have water atleast, {name}")
```

```
In [10]: tea_maker()
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[10], line 1
----> 1 tea_maker()

TypeError: tea_maker() missing 2 required positional arguments: 'name' and 'tea'
```

```
In [12]: tea_maker("katti", "Yes")
```

```
Here, Have a tea katti
```

```
In [13]: tea_maker("katti", "No")
```

```
Have water atleast, katti
```

```
In [11]: def addition(x, y):
        return f"Addition of {x} and {y} is {x + y}"
```

```
In [12]: addition(10, 20)
```

```
Out[12]: 'Addition of 10 and 20 is 30'
```

```
In [14]: def info():
        return "Welcome to Civil Engineering"
        return "Something"
```

```
In [15]: def info_2():
        print("Welcome to CIT")
        print("Something")
```

```
In [16]: info()
```

```
Out[16]: 'Welcome to Civil Engineering'
```

```
In [17]: info_2()
```

```
Welcome to CIT
Something
```

Functions with mandatory arguments are crucial for scenarios where specific data must be provided to execute a function's task. Mandatory arguments ensure that the function receives the necessary input to operate correctly.

Functions with Mandatory and Optional Arguments

```
In [18]: def tea_maker(name, liquid = "Milk", sweetener = "Sugar"):

    if liquid == "Milk" and sweetener == "Sugar":
        return f"{name} will have a tea with Milk and Sugar"
    elif liquid == "Water" and sweetener == "Sugar":
        return f"{name} will have a black tea with Sugar"
    elif liquid == "Milk" and sweetener == "Honey":
        return f"{name} will have a tea with Milk and Honey"
    elif liquid == "Water" and sweetener == "Honey":
        return f"{name} will have a black tea with Honey"
    elif liquid == "Water" and (sweetener != "Sugar" or sweetener != "Honey"):
        return f"{name} will have a black tea"
    else:
        return f"{name} will have tea with just Milk"
```

```
In [19]: tea_maker("katti")
```

```
Out[19]: 'katti will have a tea with Milk and Sugar'
```

```
In [20]: tea_maker("katti", liquid = "Water")
```

```
Out[20]: 'katti will have a black tea with Sugar'
```

```
In [21]: tea_maker("manju", liquid = "Water", sweetener = "Honey")
```

```
Out[21]: 'manju will have a black tea with Honey'
```

```
In [22]: tea_maker("katti", sweetener = "Honey")
```

```
Out[22]: 'katti will have a tea with Milk and Honey'
```

```
In [27]: tea_maker("Indradev", liquid = "Water", sweetener = "No")
```

```
Out[27]: 'Indradev will have a black tea'
```

In Python, you can define functions with both mandatory and optional arguments. Mandatory arguments are required for the function to execute, while optional arguments have default values and can be omitted when the function is called. This flexibility allows you to create more versatile functions.

Documenting the Functions in python:

Documenting functions in Python is crucial for maintaining code readability and usability, especially in collaborative projects. Proper documentation helps other developers understand the purpose and usage of your functions. The most common way to document functions in Python is by using docstrings.

```
In [29]: def tea_maker(name, liquid = "Milk", sweetener = "Sugar"):

        """
        Purpose: To make tea based on users Choice

        Arguments:
        name (mandatory) : User name must be supplied
        liquid (optional) : Defaults "Milk", user can supply "Water"
        sweetener (optional) : Defaults "Sugar", user can also supply "Honey"

        ex:
        1. tea_maker("Gowtham") will return 'Gowtham will have a tea with Milk and Sugar'
        2. tea_maker("Gowtham", liquid = "Water") will return 'Gowtham will have a black tea with Sugar'
        """

        if liquid == "Milk" and sweetener == "Sugar":
            return f"{name} will have a tea with Milk and Sugar"
        elif liquid == "Water" and sweetener == "Sugar":
            return f"{name} will have a black tea with Sugar"
        elif liquid == "Milk" and sweetener == "Honey":
            return f"{name} will have a tea with Milk and Honey"
        elif liquid == "Water" and sweetener == "Honey":
            return f"{name} will have a black tea with Honey"
        elif liquid == "Water" and (sweetener != "Sugar" or sweetener != "Honey"):
            return f"{name} will have a black tea"
        else:
            return f"{name} will have tea with just Milk"
```

