Python Programming

Subject Code: DI02032011 / 1323203

Comprehensive Syllabus

Diploma in Engineering
Information & Communication Technology

Python

Milav Dabgar
AICTE QIP PG Certificate in Deep Learning (SVNIT Surat)
Diploma in Data Science (IIT Madras)

March 30, 2026

Preface

Python has rapidly become one of the most dominant and accessible programming languages across the globe, powering everything from massive server-side web applications to advanced Data Analytics and Machine Learning systems. Its philosophy heavily emphasizes code readability, characterized by its notable use of significant indentation.

This manual, designed specifically for the **Python Programming (DI02032011 / 1323203)** course within the Diploma Engineering curriculum, bridges the critical gap between abstract coding theory and concrete, high-impact industrial applications.

The core objectives of this book are to enable students to:

1. Prepare logical flowcharts and write structurally sound algorithms for computational problem-solving.
2. Master Python's intrinsic data types, robust control structures, and rich set of operators.
3. Develop and deploy modular user-defined functions and leverage standard standard library modules like `math` and `random`.
4. Execute extremely fast data manipulation using core structures like Dictionaries, Lists, Sets, and Tuples.
5. Pass University examinations with perfect clarity by studying the highly integrated "PYQ Exam Focus" and "Complete Cheatsheet" matrices at the end of each unit.

Every unit has been meticulously crafted to align directly with the GTU weightage matrix, guaranteeing that examination preparation is highly optimized and profoundly effective.

Milav Dabgar
March 30, 2026

About the Author

Milav Dabgar

Lecturer and Researcher in Mathematics, AI, and Software Engineering.
Passionate about engineering education and advanced computational systems.

Milav Dabgar is a dedicated educator holding vast, rigorous experience in teaching Mathematics, Information Technology, and core computing paradigms. He is profoundly committed to open-source knowledge and pedagogical innovation, consistently striving to develop resources that simplify hyper-complex engineering principles into intuitively accessible logic for engineering students.

Recent Academic Milestones:

- **Diploma in Data Science, Indian Institute of Technology (IIT) Madras**
  Graduated with an extraordinary, flawless **10/10 CGPA**, officially recognized as the Batch Topper. Developed deep expertise in Machine Learning mechanics, massive data pipelines, and predictive statistical modelling.
- **AICTE QIP PG Certificate in Deep Learning, SVNIT Surat**
  Mastered advanced neural network architectures, optimizing convolutional/recurrent networks and accelerating AI training models.

Beyond his formal academic pursuits, Milav actively designs large-scale, enterprise-ready documentation structures—specifically utilizing powerful LaTeX typography—to elevate the standard of study manuals provided to Diploma Engineering candidates. This Python Programming reference guide acts as the definitive iteration of that architectural philosophy.

Contents

CHAPTER 1

Problem Solving using Flowchart and Algorithm

1.1 Introduction to Problem Solving

Before writing a single line of Python code, a programmer must rigorously define the problem statement and structure a logical path to the solution. Programming fundamentally consists of breaking a large, complex problem into a sequence of small, mathematically executable steps.

Algorithm

An **Algorithm** is a finite, step-by-step sequence of well-defined instructions written in plain human language to solve a specific computational problem.

1.1.1 Characteristics of an Algorithm

An algorithm is mathematically sound only if it satisfies the following rigid characteristics:

- **Finiteness:** It must unconditionally terminate after a finite number of steps. Infinite loops are fatal errors.
- **Definiteness:** Every single step must be unambiguously defined and crystal clear.
- **Input:** It must accept zero or more well-defined inputs.
- **Output:** It must produce at least one specific, identifiable output.
- **Effectiveness:** Every operation must be elemental enough that it could theoretically be performed exactly using just a pencil and paper.

1.1.2 Importance of Algorithms and Flowcharts

- **Logic Blueprint:** They act as architectural blueprints. Just as you do not build a skyscraper without blueprint equations, you do not write complex software without algorithmic design.
- **Language Independence:** They are universal. A well-designed algorithm can be coded into Python, Java, C++, or any future programming language.

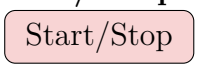
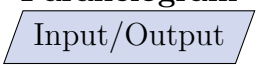
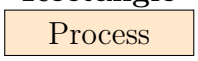
- **Debugging:** It is drastically faster to detect structural logic errors in a flowchart than tracing errors deep inside thousands of lines of syntax code.

1.2 Flowcharts: Visual Execution

Flowchart

A **Flowchart** is the precise, graphical representation of an algorithm. It uses highly specific geometric shapes to represent different types of computational actions (like input, processing, and decision-making) connected by directional arrows.

1.2.1 Standard Flowchart Symbols

Symbol Shape	Purpose and Functionality
Oval / Ellipse 	**Terminal:** Indicates the absolute beginning (START) or end (STOP/END) of the program execution sequence.
Parallelogram 	**Input / Output:** Represents reading external data (e.g., scanning a keyboard number) or printing data to the screen.
Rectangle 	**Processing Box:** Represents mathematical calculations, variable assignments, or data manipulation (e.g., $C = A + B$).
Diamond 	**Decision / Selection:** Evaluates a Boolean (True/False) condition. Splits the flow into two distinct paths based on the answer.
Arrows	**Flowline:** Explicitly dictates the directional sequence of execution.

1.2.2 Differences: Algorithm vs Flowchart

Parameter	Algorithm	Flowchart
Definition	Step-by-step written logical instructions.	Graphical diagram of instructions.
Complexity	Extremely easy to write natively, highly compact for complex algorithms.	Requires massive page space for complex logic; intricate drawing.
Readability	Highly abstract, often requires deep reading.	Visually intuitive, easily grasped at a glance.
Modification	Extremely easy to update or edit text lines.	Exceedingly difficult to modify; often requires redrawing entire chains.

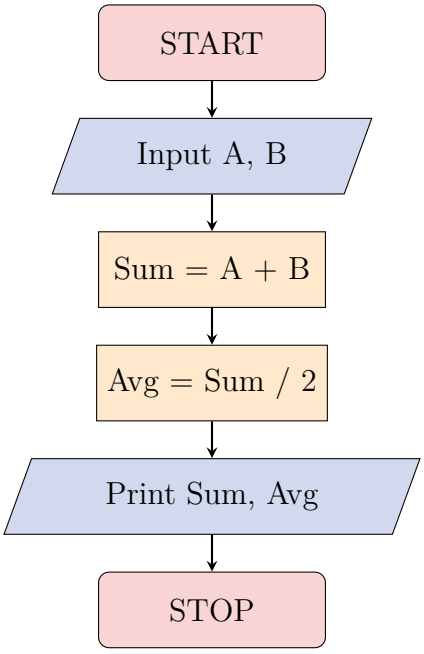
1.3 Essential Flowchart & Algorithm Designs

Based heavily on historical university examinations, the following standard algorithm structures must be mastered.

