📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 1 — Introduction to Problem-Solving

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Understand what **problem-solving** means in computer science.
- Identify the different stages involved in solving a problem.
- Understand the role of a computer in problem-solving.
- Differentiate between a **problem**, **solution**, and **program**.
- Understand why systematic problem-solving is important.

---

1. 🧩 What is Problem-Solving?

Problem-solving is the process of finding a suitable solution to a given problem.

In computer science, problem-solving means:

> **Understanding a problem, developing a logical solution, and implementing that solution using a computer.**

Simple Example

Suppose we want to calculate the area of a rectangle.

We need to:

1. Get the length.
2. Get the breadth.
3. Calculate the area.
4. Display the result.

The formula is:

Area = Length × Breadth

This complete process of finding and implementing the solution is called **problem-solving**.

---

2. 💻 Problem-Solving in Computer Science

A computer cannot solve a problem simply because we give it a question.

We must provide **clear and precise instructions**.

For example, if we want a computer to calculate the average of three numbers:

Input

Three numbers:

```
10, 20, 30
```

Processing

Average = (10 + 20 + 30) / 3

Output

```
Average = 20
```

Therefore, a computer-based solution generally involves:

Input → Processing → Output

Example

```
       INPUT
         ↓
   Three Numbers
         ↓
      PROCESS
         ↓
 Calculate Average
         ↓
      OUTPUT
         ↓
        20
```

---

3. 🔍 Understanding a Problem

Before writing any program, we must first understand the problem properly.

A problem should be analyzed by identifying:

1. Input

What information is provided to the program?

2. Processing

What calculations or operations need to be performed?

3. Output

What result is expected from the program?

Example

Problem: Find the area of a circle.

Input: Radius

Processing:

Area = $\pi \times r^2$

Output: Area of the circle

⭐ Remember

> **Input → What is given? Processing → What needs to be done? Output → What is required?**

---

4. 🧠 What is a Solution?

A **solution** is a systematic method or set of steps used to solve a problem.

A good solution should be:

- **Correct** — produces the required result.
- **Clear** — easy to understand.
- **Finite** — should eventually terminate.
- **Efficient** — should not waste unnecessary time or resources.
- **Practical** — should be possible to implement.

Example

Problem:

Find the larger of two numbers.

A possible solution is:

1. Input two numbers.
2. Compare the numbers.
3. If the first number is greater, display it.
4. Otherwise, display the second number.

This is a systematic solution to the problem.

---

5. ⚙️ Problem-Solving Using a Computer

A computer-based problem-solving process can be represented as:

```
        PROBLEM
           ↓
   Understand the Problem
           ↓
    Analyze the Problem
           ↓
   Develop a Solution
           ↓
       Algorithm
           ↓
   Flowchart / Pseudocode
           ↓
        Coding
           ↓
   Testing & Debugging
           ↓
        FINAL SOLUTION
```

Each stage is important.

If the problem is not understood correctly, even a perfectly written program may produce the wrong result.

---

6. 🌍 Real-Life Example of Problem-Solving

Suppose a student wants to determine whether they have passed an examination.

Problem

Determine whether a student has passed based on marks.

Input

Student's marks.

Processing

If marks are **40 or above**, the student passes.

Otherwise, the student fails.

Output

```
Pass
```
or

```
Fail
```

Logical Solution

```
Input marks
     ↓
Are marks ≥ 40?
   ↙       ↘
 YES       NO
  ↓         ↓
PASS      FAIL
```
This same logical thinking can later be converted into a **Python program**.

---

7. 💡 Why is Problem-Solving Important?

Problem-solving is one of the most important skills in computer science because programming is not only about writing code.

A programmer must first determine:

- What is the actual problem?
- What information is available?
- What result is required?
- What steps are needed?
- Which method is most efficient?
- How can the solution be tested?

⭐ Important Point

> **Programming is the implementation of a solution; problem-solving comes before programming.**

A student who understands the logic of a problem can usually write the program more easily.

---

8. 📝 Quick Revision & Practice

Quick Revision

Problem-Solving: The process of finding and implementing a suitable solution to a problem.

Input: Data provided to the computer.

Processing: Operations performed on the input.

Output: The final result produced by the computer.

Solution: A systematic method of solving a problem.

Practice Questions

Q1. What is problem-solving?

Q2. Why is problem-solving important in computer science?

Q3. Explain the terms:

a) Input b) Processing c) Output

Q4. Identify the input, processing and output for:

> Write a program to calculate the simple interest.

Q5. Identify the input, processing and output for:

> Write a program to find the average of five numbers.

Q6. Why should a programmer understand a problem before writing code?

Q7. Differentiate between a **problem**, **solution**, and **program**.

⭐ Key Takeaway

> **First understand the problem, then develop the solution, and only after that write the program.**

🔜 Next Lecture

Lecture 2 — Steps for Problem-Solving

We will learn how to systematically move from **problem analysis → solution development → coding → testing and debugging**.

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 2 — Steps for Problem-Solving & Analyzing a Problem

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Understand the major steps involved in problem-solving.
- Understand how to analyze a problem before writing a program.
- Identify **input, processing, and output**.
- Identify the requirements and constraints of a problem.
- Convert a problem statement into a clear programming task.

---

1. 🔄 Steps for Problem-Solving

Solving a programming problem should be done systematically rather than immediately writing code.

The major steps are:

```
Problem
   ↓
Analyze the Problem
   ↓
Develop the Solution
   ↓
Write the Code
   ↓
Test the Program
   ↓
Debug the Program
   ↓
Final Solution
```

The main steps are:

1. **Analyzing the Problem**
2. **Developing an Algorithm**
3. **Coding**

4. **Testing and Debugging**

---

2. 🔍 Step 1 — Analyzing the Problem

Problem analysis means understanding exactly what the problem is asking before attempting to solve it.

Before writing code, we should answer:

- What information is given?
- What information is required?
- What calculations or operations are needed?
- Are there any conditions or restrictions?
- What should the final output look like?

⭐ Remember

> **Don't start coding until you understand the problem.**

---

3. 📥 Identify the Input

The **input** is the data that is provided to the program.

Example

Problem:

> Write a program to calculate the area of a rectangle.

The required inputs are:

- Length
- Breadth

```
Input:
Length = 10
Breadth = 5
```

---

4. ⚙️ Identify the Processing

Processing refers to the operations or calculations performed on the input to obtain the required result.

For the rectangle example:

```
Area = Length × Breadth
```

If:

```
Length = 10
Breadth = 5
```

Then:

```
Area = 10 × 5
     = 50
```

Therefore:

Processing = Multiplication of length and breadth

---

5. 📤 Identify the Output

The **output** is the information or result produced by the program.

For the rectangle example:

```
Output:
Area = 50
```

Therefore:

```
INPUT → PROCESSING → OUTPUT
Length, Breadth
      ↓
Length × Breadth
      ↓
     Area
```

---

6. 🧩 Example — Calculate Simple Interest

Problem

Write a program to calculate the simple interest.

Step 1 — Identify Input

We need:

- Principal (`P`)

- Rate (`R`)
- Time (`T`)

Step 2 — Identify Processing

Formula:

SI = (P × R × T) / 100

Step 3 — Identify Output

The required output is:

Simple Interest

Example

```
P = 5000
R = 10
T = 2
```

Processing:

```
SI = (5000 × 10 × 2) / 100
   = 1000
```

Output:

```
Simple Interest = 1000
```

---

7. 📋 Problem Analysis Table

A useful way to analyze a problem is to create an **IPO table**.

IPO stands for:

I → Input

P → Processing

O → Output

Example: Area of a Circle

Input	Processing	Output
Radius	Area = π × r^2	Area

Example: Average of Three Numbers

Input	Processing	Output
Three numbers	Sum ÷ 3	Average

Example: Largest of Two Numbers

Input	Processing	Output
Two numbers	Compare the numbers	Larger number

---

8. 🧠 Identify the Requirements

While analyzing a problem, we should identify its **requirements**.

Requirements describe what the program must do.

Example

Problem:

> Write a program to calculate the percentage of a student from marks obtained in five subjects.

Requirements:

- Accept marks of five subjects.
- Calculate total marks.
- Calculate percentage.
- Display total and percentage.

Formula

Percentage = (Total Marks / Maximum Marks) × 100

---

9. ⚠ Identify Constraints

A **constraint** is a restriction or condition that must be followed while solving a problem.

Example

Problem:

> Enter marks between 0 and 100 and calculate the grade.

Constraint:

```
0 ≤ marks ≤ 100
```

Another example:

> Find the square root of a number.

Constraint:

```
Number should not be negative
```

Understanding constraints helps us prevent invalid input and incorrect results.

---

10. 📝 Example — Analyze a Complete Problem

Problem

> Write a program to determine whether a number is even or odd.

Analysis

Input:

One integer.

Processing:

Check the remainder when the number is divided by 2.

```
Number % 2
```

If the remainder is `0`:

```
Even
```

Otherwise:

```
Odd
```

Output:

Either:

```
Even
```

or:

```
Odd
```

IPO Representation

Input	Processing	Output
One integer	Check `number % 2`	Even/Odd

---

11. 🔎 From Problem Statement to Solution

A programming problem is usually given in the form of a **problem statement**.

We need to convert it into smaller, understandable parts.

Example

> Write a program to calculate the total and average marks of three subjects.

