📘 Chapter 4 – Python Programming Fundamentals

Lecture 1: Python Basics, Variables, Data Types, Expressions & Operators

---

🎯 Learning Objectives

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

- Understand the basic fundamentals of Python programming.
- Identify the Python character set.
- Understand Python tokens.
- Understand the concept of objects in Python.
- Create and use variables.
- Identify different Python data types.
- Use the `type()` function.
- Perform multiple assignments.
- Understand mutable and immutable data types.
- Identify Python keywords.
- Understand expressions and type conversion.
- Use different types of Python operators.

---

1. Introduction to Python Programming

Python is a high-level, interpreted, general-purpose programming language.

Python is widely used in:

- Data Science
- Artificial Intelligence
- Machine Learning
- Web Development
- Automation
- Software Development

Example

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

Output

```
Hello, World!
```

Python programs are generally easy to read because Python uses simple and clear syntax.

---

2. Python Character Set

A **character set** is a collection of valid characters that can be used in a programming language.

The Python character set includes:

1. Alphabets

```
A-Z
a-z
```

Example:

```
name = "Haris"
```

2. Digits

```
0-9
```

Example:

```
age = 20
```

3. Special Symbols

Examples:

```
+  -  *  /  %  =  <  >  (  )  [  ]  {  }  :  ,  .  _  #  "  '
```

4. Whitespace Characters

Whitespace includes:

- Space
- Tab
- Newline

Python uses indentation (whitespace) to define blocks of code.

Example:

```
if age >= 18:
    print("Adult")
```

The spaces before `print()` are important.

---

3. Tokens in Python

A **token** is the smallest individual unit of a Python program.

Python tokens are mainly classified into:

1. Keywords
2. Identifiers
3. Literals
4. Operators
5. Punctuators / Delimiters

Example

```
age = 20
```

Here:

- `age` → Identifier
- `=` → Operator
- `20` → Literal

3.1 Keywords

Keywords are reserved words that have a special meaning in Python.

Examples:

```
if
else
for
while
def
return
class
True
False
None
and
or
not
```

We cannot normally use a keyword as a variable name.

❌ Incorrect:

```
if = 10
```

3.2 Identifiers

An **identifier** is the name given to a variable, function, class, etc.

Examples:

```
name = "Haris"
age = 20
total_marks = 450
```

Here:

- `name`
- `age`
- `total_marks`

are identifiers.

Rules for Identifiers

1. An identifier can contain letters, digits and underscore `_`.
2. It cannot start with a digit.
3. It cannot contain spaces.
4. It cannot be a Python keyword.
5. Python identifiers are case-sensitive.

Examples

✅ Valid:

```
name
student_name
marks1
_total
```

❌ Invalid:

```
1name
student name
student-name
class
```

Case Sensitivity

Python treats uppercase and lowercase names as different.

```
name = "Haris"
Name = "Ali"

print(name)
print(Name)
```

Both variables are different.

3.3 Literals

A **literal** is a fixed value written directly in a Python program.

Examples:

```
10
25.5
"Hello"
True
```

Common types of literals:

- Integer literals → `10`, `25`, `-5`
- Floating-point literals → `10.5`, `3.14`
- String literals → `"Python"`, `'Hello'`
- Boolean literals → `True`, `False`

3.4 Operators

Operators are symbols or keywords used to perform operations.

Examples:

```
+
-
*
/
>
<
==
and
or
```

Example:

```
a = 10
b = 5

print(a + b)
```

Output:

```
15
```

3.5 Punctuators / Delimiters

Punctuators are symbols used to structure Python programs.

Examples:

```
( )    [ ]    { }
:      ,      .
```

Example:

```
numbers = [10, 20, 30]
```

---

4. Concept of an Object

In Python, **everything is an object**.

An object is a piece of data stored in memory.

Every object has:

- **Identity** – unique identity of the object
- **Type** – type/class of the object
- **Value** – actual data stored in the object

Example

```
x = 25
```

Here:

- `x` → reference/name
- `25` → value
- `int` → type of object

We can check the type using:

```
type(x)
```

Output:

```
<class 'int'>
```

We can also check the identity using:

```
id(x)
```

The value returned by `id()` represents the object's identity during its lifetime.

---

5. Variables and Data Types

5.1 Variables

A **variable** is a name that refers to an object/value in Python.