```
In [ ]: tea_maker()
```

```
In [30]: print(tea_maker.__doc__)
```

```
        Purpose: To make tea based on users Choice

        Arguments:
        name (mandatory) : User name must be supplied
        liquid (optional) : Defaults "Milk", user can supply "Water"
        sweetener (optional) : Defaults "Sugar", user can also supply "Honey"

        ex:
        1. tea_maker("Gowtham") will return 'Gowtham will have a tea with Milk and Sugar'
        2. tea_maker("Gowtham", liquid = "Water") will return 'Gowtham will have a black tea with Sugar'
```

```
In [23]: tea_maker("KATTI", 100)
```

```
Out[23]: 'KATTI will have tea with just Milk'
```

```
In [24]: tea_maker("KATTI", 250)
```

```
Out[24]: 'KATTI will have tea with just Milk'
```

Hands on Activity

1. Checking whether the input is greater than set of values in a list or not.

```
In [1]: import random
a = [random.randint(10, 100) for i in range(10)]
print(a)

[15, 23, 94, 88, 11, 53, 54, 40, 18, 85]

In [3]: user_ip = int(input("Supply a number of your choice: "))

for i in a:
    if user_ip > i:
        print(f"User Input {user_ip} is greater than the value {i}")

Supply a number of your choice: 55
User Input 55 is greater than the value 15
User Input 55 is greater than the value 23
User Input 55 is greater than the value 11
User Input 55 is greater than the value 53
User Input 55 is greater than the value 54
User Input 55 is greater than the value 40
User Input 55 is greater than the value 18
```

2. Check whether the given number is an Armstrong number or not.

```
In [ ]: 153

1 ** 3 + 5 ** 3 + 3 ** 3
1 + 125 + 27
126 + 27
153

In [50]: def armstrong_num():

    ip = input("Enter a Number: ")

    tot = 0

    for i in ip:
        tot += int(i) ** len(ip)

    if tot == int(ip):
        return "It is an Armstrong Number"
    else:
        return "It is not an Armstrong Number"

In [52]: armstrong_num()

Enter a Number: 153

Out[52]: 'It is an Armstrong Number'

In [53]: armstrong_num()

Enter a Number: 123

Out[53]: 'It is not an Armstrong Number'
```

3. Prime Number or Not

If the number is divisible by itself and 1, then it is a prime number

```
In [1]: def prime_number():  
        ip = int(input("Supply a number greater than 1: "))  
        for i in range(2, ip):  
            if ip % i == 0:  
                print(f"{ip} is not a prime number")  
                break  
        else:  
            print(f"{ip} is a Prime Number")
```

```
In [2]: prime_number()  
  
Supply a number greater than 1: 30  
30 is not a prime number
```

```
In [67]: prime_number()  
  
Supply a number greater than 1: 97  
97 is a Prime Number
```

```
In [68]: prime_number()  
  
Supply a number greater than 1: 12  
12 is not a prime number
```

2. Input, output, data types, operators and simple engineering calculations.

Example 1: Area of a Rectangle (Plot Area)

Formula

$$\text{Area} = \text{Length} \times \text{Width}$$

Program

```
length = float(input("Enter length (m): "))
width = float(input("Enter width (m): "))

area = length * width

print(f"Area = {area:.2f} square meters")
```

Sample Output

```
Enter length (m): 25
Enter width (m): 18
Area = 450.00 square meters
```

Example 2: Volume of Concrete

Formula

$$\text{Volume} = \text{Length} \times \text{Width} \times \text{Depth}$$

Program