Break the problem into:

```
1. Input marks of Subject 1
2. Input marks of Subject 2
3. Input marks of Subject 3
4. Calculate total
5. Calculate average
6. Display total
7. Display average
```

This process makes the problem easier to solve.

---

12. 🧱 Breaking a Complex Problem into Smaller Problems

A large problem can be divided into smaller and simpler tasks.

For example:

Problem

> Develop a student result program.

Instead of solving everything at once:

```
Student Result
      ↓
  ┌──┬──┬──┐
  ↓  ↓  ↓  ↓
Input  Total  %  Grade
Marks
```

We can solve each part separately.

This idea of breaking a problem into smaller parts is called **Decomposition**.

> Decomposition will be discussed in greater detail later in this chapter.

---

13. 🧮 Example — Student Result

Problem

A student has marks in three subjects. Calculate total, percentage and display the result.

Analysis

Input:

```
Marks of 3 subjects
```

Processing:

```
Total = M1 + M2 + M3

Percentage = Total / 300 × 100
```

Output:

```
Total
Percentage
```

Program Flow

```
      START
        ↓
  Input M1, M2, M3
        ↓
   Calculate Total
        ↓
Calculate Percentage
        ↓
   Display Results
        ↓
       STOP
```

---

14. 🚫 Why Should We Not Start Coding Immediately?

Starting to code without analyzing the problem can lead to:

- Wrong formulas
- Missing inputs
- Incorrect conditions
- Unnecessary code
- Logical errors
- Difficulty in debugging

Example

Suppose the question asks:

> Calculate the area of a rectangle.

A student writes:

```
area = length + breadth
```

The program may run successfully, but the answer is incorrect.

The correct formula is:

```
area = length * breadth
```

This is why **problem analysis must come before coding**.

---

15. ⭐ Problem Analysis Checklist

Before writing a program, ask:

✅ Input

What data do I have to take?

✅ Output

What result do I have to display?

✅ Processing

What formula, calculation or operation is required?

✅ Conditions

Is there any `if`, `elif` or `else` requirement?

✅ Repetition

Do I need a loop?

✅ Constraints

Are there restrictions on the input?

---

16. 🧠 Practice — Analyze the Following Problems

For each problem, identify:

Input → Processing → Output

Q1.

Write a program to calculate the area of a triangle.

Q2.

Write a program to calculate the circumference of a circle.

Q3.

Write a program to find the largest of two numbers.

Q4.

Write a program to calculate the average of five numbers.

Q5.

Write a program to determine whether a number is positive, negative or zero.

Q6.

Write a program to calculate the total bill amount after adding 18% GST.

🎯 Quick Revision

Problem-Solving

The process of finding and implementing a suitable solution to a problem.

Problem Analysis

Understanding the problem and identifying what needs to be done before writing code.

Input

Data given to the program.

Processing

Operations performed on the input.

Output

Result produced by the program.

Constraint

A restriction or condition that must be followed.

IPO

Input → Processing → Output

⭐ KEY TAKEAWAY

> **A good programmer does not immediately start writing code.**

The correct approach is:

```
UNDERSTAND
    ↓
ANALYZE
    ↓
PLAN
    ↓
CODE
    ↓
TEST
    ↓
DEBUG
```

🔜 Next Lecture

Lecture 3 — Developing an Algorithm

We will learn:

- What is an algorithm?
- Characteristics of a good algorithm
- How to write an algorithm
- Step-by-step algorithm development
- Examples of algorithms
- Converting a problem into an algorithm

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 3 — Developing an Algorithm

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Define an **algorithm**.
- Understand the need for an algorithm.
- Identify the characteristics of a good algorithm.
- Write algorithms using simple and clear steps.
- Develop an algorithm from a given problem.
- Understand the relationship between a problem, algorithm and program.

---

1. 🧠 What is an Algorithm?

An **algorithm** is a finite sequence of clear and well-defined steps used to solve a problem or perform a task.

In simple words:

> **An algorithm is a step-by-step procedure for solving a problem.**

Example

Problem:

> Find the sum of two numbers.

Algorithm:

```
1. Start
2. Input two numbers A and B
3. Calculate Sum = A + B
4. Display Sum
5. Stop
```

The above steps provide a clear method for solving the problem.

---

2. 💡 Why Do We Need an Algorithm?

Before writing a program, it is useful to first develop an algorithm.

An algorithm helps us to:

- Understand the problem clearly.
- Plan the solution.
- Identify the required steps.
- Find logical errors before coding.
- Make coding easier.
- Communicate the solution to others.
- Convert the solution into a program.

⭐ Remember

> **Algorithm comes before coding.**

```
Problem
   ↓
Algorithm
   ↓
Program
   ↓
Testing
```

---

3. 🖇️ Algorithm vs Program

An **algorithm** describes the solution using logical steps.

A **program** implements those steps using a programming language such as Python.

Example

Problem

Find the area of a rectangle.

Algorithm

```
1. Start
2. Input length and breadth
3. Calculate Area = length × breadth
4. Display Area
5. Stop
```

Python Program

```python
length = float(input("Enter length: "))
breadth = float(input("Enter breadth: "))

area = length * breadth

print("Area =", area)
```

Difference

Algorithm	Program
Step-by-step solution	Implementation of the solution
Usually written in simple language	Written in a programming language
Does not depend on a specific programming language	Depends on programming language
Easier to understand	Executed by the computer

---

4. ⭐ Characteristics of a Good Algorithm

A good algorithm should have the following characteristics:

1. Input

An algorithm may accept **zero or more inputs**.

Example:

```
Input: Two numbers
```

2. Output

An algorithm should produce at least one result/output.

Example:

```
Output: Sum of two numbers
```

3. Definiteness

Every step should be **clear and unambiguous**.

❌ Incorrect:

```
Add some numbers
```

✅ Correct:

```
Add A and B and store the result in Sum
```

4. Finiteness

An algorithm must terminate after a **finite number of steps**.

It should not continue forever.

5. Effectiveness

Every step should be simple enough to be performed or executed in a practical manner.

---

5. 📝 Steps for Developing an Algorithm

To develop an algorithm, follow these steps:

```
Understand the Problem
        ↓
Identify Input
        ↓
Identify Output
        ↓
Determine Processing
        ↓
Write Step-by-Step Instructions
        ↓
Check the Algorithm
```

Example Problem

> Calculate the average of three numbers.

Step 1 — Input

Three numbers:

```
A, B, C
```

Step 2 — Processing

```
Sum = A + B + C
Average = Sum / 3
```

Step 3 — Output

```
Average
```

Step 4 — Algorithm

```
1. Start
2. Input A, B and C
3. Calculate Sum = A + B + C
4. Calculate Average = Sum / 3
5. Display Average
6. Stop
```

---

6. 🧮 Example — Calculate Area of a Circle

Problem

Write an algorithm to calculate the area of a circle.

Formula

Area = π × r²

where `r` is the radius.

Algorithm

```
1. Start
2. Input radius R
3. Calculate Area = π × R²
4. Display Area
5. Stop
```

Python Implementation

```
r = float(input("Enter radius: "))

area = 3.14 * r ** 2

print("Area =", area)
```

---

7. 🔢 Example — Check Even or Odd

Problem

Write an algorithm to determine whether a number is even or odd.

Logic

If the remainder after dividing by 2 is zero:

`Number % 2 = 0`
the number is **Even**.

Otherwise, it is **Odd**.

Algorithm

```
1. Start
2. Input number N
3. If N % 2 = 0, then
       Display "Even"
   Otherwise
       Display "Odd"
4. Stop
```

Python Implementation

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

if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

---

8. 🔍 Example — Find the Largest of Two Numbers

Problem

Write an algorithm to find the larger of two numbers.

Algorithm

```
1. Start
2. Input A and B
3. If A > B, then
       Display A as the largest
4. Otherwise, if B > A, then
       Display B as the largest
5. Otherwise
       Display "Both are equal"
6. Stop
```

Python Implementation

```python
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

if a > b:
    print("Largest =", a)
elif b > a:
    print("Largest =", b)
else:
    print("Both numbers are equal")
```

---

9. 🔢 Example — Sum of Digits

Problem

Write an algorithm to find the sum of digits of a positive integer.

Example

```
Input: 345

3 + 4 + 5 = 12
```

Logic

We can extract the last digit using:

`N % 10`

and remove the last digit using:

`N // 10`

Algorithm

```
1. Start
2. Input number N
3. Set Sum = 0
4. Repeat while N > 0:
      Digit = N % 10
      Sum = Sum + Digit
      N = N // 10
5. Display Sum
6. Stop
```

Python Implementation

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

sum_digits = 0

while n > 0:
    digit = n % 10
    sum_digits += digit
    n //= 10

print("Sum of digits =", sum_digits)
```

---

10. 🔄 Algorithm with Repetition

Some problems require the same operation to be performed repeatedly.

For example:

> Print numbers from 1 to 10.

Algorithm

```
1. Start
2. Set I = 1
3. Repeat while I <= 10:
      Display I
      Increase I by 1
4. Stop
```

Python

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

This shows that an algorithm can contain:

- Sequence
- Selection/condition
- Repetition/loop

These are the basic building blocks of programming logic.

---

11. 🧩 Three Basic Structures in Algorithms

Most algorithms can be constructed using three basic control structures.

1. Sequence

Instructions are executed one after another.

```
Input A
Input B
Calculate Sum
Display Sum
```

---

2. Selection

A condition determines which instruction should be executed.