1.3.1 Example 1: Sum and Average of Two Numbers

Algorithm:

1. START
2. Input values for variable A and variable B.
3. Calculate Sum: $Sum = A + B$.
4. Calculate Average: $Avg = Sum/2$.
5. Print the calculated Sum and Avg.
6. STOP

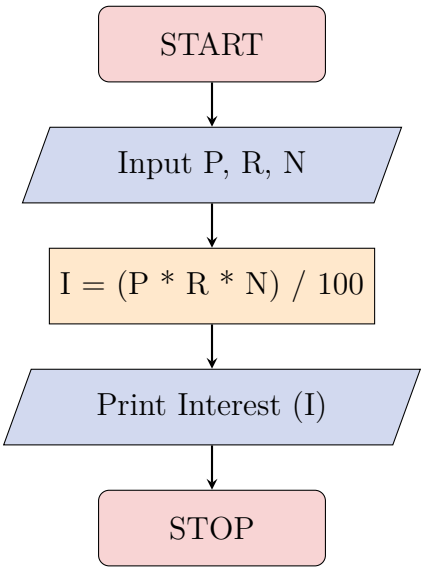


1.3.2 Example 2: Simple Interest Calculation

Formula: $I = \frac{P \times R \times N}{100}$ (Where P = Principal, R = Rate, N = Time).

Algorithm:

1. START
2. Input values for P, R, N.
3. Process strictly via formula: $I = (P \times R \times N)/100$.
4. Print the calculated Interest (I).
5. STOP

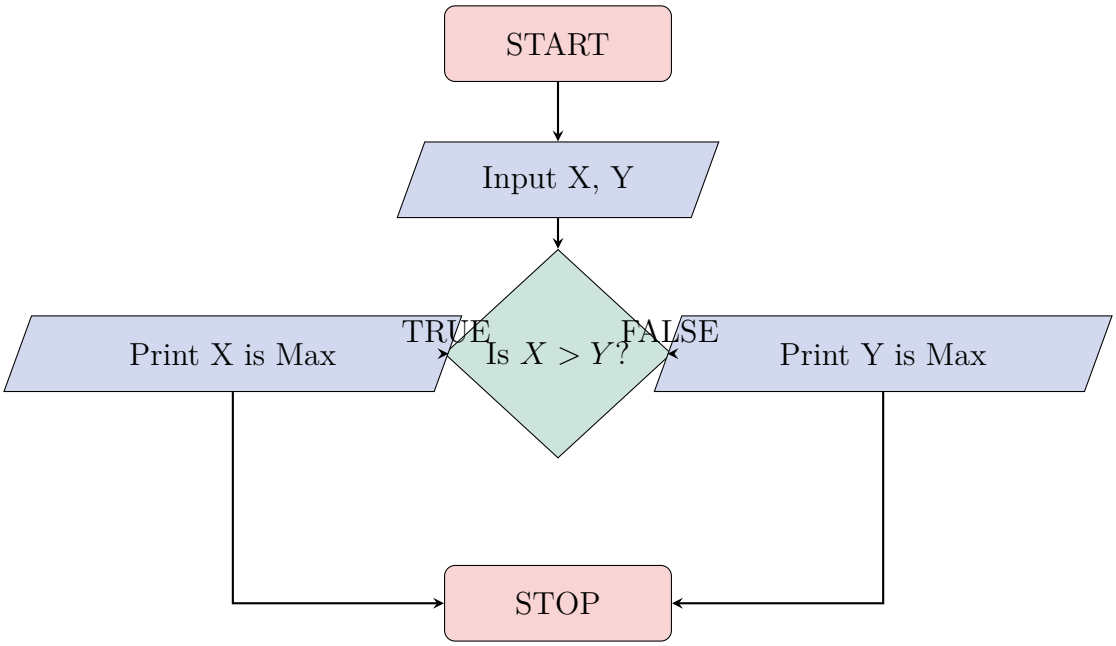


1.3.3 Example 3: Decision Making — Finding the Maximum of Two Numbers

This specifically tests the "Decision Diamond" pathway split.

Algorithm:

1. START
2. Input X and Y.
3. Check Condition: IS $X > Y$?
4. IF condition is TRUE, then Print "X is Largest".
5. IF condition is FALSE, then Print "Y is Largest".
6. STOP



1.3.4 Example 4: Repetition — Print Even Numbers from 1 to 20

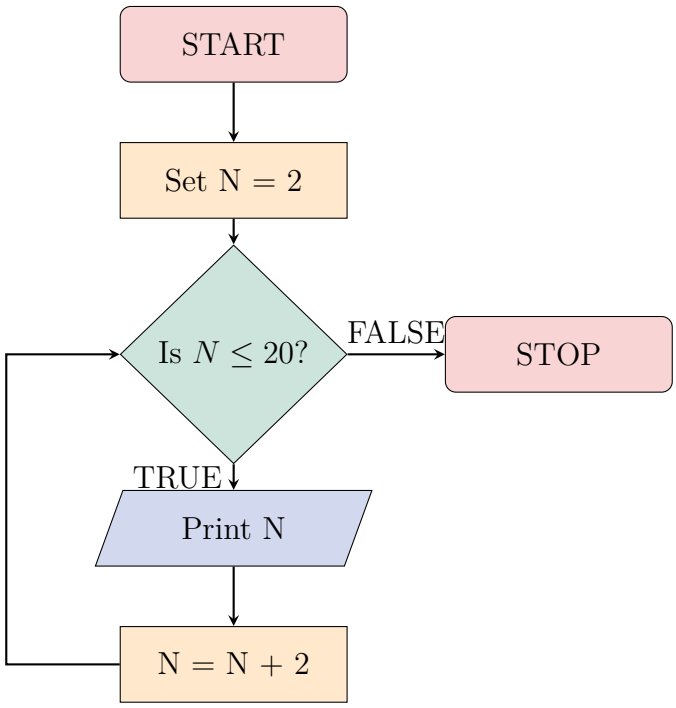
This critically tests logical looping capabilities via variable counters.

Algorithm:

1. START
2. Initialize variable counter $N = 2$.
3. Check looping limit condition: IS $N \leq 20$?
4. IF TRUE:
   (a) Print the current value of N.
   (b) Increment counter: $N = N + 2$.

(c) Loop heavily back to step 3.

5. IF FALSE: Terminate loops → Go to STOP.
6. STOP



1.4 Pseudocode: The Middle Ground

Pseudocode

Pseudocode (False/Fake Code) is an informal, high-level structural description of a computer program. It sits directly between pure human-language algorithms and strict machine syntax (like Python). It ignores rigid syntax rules entirely (no brackets or semicolons needed) but rigidly enforces logical structural intent.