```
length = float(input("Length (m): "))
width = float(input("Width (m): "))
depth = float(input("Depth (m): "))

volume = length * width * depth

print(f"Concrete Volume = {volume:.3f} m³")
```

Sample Output

```
Length = 10
Width = 5
Depth = 0.15
Volume = 7.500 m³
```

3. Conditional Statements and Looping Constructs for Civil Engineering Problem Solving

Many Civil Engineering problems require **decision-making** and **repetitive calculations**. For example:

- Determining whether concrete strength meets the design requirement.
- Checking if a beam is safe under the applied load.
- Calculating quantities for multiple rooms or floors.
- Processing test results for several soil samples.

Python provides **conditional statements** for decision-making and **loops** for repetitive tasks, making engineering calculations faster

Conditional Statements

Conditional statements allow a program to execute different blocks of code based on whether a condition is **True** or **False**.

Types of Conditional Statements

i. if

Check Concrete Grade

Suppose the required compressive strength is **25 MPa**.

```
strength = float(input("Enter concrete strength (MPa): "))  
  
if strength >= 25:  
    print("Concrete satisfies the design requirement.")
```

Sample Output

```
Enter concrete strength: 28  
Concrete satisfies the design requirement.
```

ii. if-else

Beam Safety Check

```
applied_load = float(input("Applied Load (kN): "))  
capacity = float(input("Beam Capacity (kN): "))  
  
if applied_load <= capacity:  
    print("Beam is Safe")  
else:  
    print("Beam is Unsafe")
```

Sample Output

```
Applied Load = 180  
Beam Capacity = 200  
  
Beam is Safe
```

iii. if-elif-else

Soil Classification Based on Bearing Capacity

```
bearing = float(input("Enter Safe Bearing Capacity (kN/m²): "))

if bearing >= 300:
    print("Hard Soil")
elif bearing >= 150:
    print("Medium Soil")
else:
    print("Soft Soil")
```

iv. Nested if: An if statement inside another if.

Building Approval

```
height = float(input("Building Height (m): "))
setback = float(input("Setback Distance (m): "))

if height <= 15:
    if setback >= 3:
        print("Building Approved")
    else:
        print("Increase Setback")
else:
    print("High-rise Building Rules Apply")
```

Looping Constructs

Loops repeat a block of code multiple times.

Python provides:

- i. for loop
Used when the number of iterations is known.
Example: Calculate Area of Five Plots

```
for i in range(1, 6):
    length = float(input(f"Enter length of Plot {i}: "))
    width = float(input(f"Enter width of Plot {i}: "))

    area = length * width

    print(f"Area of Plot {i} = {area:.2f} m²")
```

Example: Multiplication Table

```
length = float(input("Enter beam length: "))

for i in range(1, 11):
    print(length, "x", i, "=", length * i)
```

4. Functions and modular programming for reusable engineering computations

A function is a block of code that performs a specific task. Instead of writing the same code repeatedly, we write it once inside a function and call it whenever needed.

```
# Syntax
def function_name(parameters):
    # Function body
    return value

# def → Keyword used to define a function.
# function_name → Name of the function.
# parameters → Inputs given to the function.
# return → Sends the result back to the caller
```

Why Use Functions?

Functions help us:

1. Avoid writing repetitive code.
2. Make programs easier to understand.
3. Reduce programming errors.
4. Reuse the same calculations in different projects.
5. Divide a large program into smaller modules.

In Civil Engineering, many formulas such as area, volume, stress, load, and concrete mix calculations are repeatedly used. Functions make these calculations simple and reusable.