```
IF marks >= 40
    Display "Pass"
ELSE
    Display "Fail"
```

---

3. Iteration

A set of instructions is repeated.

```
Repeat 10 times
    Display number
```

⭐ Remember

```
Sequence → Step by step
Selection → Decision
Iteration → Repetition
```

---

12. 📊 Algorithm Development Example — Student Grade

Problem

Write an algorithm to display the grade of a student based on marks.

Conditions

```
90 or above → A
75–89       → B
60–74       → C
40–59       → D
Below 40    → F
```

Algorithm

```
1. Start
2. Input Marks
3. If Marks >= 90:
       Display "A"
4. Else if Marks >= 75:
       Display "B"
5. Else if Marks >= 60:
       Display "C"
6. Else if Marks >= 40:
       Display "D"
7. Else:
       Display "F"
8. Stop
```

Python Implementation

```python
marks = float(input("Enter marks: "))

if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 60:
    print("Grade C")
elif marks >= 40:
    print("Grade D")
else:
    print("Grade F")
```

---

13. ❌ Common Mistakes While Writing Algorithms

Mistake 1 — Unclear Steps

❌

```
Calculate result
```

✅

```
Calculate Average = Total / Number of Subjects
```

Mistake 2 — Missing Input

The algorithm should clearly mention what data is required.

Mistake 3 — Missing Output

The final result should be clearly specified.

Mistake 4 — Infinite Repetition

A loop should have a condition that eventually becomes false.

Mistake 5 — Wrong Order of Steps

The steps should follow the actual sequence required to solve the problem.

---

14. 🧠 Practice Questions

Write an algorithm for each of the following:

Q1.

Calculate the area and perimeter of a rectangle.

Q2.

Calculate the simple interest.

Q3.

Check whether a number is positive, negative or zero.

Q4.

Find the largest of three numbers.

Q5.

Calculate the factorial of a number.

Q6.

Print the multiplication table of a given number.

Q7.

Calculate the sum of the first `N` natural numbers.

Q8.

Check whether a number is a palindrome.

🎯 Quick Revision

Algorithm

A finite sequence of clear and well-defined steps used to solve a problem.

Five Important Characteristics

Input → Output → Definiteness → Finiteness → Effectiveness

Three Basic Structures

Sequence → Selection → Iteration

Development Process

```
Problem
   ↓
Analyze
   ↓
Identify Input/Output
   ↓
Develop Algorithm
   ↓
Write Program
```

```
   ↓
Test & Debug
```

⭐ KEY TAKEAWAY

> **An algorithm is the blueprint of a program.**

Just as a building is constructed using a blueprint, a program can be developed using an algorithm.

```
PROBLEM
   ↓
ALGORITHM
   ↓
CODE
   ↓
TEST
   ↓
SOLUTION
```

🔜 Next Lecture

Lecture 4 — Coding, Testing, Debugging & Need for an Algorithm

We will learn:

- Converting an algorithm into code
- What is coding?
- Testing a program
- Debugging
- Types of errors
- Why algorithms are important before coding

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 4 — Coding, Testing, Debugging & Need for an Algorithm

---

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Understand the meaning of **coding**.
- Convert an algorithm into a Python program.
- Understand **testing** and why it is necessary.
- Understand **debugging** and identify common types of errors.
- Understand why an algorithm is needed before coding.
- Follow the complete problem-solving process.

---

1. 💻 Coding — Converting an Algorithm into a Program

Coding is the process of converting an algorithm or solution into instructions written in a programming language.

For example, suppose the problem is:

> Find the sum of two numbers.

Algorithm

```
1. Start
2. Input A and B
3. Calculate Sum = A + B
4. Display Sum
5. Stop
```

Python Code

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

sum = a + b

print("Sum =", sum)
```

Important Point

The **algorithm tells us what to do**, while the **program tells the computer how to do it using a programming language**.

```
Problem
   ↓
```

```
Algorithm
   ↓
Python Code
   ↓
Execution
```

---

2. 🧩 Example — From Problem to Code

Problem

Write a program to calculate the area of a rectangle.

Step 1 — Analyze

Input:

- Length
- Breadth

Processing:

Area = Length × Breadth

Output:

- Area

Step 2 — Algorithm

```
1. Start
2. Input length and breadth
3. Calculate Area = length × breadth
4. Display Area
5. Stop
```

Step 3 — Code

```python
length = float(input("Enter length: "))
breadth = float(input("Enter breadth: "))

area = length * breadth

print("Area =", area)
```

Example Output

```
Enter length: 10
Enter breadth: 5
Area = 50.0
```

⭐ Key Idea

> **A well-designed algorithm makes coding easier.**

---

3. 🧪 Testing a Program

Testing is the process of executing a program with different inputs to check whether it produces the expected output.

A program should not be considered correct just because it runs without showing an error.

Example

Consider:

```python
a = int(input("Enter a number: "))
b = int(input("Enter a number: "))

result = a - b

print("Sum =", result)
```

The program may run successfully, but it gives the wrong answer because subtraction has been used instead of addition.

Testing can help us identify this problem.

Types of Test Data

1. Normal Input

Valid input that is expected during normal use.

Example:

```
10
20
```

2. Boundary Input

Input near the limits of the allowed range.

Example:

If marks must be between 0 and 100:

```
0
100
```

3. Invalid Input

Input that does not satisfy the requirements.

Example:

```
Marks = 150
```

if the valid range is 0–100.

⭐ Remember

> **Testing checks whether the program behaves correctly for different inputs.**

---

4. 🐞 Debugging and Types of Errors

Debugging is the process of finding, identifying and correcting errors in a program.

The errors in a program can generally be classified into three important types.

1. Syntax Error

A **syntax error** occurs when Python rules or grammar are violated.

Example:

```
if x > 10
    print("Greater")
```

The colon `:` is missing.

Correct version:

```
if x > 10:
    print("Greater")
```

Python will normally report a syntax error before executing the program.

---

2. Runtime Error

A **runtime error** occurs while the program is executing.

Example:

```
a = 10
b = 0

print(a / b)
```

This causes a **ZeroDivisionError** because division by zero is not allowed.

Other examples include:

- Invalid conversion
- Accessing an invalid index
- Using an undefined variable

---

3. Logical Error

A **logical error** occurs when the program runs but produces an incorrect result.

Example:

```
length = 10
breadth = 5

area = length + breadth

print("Area =", area)
```

The program runs, but the formula is wrong.

Correct:

```
area = length * breadth
```

⭐ Important Comparison

Error	When does it occur?	Example
Syntax Error	Python syntax is incorrect	Missing `:`
Runtime Error	During execution	Division by zero
Logical Error	Program gives wrong result	Wrong formula

---

5. 🔍 Debugging Process

A systematic debugging process can be followed:

```
Run the Program
      ↓
Observe the Output/Error
```

```
      ↓
Identify the Problem
      ↓
Find the Cause
      ↓
Correct the Code
      ↓
Run the Program Again
      ↓
Test With Different Inputs
```

Example

Suppose the program is:

```
a = 10
b = 20

average = a + b / 2

print("Average =", average)
```

The expected average is:

```
15
```

But Python calculates:

```
10 + (20 / 2)
= 20
```

The problem is the **order of operations**.

Correct program:

```
average = (a + b) / 2
```

This is a **logical error**.

---

6. 📐 Need for an Algorithm

An algorithm is needed because it provides a clear plan before coding.

Advantages of Using an Algorithm

1. Easy to Understand

The solution is written in simple steps.

2. Makes Coding Easier

The programmer can directly convert the steps into code.

3. Helps Find Logical Errors

Problems in the solution can be identified before coding.

4. Language Independent

An algorithm is not tied to Python, C++, Java, etc.

5. Easy to Modify

Steps can be changed before writing the actual program.

6. Easy to Communicate

Other programmers can understand the proposed solution.

7. Helps in Testing

Expected results can be determined from the algorithm.

Example

Without an algorithm:

```
Problem → Immediately start coding
```

With an algorithm:

```
Problem
   ↓
Understand
   ↓
Analyze
   ↓
Algorithm
   ↓
Code
   ↓
Test
   ↓
Debug
```

⭐ Key Point

> **An algorithm acts as a blueprint for a program.**

---

7. 🧠 Complete Problem-Solving Process

Let's combine everything learned so far.

Example Problem

> Write a program to determine whether a number is even or odd.

1. Analyze the Problem

Input: One number

Processing: Check whether `number % 2 == 0`

Output: Even or Odd

2. Develop Algorithm

```
1. Start
2. Input number N
3. If N % 2 == 0:
       Display "Even"
4. Otherwise:
       Display "Odd"
5. Stop
```

3. Write Code

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

if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

4. Test

Test with:

```
Input: 10
Expected Output: Even

Input: 7
Expected Output: Odd

Input: 0
Expected Output: Even
```

5. Debug

If the output is incorrect, check:

- Condition
- Operator

- Input
- Variables
- Indentation
- Program logic

---

8. 📝 Practice & Quick Revision

Practice Questions

Q1. What is coding?

Q2. Why should an algorithm be prepared before coding?

Q3. What is testing?

Q4. What is debugging?

Q5. Differentiate between syntax error, runtime error and logical error.

Q6. Identify the type of error:

```
print("Hello"
```

Q7. Identify the type of error:

```
x = 10
y = 0
print(x / y)
```

Q8. Identify the type of error:

```
length = 10
breadth = 5
area = length + breadth
```

Q9. Write an algorithm and Python program to calculate the square of a number.