1.4.1 Example: Pseudocode to Check Positive/Negative Number

Pseudocode Layout

```
BEGIN
    READ inputNumber
    IF inputNumber > 0 THEN
        PRINT "Number is Positive"
    ELSE IF inputNumber < 0 THEN
        PRINT "Number is Negative"
    ELSE
        PRINT "Number is Exactly Zero"
    END IF
```

```
END
```

Unit 1: Flowchart & Logic — Complete Cheatsheet

1. Core Logic Concepts

- **Algorithm:** Step-by-step universal human language instructions. (Pros: Universal, Logical / Cons: Hard to read for complex scaling).
- **Flowchart:** Graphical diagram execution using established geometric templates. (Pros: Extremely intuitive visual verification / Cons: Annoyingly hard to rigidly manually redraw if logic fails midway).
- **Pseudocode:** Fake syntax-free structured code utilizing capitalized intent actions (READ, IF, PRINT).

2. Exact Shape Formats (Extremely Tested)

- **Oval:** Hardware Terminal Control (START/STOP).
- **Parallelogram:** I/O Bus Control (Scan Keyboard Input / Display Print Output).
- **Rectangle:** System Processing Engine (Formulas, assignments, array shifts).
- **Diamond:** Mathematical Boolean Splitter (Is Val > 0?). Evaluates strictly mapping exactly to TRUE branch and FALSE branch.

3. Three Core Logic Pathways

- **Sequence:** Code runs flawlessly line by line from top to bottom.
- **Selection (Decision):** Code visually hits a diamond and permanently splits directions based on a boolean math condition.
- **Repetition (Looping):** Code hits a diamond and mathematically cycles backwards up the execution line until the target counter threshold shatters.

Unit 1: PYQ Exam Focus

In GTU examinations, Unit 1 rigidly evaluates pure geometric logic mapping. Ensure absolute mastery over the following historical vectors:

- **3-Mark Essentials:**
  - Explicitly drawing four standard Flowchart shape symbols and defining

their purpose.

- Highlighting the precise difference between Algorithm and Flowchart.
- Writing tight Pseudocode for checking Positive/Negative or finding the Smallest of two numbers.

- **4-Mark Flowcharts/Algorithms:**
  - **Simple Interest Algorithm/Flowchart** (Formulas demanded: $I = (P * R * N)/100$). Highly repeated.
  - Max of three variables ($A > B$ AND $A > C$).
  - Even / Odd detection algorithm.
- **7-Mark Advanced Flowcharts:**
  - Complex looping logic: Design flowchart tracking the **sum of first 20 even natural numbers**.
  - Creating tracking variables and incrementing precisely.

CHAPTER 2
Introduction to Python

2.1 Overview and Features of Python

Python is a dynamically-typed, high-level, interpreted programming language created by **Guido van Rossum** and officially released in 1991. Its design philosophy emphasizes uncompromised code readability through the rigid use of significant indentation.

2.1.1 Key Features of Python

- **High-Level & Readable:** Extremely close to plain English. Syntactical noise (like curly braces {} and semicolons ;) is entirely eliminated.
- **Interpreted:** Code is executed line-by-line automatically by the Python Interpreter. It does not require a manual compilation step (unlike C++ or Java).
- **Dynamically Typed:** There is no need to declare the data type of a variable mathematically before using it. The type is assigned dynamically at runtime.
- **Open Source & Free:** Entirely free for commercial and academic use, backed by a massive global community.
- **Vast Standard Library:** Python strictly follows a "Batteries Included" philosophy, offering massive built-in libraries for math, random numbers, web networking, and data manipulation.

2.1.2 Industrial Applications of Python

- **Web Development:** Powering massive server backends using frameworks like Django and Flask (e.g., Instagram, Spotify).
- **Data Science & Machine Learning:** The undisputed king of AI engineering utilizing massive computational libraries like Pandas, NumPy, and TensorFlow.
- **Automation & Scripting:** Automating redundant operating system tasks via ultra-fast scripts.
- **Game Development & GUI:** Building visual interactive frameworks via PyGame and Tkinter.

2.2 Variables and Identifiers

Identifier

An **Identifier** is a user-defined name given to program elements like variables, functions, classes, or modules. Identifiers must follow incredibly strict grammatical rules.

Rules for Naming Identifiers in Python:

1. Must begin strictly with an alphabet letter (A-Z or a-z) or an underscore (_).
2. Cannot start with a pure number (e.g., `1variable` is illegal).
3. Only alphanumeric characters and underscores are permitted. No special symbols ($, %, @) are allowed.
4. Identifiers are **case-sensitive** (`Age` is mathematically entirely different from `age`).
5. Reserved keywords (like `if`, `else`, `import`) absolutely cannot be used.

2.3 Core Data Types

Because Python is dynamically typed, the runtime engine automatically detects the variable type based purely on the injected value.

Type Category	Keyword	Description & Example
Numeric	`int`	Whole numbers without fractions. Example: `5, -99`
	`float`	Floating-point decimal numbers. Example: `3.14, -0.001`
Text Sequence	`str`	Text encapsulated in single, double, or triple quotes. Example: `"Hello"`, `'Python'`
Boolean	`bool`	Strict binary Truth values. Holds exactly `True` or `False` (must be capitalized).
Sequence	`list`	Ordered, highly mutable collection of heterogeneous items. Extensively used. Example: `[1, "Apple", 3.14]`

2.3.1 Type Conversion (Typecasting)

Type conversion is the process of physically transforming a variable from one data type to another.

- **Implicit Type Conversion:** Automatically performed unilaterally by the Python interpreter without user intervention (e.g., mixing an `int` and a `float` yields a `float` to prevent data loss).
- **Explicit Type Conversion (Typecasting):** Manually forced conversion using built-in programmatic tools like `int()`, `float()`, `str()`.

Demonstrating Explicit Typecasting

```
x = 10.99          # This is fundamentally a float
y = int(x)         # Explicitly forces float to drop
    decimals

print(y)           # Output: 10
print(type(y))     # Output: <class 'int'>
```

2.4 Operators in Python

Operators are special symbols that execute structural arithmetic or logical computations upon assigned values (operands).

2.4.1 Arithmetic Operators

Used for standard mathematical actions.

Symbol	Operation Description	Example ($x = 10, y = 3$)
`+`	**Addition:** Adds two operands	`x + y = 13`
`-`	**Subtraction:** Subtracts the second operand from the first	`x - y = 7`
`*`	**Multiplication:** Multiplies both operands	`x * y = 30`
`/`	**Division:** Normal mathematical division (always strictly returns a `float`)	`x / y = 3.3333`
`//`	**Floor Division:** Eliminates decimals completely, returning the closest lowest integer	`x // y = 3`
`%`	**Modulus:** Returns purely the remainder of the division	`x % y = 1`
`**`	**Exponentiation:** Resolves exactly as x to the power of y (x^y)	`x ** y = 1000`

2.4.2 Relational (Comparison) Operators

Compares two distinct values mathematically and unconditionally returns a strict Boolean (`True` / `False`). Highly tested in PYQs.