Example:

```
age = 20
name = "Haris"
height = 5.8
```

Here:

- `age` refers to `20`
- `name` refers to `"Haris"`
- `height` refers to `5.8`

Python does not require us to declare the data type separately.

Example

```
x = 10
print(x)

x = "Python"
print(x)
```

Output:

```
10
Python
```

The same variable can refer to objects of different types at different times.

5.2 Common Python Data Types

Integer – `int`

Used for whole numbers.

```
age = 20
marks = 95
```

Float – `float`

Used for decimal numbers.

```
price = 99.50
height = 5.8
```

Complex – `complex`

Used for complex numbers.

```
z = 3 + 4j
```

A complex number has:

- Real part → `3`
- Imaginary part → `4j`

Boolean – `bool`

Represents either:

```
True
False
```

Example:

```
is_student = True
```

String – `str`

Used to store text.

```
name = "Haris"
city = 'Delhi'
```

5.3 Usage of Python Data Types

Data Type	Example	Used For
`int`	`25`	Whole numbers
`float`	`25.5`	Decimal numbers
`complex`	`2+3j`	Complex numbers
`bool`	`True`	Logical values
`str`	`"Python"`	Text

Example

```
student_name = "Haris"
age = 20
```

```
percentage = 87.5
passed = True
```

5.4 `type()` Function

The `type()` function is used to determine the data type/class of an object.

Syntax

```
type(object)
```

Example

```
x = 25
y = 10.5
name = "Haris"
status = True

print(type(x))
print(type(y))
print(type(name))
print(type(status))
```

Output

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

5.5 Multiple Assignments

Python allows us to assign values to multiple variables in a single statement.

Example 1

```
a, b, c = 10, 20, 30

print(a)
print(b)
print(c)
```

Output:

```
10
20
30
```

Example 2 – Same Value to Multiple Variables

```
x = y = z = 100

print(x)
print(y)
print(z)
```

Output:

```
100
100
100
```

5.6 Variable Names

Variable names should be meaningful and easy to understand.

Good Examples

```
student_name = "Haris"
total_marks = 450
student_age = 18
```

Poor Examples

```
x = "Haris"
a = 450
n = 18
```

Although these may be valid, meaningful names improve readability.

Important Rules

- Use meaningful names.
- Use lowercase letters for normal variables.
- Use `_` to separate words.
- Do not start with a digit.
- Do not use keywords.

Example:

```
total_marks = 500
student_name = "Rahul"
```

---

6. Mutable and Immutable Data Types

Mutable Data Type

An object whose value can be changed after it is created is called **mutable**.

Example:

```
my_list = [10, 20, 30]
my_list[0] = 100

print(my_list)
```

Output:

```
[100, 20, 30]
```

Lists are mutable.

Immutable Data Type

An object whose value cannot be changed after it is created is called **immutable**.

Examples include:

- `int`
- `float`
- `bool`
- `str`
- `tuple`

Example:

```
x = 10
```

We cannot directly modify the integer object `10`.

Instead, assigning a new value makes `x` refer to another object:

```
x = 20
```

Quick Comparison

Mutable	Immutable
Can be changed after creation	Cannot be changed after creation
Example: `list`	Example: `int`
Example: `dict`	Example: `str`
Example: `set`	Example: `tuple`

> **Remember:** Mutability is a property of the object, not simply the variable name.

7. Keywords in Python

Keywords are reserved words that have predefined meanings in Python.

Examples:

```
and       as        assert
break     class     continue
def       del       elif
else      except    False
finally   for       from
global    if        import
in        is        lambda
None      not       or
pass      raise     return
True      try       while
with      yield
```

Example

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

Here, `if` is a keyword.

We should not use keywords as variable names.

❌ Wrong:

```
for = 10
```

8. Expressions, Type Conversion & Operators

8.1 Expressions

An **expression** is a combination of values, variables, operators and function calls that produces a result.

Example:

```
a = 10
b = 20

result = a + b
print(result)
```

Output:

```
30
```

Here:

`a + b`
is an expression.

Other examples:

```
10 + 5
x * 2
marks >= 40
a and b
```

8.2 Type Conversion

Type conversion means converting a value from one data type to another.