```
# Function Without Parameters
def department():
    print("Department of Civil Engineering")

department()

output
department()
Department of Civil Engineering

# Function with Parameters
# Calculate the area of a rectangular footing.

def footing_area(length, width):
    area = length * width
    print("Footing Area =", area, "m²")

footing_area(4, 3)

output
footing_area(12, 2)
Footing Area = 24 m²
```

Modular Programming

Modular programming is the practice of dividing a large program into small, independent modules (functions). Each module performs one specific task.

Instead of writing one long program, we divide it into smaller reusable functions.

Advantages

- Easy to understand
- Easy to test
- Easy to debug
- Code can be reused
- Easy to maintain
- Suitable for large engineering projects

```
# Modular Program for Building Material Estimation
def slab_volume(length, width, thickness):
    return length * width * thickness

def concrete_weight(volume):
    density = 2400
    return volume * density

def cement_bags(volume):
    return volume * 7

# Main Program
length = 8
width = 5
thickness = 0.15

volume = slab_volume(length, width, thickness)
weight = concrete_weight(volume)
bags = cement_bags(volume)

print("Concrete Volume =", volume, "m³")
print("Concrete Weight =", weight, "kg")
print("Approximate Cement Bags =", bags)

output
Concrete Volume = 6.0 m³
Concrete Weight = 14400.0 kg
Approximate Cement Bags = 42.0
```

5. Arrays, lists, tuples, and dictionaries for data handling.

In Civil Engineering, arrays, lists, tuples, and dictionaries are useful for storing and handling engineering data such as survey measurements, material properties, rainfall data, structural loads, soil parameters, and project information.

1. Arrays

An **array** stores multiple values of the same type and is particularly useful for numerical calculations. In Python, the **NumPy** library is commonly used for arrays.

Example: Survey measurements

Suppose the measured elevations at five points are:

```
import numpy as np
|
elevation = np.array([102.5, 103.2, 101.8, 104.1, 102.9])

print("Elevations:", elevation)
print("Average elevation:", np.mean(elevation))
print("Maximum elevation:", np.max(elevation))
print("Minimum elevation:", np.min(elevation))
```

Output:

```
Elevations: [102.5 103.2 101.8 104.1 102.9]
Average elevation: 102.9
Maximum elevation: 104.1
Minimum elevation: 101.8
```

2. Lists

A list is a collection that can store multiple values. Lists are ordered and changeable, meaning we can add, remove, or modify elements.

Example: Concrete cube test results

```
strength = [28.5, 30.2, 29.8, 31.5, 27.9]

print("Strength values:", strength)

strength.append(32.1)

print("After adding a new value:", strength)
print("Number of specimens:", len(strength))
```

We can also calculate the average:

```
average = sum(strength) / len(strength)

print("Average compressive strength:", average, "MPa")
```

Accessing individual values:

```
print(strength[0])    # First value
print(strength[2])    # Third value
print(strength[-1])   # Last value
```

3. Tuples

A tuple is similar to a list, but its values cannot normally be changed after creation. Tuples are useful when data should remain fixed.

Example: Dimensions of a beam

```
beam_dimensions = (300, 450, 5000)

print("Beam width:", beam_dimensions[0], "mm")
print("Beam depth:", beam_dimensions[1], "mm")
print("Beam length:", beam_dimensions[2], "mm")
```

```
beam_dimensions[0] = 350
```

Python will produce an error because tuples are not designed for modification.

```
point_A = (125.5, 250.8, 105.2)

print("X coordinate:", point_A[0])
print("Y coordinate:", point_A[1])
print("Elevation:", point_A[2])
```

4. Dictionaries

A dictionary stores information as key-value pairs. It is very useful in Civil Engineering because engineering data often has a parameter name and corresponding value.

Example: Concrete properties

```
concrete = {  
    "grade": "M25",  
    "density": 25,  
    "compressive_strength": 25,  
    "unit": "kN/m3"  
}  
  
print("Grade:", concrete["grade"])  
print("Density:", concrete["density"], "kN/m3")  
print("Compressive strength:", concrete["compressive_strength"], "MPa")
```

Example: Soil properties