Symbol	Operation Description	Example ($x = 5, y = 10$)
`==`	**Equal To:** True if values are strictly identical	`(x == y)` is False
`!=`	**Not Equal To:** True if values differ unconditionally	`(x != y)` is True
`>`	**Greater Than**	`(x > y)` is False
`<`	**Less Than**	`(x < y)` is True
`>=`	**Greater Than or Equal**	`(x >= 5)` is True
`<=`	**Less Than or Equal**	`(x <= 1)` is False

2.4.3 Logical Operators

Used to connect heavy mathematical conditions together.

- `and` : Returns `True` **only if BOTH** conditions perfectly evaluate to True.
- `or` : Returns `True` **if AT LEAST ONE** condition evaluates to True.
- `not` : Mathematically reverses the Boolean output (`not True` forces `False`).

2.4.4 Membership Operators

Critically used to verify if a sequence (like a String, List, or Tuple) structurally contains a precise value.

- `in` : Returns `True` if the specified value is physically present inside the sequence.
- `not in` : Returns `True` if the specified value is structurally absent.

Membership Operator Demo

```
name = "Python"
print("P" in name)       # Output: True (Capital P exists)
print("x" not in name)   # Output: True (x does not exist)
```

2.4.5 Assignment Operators

Used universally to assign computed values heavily back into variables.

- **Basic Assignment:** `x = 5`
- **Shorthand Operators:** `x += 3` (Mathematically identical to `x = x + 3`)
- **Shorthand Exponent:** `x **= 2` (Mathematically identical to `x = x ** 2`)

2.5 Essential Standard Input / Output

Python radically simplifies scanning hardware data via the `input()` and `print()` functions.

Input is always a String

The `input()` function universally snatches data exactly as a `<str>` String. If you are calculating sheer math (like Simple Interest), you must manually trap the input via explicit Typecasting (`int()` or `float()`).

2.5.1 Highly Tested Program: Calculating the Average of 3 Numbers

PYQ: Number Averaging Module

```
# Program to calculate the definitive average of three
    scanned numbers
print("Enter three distinct numbers:")

n1 = float(input("Enter first number: "))
n2 = float(input("Enter second number: "))
n3 = float(input("Enter third number: "))

total_sum = n1 + n2 + n3
average = total_sum / 3

print("Total Sum structurally equals:", total_sum)
print("Computed Average equals:", average)
```

Unit 2: Introduction to Python — Complete Cheatsheet

1. Core Mechanics & Identifiers

- **Features:** Interpreted, High-level, Dynamically Typed, Massive standard packages.
- **Identifier Rules:** Alphabet or Underscore origin strictly required. Zero special characters ($, @) permitted natively. Deeply case sensitive (`App` $\neq$ `app`).

2. Operator Hierarchy & Mapping

- **Floor Division (`//`):** Shreds off decimals. ($10 \,//\, 3 \rightarrow 3$).

- **Modulus (`%`):** Captures strictly the remainder. ($10\ \%3 \rightarrow 1$).
- **Exponentiation (`**`):** Resolves the powers equation. ($2 * * 3 \rightarrow 8$).
- **Relational (`==, !=`):** Generates raw `True` or `False` outputs based on direct magnitude comparison.
- **Membership (`in, not in`):** Highly optimized string/list searching sequence checker.

3. Mathematical Execution Framework

- **I/O System:** The standard `input()` defaults permanently to text String (`str`).
- **Typecasting Protocol:** Enforcing numbers demands `int(input())` or `float(input())`.

Unit 2: PYQ Exam Focus

In GTU examinations, Unit 2 rigidly checks syntax identification and direct mathematical application modeling. Ensure absolute mastery over:

- **3 & 4-Mark Theory Rules:**
  - Explaining the differences between Membership Operators (`in`) and Mathematical Operators.
  - Listing all fundamental Python applications (Web, AI, DS) and defining features.
- **Expression Output Tracing:**
  - Memorize the difference between `/` (Returns Float automatically) and `//` (Returns Integer Floor). Example: `5 / 2 = 2.5`, whereas `5 // 2 = 2`.
  - Evaluating `x % y` explicitly.
- **7-Mark Program Integrations:**
  - Creating a full framework program that intakes user data and calculates **Simple Interest**.
  - Demonstrating the three logical operators (`and`, `or`, `not`) with code samples.
  - Constructing the script to ingest three temperatures or variables and calculating the **structural average** or computing Fahrenheit to Celsius matrix shifts.

CHAPTER 3

Flow of Control

3.1 Introduction to Flow of Control

In a standard script, Python structurally executes code strictly line-by-line from top to bottom (Sequential Flow). However, practically all useful software requires the ability to intelligently skip lines (Selection) or repeat blocks of code heavily (Iteration). These redirection pathways define the **Flow of Control**.

3.2 Selection Statements (Decision Making)

Selection statements force the compiler to evaluate a mathematical Boolean condition (`True` / `False`). If the logic equates to `True`, a specific block is executed; otherwise, it is totally bypassed.

3.2.1 The `if` Statement

The simplest decision-making block. It tests a solitary condition.

`if` statement Structure

```
x = 10
if (x > 5):
    # This block executes strictly because 10 is greater
    than 5
    print("X is a large number")
```

3.2.2 The `if-else` Statement

Provides a definitive alternative pathway when the primary `if` condition explicitly registers as `False`.

PYQ: Odd or Even Number Detection

```
# Program to evaluate Odd vs Even states
num = int(input("Enter an integer: "))
```

```
3
4 if (num % 2 == 0):
5     print("The number is Even")
6 else:
7     print("The number is distinctly Odd")
```

3.2.3 The `if-elif-else` Ladder

Used exclusively to test multiple distinct conditions in a rigid sequence. As soon as one `elif` chain registers `True`, the remainder of the ladder is physically aborted.

PYQ: Positive, Negative, or Zero

```
1 n = float(input("Enter number: "))
2
3 if (n > 0):
4     print("Positive Number")
5 elif (n < 0):
6     print("Negative Number")
7 else:
8     print("Number is exactly Zero")
```

3.2.4 Nested `if` Statements

Injecting an entire `if` matrix directly inside another `if` block. Used to filter sequential prerequisite logic.