Python provides functions such as:

```
int()
float()
str()
bool()
```

String to Integer

```
x = "25"
y = int(x)

print(y)
print(type(y))
```

Output:

```
25
<class 'int'>
```

Integer to Float

```
x = 10
y = float(x)

print(y)
```

Output:

```
10.0
```

Number to String

```
x = 100
y = str(x)

print(y)
print(type(y))
```

Output:

```
100
<class 'str'>
```

Important Example

The `input()` function returns data as a string.

```
age = input("Enter your age: ")
print(type(age))
```

Even if the user enters:

```
20
```

the type will normally be:

```
<class 'str'>
```

Therefore, for numerical calculations:

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

---

9. Python Operators

Operators are used to perform operations on values and variables.

Python provides several types of operators.

9.1 Arithmetic Operators

Used for mathematical calculations.

Operator	Meaning	Example	Result
`+`	Addition	`10 + 3`	`13`
`-`	Subtraction	`10 - 3`	`7`
`*`	Multiplication	`10 * 3`	`30`
`/`	Division	`10 / 3`	`3.333...`
`//`	Floor Division	`10 // 3`	`3`
`%`	Modulus	`10 % 3`	`1`

Operator	Meaning	Example	Result
`**`	Exponentiation	`10 ** 2`	`100`

Example

```
a = 10
b = 3

print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a // b)
print(a % b)
print(a ** b)
```

9.2 Relational / Comparison Operators

These operators compare two values and return:

```
True
False
```

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

Example

```
a = 10
b = 20

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

Output:

```
False
True
False
```

Important

Do not confuse:

`=`
with:

`==`
`=` → Assignment operator

`==` → Comparison operator

Example:

```
x = 10
print(x == 10)
```
Output:

```
True
```

9.3 Logical Operators

Logical operators are used to combine conditions.

Python has three logical operators:

`and`

Returns `True` only when both conditions are true.

```
age = 20

print(age >= 18 and age <= 25)
```
Output:

```
True
```

`or`

Returns `True` if at least one condition is true.

```
age = 20

print(age < 18 or age > 18)
```
Output:

```
True
```

`not`

Reverses the Boolean result.

```
x = True

print(not x)
```

Output:

```
False
```

Truth Table

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

For `not`:

A	not A
True	False
False	True

9.4 Shorthand / Augmented Assignment Operators

These operators combine an arithmetic operation with assignment.

Operator	Equivalent To
`+=`	`x = x + value`
`-=`	`x = x - value`
`*=`	`x = x * value`
`/=`	`x = x / value`
`//=`	`x = x // value`
`%=`	`x = x % value`
`**=`	`x = x ** value`

Example

```
x = 10

x += 5
print(x)
```

Output:

```
15
```

It is equivalent to:

```
x = x + 5
```

9.5 Membership Operators

Membership operators check whether a value exists inside a sequence such as a string, list, tuple, etc.

Python provides:

```
in
not in
```

Example

```
name = "Haris"

print("H" in name)
print("z" in name)
```

Output:

```
True
False
```

`not in`

```
name = "Haris"

print("z" not in name)
```

Output:

```
True
```

9.6 Identity Operators

Identity operators are used to check whether two variables refer to the **same object**.

Python provides:

```
is
is not
```

Example

```
a = [10, 20]
b = a

print(a is b)
```

Output:

```
True
```

Here, both `a` and `b` refer to the same list object.

`is not`

```
a = [10, 20]
b = [10, 20]

print(a is not b)
```

Output:

```
True
```

The two lists have the same values, but they are separate objects.

Important Difference

`==`

checks **whether values are equal**.

`is`

checks **whether two references point to the same object**.

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)
print(a is b)
```

Output:

```
True
False
```

🧠 Quick Revision

Python Character Set

Characters that can be used in Python programs.

Token

Smallest individual unit of a Python program.

Identifier

Name given to variables, functions, classes, etc.

Literal

Fixed value written directly in a program.

Object

A data entity in Python having identity, type and value.

Variable

A name/reference used to refer to an object.

`type()`

Used to determine the type of an object.

Mutable

Object whose contents can be changed.

Immutable

Object whose contents cannot be changed after creation.

Keyword

Reserved word with a special meaning in Python.

Expression

Combination of values, variables, operators and other elements that produces a result.

Type Conversion

Changing a value from one data type to another.