```
soil = {  
    "type": "Clay",  
    "moisture_content": 18.5,  
    "specific_gravity": 2.65,  
    "liquid_limit": 45,  
    "plastic_limit": 22  
}  
  
print("Soil type:", soil["type"])  
print("Moisture content:", soil["moisture_content"], "%")  
print("Specific gravity:", soil["specific_gravity"])
```

6. Plotting using Python libraries for basic civil engineering data visualization.

Data visualization is an important skill for Civil Engineers because engineering data is often easier to understand through **graphs and charts** rather than tables of numbers.

Python provides several libraries for visualization. The most commonly used are:

- **Matplotlib** – basic and highly flexible plotting
- **Seaborn** – attractive statistical graphs
- **Plotly** – interactive graphs

1. Line Plot – Rainfall Data

A line graph is useful for showing how a quantity changes with time.

For example, consider monthly rainfall:

```
import matplotlib.pyplot as plt
|
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
rainfall = [5, 12, 18, 45, 110, 180]

plt.plot(months, rainfall, marker='o')

plt.xlabel("Month")
plt.ylabel("Rainfall (mm)")
plt.title("Monthly Rainfall")
plt.grid(True)

plt.show()
```

2. Bar Chart – Compressive Strength

```
import matplotlib.pyplot as plt

specimens = ["Cube 1", "Cube 2", "Cube 3", "Cube 4", "Cube 5"]
strength = [24.5, 26.2, 25.8, 27.1, 25.4]

plt.bar(specimens, strength)

plt.xlabel("Specimen")
plt.ylabel("Compressive Strength (MPa)")
plt.title("Concrete Compressive Strength")

plt.show()
```

3. Scatter Plot – Relationship Between Variables

```
import matplotlib.pyplot as plt

water_cement_ratio = [0.35, 0.40, 0.45, 0.50, 0.55]
strength = [38, 35, 32, 28, 24]

plt.scatter(water_cement_ratio, strength)

plt.xlabel("Water-Cement Ratio")
plt.ylabel("Compressive Strength (MPa)")
plt.title("Water-Cement Ratio vs Compressive Strength")

plt.grid(True)
plt.show()
```

4. Histogram – Distribution of Data

```
import matplotlib.pyplot as plt

strength = [24, 26, 25, 28, 27, 25, 29, 26, 24, 27,
            28, 25, 26, 27, 29, 24, 25, 28, 26, 27]

plt.hist(strength, bins=5)

plt.xlabel("Compressive Strength (MPa)")
plt.ylabel("Number of Specimens")
plt.title("Distribution of Concrete Strength")

plt.show()
```

7. Solution of simultaneous equations relevant to structural or water resources problems.

Structural Engineering Example – Beam Reactions

```
1 import numpy as np
2
3 # Example: simple 3-member truss, unknowns = [F1, F2, F3]
4 # Coefficients come from cos/sin of member angles at each joint
5 A = np.array([
6     [1,      0.866,  0],      # Joint A: sum Fx
7     [0,      0.5,   -1],      # Joint A: sum Fy
8     [-1,     0,     1]        # Joint B: sum Fx
9 ])
10 B = np.array([0, 10, 0])      # applied loads (kN)
11
12 F = np.linalg.solve(A, B)
13 print("Member forces (kN):", F)
```

Member forces (kN): [-6.33967789 7.32064422 -6.33967789]

```
1 import numpy as np
2
3 # Example: 3 equations, 3 unknowns
4 A = np.array([[4, -1, 0],
5              [-1, 4, -1],
6              [0, -1, 4]])
7 B = np.array([15, 10, 10])
8
9 x = np.linalg.solve(A, B)
10 print("Solution:", x)
```

Solution: [4.91071429 4.64285714 3.66071429]

```
1 A = np.array([[4, -1, 0],
2              [-1, 4, -1],
3              [0, -1, 4]])
4 B = np.array([15, 10, 10])
5
6 x = np.linalg.solve(A, B)
7 print("Solution: x1 = ", round(x[0],3))
8 print("Solution: x2 = ", round(x[1],3))
9 print("Solution: x3 = ", round(x[2],3))
```

Solution: x1 = 4.911
Solution: x2 = 4.643
Solution: x3 = 3.661