Nested `if` Demo

```
1 age = 25
2 has_license = True
3
4 if (age >= 18):
5     if (has_license == True):
6         print("You are fully authorized to drive.")
7     else:
8         print("You are old enough, but lack a valid license
  .")
```

3.3 Repetition (Loop Statements)

Looping continuously repeats a targeted block of code until a terminating condition fundamentally shatters.

3.3.1 The `while` Loop

The `while` loop aggressively evaluates a condition. If the condition is `True`, the locked code executes. It repeats unconditionally until the tracked condition flips to `False`.

`while` Loop Logic

```
count = 1
# Print sequence 1 through 5
while (count <= 5):
    print(count)
    count = count + 1    # Crucial step: Increment the
   tracker!
```

Infinite Loops

If you mathematically forget to increment the variable (e.g., omitting `count = count + 1`), the condition never becomes `False`. The loop locks into an **Infinite Loop**, instantly crashing the software.

3.3.2 The `for` Loop

Unlike `while` loops which operate on vague conditions, a Python `for` loop systematically iterates exactly over a fixed sequence (like a String, List, Tuple, or mathematical `range()`).

`range()` Function

The `range(start, stop, step)` function dynamically generates a sequence of numbers. It begins natively at `start` and absolutely terminates just *before* the `stop` boundary.

`for` Loop printing numbers 1 to 10

```
# Prints values 1 through 10. The upper limit (11) is
   excluded!
for i in range(1, 11):
    print(i)
```

3.3.3 Difference: `for` Loop vs `while` Loop

Metric	`for` Loop	`while` Loop
Usage Context	Heavily preferred when the absolute number of iterations is completely **known** beforehand.	Preferred when termination heavily relies on dynamic future conditions (unknown iteration limit).
Structure	Iterates uniformly directly over sequences/ranges natively.	Iterates exclusively over a singular boolean expression.

3.4 Jump Statements: Altering Execution Flow

Python provides structural jump commands to instantaneously modify a running loop's inherent trajectory.

- **`break`:** Violently terminates the entire loop infrastructure and shifts execution to the very next line totally outside the loop.
- **`continue`:** Skips only the *current iteration*. It aborts the remaining code chunk for the current cycle and violently forces the loop back to the top for the next sequence.

Jump Statement Demonstrations

```
# Demonstration of BREAK (Terminates at 3)
for i in range(1, 6):
    if (i == 3):
        break
    print(i)
# Output: 1, 2

# Demonstration of CONTINUE (Skips exactly 3)
for i in range(1, 6):
    if (i == 3):
        continue
    print(i)
# Output: 1, 2, 4, 5
```

3.5 Mastering Standard Loop Programs

The following algorithms heavily probe your mastery of loop boundaries, primarily testing numeric theory and visual matrix generation.

3.5.1 Program: Finding Prime Numbers

A number is Prime solely if it divides identically by exactly 1 and itself without remainder.

PYQ: Prime Number Verification

```python
num = int(input("Enter number: "))
is_prime = True

# We only need to check division up to the number itself
if (num > 1):
    for i in range(2, num):
        if (num % i == 0):
            is_prime = False
            break    # Divisor found, loop termination

if (is_prime and num > 1):
    print(num, "is profoundly Prime")
else:
    print(num, "is NOT Prime")
```

3.5.2 Pattern Printing Structures

Pattern printing critically relies on **Nested Loops**. The outer loop uniformly controls row generation, while the inner loop dictates column tracking (or characters deposited per row).

Pattern Type A: Star Triangles

Generating Right-Aligned Start Triangles

```python
# Target Pattern:
# *
# * *
# * * *

rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()  # Forces execution onto the next visual line
```

Pattern Type B: Number Cascades

Generating Number Cascades

```python
# Target Pattern:
# 1
```

```
# 1 2
# 1 2 3

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

Unit 3: Flow of Control — Complete Cheatsheet

1. Selection Core Hierarchy

- **`if`:** Baseline singular checkpoint.
- **`if-else`:** Dual strict pathway (True/False splits).
- **`elif`:** Chained ladder matrices to prevent deep nesting.
- **Nested `if`:** Condition explicitly locked inside an outer valid condition threshold.

2. Loop Execution Theory

- **`while` Loop:** Evaluates `True` state initially. Triggers until `False`. Heavily relies on tracking variables counting upward.
- **`for` Loop *in* `range(x,y)`:** The backbone of matrix traversal. Automatically pulls variables linearly strictly from `x` to mathematically `y-1`. (Example: `range(1,6)` strictly generates `[1,2,3,4,5]`).

3. Jump Execution Commands

- **`break`:** Utterly shatters the loop barrier. Execution escapes entirely regardless of remaining iteration limits.
- **`continue`:** Skips exactly the remaining current cycle code lines and immediately re-evaluates the top-level loop logic sequence.

Unit 3: PYQ Exam Focus

In GTU examinations, Unit 3 relentlessly tests visual flowchart execution mapping via Pattern Printing. Target areas include:

- **3 & 4-Mark Programming Basics:**
  - Distinguishing cleanly between `break` vs `continue` utilizing a sharp 3-line code example.
  - Defining nested loops and identifying their importance.
  - Comparing `for` loops explicitly against `while` loops.
- **7-Mark Program Designs:**
  - **Prime Number Programs:** Utilizing limits and `%` calculations alongside an `is_prime` flag tracker. Very highly repeatedly tested.
  - **Pattern Printing Engines:**
    * Inverted Star / Character matrices (e.g., Decreasing triangle `****` down to `*`).
    * Ascending numeric sequences (e.g., `1`, `1 2`, `1 2 3`).
    * Squared Output Patterns (e.g., `1`, `1 4`, `1 4 9 16` — accomplished easily via `print(j*j, end=" ")`).
    * Character increments (e.g., `A`, `AB`, `ABC` utilizing `chr(64 + j)` logic arrays).

CHAPTER 4

Functions and Modules

4.1 Introduction to Functions

As a Python program grows significantly larger, writing linear sequential code becomes fundamentally unmanageable. **Functions** solve this by breaking massive programs into smaller, isolated, and highly reusable blocks of execution logic.

Function

A **Function** is a named, structurally isolated block of reusable code engineered to perfectly perform exactly one specific task. Functions execute strictly **only when they are mathematically called**.

4.1.1 Benefits of Utilizing Functions

- **High Reusability:** Write the complex logic once, and call it infinitely across different program files.
- **Modularity:** Complex systems can be cleanly compartmentalized into highly distinct, isolated modules.
- **Maintenance:** Tracking logic bugs in a 10-line function is drastically simpler than surveying 10,000 lines of sequential script.

4.2 User-Defined Functions

Programmers absolutely must declare, structure, and invoke their own custom logic architectures using the `def` keyword.

4.2.1 Syntax and `return` Statement

Creating and Calling a Function

```
# 1. Defining the Function
def greet_student(name):
    # This block executes ONLY when called
```