`=`

Assignment

`==`

Equality comparison

`is`

Identity comparison

`in`

Membership checking

💻 Practice Questions

Basic

1. What is a character set?
2. What is a token?
3. Define identifier and literal.
4. What is an object in Python?
5. What is a variable?
6. Write any five Python data types.
7. What is the use of `type()`?
8. What is multiple assignment?
9. What are mutable and immutable data types?
10. What are Python keywords?

Programming

Q1. Create variables to store:

- Your name
- Age
- Percentage
- Whether you are a student

Then print their values and data types.

Q2. Take two numbers and display:

- Addition
- Subtraction
- Multiplication
- Division
- Floor division
- Remainder
- Power

Q3. Predict the output:

```
x = 10
x += 5
x *= 2

print(x)
```

Q4. Predict the output:

```
a = 10
b = 20

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

Q5. Predict the output:

```
name = "Python"

print("P" in name)
print("z" in name)
print("x" not in name)
```

Q6. Explain the difference:

```
=
==
is
in
```

🎤 Viva Questions

1. What is Python?
2. What is a token?

3. What is an identifier?
4. Can an identifier start with a number?
5. Is Python case-sensitive?
6. What is a variable?
7. Does Python require explicit variable declaration?
8. What does `type()` return?
9. What is multiple assignment?
10. What is the difference between mutable and immutable objects?
11. Name five Python keywords.
12. What is an expression?
13. What is type conversion?
14. What is the difference between `/` and `//`?
15. What is the difference between `=` and `==`?
16. What is the difference between `==` and `is`?
17. What are membership operators?
18. Name the three logical operators.
19. What does the `%` operator do?
20. What does `**` mean in Python?

✅ Lecture 1 Summary

In this lecture, we covered:

✔ Introduction to Python ✔ Python Character Set ✔ Tokens ✔ Keywords ✔ Identifiers ✔ Literals ✔ Operators ✔ Concept of Objects ✔ Variables ✔ Python Data Types ✔ `type()` Function ✔ Multiple Assignments ✔ Variable Naming Rules ✔ Mutable and Immutable Data Types ✔ Expressions ✔ Type Conversion ✔ Arithmetic Operators ✔ Relational Operators ✔ Logical Operators ✔ Augmented Assignment Operators ✔ Membership Operators ✔ Identity Operators

⭐ Remember

> **Variable → refers to an Object → Object has Type and Value**

And remember these important operators:

```
=       Assignment
==      Comparison
is      Identity
in      Membership
```

📌 Next Lecture

Lecture 2 – Programming, Functions & Debugging

We will cover:

- User Input
- User-defined Functions
- Indentation
- Rules and Conventions for Python Programs
- Comments
- Debugging
- Syntax Errors
- Run-time Errors
- Logical Errors
- Practical Python Programs

📘 Chapter 4 – Python Programming Fundamentals

Lecture 2: User Input, Functions, Indentation, Comments & Debugging

---

🎯 Learning Objectives

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

- Take input from the user using `input()`.
- Perform type conversion with user input.
- Understand user-defined functions.
- Define and call functions using `def`.
- Understand parameters, arguments and `return`.
- Understand indentation in Python.
- Follow basic rules and conventions for writing Python programs.
- Use comments in Python.
- Understand debugging.
- Identify Syntax, Run-time and Logical errors.
- Differentiate between different types of errors.

4.10 User Input

A Python program can take data from the user using the **`input()`** function.

Syntax

```
variable = input("Message")
```

Example

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

Sample Output

```
Enter your name: Haris
Hello Haris
```

Important Point About `input()`

The `input()` function **normally returns the entered value as a string (`str`)**.

Example:

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

print(age)
print(type(age))
```

If the user enters:

```
20
```

Output:

```
20
<class 'str'>
```

Therefore, if we want to perform numerical calculations, we need **type conversion**.

Taking Integer Input

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

print(age)
print(type(age))
```

If the user enters:

```
20
```

Output:

```
20
<class 'int'>
```

Taking Float Input

```
price = float(input("Enter the price: "))

print(price)
print(type(price))
```

Taking Two Numbers

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

print("Sum =", a + b)
```

Sample Output

```
Enter first number: 10
Enter second number: 20
Sum = 30
```

Taking Multiple Inputs

We can take multiple values in one line using `split()`.

```
a, b = input("Enter two numbers: ").split()

print(a)
print(b)
```

> **Note:** Values obtained using `input()` are strings unless explicitly converted.