Q10. Write an algorithm and Python program to check whether a number is positive, negative or zero.

🎯 Quick Revision

Coding

Converting an algorithm into a program using a programming language.

Testing

Executing a program with different inputs to check its correctness.

Debugging

Finding and correcting errors in a program.

Three Important Errors

Syntax Error → Wrong Python syntax

Runtime Error → Error during execution

Logical Error → Program runs but gives wrong result

Need for Algorithm

Understand → Plan → Code → Test → Debug

⭐ KEY TAKEAWAY

> **Never think that a program is correct just because it runs successfully.**

A good programmer follows:

```
ANALYZE
   ↓
ALGORITHM
   ↓
CODE
   ↓
TEST
   ↓
DEBUG
   ↓
FINAL SOLUTION
```

🔜 Next Lecture

Lecture 5 — Flowcharts: Visual Representation of an Algorithm

We will learn:

- What is a flowchart?

- Need and advantages of flowcharts
- Standard flowchart symbols
- Rules for drawing flowcharts
- Flowcharts for common problems

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 5 — Flowcharts: Visual Representation of an Algorithm

---

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Define a **flowchart**.
- Understand the need and advantages of flowcharts.
- Identify common flowchart symbols.
- Understand the purpose of each symbol.
- Represent an algorithm using a flowchart.
- Create flowcharts for simple problems involving sequence, decision and repetition.

---

1. 🧠 What is a Flowchart?

A **flowchart** is a graphical representation of an algorithm or process.

It represents the steps of a solution using **standard symbols connected by arrows**.

In simple words:

> **A flowchart shows the flow of a solution visually.**

Example

Problem:

> Add two numbers.

The flow can be represented as:

```
   START
     ↓
Input A, B
     ↓
Sum = A + B
     ↓
Display Sum
     ↓
    STOP
```

The flowchart makes the sequence of operations easy to understand.

---

2. 💡 Why Do We Use Flowcharts?

Flowcharts are useful because they:

- Make the logic of a problem easy to understand.
- Provide a visual representation of an algorithm.
- Help in planning a program before coding.
- Help identify logical errors.
- Make communication of a solution easier.
- Help programmers understand complex processes.
- Can be used as a guide while writing code.

⭐ Remember

> **Algorithm → Written steps**

> **Flowchart → Graphical representation of those steps**

---

3. 🔷 Standard Flowchart Symbols

Flowcharts use standard symbols for different types of operations.

Symbol	Name	Purpose
Oval	Terminator	Start/Stop
Parallelogram	Input/Output	Input or display output
Rectangle	Process	Calculation or processing
Diamond	Decision	Condition/decision
Arrow	Flow Line	Shows direction of flow
Circle	Connector	Connects different parts of a flowchart

Easy Memory Trick

Oval → Start/Stop

Parallelogram → Input/Output

Rectangle → Process

Diamond → Decision

Arrow → Flow

---

4. 🟢 Terminator — Start/Stop

The **Terminator** symbol is represented by an **oval**.

It is used to indicate:

- Start of a flowchart.
- End of a flowchart.

Example:



and:



⭐ Remember

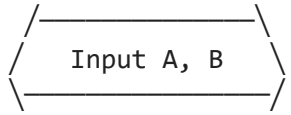
> **Oval = Start or Stop**

---

5. 🟦 Input/Output Symbol

The **parallelogram** is used for input and output operations.

Input Example



Output Example



It is used for operations such as:

```
Input number
Input name
Display result
Print average
```

⭐ Remember

> **Parallelogram = Input / Output**

---

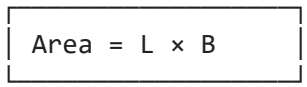
6. 🟨 Process Symbol

The **rectangle** represents a processing operation.

It is used for:

- Calculations
- Assignments
- Data processing

Example



Other examples:

```
Total = A + B + C
Average = Total / 3
Sum = Sum + Number
```

⭐ Remember

> **Rectangle = Process / Calculation**

---

7. 🔶 Decision Symbol

The **diamond** represents a decision or condition.

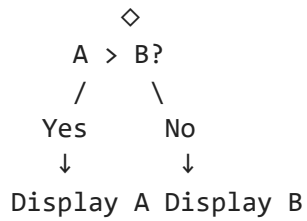
It usually has two possible paths:

Yes / No

or:

True / False

Example



A decision is commonly used with:

```
if
elif
else
```

Example Condition

```
Marks >= 40?
```

If True:

```
Pass
```

If False:

```
Fail
```

⭐ Remember

> **Diamond = Decision / Condition**

---

8. ➡️ Flow Lines and Connectors

Flow Line

An **arrow** shows the direction in which the process moves.

Example:

```
START
  ↓
Input
  ↓
```

```
Process
  ↓
Output
  ↓
STOP
```

Connector

A **connector** is used when a flowchart becomes large or when connecting different parts of the same flowchart.

It helps avoid excessive crossing of flow lines.

⭐ Remember

> **Arrow = Direction of flow**

---

9. 🧮 Flowchart — Addition of Two Numbers

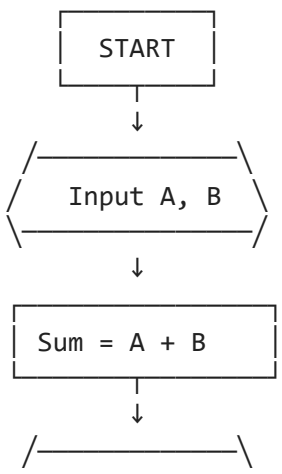
Problem

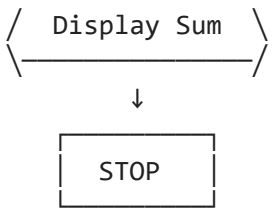
Draw a flowchart to input two numbers and display their sum.

Algorithm

```
1. Start
2. Input A and B
3. Calculate Sum = A + B
4. Display Sum
5. Stop
```

Flowchart





Python Program

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

sum = a + b

print("Sum =", sum)
```

---

10. 🔍 Flowchart — Even or Odd

Problem

Draw a flowchart to determine whether a number is even or odd.

Logic

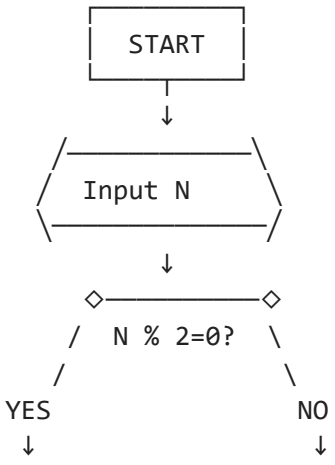
`Number % 2 == 0?`
If Yes:

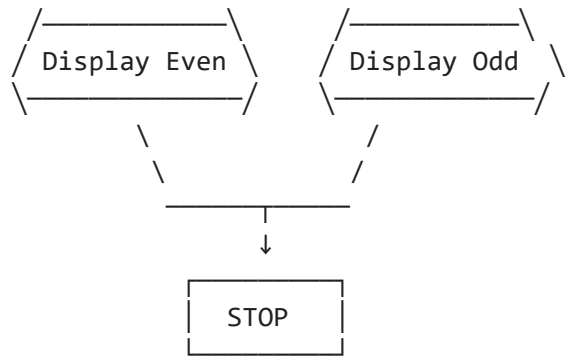
Even

If No:

Odd

Flowchart





Python Program

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

if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

---

11. 🧠 Flowchart — Largest of Two Numbers

Problem

Draw a flowchart to find the larger of two numbers.

Flowchart Logic

```
START
  ↓
Input A, B
  ↓
Is A > B?
 ↙       ↘
YES       NO
 ↓         ↓
A larger  B larger
 ↘       ↙
   STOP
```

Detailed Flow

```
      START
        ↓
   Input A, B
        ↓
     ◇ A > B?
```

```
     /        \
   YES         NO
    ↓           ↓
Display A   Display B
as larger   as larger
     \        /
      \      /
       ↓    ↓
        STOP
```

Python Program

```python
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

if a > b:
    print("A is larger")
elif b > a:
    print("B is larger")
else:
    print("Both are equal")
```

---

12. 🔄 Flowchart with Repetition

A flowchart can also represent a **loop**.

Consider:

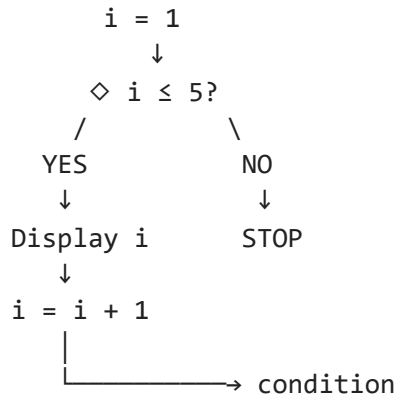
> Print numbers from 1 to 5.

Logic

```
Set i = 1
     ↓
Is i ≤ 5?
  ↙       ↘
YES       NO
 ↓         ↓
Print i   STOP
 ↓
i = i + 1
 ↓
 └──────────────→ back to condition
```

Flowchart

```
          START
            ↓
```



Python Program

```
i = 1

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

⭐ Important

A loop in a flowchart creates a **backward flow line** to the decision.

---

13. 🔗 Algorithm vs Flowchart

Algorithm	Flowchart
Written representation	Graphical representation
Uses statements/steps	Uses symbols
Easy to write	Easy to visualize
Does not require symbols	Uses standard symbols
Describes the solution	Shows the flow of the solution

Example

Algorithm:

```
Input A and B
Calculate Sum
Display Sum
```

Flowchart:

```
Input → Process → Output
```

Both represent the **same solution**.