```
    greeting = "Hello, " + name
    return greeting

# 2. Calling the Function from Main Execution
message = greet_student("Milav")
print(message)  # Output: Hello, Milav
```

- **def:** The mandatory keyword that initializes a strict function definition.
- **Arguments/Parameters:** Highly specialized variables injected continuously into the function parenthesis (`name`).
- **return:** Violently exits the function structure safely and instantaneously ejects the calculated data back to the calling line. If omitted, the function natively returns `None`.

4.3 Scope of Variables

The **Scope** absolutely defines the strict mathematical regions of a script where a variable is structurally accessible.

Scope Matrix: Local vs Global

Global Variables are declared radically outside all functions. They exist unconditionally everywhere.
Local Variables are declared strictly inside a specific function framework. They structurally annihilate the moment the function terminates.

PYQ: Variable Scope Execution Matrix

```
x = 50    # GLOBAL Variable. Accessible universally.

def test_scope():
    y = 100  # LOCAL Variable. Exists ONLY inside this
   function.
    print("Inside function, accessing Global x:", x)
    print("Inside function, accessing Local y:", y)

# Executing the function
test_scope()

# print(y) -> CRASH! y is strictly annihilated here.
```

4.4 Built-in Functions

Python pre-packages a robust core of functions requiring absolutely no `import` declarations.

Function	Mathematical Description	Example Execution
`abs(x)`	Returns the strictly positive Absolute Value.	`abs(-15.5)` → `15.5`
`max()`	Snipes the absolute largest item in an iterable matrix.	`max([1, 99, 5])` → `99`
`min()`	Snipes the absolute smallest item.	`min([1, 99, 5])` → `1`
`pow(x,y)`	Accurately resolves x to the power of y.	`pow(2, 3)` → `8`
`sum(x)`	Recursively adds all numerical elements inside a sequence.	`sum([10, 20])` → `30`
`divmod()`	Returns both the quotient and strictly the remainder as a structured tuple.	`divmod(10, 3)` → `(3, 1)`

4.5 Essential Python Modules

A **Module** is a highly structured file housing massive libraries of pre-built functions and variables. You absolutely must `import` modules to extract their architectural capabilities.

4.5.1 The `math` Module

For high-level rigid mathematical arrays.

`math` Implementation

```
import math

print(math.sqrt(64))       # Output: 8.0 (Square Root)
print(math.factorial(5))   # Output: 120 (5 * 4 * 3 * 2 * 1)
print(math.ceil(4.1))      # Output: 5 (Forces rounding UP)
print(math.floor(4.9))     # Output: 4 (Forces rounding DOWN
    )
print(math.pi)             # Output: 3.14159...
```

4.5.2 The `random` Module

For stochastic and cryptographically random generators.

`random` Implementation

```
import random
```

```
print(random.randint(1, 10))    # Targets random integer
    between 1 and 10
items = ["Apple", "Banana", "Cherry"]
print(random.choice(items))     # Snipes one random String
    strictly from list
```

4.5.3 The statistics Module

For deep mathematical predictive analytics.

statistics Implementation

```
import statistics

data = [10, 20, 30, 40, 50, 50]
print(statistics.mean(data))    # Evaluates Average (33.33)
print(statistics.median(data)) # Evaluates structural
    Center (35.0)
print(statistics.mode(data))    # Extracts the most highly
    frequent variable (50)
```

4.6 Mastering Complex Structural Algorithms (PYQ Focus)

These explicit highly complex programmatic designs rely heavily on custom mathematical functions and are rigorously tested for 7 marks.

4.6.1 Algorithm 1: Factorial using a Function

The factorial of N ($N!$) is precisely $1 \times 2 \times 3 \cdots \times N$.

Factorial Matrix

```
def calculate_factorial(n):
    if n < 0:
        return None
    fact = 1
    for i in range(1, n + 1):
        fact = fact * i
    return fact

# Execution Test
num = int(input("Enter number: "))
print("Factorial evaluates to:", calculate_factorial(num))
```

4.6.2 Algorithm 2: Fibonacci Sequence generation

Fibonacci explicitly adds the last two identical preceding numbers to generate strictly the next sequence block $(0, 1, 1, 2, 3, 5, 8 \dots)$.

Fibonacci Matrix

```
def print_fibonacci(limit):
    n1 = 0
    n2 = 1
    print(n1, n2, end=" ")

    # We heavily loop until limit is bridged
    for _ in range(2, limit):
        n3 = n1 + n2
        print(n3, end=" ")
        # Sequence Shift Mechanics
        n1 = n2
        n2 = n3

print_fibonacci(10) # Print strictly the first 10 structures
```

4.6.3 Algorithm 3: Armstrong Number logic

An Armstrong number mathematically equals the exact structural sum of its own specific digits raised physically to the power of the length of digits ($153 \to 1^3 + 5^3 + 3^3 = 153$).

Armstrong Verification Matrix

```
def is_armstrong(num):
    # Convert natively to String to easily trace string length
    num_str = str(num)
    length = len(num_str)

    sum_value = 0
    for digit in num_str:
        # Heavily extract the integer and explicitly power it
        sum_value += pow(int(digit), length)

    if sum_value == num:
        return True
    return False
```

```
14
15 print(is_armstrong(153)) # Output: True
```

Unit 4: Functions — Complete Cheatsheet

1. Function Mechanics

- **Definition Boundary:** Constructed unconditionally using the `def` trigger.
- **Parameters:** Strict variables explicitly accepted safely by the function header.
- **Return Protocol:** Ejects computed numerical states out violently back to the caller using `return`. Local variables shatter instantly afterward.

2. Library Mastery (Modules)

- **Built-in:** `max()`, `min()`, `abs()`, `pow()`, `sum()` — Execute immediately without any arbitrary initialization.
- **`math`:** Handles profound geometric values (`sqrt`, `pi`, `factorial`, `ceil`).
- **`random`:** Stochastic generators (`randint`, `choice`).
- **`statistics`:** High-level demographic aggregation (`mean`, `median`, `mode`).

3. Scope Mechanics

- **Global Status:** Lives unconditionally forever. Highly accessible everywhere.
- **Local Status:** Lives purely inside the executed `def` brackets. Instantly erased upon function conclusion unconditionally.

Unit 4: PYQ Exam Focus

In GTU examinations, Unit 4 heavily punishes students lacking explicit algorithmic programming architecture. Target areas include:

- **3 & 4-Mark Programming Basics:**
  - Distinguishing cleanly between Local and Global scope limits utilizing a sharp code block.
  - Listing mathematically standard python functions (`max, min`) or modules (`random`).
  - Defining user-defined execution rules.

- **7-Mark Program Designs (MUST MASTER):**
  - **Factorial Sequence:** Using `fact = fact * i` loops. Highly tested.
  - **Fibonacci Generator:** Shifting positional trackers (`n1 = n2`, `n2 = n3`).
  - **Armstrong & Palindrome Verification:** Master the mathematical looping string transformations.
  - **Module Scripting:** Writing code explicitly calling 3 capabilities of `math` or `statistics`.

CHAPTER 5

Dictionary, List, Set, String and Tuple

5.1 Introduction to Data Structures

Primitive data types (like `int` or `float`) can only inherently hold a singular, isolated value. To architect massive software, we require structures capable of holding thousands of dynamically linked values simultaneously. Python provides 4 incredibly powerful core built-in Data Structures: **List**, **Tuple**, **Set**, and **Dictionary**, alongside highly advanced **String** manipulations.

5.2 Strings: Textual Arrays

A String (`str`) is a linear mathematical sequence of unicode characters. Strings are absolutely **immutable** (fundamentally unchangeable after physical creation).

String Operations and Indexing

```
text = "Python"

# Indexing (Extracting specific characters)
# Positional Index operates from 0 to N-1
print(text[0])    # Output: 'P'
print(text[-1])   # Output: 'n' (Negative Indexing snipes
    from the end)

# Traversing specifically via a For Loop
for char in text:
    print(char, end="-")
# Output: P-y-t-h-o-n-
```

5.2.1 Essential String Methods

- `len(s)`: Calculates total character length.
- `s.upper()`: Forces purely uppercase.
- **`s.count(char)`:** Calculates exact occurrences of a substring.

- `s.replace(old, new)`: Swaps out internal characters.

5.3 Lists: The Dynamic Array

The **List** is historically the most powerful, heavily utilized structure in Python. It is an ordered, highly **mutable** (changeable) collection of heterogeneous data.

List Initialization and Slicing

```
my_list = [10, "Apple", 3.14, True]

# Slicing extracts specifically a sub-list [start:stop]
subset = my_list[1:3]
print(subset)     # Output: ['Apple', 3.14]

# Mutability: You can directly overwrite internal data
my_list[0] = 99
print(my_list)    # Output: [99, 'Apple', 3.14, True]
```

5.3.1 Nested Lists

A Nested List is literally a structural list entirely contained directly inside another list, simulating multi-dimensional matrices ($M \times N$).

Nested List Traversal

```
matrix = [[1, 2], [3, 4], [5, 6]]

print(matrix[1])      # Access entire second row: [3, 4]
print(matrix[1][0])   # Access first element of second row:
   3
```

5.3.2 Highly Tested Base List Functions

- `list.append(x)`: Injects `x` strictly at the absolute end.
- `list.insert(idx, x)`: Forcefully shoves `x` exactly at the `idx` position.
- `list.pop()`: Ejects and drastically deletes the absolute final element.
- `list.sort()`: Mathematically orders numeric files natively.
- `list.reverse()`: Inverts the structural layout sequence.

5.3.3 PYQ Priority Algorithms: List Computations

1. Swapping Two Elements:

```python
L = [10, 20, 30, 40]
# Python allows absolute parallel assignment mapping
L[0], L[3] = L[3], L[0]
print(L)  # Output: [40, 20, 30, 10]
```

2. Sum and Multiplication of All Elements:

```python
num_list = [2, 4, 5]
total_sum = 0
total_mult = 1

for num in num_list:
    total_sum += num
    total_mult *= num

print("Sum:", total_sum)       # Output: 11
print("Product:", total_mult) # Output: 40
```

3. Finding the Largest/Smallest Element (Without built-in functions):

```python
data = [21, 6, 3, 18, 15]
smallest = data[0]
largest = data[0]

for x in data:
    if x < smallest:
        smallest = x
    if x > largest:
        largest = x

print("Smallest:", smallest)  # Output: 3
print("Largest:", largest)    # Output: 21
```

5.4 Tuples: The Immutable Array

A **Tuple** acts mathematically identically to a List, however, it is structurally **immutable**. Once born, its internal elements absolutely cannot be altered, appended to, or deleted.

Tuple Structure

```python
my_tuple = (10, 20, 30)

print(my_tuple[1]) # Accessing works perfectly (Output: 20)
# my_tuple[0] = 99 -> CRASH! Mutation is strictly illegal.
```

Metric	List []	Tuple ()
Mutability	Fully Mutable (Can append/delete dynamically).	Strictly Immutable (Locked completely at creation).
Speed/Memory	Extremely heavy, demands higher RAM overhead.	Extremely fast, heavily optimized by Python compilation.
Application	Used when data counts violently fluctuate.	Used for locked static databases (like GPS coordinate arrays).

5.5 Sets: Unordered Unique Math Matrices

A **Set** natively forces strict mathematical uniqueness. It completely ignores duplicate insertions and has absolutely no sequential numbering index (unordered).

Set Instantiation and Operation

```
# Automatic duplicate obliteration
my_set = {1, 2, 2, 3, 4, 4}
print(my_set)  # Output: {1, 2, 3, 4}

# Addition and Deletion
my_set.add(99)
my_set.remove(1)

# Advanced Set Operations
A = {1, 2, 3}
B = {3, 4, 5}
print(A.union(B))          # Output: {1, 2, 3, 4, 5}
print(A.intersection(B))   # Output: {3}
```

5.6 Dictionaries: The Hash-Map Architecture

Dictionaries map arbitrary unique **Keys** to specific corresponding **Values**. They are heavily used to mimic fast associative arrays or database rows.

Dictionary Setup and Alteration

```
student = {
    "roll_no": 101,
    "name": "Milav",
    "marks": 95
}

```

```python
# Accessing Values
print(student["name"])     # Output: Milav

# Modifying and Adding
student["marks"] = 99      # Overwrites existing key
student["city"] = "Surat" # Initializes radically new key