For integer values:

```
a, b = map(int, input("Enter two numbers: ").split())

print(a + b)
```

Sample Input

```
10 20
```

Output

```
30
```

4.11 User-defined Functions

A **function** is a block of reusable code designed to perform a particular task.

Python provides many built-in functions such as:

```
print()
input()
type()
len()
```

We can also create our own functions. These are called **user-defined functions**.

Why Use Functions?

Functions help us to:

- Reuse code
- Reduce repetition
- Organize programs
- Make programs easier to understand
- Make debugging easier

Creating a Function

A user-defined function is created using the `def` keyword.

Syntax

```
def function_name():
    statements
```

Example

```
def greet():
    print("Hello Students!")

greet()
```

Output

```
Hello Students!
```

Here:

- `def` → keyword used to define a function
- `greet` → function name
- `()` → parameter list
- `:` → indicates the beginning of the function body
- `greet()` → function call

Function Definition and Function Call

Function Definition

The code used to create a function is called its **definition**.

```
def welcome():
    print("Welcome to Python")
```

Function Call

Using the function is called a **function call**.

```
welcome()
```

Complete Example

```
def welcome():
    print("Welcome to Python")

welcome()
welcome()
```

Output

```
Welcome to Python
Welcome to Python
```

The same function can be called multiple times.

Function with Parameters

A function can receive values through **parameters**.

Example

```
def greet(name):
    print("Hello", name)

greet("Haris")
```

Output

```
Hello Haris
```

Here:

- `name` → parameter
- `"Haris"` → argument

Parameter vs Argument

Parameter	Argument
Variable written in function definition	Actual value passed during function call
Example: `name`	Example: `"Haris"`

Example:

```
def square(n):
    print(n * n)

square(5)
```

Here:

- `n` → parameter
- `5` → argument

Output:

```
25
```

Function with Multiple Parameters

```
def add(a, b):
    print(a + b)

add(10, 20)
```

Output

```
30
```

Here:

- `a` and `b` → parameters
- `10` and `20` → arguments

`return` Statement

The `return` statement is used to send a result back to the place where the function was called.

Example

```
def add(a, b):
    return a + b

result = add(10, 20)

print(result)
```

Output

```
30
```

`print()` vs `return`

`print()` displays a value on the screen.

`return` sends a value back from the function.

Example:

```
def square(n):
    return n * n

x = square(5)

print(x)
```

Output:

```
25
```

Example: Function to Find Even or Odd

```python
def check_even_odd(n):
    if n % 2 == 0:
        return "Even"
    else:
        return "Odd"

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

result = check_even_odd(number)

print(result)
```

Sample Output

```
Enter a number: 7
Odd
```

4.12 Indentation in Python

Indentation means the spaces at the beginning of a line of code.

Python uses indentation to define a **block of code**.

Unlike many other programming languages, indentation is an important part of Python syntax.

Example

```python
age = 20

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

The `print()` statement belongs to the `if` block because it is indented.

Standard Indentation

The commonly used indentation is **4 spaces**.

Example:

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

Function Indentation

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

Both `print()` statements are part of the function because they are indented.

Incorrect Indentation

❌ Example:

```
if age >= 18:
print("Adult")
```

This will produce an indentation-related error because the statement inside the `if` block is not properly indented.

Nested Indentation

Indentation is also used for nested blocks.

```
if age >= 18:
    if age >= 60:
        print("Senior Citizen")
    else:
        print("Adult")
```

Each level of the block has its own indentation.

Remember

> **In Python, indentation is not just for readability; it is part of the syntax.**

4.13 Rules and Conventions for Writing Python Programs

Writing clean and readable code is an important programming practice.

Important Rules

1. Use Meaningful Names

❌ Poor:

```
x = 500
```

✅ Better:

```
total_marks = 500
```

---

2. Follow Proper Indentation

Use consistent indentation.

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

---

3. Use Meaningful Function Names

❌ Poor:

```
def f():
```

✅ Better:

```
def calculate_total():
```

---

4. Use Snake Case for Variables and Functions

Examples:

```
student_name
total_marks
calculate_average()
```

---

5. Avoid Unnecessary Code

Write only the code that is required to solve the problem.

---

6. Use Comments Where Helpful

Comments can explain complicated or important parts of the program.