```
1 """
2 Solving Simultaneous Equations in Water Resources Engineering
3 -----
4 Problem:
5 A main canal supplies water to two secondary canals, C1 and C2.
6
7 Total discharge delivered      :  $Q_1 + Q_2 = 18$  (m3/s)
8 Design / head ratio constraint :  $Q_1 - 2Q_2 = -3$  (m3/s)
9
10 Find  $Q_1$  and  $Q_2$ .
11 """
12 # Coefficient matrix A and right-hand-side vector b, from:
13 #    $1Q_1 + 1Q_2 = 18$ 
14 #    $1Q_1 - 2Q_2 = -3$ 
15 A = np.array([[1, 1],
16               [1, -2]])
17 b = np.array([18, -3])
18
19 solution_numpy = np.linalg.solve(A, b)
20 print(f"   $Q_1 = \{solution\_numpy[0]:.3f\}$  m3/s")
21 print(f"   $Q_2 = \{solution\_numpy[1]:.3f\}$  m3/s\n")
```

$Q_1 = 11.000$ m³/s
 $Q_2 = 7.000$ m³/s

8. Computation of area, volume, centroid, and moment of inertia for standard sections.

```
1 # Area, Volume, Centroid and Moment of Inertia of a Rectangle
2
3 b = float(input("Enter width of rectangle (m): "))
4 h = float(input("Enter height of rectangle (m): "))
5 L = float(input("Enter length of member (m): "))
6
7 # Area
8 A = b * h
9
10 # Volume
11 V = A * L
12
13 # Centroid
14 x_bar = b / 2
15 y_bar = h / 2
16
17 # Moment of inertia
18 Ix = (b * h**3) / 12
19 Iy = (h * b**3) / 12
20
21 print("\n--- RESULTS ---")
22 print("Area =", A, "m2")
23 print("Volume =", V, "m3")
24 print("Centroid = (", x_bar, ",", y_bar, ") m")
25 print("Ix =", Ix, "m4")
26 print("Iy =", Iy, "m4")
```

```
1 # Circular Section
2 import math
3
4 # Circular section
5
6 d = float(input("Enter diameter (m): "))
7 L = float(input("Enter length of cylinder (m): "))
8
9 r = d / 2
10
11 # Area
12 A = math.pi * d**2 / 4
13
14 # Volume
15 V = A * L
16
17 # Centroid
18 x_bar = 0
19 y_bar = 0
20
21 # Moment of inertia
22 I = math.pi * d**4 / 64
23
24 print("\n--- RESULTS ---")
25 print("Area =", A, "m2")
26 print("Volume =", V, "m3")
27 print("Centroid is at the centre of the circle")
28 print("Ix =", I, "m4")
29 print("Iy =", I, "m4")
```

```
1 # Properties of a triangular section
2
3 b = float(input("Enter base of triangle (m): "))
4 h = float(input("Enter height of triangle (m): "))
5
6 # Area
7 A = 0.5 * b * h
8
9 # Centroid from base
10 y_bar = h / 3
11
12 # Centroidal moment of inertia
13 Ix = b * h**3 / 36
14
15 # Moment of inertia about base
16 I_base = b * h**3 / 12
17
18 print("\n--- RESULTS ---")
19 print("Area =", A, "m²")
20 print("Centroid from base =", y_bar, "m")
21 print("Moment of inertia about centroid =", Ix, "m⁴")
22 print("Moment of inertia about base =", I_base, "m⁴")
```

9. Program for slope, deflection, or load calculation in simple structural members.

1. Slope and Deflection of a Simply Supported Beam (Point Load at Centre)

For a simply supported beam carrying a concentrated load **P** at the center:

- Maximum Slope:

$$\theta = \frac{PL^2}{16EI}$$

- Maximum Deflection:

$$Y = \frac{PL^3}{48EI}$$

Where:

Symbol	Meaning
P	Point Load (N)
L	Length of Beam (m)
E	Young's Modulus (N/m ²)
I	Moment of Inertia (m ⁴)