# Accessing Keys and Values globally
print(student.keys())      # Output: dict_keys(['roll_no', 'name', 'marks', 'city'])
print(student.values())    # Output: dict_values([101, 'Milav', 99, 'Surat'])
```

Unit 5: Data Structures — Complete Cheatsheet

1. The Four Pillars of Collections

- **List `[ ]`:** Ordered, Indexable, Mutable. (The Swiss Army Knife).
- **Tuple `( )`:** Ordered, Indexable, IMMUTABLE. (Read-only arrays).
- **Set `{ }`:** Unordered, **No Duplicates Allowed**. (Used for deep Intersection/Union math).
- **Dictionary `{ k:v }`:** Key-Value pairs. Keys must be strictly unique. Immensely fast lookup speeds.

2. Vital Indexing Commands

- **0-Based Logging:** `L[0]` accesses the prime target.
- **Negative Bounds:** `L[-1]` flawlessly retrieves the absolute last entry mechanically.
- **Slicing Limiters:** `L[start:stop]` strictly extracts up exactly to `stop - 1`.

3. Crucial Python Core Functions

- `list.append(item)` vs `list.insert(idx, item)`.
- `dict.get(key)` avoids severe program crashes perfectly if the key evaluates empty.
- `string.split(" ")` mathematically shatters a sentence into a pure List of literal words.

Unit 5: PYQ Exam Focus

In GTU examinations, Unit 5 holds a colossal 33% weighting (Highest Priority). Target exact precision on:

- **3 & 4-Mark Programming Capabilities:**
  - Distinguishing cleanly between List and Tuple (Must state Mutability laws).
  - Writing 4-line python code specifically to **swap elements** natively using `L[a], L[b] = L[b], L[a]`.
  - Explaining Nested Lists and indexing them via `matrix[x][y]`.
  - Extracting the smallest number in an array utilizing a `for` loop (e.g., `min_val`).
- **7-Mark Advanced Data Program Sequences:**
  - **Mathematical Array Sweeps:** Formulating `total_sum` and `total_mult` across list components natively without dependencies.
  - **String Extraction Mechanics:** Building dictionaries mapping frequency counts of vocal characters dynamically inside a sequence block.
  - Defining set intersections (`&`) and dictionary indexing mechanics.

CHAPTER A
Practical Exercises Manual

This appendix provides the strict, laboratory-ready Python code implementations for the prescribed Gujarat Technological University (GTU) Practical Outcomes (PrOs) for DI02032011. Students should actively retype these codes natively in `IDLE` or `VS Code` to verify the structural outputs.

A.1 Unit 2: Basic IO and Operators

Practical 3: Print Personal Identity Matrix

Aim: Write a Program to print your name, mobile number, and date of birth.

Example Python Code

```
name = "Milav Dabgar"
mobile = "+91-9876543210"
dob = "01-Jan-2005"

print("----- IDENTITY MATRIX -----")
print("Name   :", name)
print("Mobile :", mobile)
print("D.O.B  :", dob)
```

Terminal Output

```
--- IDENTITY MATRIX ---
Name :  Milav Dabgar
Mobile :  +91-9876543210
D.O.B : 01-Jan-2005
```

Practical 6: Simple and Compound Interest Engine

Aim: Develop a Program that can calculate simple interest and compound interest on given data.

Example Python Code

```
p = float(input("Enter Principal Amount: "))
r = float(input("Enter Rate of Interest: "))
n = float(input("Enter Time (Years): "))

# Simple Interest Formula
si = (p * r * n) / 100

# Compound Interest Formula [A = P(1 + R/100)^N]
amount = p * ((1 + (r / 100)) ** n)
ci = amount - p

print("Calculated Simple Interest:   ", si)
print("Calculated Compound Interest: ", ci)
```

Practical 7: Thermal Matrix Conversion (F to C)

Aim: Write a Program to convert temperature from Fahrenheit to Celsius unit using eq: $C = (F - 32)/1.8$.

Example Python Code

```
fahrenheit = float(input("Enter Temperature in Fahrenheit:
    "))

# Direct Formula Application
celsius = (fahrenheit - 32) / 1.8

print("Temperature in Celsius evaluates to:", round(celsius
    , 2))
```

A.2 Unit 3: Flow of Control Protocols

Practical 8: Odd vs Even Detector

Aim: Identify whether the scanned number is even or odd and print an appropriate message.

Example Python Code

```
num = int(input("Enter an integer limit: "))

if (num % 2 == 0):
    print(num, "is fundamentally Even.")
else:
```

```
    print(num, "is unconditionally Odd.")
```

Practical 13: Mathematical Prime Locator

Aim: Write a Program to show whether the entered number is prime or not.

Example Python Code

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

if n == 1:
    print(n, "is neither prime nor composite")
elif n > 1:
    # Scan dynamically for any valid divisor
    for i in range(2, n):
        if (n % i == 0):
            flag = True # Devastating divisor located
            break

    if flag:
        print(n, "is NOT a prime number")
    else:
        print(n, "is a PRIME number")
```

Practical 14: Arbitrary Pattern Printing

Aim: Write a Program to display the descending structural Star Pattern.

Example Python Code

```
# Target:
# ****
# ***
# **
# *

rows = 4

# Outer loop drastically counts completely backwards!
for i in range(rows, 0, -1):
    for j in range(0, i):
        print("*", end="")
    print()
```

A.3 Unit 4: Functions Architecture

Practical 15: Fibonacci Generator via Functions

Aim: Create a User-defined function to print the Fibonacci series up to N numbers.

Example Python Code

```
def fibonacci_matrix(n):
    n1 = 0
    n2 = 1
    print("Fibonacci Sequence:", n1, n2, end=" ")

    for i in range(2, n):
        n3 = n1 + n2
        print(n3, end=" ")
        # Sequence Replacement Protocol
        n1 = n2
        n2 = n3

fibonacci_matrix(8)
```

Practical 18: Armstrong Number Function

Aim: Write a Program that determines whether a given number is an Armstrong number.

Example Python Code

```
def is_armstrong(number):
    total = 0
    temp = number
    # Power calculates length
    power = len(str(number))

    while temp > 0:
        digit = temp % 10
        total += digit ** power
        temp //= 10

    if number == total:
        print(number, "is an Armstrong Number")
    else:
        print(number, "is NOT an Armstrong Number")

is_armstrong(153)
```

A.4 Unit 5: Advanced Data Structures

Practical 23: Swapping Elements within a List

Aim: Develop Programs to swap two given elements natively in a list.

Example Python Code

```
def swap_positions(data_list, pos1, pos2):
    print("Original Matrix:", data_list)

    # Python Unpacking Assignment Sequence
    data_list[pos1], data_list[pos2] = data_list[pos2],
   data_list[pos1]

    print("Swapped Matrix:", data_list)

my_list = [10, 20, 30, 40, 50]
swap_positions(my_list, 1, 3)
# Evaluates successfully swapping 20 and 40
```

Practical 29: Dictionary Threshold Filtering

Aim: Create a Dictionary with roll numbers, names, and marks of students. Display names of students who scored strictly above 75.

Example Python Code

```
# Dictionary Initialization
students = {
    101: {"name": "Milav", "marks": 96},
    102: {"name": "Rohan", "marks": 64},
    103: {"name": "Sanya", "marks": 88},
    104: {"name": "Arjun", "marks": 72}
}

print("Students scoring strictly ABOVE 75:")

# Core Iteration Sequence
for roll, data in students.items():
    if data["marks"] > 75:
        print(f"[{roll}] {data['name']} scored {data['marks
   ']}")
```

Terminal Output

```
Students scoring strictly ABOVE 75:
[101] Milav scored 96
[103] Sanya scored 88
```