```
# Calculate the total marks
total = marks1 + marks2
```

---

7. Maintain Consistent Spacing

Example:

```
a = 10
b = 20
result = a + b
```

This is easier to read than:

```
a=10
b=20
result=a+b
```

---

8. Keep Programs Readable

A good program should be:

- Simple
- Clear
- Well-structured
- Properly indented
- Easy to understand

4.14 Comments

A **comment** is a part of a program that is ignored by the Python interpreter.

Comments are written to explain the code.

They improve the **readability and understanding** of programs.

Single-line Comment

A single-line comment begins with `#`.

Example

```
# This program calculates the sum
a = 10
b = 20

print(a + b)
```

Everything after `#` on that line is treated as a comment.

Comment After a Statement

```
x = 10  # Store the value 10 in x
```

Multiple-line Explanations

Python does not have a separate multi-line comment syntax using a special comment symbol.

For multiple lines of comments, `#` can be used on each line:

```
# This program calculates
# the area of a rectangle
# using length and breadth.
```

Triple-quoted Strings

Triple-quoted strings can span multiple lines:

```
"""
This is a multi-line string.
It can contain multiple lines.
"""
```

Triple-quoted strings are commonly used for **docstrings** when placed appropriately inside functions, classes or modules.

Example:

```
def greet():
    """This function displays a greeting."""
    print("Hello")
```

4.15 Debugging

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

Errors in programs can prevent the program from running correctly or can cause it to produce incorrect results.

Basic Debugging Process

```
Write Program
     ↓
Run Program
     ↓
Find Error
     ↓
Understand Error
     ↓
Correct Error
     ↓
Run Again
```

Types of Errors

According to this chapter, we mainly study:

1. Syntax Error
2. Run-time Error
3. Logical Error

4.15.1 Syntax Error

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

In simple words:

> The program is not written according to Python's syntax rules.

Example

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

The colon `:` is missing.

Python will report a syntax error.

Another Example

```
print("Hello"
```

The closing parenthesis is missing.

Common Causes

- Missing colon `:`
- Missing brackets
- Incorrect syntax
- Incorrect indentation
- Misspelled keywords

4.15.2 Run-time Error

A **run-time error** occurs while the program is executing.

The program may start successfully but stop because of an error during execution.

Example: Division by Zero

```
a = 10
b = 0

print(a / b)
```

This produces:

```
ZeroDivisionError
```

Example: Invalid Integer Conversion

```
age = int("abc")
```

This produces:

```
ValueError
```

The syntax is correct, but an error occurs while the program is running.

4.15.3 Logical Errors

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

Example

Suppose we want to calculate the average of two numbers.

Incorrect program:

```
a = 10
b = 20

average = a + b / 2

print(average)
```

The program runs, but the result is incorrect because the intended calculation should be:

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

Output of Correct Program

```
15.0
```

There may be **no error message** for a logical error.

The programmer must identify the incorrect result and correct the logic.

🔍 Difference Between Types of Errors

Error Type	When It Occurs	Result
Syntax Error	Python syntax is incorrect	Program cannot execute normally
Run-time Error	During program execution	Program stops/raises an exception
Logical Error	Program logic is incorrect	Program runs but gives wrong result

Easy Way to Remember

> **Syntax Error → Wrong Python Grammar Run-time Error → Problem While Running Logical Error → Wrong Answer**

🧪 Practical Examples

Example 1 – User Input and Addition

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

sum = a + b

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

Sample Output

```
Enter first number: 15
Enter second number: 25
Sum = 40
```

Example 2 – Function to Calculate Square

```
def square(n):
    return n * n

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

print("Square =", square(number))
```

Sample Output

```
Enter a number: 5
Square = 25
```

Example 3 – Function to Calculate Simple Interest

Formula:

SI = (P × R × T) / 100

```
def simple_interest(p, r, t):
    return (p * r * t) / 100

p = float(input("Enter principal: "))
r = float(input("Enter rate: "))
t = float(input("Enter time: "))

si = simple_interest(p, r, t)

print("Simple Interest =", si)
```

🧠 Quick Revision

`input()`

Used to take input from the user.