14. ⚠️ Common Mistakes in Flowcharts

Mistake 1 — Missing Start or Stop

Every basic flowchart should clearly indicate where the process begins and ends.

Mistake 2 — Wrong Symbol

Do not use a rectangle for a decision.

❌

```
[ Is A > B? ]
```

✅

Use a **diamond** for the decision.

Mistake 3 — Missing Arrows

Every step should have a clear flow direction.

Mistake 4 — Unclear Decision Paths

A decision should clearly indicate:

Yes/No

or:

True/False

Mistake 5 — Crossing Flow Lines

Avoid unnecessary crossing of arrows because it makes the flowchart difficult to understand.

---

15. 📝 Practice Questions

Q1.

Draw a flowchart to calculate the **area of a rectangle**.

Q2.

Draw a flowchart to calculate the **simple interest**.

Q3.

Draw a flowchart to determine whether a number is **positive, negative or zero**.

Q4.

Draw a flowchart to find the **largest of three numbers**.

Q5.

Draw a flowchart to calculate the **average of three numbers**.

Q6.

Draw a flowchart to calculate the **factorial of a number**.

Q7.

Draw a flowchart to print numbers from **1 to 10**.

Q8.

Draw a flowchart to calculate the **sum of the first N natural numbers**.

🎯 Quick Revision

Flowchart

A graphical representation of an algorithm using standard symbols.

Important Symbols

Symbol	Meaning
Oval	Start/Stop
Parallelogram	Input/Output
Rectangle	Process
Diamond	Decision
Arrow	Flow direction

Symbol	Meaning
Circle	Connector

⭐ Most Important

```
Oval          → Start / Stop
Parallelogram → Input / Output
Rectangle     → Process
Diamond       → Decision
Arrow         → Flow
```

🏆 Remember the Relationship

```
PROBLEM
   ↓
ANALYSIS
   ↓
ALGORITHM
   ↓
FLOWCHART
   ↓
CODING
   ↓
TESTING
   ↓
DEBUGGING
```

> **Algorithm = What steps should be performed?**

> **Flowchart = How do those steps flow visually?**

> **Program = How are those steps implemented in a programming language?**

🔜 Next Lecture

Lecture 6 — Pseudocode & Decomposition

We will learn:

- What is pseudocode?
- Rules for writing pseudocode
- Algorithm vs pseudocode
- Examples of pseudocode

- What is decomposition?
- Breaking a complex problem into smaller problems
- Solving problems using decomposition

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 6 — Pseudocode & Decomposition

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Define **pseudocode**.
- Understand the purpose of pseudocode.
- Write simple pseudocode for programming problems.
- Differentiate between an algorithm, flowchart and pseudocode.
- Understand **decomposition**.
- Break a complex problem into smaller and manageable sub-problems.
- Apply decomposition to simple programming problems.

---

1. 🧠 What is Pseudocode?

Pseudocode is a simple, informal way of describing the steps of an algorithm using a mixture of ordinary language and programming-like statements.

It is **not an actual programming language** and cannot be directly executed by a computer.

Simple Definition

> **Pseudocode is a structured description of a solution written in simple language without following the exact syntax of a programming language.**

Example

Problem:

> Find the sum of two numbers.

Pseudocode

```
START
INPUT A, B
SUM ← A + B
OUTPUT SUM
STOP
```

This describes the logic without using Python syntax.

---

2. 💡 Why Do We Use Pseudocode?

Pseudocode helps us:

- Plan a program before coding.
- Understand the logic clearly.
- Focus on the solution rather than programming syntax.
- Identify logical mistakes early.
- Communicate the solution to other people.
- Easily convert a solution into a programming language.

⭐ Remember

> **Pseudocode focuses on logic, not programming syntax.**

For example:

```
IF marks >= 40
    OUTPUT "Pass"
ELSE
    OUTPUT "Fail"
```

This is pseudocode.

It can later be converted into Python:

```python
if marks >= 40:
    print("Pass")
else:
    print("Fail")
```

---

3. 📝 Common Pseudocode Keywords

There is no single compulsory syntax for pseudocode, but some common keywords are:

Keyword	Meaning
`START`	Beginning of the algorithm
`STOP`	End of the algorithm
`INPUT`	Accept data
`OUTPUT`	Display result
`SET`	Assign a value
`IF`	Check a condition
`ELSE`	Alternative condition
`ELSE IF`	Additional condition
`WHILE`	Repeat while a condition is true
`FOR`	Repeat a specified number of times

Example

```
START
INPUT number

IF number % 2 = 0 THEN
    OUTPUT "Even"
ELSE
    OUTPUT "Odd"
END IF

STOP
```

---

4. 🖇️ Algorithm vs Pseudocode

An algorithm and pseudocode both describe a solution, but their presentation is different.

Algorithm	Pseudocode
Written as logical steps	Written in a structured programming-like form
Can use ordinary language	Uses keywords such as INPUT, OUTPUT, IF
Less formal	More structured
Easy for beginners	Closer to actual programming
Focuses on steps	Focuses on logic and structure

Example

Algorithm

```
1. Start
2. Input two numbers
3. Add the numbers
4. Display the result
5. Stop
```

Pseudocode

```
START
INPUT A, B
SUM ← A + B
OUTPUT SUM
STOP
```

Both represent the same solution.

---

5. 🔄 Algorithm vs Flowchart vs Pseudocode

These are three different ways of representing a solution.

Method	Representation
Algorithm	Written steps
Flowchart	Graphical symbols
Pseudocode	Structured English/programming-like statements

Example Problem

> Find whether a number is even or odd.

Algorithm

```
1. Start
2. Input number N
3. If N is divisible by 2, display Even
4. Otherwise display Odd
5. Stop
```

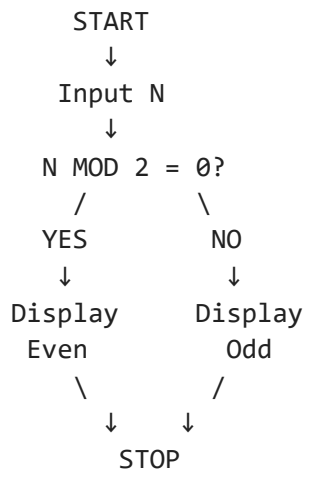
Pseudocode

```
START
INPUT N
```

```
IF N MOD 2 = 0 THEN
    OUTPUT "Even"
ELSE
    OUTPUT "Odd"
END IF

STOP
```

Flowchart



⭐ Remember

```
Algorithm  → Words / Steps
Flowchart  → Symbols / Diagram
Pseudocode → Structured Instructions
```

---

6. 🧮 Example — Calculate Area of Rectangle

Problem

Write pseudocode to calculate the area of a rectangle.

Formula

Area = Length × Breadth

Pseudocode

```
START

INPUT Length
INPUT Breadth
```

```
Area ← Length × Breadth

OUTPUT Area

STOP
```

Python Equivalent

```python
length = float(input("Enter length: "))
breadth = float(input("Enter breadth: "))

area = length * breadth

print("Area =", area)
```

Notice that the pseudocode describes the logic without worrying about Python syntax.

---

7. 🔍 Example — Find Largest of Two Numbers

Problem

Write pseudocode to find the largest of two numbers.

Pseudocode

```
START

INPUT A, B

IF A > B THEN
    OUTPUT A
ELSE IF B > A THEN
    OUTPUT B
ELSE
    OUTPUT "Both are equal"
END IF

STOP
```

Python Equivalent

```python
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

if a > b:
    print(a)
```

```
elif b > a:
    print(b)
else:
    print("Both are equal")
```

---

8. 🔄 Pseudocode Using a Loop

Pseudocode can also represent repetition.

Problem

Print numbers from 1 to 5.

Pseudocode Using `FOR`

```
START

FOR I ← 1 TO 5
    OUTPUT I
END FOR

STOP
```

Python Equivalent

```
for i in range(1, 6):
    print(i)
```

---

9. 🔁 Pseudocode Using WHILE

Problem

Print numbers from 1 to 5 using a while loop.

Pseudocode

```
START

SET I ← 1

WHILE I <= 5
    OUTPUT I
    SET I ← I + 1
END WHILE
```

STOP

Python Equivalent

```
i = 1

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

⭐ Important

A loop must have a condition or control mechanism that eventually allows it to stop.

---

10. 🧩 What is Decomposition?

Decomposition is the process of breaking a large or complex problem into smaller, simpler and manageable sub-problems.

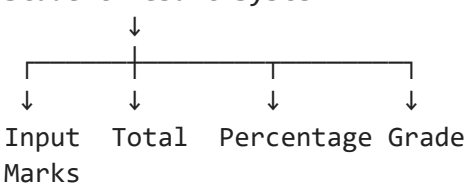
Simple Definition

> **Decomposition means breaking a complex problem into smaller problems that are easier to solve.**

Example

Suppose we want to create a **Student Result System**.

Instead of solving the entire problem at once, divide it into:



Each smaller task can then be solved separately.

---

11. 💡 Why is Decomposition Important?

Decomposition makes complex problems easier to manage.

Advantages

1. Simplifies the Problem

A large problem becomes a collection of smaller problems.

2. Easier to Understand

Each part can be understood separately.

3. Easier to Code

Smaller tasks are easier to convert into programs.

4. Easier to Test

Each part can be tested separately.

5. Easier to Debug

If something goes wrong, we can identify which part caused the problem.

6. Encourages Reusability

A solution to one smaller problem may be reused elsewhere.

7. Makes Teamwork Easier

Different people can work on different parts of a large problem.

---

12. 🏫 Real-Life Example of Decomposition

Suppose a school wants to create an **Online Examination System**.

This is a large problem.

We can decompose it into:

```
Online Examination System
          ↓
  ┌──────┬──────┬──────┐
  ↓      ↓      ↓      ↓
Login  Questions Evaluation Result
```

Further decomposition:

```
Login
 ↓
Enter Username
 ↓
Enter Password
 ↓
```

```
Verify Details
 ↓
Allow / Deny Access
```

This makes the complete system much easier to design.

---

13. 🎑 Decomposition — Student Result Program

Large Problem

> Develop a program to generate a student's result.

Instead of treating it as one huge task, divide it into:

Sub-problem 1

Input marks.

Sub-problem 2

Calculate total.