```

1 # Slope and Deflection of Simply Supported Beam
2 P = float(input("Enter point load P (N): "))
3 L = float(input("Enter beam length L (m): "))
4 E = float(input("Enter Young's Modulus E (N/m²): "))
5 I = float(input("Enter Moment of Inertia I (m⁴): "))
6
7 # Calculations
8 slope = (P * L**2) / (16 * E * I)
9 deflection = (P * L**3) / (48 * E * I)
10
11 print("\nResults")
12 print("Maximum Slope =", slope, "radians")
13 print("Maximum Deflection =", deflection, "m")

```

```

Enter point load P (N): 500
Enter beam length L (m): 5
Enter Young's Modulus E (N/m²): 2000000
Enter Moment of Inertia I (m⁴): 1500

```

Results

```

Maximum Slope = 2.604166666666667e-07 radians
Maximum Deflection = 4.340277777777778e-07 m

```

2. Deflection of a Cantilever Beam (Point Load at Free End)

- Slope at free end:

$$\theta = \frac{PL^2}{2EI}$$

- Deflection at free end:

$$Y = \frac{PL^3}{3EI}$$

```
1 # Cantilever Beam Deflection
2
3 P = float(input("Enter load P (N): "))
4 L = float(input("Enter length L (m): "))
5 E = float(input("Enter Young's Modulus E (N/m²): "))
6 I = float(input("Enter Moment of Inertia I (m⁴): "))
7
8 slope = (P * L**2) / (2 * E * I)
9 deflection = (P * L**3) / (3 * E * I)
10
11 print("\nResults")
12 print("Slope at free end =", slope, "radians")
13 print("Deflection at free end =", deflection, "m")
```

```
Enter load P (N): 250
Enter length L (m): 3
Enter Young's Modulus E (N/m²): 2050000
Enter Moment of Inertia I (m⁴): 120000
```

```
Results
Slope at free end = 4.573170731707317e-09 radians
Deflection at free end = 9.146341463414635e-09 m
```

10 Program for beam reactions under point loads and uniformly distributed loads

BEAM REACTIONS – SIMPLY SUPPORTED BEAM

1. SIMPLY SUPPORTED BEAM WITH A POINT LOAD	2. SIMPLY SUPPORTED BEAM WITH UDL	3. POINT LOAD + UDL
Diagram Equations Where, $L = a + b$ $R_A = \frac{Pb}{L} \quad R_B = \frac{Pa}{L}$ Where, P = Point load (kN) a = Distance of load from support A (m) b = Distance of load from support B (m) L = Total span of the beam (m) R_A, R_B = Reactions at supports A and B (kN) Equilibrium Check $R_A + R_B = P$	Diagram Equations Total UDL (W) = wL $R_A = R_B = \frac{wL}{2}$ Where, w = UDL intensity (kN/m) L = Total span of the beam (m) R_A, R_B = Reactions at supports A and B (kN) Equilibrium Check $R_A + R_B = wL$ (Total Load)	Diagram Equations Total UDL (W) = wL $R_B = \frac{Pa + wL(L/2)}{L}$ $R_A = P + wL - R_B$ Where, P = Point load (kN) w = UDL intensity (kN/m) L = Total span of the beam (m) a = Distance of point load from support A (m) R_A, R_B = Reactions at supports A and B (kN) Equilibrium Check $R_A + R_B = P + wL$ (Total Load)

Note: All reactions are in kN if loads are in kN and dimensions are in meters.

1. Simply Supported Beam with a Point Load

```
1 # 1. Simply Supported Beam with a Point Load
2 # Beam Reactions for a Point Load
3
4 P = float(input("Enter point load P (kN): "))
5 L = float(input("Enter beam length L (m): "))
6 a = float(input("Enter distance of load from support A (m): "))
7
8 b = L - a
9
10 RA = (P * b) / L
11 RB = (P * a) / L
12
13 print("\nBeam Reactions")
14 print("Reaction at A =", RA, "kN")
15 print("Reaction at B =", RB, "kN")
```

```
Enter point load P (kN): 20
Enter beam length L (m): 3
Enter distance of load from support A (m): 10
```

```
Beam Reactions
Reaction at A = -46.666666666666664 kN
Reaction at B = 66.66666666666667 kN
```

2. Simply Supported Beam with UDL

```
1 # 2. Simply Supported Beam with UDL
2 # Beam Reactions for UDL
3
4 w = float(input("Enter UDL w (kN/m): "))
5 L = float(input("Enter beam length L (m): "))
6
7 total_load = w * L
8
9 RA = total_load / 2
10 RB = total_load / 2
11
12 print("\nBeam Reactions")
13 print("Total UDL =", total_load, "kN")
14 print("Reaction at A =", RA, "kN")
15 print("Reaction at B =", RB, "kN")
```

```
Enter UDL w (kN/m): 10
Enter beam length L (m): 3
```

```
Beam Reactions
Total UDL = 30.0 kN
Reaction at A = 15.0 kN
Reaction at B = 15.0 kN
```

3. Point Load + UDL

```
1 # 3. Point Load + UDL
2 # Beam Reactions for Point Load + UDL
3 |
4 P = float(input("Enter point load P (kN): "))
5 w = float(input("Enter UDL w (kN/m): "))
6 L = float(input("Enter beam length L (m): "))
7 a = float(input("Enter distance of point load from A (m): "))
8
9 # Total UDL
10 W = w * L
11
12 # Reaction at B using moment equilibrium about A
13 RB = (P * a + W * (L / 2)) / L
14
15 # Reaction at A using vertical equilibrium
16 RA = P + W - RB
17
18 print("\nBeam Reactions")
19 print("Total UDL =", W, "kN")
20 print("Reaction at A =", RA, "kN")
21 print("Reaction at B =", RB, "kN")
22
23 # Check equilibrium
24 print("\nEquilibrium Check")
25 print("RA + RB =", RA + RB, "kN")
26 print("Total Load =", P + W, "kN")
```

- Enter point load P (kN): 10
Enter UDL w (kN/m): 10
Enter beam length L (m): 10
Enter distance of point load from A (m): 5

```
Beam Reactions
Total UDL = 100.0 kN
Reaction at A = 55.0 kN
Reaction at B = 55.0 kN
```

```
Equilibrium Check
RA + RB = 110.0 kN
Total Load = 110.0 kN
```

Viva Questions

1. What is Python and what are its key features?
2. Why is Python called an interpreted language?
3. What is the significance of indentation in Python?
4. What is the difference between dynamically typed and statically typed languages?
5. What is PEP 8?
6. What are tokens and keywords in Python?
7. What is the purpose of the type() function?
8. What are comments in Python and how do you write them?
9. What are the primary built-in data types in Python?
10. What is the difference between mutable and immutable objects?
11. Differentiate between a list and a tuple.
12. What are the properties of a Python set?
13. What is a dictionary in Python?
14. What is the difference between the == operator and the is operator?
15. How does string slicing work in Python?
16. What is the conditional structure if-elif-else used for?
17. What is the difference between a for loop and a while loop?
18. Explain the purpose of break, continue, and pass.
19. What does the range() function do?
20. How do you define a function in Python?
21. What is a lambda function?
22. What is the difference between local and global variables?
23. What is the purpose of the return statement?
24. What is runtime error/exception handling in Python?
25. What do the else and finally blocks do in exception handling?
26. What is the safest way to open and handle files in Python?
27. What is a class and an object?
28. What is the purpose of the __init__ method?
29. What does the self parameter represent in class methods?
30. What is inheritance?