```
name = input("Enter name: ")
```

`def`

Used to define a function.

```
def greet():
    print("Hello")
```

Function Call

Used to execute a function.

```
greet()
```

Parameter

Variable written in a function definition.

```
def square(n):
```

Here, `n` is a parameter.

Argument

Actual value passed during a function call.

```
square(5)
```
Here, `5` is an argument.

`return`

Sends a value back from a function.

```
def add(a, b):
    return a + b
```

Indentation

Spaces at the beginning of a line used to define blocks of code.

Comment

Text ignored by the interpreter and used to explain code.

```
# This is a comment
```

Debugging

Process of finding and correcting errors.

🎯 Important Exam Questions

Short Answer Questions

1. What is the use of the `input()` function?
2. What type of value does `input()` normally return?
3. How can we take an integer as input?
4. What is a user-defined function?
5. What is the purpose of the `def` keyword?
6. Differentiate between a parameter and an argument.
7. What is the purpose of the `return` statement?
8. What is indentation?
9. Why is indentation important in Python?
10. What are comments?
11. What is debugging?
12. Define syntax error.

13. Define run-time error.
14. Define logical error.
15. Differentiate between syntax, run-time and logical errors.

💻 Programming Practice

Q1. User Input

Write a program to input two numbers and display their sum, difference and product.

Q2. User Details

Write a program to input:

- Name
- Age
- Percentage

and display all the details.

Q3. Function

Write a function `square()` that accepts a number and returns its square.

Q4. Function with Multiple Parameters

Write a function `maximum()` that accepts two numbers and returns the greater number.

Q5. Simple Interest

Write a function to calculate simple interest.

Q6. Debugging

Find and correct the errors:

```
def add(a,b)
    return a+b

print(add(10,20)
```

Q7. Logical Error

Find the logical error:

```
a = 10
b = 20

average = a + b / 2

print(average)
```

Write the corrected statement.

🎤 Viva Questions

1. What is the purpose of `input()`?
2. What is the default data type returned by `input()`?
3. Why do we use `int(input())`?
4. What is a function?
5. Why are functions useful?
6. Which keyword is used to define a function?
7. What is a function call?
8. What is a parameter?
9. What is an argument?
10. What is the purpose of `return`?
11. What is indentation?
12. How many spaces are commonly used for one indentation level?
13. What happens if indentation is incorrect?
14. What is a comment?
15. Which symbol is used for a single-line comment?
16. What is debugging?
17. What is a syntax error?
18. What is a run-time error?
19. What is a logical error?
20. Which error can produce a wrong answer without displaying an error message?

⭐ Chapter 4 – Complete Revision

After completing Lecture 1 and Lecture 2, you have covered the complete Chapter 4 syllabus:

Lecture 1

- 4.1 Introduction
- 4.2 Python Character Set
- 4.3 Tokens
- 4.4 Concept of an Object
- 4.5 Variables and Data Types
- 4.5.1 Usage of Python Data Types
- 4.5.2 `type()`
- 4.5.3 Multiple Assignments
- 4.5.4 Variable Names
- 4.6 Mutable and Immutable Data Types
- 4.7 Keywords in Python
- 4.8 Expressions
- 4.8.1 Type Conversion
- 4.9 Operators
- 4.9.1 Arithmetic Operators
- 4.9.2 Relational Operators
- 4.9.3 Logical Operators
- 4.9.4 Shorthand/Augmented Assignment Operators
- 4.9.5 Membership Operators
- 4.9.6 Identity Operators

Lecture 2

- 4.10 User Input
- 4.11 User-defined Functions
- 4.12 Indentation in Python
- 4.13 Rules and Conventions for Writing Python Programs
- 4.14 Comments
- 4.15 Debugging
- 4.15.1 Syntax Error
- 4.15.2 Run-time Error
- 4.15.3 Logical Errors

🏆 Final Takeaway

A basic Python program follows this general flow:

```
Input
  ↓
Processing
  ↓
Output
```

Functions help us organize and reuse the processing part.

Errors can be remembered as:

```
Syntax Error  → Program structure is wrong
Run-time Error → Problem occurs during execution
Logical Error → Program runs but gives wrong result
```

> 💡 **Good Python programming = Correct Syntax + Clear Logic + Proper Indentation + Readable Code**