```
Total = M1 + M2 + M3 + M4 + M5
```

Sub-problem 3

Calculate percentage.

```
Percentage = Total / 500 × 100
```

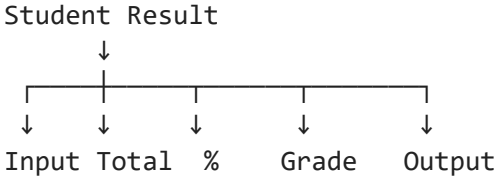
Sub-problem 4

Determine grade.

Sub-problem 5

Display result.

Complete Structure



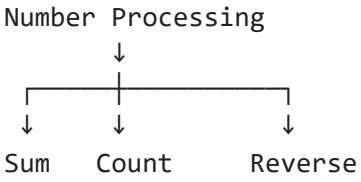
This is **decomposition**.

14. 🔢 Decomposition — Number Problem

Problem

> Write a program that accepts a number and finds its sum of digits, number of digits and reverse.

This can be divided into:



Sub-problem 1

Find sum of digits.

Sub-problem 2

Count digits.

Sub-problem 3

Reverse the number.

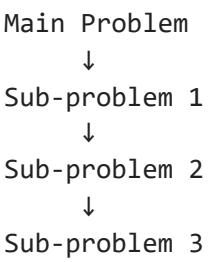
Each sub-problem can be designed and tested separately.

---

15. 🧠 Decomposition and Modular Thinking

When a complex problem is divided into smaller parts, each part can be treated as a separate logical unit.

For example:



```
     ↓
Final Solution
```

This approach is useful in real-world software development because large programs often contain many smaller components.

⭐ Key Idea

> **Solve the small problems first, then combine their solutions to solve the complete problem.**

---

16. 🔗 Combining Pseudocode and Decomposition

Pseudocode becomes especially useful after decomposition.

Problem

> Create a program that accepts marks of three subjects, calculates total and average, and displays whether the student passed.

Decomposition

```
Student Result
      ↓
Input Marks
      ↓
Calculate Total
      ↓
Calculate Average
      ↓
Check Pass/Fail
      ↓
Display Result
```

Pseudocode

```
START

INPUT M1, M2, M3

TOTAL ← M1 + M2 + M3

AVERAGE ← TOTAL / 3

IF AVERAGE >= 40 THEN
    OUTPUT "Pass"
```

```
ELSE
    OUTPUT "Fail"
END IF

OUTPUT TOTAL
OUTPUT AVERAGE

STOP
```

Python Implementation

```python
m1 = float(input("Enter marks of subject 1: "))
m2 = float(input("Enter marks of subject 2: "))
m3 = float(input("Enter marks of subject 3: "))

total = m1 + m2 + m3
average = total / 3

if average >= 40:
    result = "Pass"
else:
    result = "Fail"

print("Total =", total)
print("Average =", average)
print("Result =", result)
```

---

17. ⚠ Common Mistakes in Pseudocode

Mistake 1 — Writing Actual Python Syntax

Pseudocode does not need exact Python syntax.

❌

```
print("Enter number")
```

Better:

```
INPUT number
```

Mistake 2 — Unclear Steps

❌

```
Do the calculation
```

✅

```
TOTAL ← A + B + C
```

Mistake 3 — Missing Conditions

If a problem involves a decision, clearly write:

```
IF condition THEN
    ...
ELSE
    ...
END IF
```

Mistake 4 — Infinite Loops

When using `WHILE`, make sure the control variable changes.

---

18. 📝 Practice Questions

Pseudocode Practice

Write pseudocode for:

Q1.

Calculate the area of a triangle.

Q2.

Find the largest of three numbers.

Q3.

Check whether a number is positive, negative or zero.

Q4.

Calculate the factorial of a number.

Q5.

Print the multiplication table of a number.

Q6.

Check whether a number is prime.

Decomposition Practice

Break the following problems into smaller sub-problems:

Q7.

Student Result Management System.

Q8.

Online Shopping System.

Q9.

Library Management System.

Q10.

ATM System.

For each problem, identify at least **4 smaller tasks**.

🎯 Quick Revision

Pseudocode

A structured, informal description of an algorithm using simple language and programming-like statements.

Decomposition

Breaking a complex problem into smaller and manageable sub-problems.

Pseudocode Example

```
START
INPUT A, B
SUM ← A + B
OUTPUT SUM
STOP
```

Decomposition Example

```
Complex Problem
      ↓
Smaller Problems
      ↓
Solve Each Part
      ↓
Combine Solutions
      ↓
Final Solution
```

⭐ KEY TAKEAWAY

Algorithm

> **Describes the solution step by step.**

Flowchart

> **Shows the solution graphically.**

Pseudocode

> **Describes the solution in a structured, programming-like form.**

Decomposition

> **Breaks a complex problem into smaller problems.**

```
PROBLEM
   ↓
ANALYZE
   ↓
DECOMPOSE
   ↓
ALGORITHM / PSEUDOCODE
   ↓
FLOWCHART
   ↓
CODE
   ↓
TEST & DEBUG
   ↓
FINAL SOLUTION
```

🔜 Next Lecture

Lecture 7 — Python Programming Language

We will learn:

- Introduction to Python
- What is Python?
- Features of Python
- Advantages of Python
- Limitations of Python
- Why Python is popular
- Applications of Python

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 7 — Python Programming Language

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Understand what **Python** is.
- Know the history and development of Python.
- Understand the major features of Python.
- Explain the advantages of Python.
- Understand the limitations of Python.
- Identify common applications of Python.
- Explain why Python is popular among programmers.

---

1. 🐍 What is Python?

Python is a high-level, general-purpose programming language used to develop a wide variety of applications.

Python was designed to make programming **simple, readable and easy to learn**.

Simple Definition

> **Python is a high-level, interpreted, general-purpose programming language known for its simple and readable syntax.**

Python can be used for:

- Web development
- Data analysis
- Artificial Intelligence
- Machine Learning
- Automation
- Scientific computing
- Game development
- Desktop applications

---

2. 👨‍💻 History of Python

Python was created by **Guido van Rossum**.

Development of Python started in the late 1980s, and the first public release of Python was made in **1991**.

The name **Python** was inspired by the British comedy group **Monty Python**, rather than the snake.

Important Timeline

```
Late 1980s
    ↓
Python development begins
    ↓
1991
    ↓
First public release
    ↓
Python 2
    ↓
Python 3
    ↓
Modern Python
```

⭐ Remember

> **Creator of Python → Guido van Rossum**

> **First public release → 1991**

---

3. 🌟 Why is Python Popular?

Python is popular because its syntax is relatively simple and readable.

Compare a simple output statement:

Python

```
print("Hello World")
```

Python requires very little code for simple tasks.

This makes it suitable for:

- Beginners
- Students
- Researchers
- Data scientists
- Software developers

---

4. ⭐ Features of Python

Python has many important features.

4.1 Easy to Learn

Python has a simple and readable syntax.

Example:

```
a = 10
b = 20

print(a + b)
```

The code is easy to understand even for beginners.

4.2 High-Level Language

Python is a **high-level programming language**.

A high-level language is designed to be closer to human language than machine language.

Programmers do not need to directly write machine-level instructions for most tasks.

4.3 Interpreted Language

Python programs are generally executed by an **interpreter**.

The Python interpreter reads and executes Python code.

Simple Representation

```
Python Program
      ↓
Python Interpreter
      ↓
Execution
      ↓
Output
```

⭐ Remember

> **Python is commonly described as an interpreted language.**

4.4 General-Purpose Language

Python is not designed for only one specific type of task.

It can be used for many different applications.

Examples:

```
Web Development
Data Science
AI / ML
Automation
Scientific Computing
Game Development
```

4.5 Portable

Python programs can generally run on different operating systems with little or no modification, provided the required Python environment is available.

For example:

- Windows
- Linux
- macOS

⭐ Remember

> **Write once → Run on different platforms**

4.6 Free and Open Source

Python is available freely and its source code is publicly available.

This allows developers to:

- Use Python without paying a license fee.
- Study its source code.
- Modify it according to applicable open-source licensing terms.
- Contribute to its development.

4.7 Dynamically Typed

In Python, the programmer does not have to explicitly declare the data type of a variable before assigning a value.

Example:

```
x = 10
```

Here Python determines that `x` contains an integer.

Later:

```
x = "Hello"
```

Now `x` contains a string.

⭐ Remember

> Python determines the type of a variable from the value assigned to it.

4.8 Object-Oriented

Python supports **Object-Oriented Programming (OOP)**.

It provides concepts such as:

- Classes
- Objects
- Inheritance
- Encapsulation

- Polymorphism

Students will study these concepts in greater detail in later topics.

4.9 Extensive Library Support

Python provides a large collection of built-in and third-party libraries.

Examples:

Library	Common Use
`math`	Mathematical operations
`random`	Random numbers
`datetime`	Date and time
`os`	Operating-system related tasks
`NumPy`	Numerical computing
`Pandas`	Data analysis
`Matplotlib`	Data visualization

Example

```
import math

print(math.sqrt(25))
```

Output:

```
5.0
```

4.10 Interactive

Python provides an interactive environment where commands can be entered and executed immediately.

Example:

```
>>> 10 + 20
30
```

This is useful for:

- Learning
- Experimenting

- Testing small pieces of code
- Debugging

---

5. 📊 Advantages of Python

1. Simple Syntax

Python syntax is relatively easy to read and write.

2. Easy to Learn

It is suitable for beginners.

3. Large Community

Python has a large worldwide developer community.

4. Large Number of Libraries

Many libraries are available for different applications.

5. Cross-Platform

Python is available for major operating systems.

6. Open Source

It can be used freely under its open-source license.

7. Wide Range of Applications

Python is used in many different fields.

8. Faster Development

Its concise syntax and extensive libraries can reduce development time.

---

6. ⚠ Limitations of Python

Although Python has many advantages, it also has some limitations.

6.1 Slower Than Some Compiled Languages

Python programs can be slower than programs written in languages such as C or C++ for certain workloads.

This is partly because Python is typically executed through a runtime rather than being directly compiled to native machine code in the same way as traditional ahead-of-time compiled languages.

6.2 Higher Memory Usage

Python programs can use more memory than some lower-level languages.

6.3 Not Ideal for Some Mobile Applications

Python is not the primary language for many native mobile applications.

6.4 Runtime Type Errors

Because Python is dynamically typed, some type-related errors may only become apparent when the relevant code is executed.

Example:

```
x = 10
y = "20"

print(x + y)
```

This results in a type error because an integer and string cannot be directly added in this way.

6.5 Less Suitable for Some Low-Level Programming

Python is generally not the first choice for:

- Operating system kernels

- Device drivers
- Embedded systems with very limited resources
- Hardware-level programming

Languages such as C and C++ are more commonly used for these tasks.

---

7. 🌍 Applications of Python

Python is used in many areas.

1. Web Development

Frameworks such as:

- Django
- Flask

are used to develop web applications.

2. Data Science

Python is widely used for:

- Data analysis
- Data visualization
- Statistical analysis

Libraries include:

```
NumPy
Pandas
Matplotlib
```

3. Artificial Intelligence

Python is widely used in:

- Machine Learning
- Deep Learning
- Natural Language Processing

4. Automation

Python can automate repetitive tasks.

Examples:

- File processing
- Data processing
- Report generation

5. Scientific Computing

Python is used for mathematical and scientific calculations.

6. Game Development

Libraries such as `pygame` can be used for creating games.

7. Education

Python is widely used to teach programming because of its readable syntax.

---

8. 🆚 Python vs Low-Level Language

Python	Low-Level Language
Easier to learn	More difficult to learn
High-level	Closer to hardware
More readable	Less human-readable
Generally slower for some workloads	Generally faster for many low-level workloads
Requires an interpreter/runtime	Can be compiled directly to machine code
Easier development	More control over hardware

---

9. 🧠 Why Python is Suitable for Beginners

Python is particularly useful for beginners because:

- Its syntax is simple.
- Programs are generally shorter.
- It is easy to test small pieces of code.
- Error messages can help identify problems.
- Many learning resources are available.
- Students can focus more on logic than complicated syntax.

Example

To print numbers from 1 to 5:

```
for i in range(1, 6):
    print(i)
```

The code is short and readable.

---

10. 🔍 Python Program Execution

A simplified view of Python program execution is:

```
 Python Source Code
         ↓
Python Interpreter
         ↓
Bytecode / Runtime
         ↓
     Execution
         ↓
      Output
```

Important

Python implementations commonly use an intermediate representation called **bytecode** before execution by the Python virtual machine.

The exact internal implementation can vary between Python implementations.

---

11. 📝 Simple Python Example

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))

print("Name:", name)
print("Age:", age)
```

Example Output

```
Enter your name: Ali
Enter your age: 17

Name: Ali
Age: 17
```

This simple program demonstrates Python's:

- Input

- Variables
- Type conversion
- Output

---

12. 📋 Python Features — Quick Table

Feature	Meaning
Easy to Learn	Simple syntax
High-Level	Human-readable programming language
Interpreted	Executed through a Python interpreter/runtime
General-Purpose	Used for many applications
Portable	Available on multiple platforms
Open Source	Freely available under an open-source license
Dynamically Typed	Variable types are determined at runtime
Object-Oriented	Supports classes and objects
Interactive	Commands can be executed interactively
Large Libraries	Many built-in and third-party libraries

---

13. 🎤 Important Viva Questions

Q1. Who developed Python?

Answer: Guido van Rossum.

Q2. When was Python first publicly released?

Answer: 1991.

Q3. Is Python a high-level language?

Answer: Yes.

Q4. Is Python a general-purpose language?

Answer: Yes.

Q5. Is Python interpreted?

Answer: Python is commonly described as an interpreted language; modern implementations typically use bytecode and a runtime.

Q6. Is Python dynamically typed?

Answer: Yes.

Q7. Is Python open source?

Answer: Yes.

Q8. Name two advantages of Python.

Answer: Easy syntax and extensive library support.

Q9. Name two limitations of Python.

Answer: It can be slower than some compiled languages and may use more memory.

Q10. Name two applications of Python.

Answer: Data Science and Web Development.

---

14. 📝 Practice Questions

Q1.

What is Python? Write its definition.

Q2.

Who developed Python and when was it first publicly released?

Q3.

Write any five features of Python.

Q4.

Explain any four advantages of Python.

Q5.

Write any three limitations of Python.

Q6.

Why is Python considered easy to learn?

Q7.

What is meant by a general-purpose programming language?

Q8.

What is meant by dynamically typed language?

Q9.

Write four areas where Python is used.

Q10.

Differentiate between a high-level language and a low-level language.

🎯 Quick Revision

Python

A high-level, interpreted, general-purpose programming language known for its simple and readable syntax.

Creator

Guido van Rossum

First Public Release

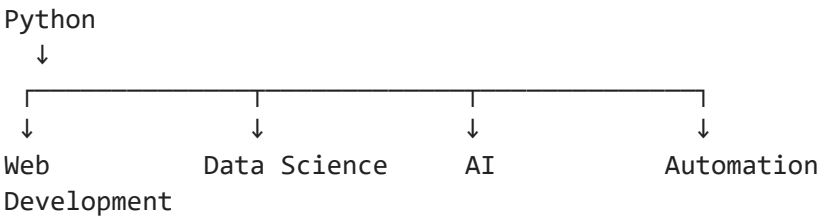
1991

Important Features

- Easy to learn
- High-level
- Interpreted
- General-purpose

- Portable
- Open source
- Dynamically typed
- Object-oriented
- Interactive
- Large library support

Major Applications



⭐ KEY TAKEAWAY

> **Python is popular because it combines simple syntax, powerful libraries and a wide range of applications.**

The most important points to remember for the exam are:

```
Python
↓
Guido van Rossum
↓
First public release: 1991
↓
High-Level
↓
Interpreted
↓
General-Purpose
↓
Dynamically Typed
↓
Open Source
↓
Widely Used
```

🔜 Next Lecture

Lecture 8 — Installing Python & Interacting with Python

We will learn:

- Installing Python
- Python IDLE
- Python Shell
- Interactive/Command Line Mode
- Script Mode
- Executing Python programs
- Saving Python programs
- Exiting Python

📘 Chapter 3: Computational Thinking and Getting Started with Python

Lecture 8 — Installing Python & Interacting with Python

🎯 Learning Objectives

By the end of this lecture, students will be able to:

- Understand how Python is installed on a computer.
- Understand **Python IDLE**.
- Differentiate between **Python Shell** and **Script Mode**.
- Execute Python commands interactively.
- Write, save and execute a Python program.
- Understand how to exit Python.
- Identify the advantages of Shell Mode and Script Mode.

---

1. 🐍 Installing Python

Python is not automatically available on every computer. If Python is not installed, it must first be installed before Python programs can be executed.

The Python installation package can be obtained from the official Python website.

During installation, Python installs the necessary components required to run Python programs.

Basic Installation Process

```
Download Python
      ↓
Run Installer
      ↓
Install Python
      ↓
Verify Installation
      ↓
Start Python
```

⭐ Important

When installing Python on Windows, the option:

`Add Python to PATH`
is useful because it allows Python to be accessed from the command prompt.

---

2. 🔎 Checking Python Installation

After installation, we can check whether Python is available.

Open **Command Prompt** and type:

`python --version`
or, depending on the system:

`py --version`
If Python is installed correctly, a version number will be displayed.

Example:

`Python 3.x.x`

> The exact version number depends on the Python version installed on the computer.

---

3. 💻 What is Python IDLE?

IDLE stands for:

> **Integrated Development and Learning Environment**

It is an environment that comes with standard Python installations and provides tools for writing and executing Python programs.

IDLE provides:

- Python Shell
- Script Editor
- Syntax highlighting
- Basic debugging support
- Program execution facilities

⭐ Remember

> **IDLE is an environment used to write and execute Python programs.**

---

4. 🐍 Python Shell

The **Python Shell** is an interactive environment where Python commands can be entered and executed immediately.

When Python Shell is opened, we commonly see:

```
>>>
```

This is called the **Python prompt**.

Example

```
>>> 10 + 20
30
```

Another example:

```
>>> print("Hello")
Hello
```

The command is executed immediately and the result is displayed.

---

5. 💡 Interactive Mode

When Python commands are entered one at a time and executed immediately, we are using **Interactive Mode**.

Example

```
>>> a = 10
>>> b = 20
>>> a + b
30
```

Each statement is executed as soon as it is entered.

Advantages

Interactive mode is useful for:

- Learning Python
- Testing small commands
- Trying expressions
- Checking the result of an operation
- Quick experimentation

⭐ Remember

> **Interactive Mode → Type → Execute Immediately → See Result**

---

6. ➡️ Command Line Interaction

Python can also be started from the operating system's command line.

For example, in Windows Command Prompt:

`python`
After Python starts, the prompt may appear as:

`>>>`
We can then enter Python commands.

Example:

```
>>> 5 * 6
30
```

To leave the interactive interpreter, we can use:

`exit()`
or:

`quit()`

---

7. 📝 Script Mode

In **Script Mode**, Python instructions are written in a file and then executed as a complete program.

Python program files normally have the extension:

`.py`

Example

Create a file:

`hello.py`
Write:

```
print("Hello World")
```
Then execute the program.

Flow

```
Write Code
    ↓
Save as .py
    ↓
Run the Program
    ↓
Get Output
```

---

8. VS Interactive Mode vs Script Mode

Interactive Mode	Script Mode
Commands are entered one at a time	Complete program is written in a file
Output appears immediately	Output appears when the program is executed
Useful for quick experiments	Useful for larger programs
Code is not automatically stored as a program file	Program can be saved
Suitable for learning/testing small commands	Suitable for developing complete programs

Example — Interactive Mode

```
>>> 2 + 3
5
```

Example — Script Mode

File: `sum.py`

```
a = 2
b = 3

print(a + b)
```

Output:

```
5
```

---

9. 👦💻 Writing a Python Program in IDLE

To create a Python program in IDLE:

Step 1

Open **IDLE**.

Step 2

Choose:

`File → New File`

Step 3

Write the Python program.

Example:

```
name = input("Enter your name: ")
print("Welcome", name)
```

Step 4

Save the file.

For example:

`welcome.py`

Step 5

Run the program using:

`Run → Run Module`
or the appropriate run command in the IDLE interface.

Step 6

The output appears in the Python Shell.

---

10. 🔢 Example — Simple Calculator

Create a Python script:

`calculator.py`
Write:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print("Sum =", a + b)
print("Difference =", a - b)
print("Product =", a * b)
```

Example Output

```
Enter first number: 20
Enter second number: 5

Sum = 25
Difference = 15
Product = 100
```

This is an example of **Script Mode** because the complete program is written and saved in a file.

---

11. 🧠 When Should We Use Interactive Mode?

Interactive Mode is useful when we want to quickly test something.

Example

Suppose we want to check:

```
>>> 25 % 4
1
```

We immediately get the answer.

We can also test:

```
>>> 2 ** 5
32
```

or:

```
>>> "Python".upper()
'PYTHON'
```

Best Uses

- Learning
- Experimenting
- Testing expressions
- Checking small pieces of code

---

12. 📚 When Should We Use Script Mode?

Script Mode is better when:

- The program contains many statements.
- We want to save the program.
- We want to run the program again later.
- We need to modify the program.
- We are developing a complete application.

Example

A program to calculate student results may contain many lines:

```
marks1 = float(input("Enter marks 1: "))
marks2 = float(input("Enter marks 2: "))
marks3 = float(input("Enter marks 3: "))

total = marks1 + marks2 + marks3
average = total / 3

print("Total =", total)
print("Average =", average)
```

It is much more convenient to save this as a `.py` file rather than typing every line again in the Shell.

---

13. 🔄 Shell Mode and Script Mode — Practical Example

Shell Mode

```
>>> a = 10
>>> b = 20
>>> a + b
30
```

The statements are entered individually.

Script Mode

```
a = 10
b = 20

result = a + b

print(result)
```

The complete program can be saved as:

`addition.py`
and executed whenever required.

---

14. 🚪 Exiting Python

When using the Python Shell, we can exit Python by using:

```
exit()
```

or:

```
quit()
```

Example

```
>>> exit()
```

Python will return to the operating system's command prompt.

⭐ Remember

> `exit()` and `quit()` can be used to leave the interactive Python interpreter.

---

15. ⚠️ Common Problems While Starting Python

Problem 1 — Python command not found

Possible reason:

- Python is not installed.
- Python was not added to PATH.
- The command being used is not the correct one for the system.

Problem 2 — Program does not run

Check:

- Syntax errors
- Indentation
- File name
- Python installation
- Whether the correct file is being executed

Problem 3 — Wrong Output

Check:

- Formula
- Conditions
- Variables
- Operators
- Input values
- Program logic

---

16. 🧪 Practical Activity

Students can perform the following activities in Python Shell.

Activity 1

Calculate:

```
>>> 25 + 15
```

Activity 2

Calculate:

```
>>> 25 * 4
```

Activity 3

Find the remainder:

```
>>> 25 % 4
```

Activity 4

Calculate:

```
>>> 2 ** 10
```

Activity 5

Display:

```
>>> print("Welcome to Python")
```

Activity 6

Create a script that displays:

```
Name: Your Name
Class: 11
Subject: Computer Science
```

---

17. 🧩 Interactive Mode vs Script Mode — Quick Revision

Interactive Mode

```
Command
   ↓
Execute Immediately
   ↓
Output
```

Script Mode

```
Write Program
     ↓
Save .py File
     ↓
Run Program
```

```
     ↓
Output
```

⭐ Easy Trick

> **Interactive = Try**

> **Script = Save & Run**

---

18. 🎤 Important Viva Questions

Q1. What is IDLE?

Answer: IDLE stands for **Integrated Development and Learning Environment** and provides an environment for writing and executing Python programs.

Q2. What is Python Shell?

Answer: It is an interactive environment where Python commands can be entered and executed immediately.

Q3. What symbol indicates the Python prompt?

Answer:

`>>>`

Q4. What is Interactive Mode?

Answer: A mode in which Python commands are entered and executed one at a time.

Q5. What is Script Mode?

Answer: A mode in which Python code is written and saved in a file, usually with a `.py` extension, and then executed.

Q6. What is the extension of a Python program file?

Answer:

`.py`

Q7. Which mode is better for writing a complete program?

Answer: Script Mode.

Q8. Which mode is useful for quickly testing an expression?

Answer: Interactive Mode.

Q9. How can we exit the Python interpreter?

**Answer:

```
exit()
```
or:

```
quit()
```

Q10. How can Python installation be checked from the command line?

**Answer:

```
python --version
```
or, depending on the system:

```
py --version
```

---

19. 📝 Practice Questions

Q1.

What is IDLE? Expand the term IDLE.

Q2.

What is Python Shell?

Q3.

What is the difference between Interactive Mode and Script Mode?

Q4.

What is the extension of a Python program file?

Q5.

Write two advantages of Interactive Mode.

Q6.

Write two advantages of Script Mode.

Q7.

Write the steps for creating and executing a Python program in IDLE.

Q8.

How can you exit the Python interpreter?

Q9.

Write a Python program to accept two numbers and display their sum.

Q10.

Explain why Script Mode is more suitable for developing large programs.

🎯 Complete Chapter 3 Revision

We have now covered the complete flow of Chapter 3:

```
Introduction to Problem-Solving
          ↓
Analyzing a Problem
          ↓
Developing an Algorithm
          ↓
Coding
          ↓
Testing & Debugging
          ↓
Need for an Algorithm
          ↓
Flowcharts
          ↓
Pseudocode
          ↓
Decomposition
          ↓
Python Programming Language
          ↓
Features / Advantages / Limitations
          ↓
Installing Python
          ↓
Python IDLE
```

```
          ↓
Python Shell
          ↓
Interactive / Command Line Mode
          ↓
Script Mode
          ↓
Exiting Python
```

⭐ FINAL REVISION TABLE

Concept	Key Point
Python	High-level, general-purpose programming language
IDLE	Integrated Development and Learning Environment
Python Shell	Interactive Python environment
`>>>`	Python prompt
Interactive Mode	Execute commands immediately
Script Mode	Write and save complete programs
`.py`	Python program file extension
`exit()`	Exit Python interpreter
`quit()`	Exit Python interpreter
`python --version`	Check Python version

🏆 KEY TAKEAWAY

> **Interactive Mode is mainly used for quick experimentation and testing, while Script Mode is used to write, save and execute complete Python programs.**

```
INTERACTIVE MODE
     ↓
Quick Testing / Learning

SCRIPT MODE
     ↓
Write → Save → Run → Reuse
```

✅ Chapter 3 Completed

The next chapter will move from **problem-solving concepts to Python programming fundamentals**, where students will start working with Python syntax, tokens, variables, data types, operators, expressions, input/output and other programming concepts.

In [ ]: