Seed Programming

Python Fundamentals

Lecture 01

1 Introduction

Python is a simple, powerful, and beginner-friendly programming language. It is widely used in web development, artificial intelligence (AI), data science, automation, game development, and many other fields.

A Python program is a collection of instructions that are normally executed one by one, from top to bottom.

```
1 print("Hello")
2 print("Welcome to Python")
```

Output

```
Hello
Welcome to Python
```

2 Displaying Output with `print()`

The built-in `print()` function displays information on the screen. It can print text, numbers, values stored in variables, and the results of calculations.

2.1 Syntax and Basic Examples

The general syntax is `print(value)`.

```
1 print("Hello World")
2 print(100)
3 print(25 + 5)
```

Output

```
Hello World
100
30
```

A variable may also be passed to `print()`:

```
1 name = "Ali"
2 print(name)
```

Output

```
Ali
```

Remark. *Text must be enclosed in quotation marks, whereas numeric literals do not need them. By*

default, each call to `print()` *ends with a newline, so the next output begins on a new line.*

3 Pausing Execution with `sleep()`

The `sleep()` function pauses a program for a specified number of seconds. It is useful for loading screens, timed messages, simple animations, and giving a user time to read the current output.

3.1 Import, Syntax, and Example

`sleep()` belongs to Python's `time` module, so it must first be imported:

```
from time import sleep

print("Loading...")
sleep(3)
print("Completed!")
```

Output

```
Loading...
(wait for 3 seconds)
Completed!
```

The general form is `sleep(seconds)`; for example, `sleep(5)` pauses execution for five seconds.

4 Clearing the Console

The `system()` function can send a command to the operating system. In a Windows console, `system("cls")` clears old output before new information is displayed.

```
from os import system

print("Old Data")
system("cls")
print("New Data")
```

Remark. *The command is platform-specific: use* `cls` *on Windows and* `clear` *on Linux or macOS. A program may therefore call* `system("clear")` *on those systems.*

5 Variables and Initialization

A *variable* is a named storage location for data. *Initialization* means giving that variable its first value. The general syntax is

`variable_name = value`.

```
name = "Ali"
age = 20
marks = 95

print(name)
print(age)
print(marks)
```

Output

```
Ali
20
95
```

Variables let a program store, reuse, and update information. Instead of repeating a literal value, store it once and refer to its name:

```
name = "Ali"
print(name)
print(name)
print(name)
```

6 The Assignment Operator

The assignment operator `=` stores the value on its right in the variable on its left. It performs assignment, not comparison.

```
name = "Ali"
age = 20
marks = 90
```

Here, `"Ali"` is stored in `name`, `20` in `age`, and `90` in `marks`. A later assignment replaces the earlier value:

```
age = 20
age = 21
print(age)
```

Output

```
21
```

7 Reading User Input

The built-in `input()` function reads information entered by the user while the program is running. Its optional prompt tells the user what to enter.

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

Output

```
Enter your name:  Ali
Welcome Ali
```

Without input, a program normally works with fixed values written by the programmer. With `input()`, different data can be supplied each time the program runs.

7.1 Converting Input to an Integer

The `input()` function always returns a string (text), even when the user types digits. Use `int()` when the value is intended for integer arithmetic.

```
1 age = int(input("Enter your age: "))
2 print(age + 5)
```

Output

```
Enter your age:  20
25
```

Without conversion, the `+` operator joins strings:

```
1 num = input("Enter number: ")
2 print(num + num)
```

If the user enters `5`, the output is `55`. With conversion, addition is numeric:

```
1 num = int(input("Enter number: "))
2 print(num + num)
```

The same input now produces `10`.

Remark. *`int()` can only convert a valid integer representation. Input such as `"five"` causes a runtime error unless the program validates or handles it.*

8 Expressions

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

```
1 a = 5 + 3
2 b = 10 - 4
3 c = 5 * 2
4 d = 20 / 5
5
6 print(a)
7 print(b)
8 print(c)
9 print(d)
```

Output

```
8
6
10
4.0
```

9 Arithmetic Operators

Operator	Meaning	Example	Result
+	Addition	5 + 3	8
-	Subtraction	8 - 2	6
*	Multiplication	5 * 4	20
/	Division	20 / 4	5.0
%	Modulus	10 % 3	1
//	Floor division	10 // 3	3
**	Power	2 ** 3	8

```
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)
```

Remark. *The / operator performs true division and returns a floating-point value. The // operator performs floor division, while % returns the remainder.*

10 Comments

Comments are notes placed inside source code. Python ignores them during execution. They explain code, record reminders, and make programs easier for other programmers—and your future self—to understand.

10.1 Single-Line Comments

A hash symbol # begins a comment that continues to the end of the line.

```
# This is a comment
print("Hello")
```

10.2 Writing Comments Across Several Lines

Python has no separate multi-line comment syntax. The clearest approach is usually to place # at the beginning of each line:

```
# Student Management System
# Python Programming
# Lecture 1
print("Welcome")
```

Triple-quoted strings are also sometimes used for long explanatory text:

```
"""
Student Management System
```

```
Lecture 1
Python Programming
"""
print("Welcome")
```

Remark. *A standalone triple-quoted value is technically a string literal, not a comment. Triple-quoted strings are primarily used for documentation strings (docstrings), while repeated # markers are preferable for ordinary multi-line comments.*

11 Variable Naming Rules

Good names communicate a variable's purpose and make a program easier to read.

- A name may contain letters, digits, and underscores (_).
- A name cannot begin with a digit.
- Spaces and most special characters are not allowed.
- Python is case-sensitive: `age`, `Age`, and `AGE` are different names.
- Python keywords such as `if`, `class`, and `for` cannot be used as names.

Valid names	Invalid names
`student_name = "Ali"`	`1student = "Ali"`
`marks1 = 90`	`student name = "Ali"`
`total_marks = 500`	`class = "Python"`

12 Common Types of Errors

12.1 Syntax Errors

A syntax error occurs when code does not follow Python's grammar, so the program cannot be parsed correctly.

```
print("Hello"
```

The closing parenthesis is missing.

12.2 Runtime Errors

A runtime error occurs after the program has started executing.

```
print(10 / 0)
```

Division by zero is not allowed, so this statement raises a `ZeroDivisionError`.

12.3 Logical Errors

A logical error allows the program to run, but the result is incorrect because the algorithm or operation is wrong.

```
length = 5
width = 4
area = length + width
print(area)
```

The program prints 9, but a rectangle's area requires multiplication:

```
area = length * width
print(area)
```

The corrected output is 20.

13 Debugging

Debugging is the process of finding, understanding, and fixing errors in a program. When a program does not produce the expected output, we should first determine why the problem occurred instead of changing the code at random. After identifying the cause, we can correct it and test the program again.

13.1 Example: Printing a Multiplication Table

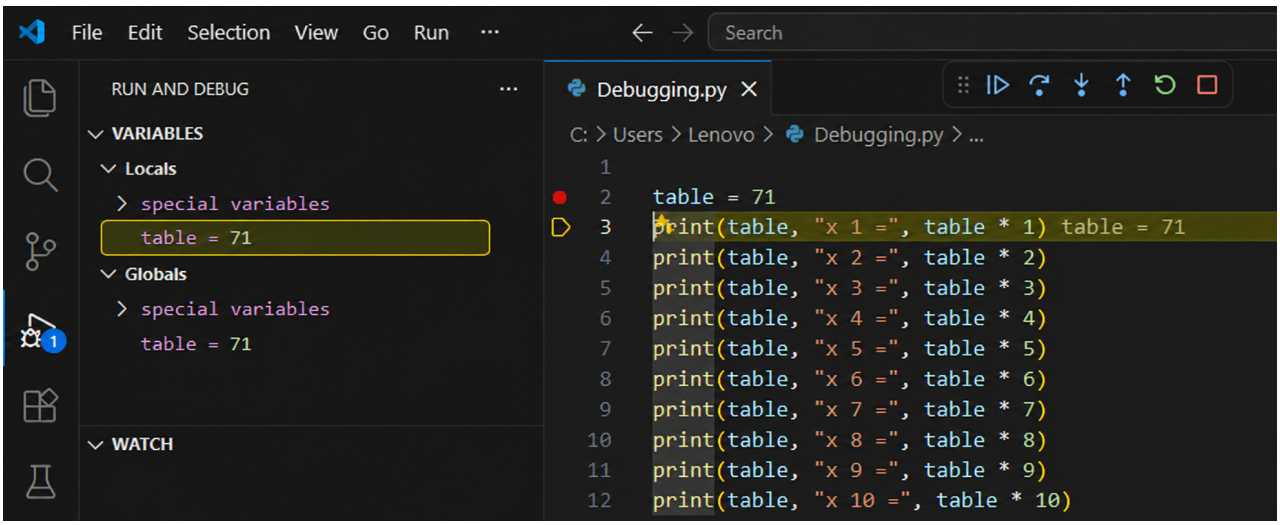
The following program prints the multiplication table of a fixed number:

```
table = 71
print(table, "x 1 =", table * 1)
print(table, "x 2 =", table * 2)
print(table, "x 3 =", table * 3)
print(table, "x 4 =", table * 4)
print(table, "x 5 =", table * 5)
print(table, "x 6 =", table * 6)
print(table, "x 7 =", table * 7)
print(table, "x 8 =", table * 8)
print(table, "x 9 =", table * 9)
print(table, "x 10 =", table * 10)
```

Output

```
71 x 1 = 71
71 x 2 = 142
71 x 3 = 213
71 x 4 = 284
71 x 5 = 355
71 x 6 = 426
71 x 7 = 497
71 x 8 = 568
71 x 9 = 639
71 x 10 = 710
```

The program works correctly because 71 is stored as an integer.



13.2 Making the Program Interactive

To let the user choose a table, we can replace the fixed value with `input()`:

```
table = input("Enter the table number: ")

print(table, "x 1 =", table * 1)
print(table, "x 2 =", table * 2)
print(table, "x 3 =", table * 3)
print(table, "x 4 =", table * 4)
print(table, "x 5 =", table * 5)
print(table, "x 6 =", table * 6)
print(table, "x 7 =", table * 7)
print(table, "x 8 =", table * 8)
print(table, "x 9 =", table * 9)
print(table, "x 10 =", table * 10)
```

If the user enters `5`, the result is unexpected:

Output

```
Enter the table number:  5
5 x 1 = 5
5 x 2 = 55
5 x 3 = 555
5 x 4 = 5555
5 x 5 = 55555
5 x 6 = 555555
5 x 7 = 5555555
5 x 8 = 55555555
5 x 9 = 555555555
5 x 10 = 5555555555
```

13.3 Finding the Cause

Python is not multiplying incorrectly; it is doing exactly what the program requests. The `input()` function always returns a string, so the entered value is stored as `"5"`, not as the integer `5`.

Therefore, the expression:

```
table * 2
```

means:

```
"5" * 2
```

Multiplying a string by an integer repeats the string:

```
print("5" * 2)
print("5" * 3)
```

Output

```
55
555
```

This explains the unexpected output and helps us identify the source of the logical error.

13.4 Fixing and Testing the Program

Convert the input to an integer before using it in a calculation. This can be done in one statement:

```
table = int(input("Enter the table number: "))
```

It can also be done in two steps:

```
table = input("Enter the table number: ")
table = int(table)
```

The corrected program is:

```
table = input("Enter the table number: ")
table = int(table)

print(table, "x 1 =", table * 1)
print(table, "x 2 =", table * 2)
print(table, "x 3 =", table * 3)
print(table, "x 4 =", table * 4)
print(table, "x 5 =", table * 5)
print(table, "x 6 =", table * 6)
print(table, "x 7 =", table * 7)
print(table, "x 8 =", table * 8)
print(table, "x 9 =", table * 9)
print(table, "x 10 =", table * 10)
```

For an input of `5`, the corrected output is:

Output

```
Enter the table number:  5
5 x 1 = 5
5 x 2 = 10
```

```
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

13.5 Let's Debug the Following Code in the Debugger

13.5.1 Code 1: Variable Value Change

```python
number = 10

print("Initial Value:", number)

number = number + 5

print("After Addition:", number)

number = number * 2

print("After Multiplication:", number)
```

Explanation

In this example, we use the Visual Studio Code Debugger to execute the program one line at a time. Instead of running the entire program at once, the debugger pauses after each line so that we can observe how the value of the variable changes.

As we press **F10 (Step Over)**, the debugger executes the current line and moves to the next one.

Step-by-Step Execution

Step 1. The first line creates a variable named `number` and stores the value `10` in it:

```
number = 10
```

The Variables panel now shows `number = 10`.

Step 2. The first `print()` statement is executed:

```
print("Initial Value:", number)
```

The Terminal displays `Initial Value: 10`. The value of `number` remains `10`.

Step 3. The following statement is executed:

```
number = number + 5
```

Python takes the current value, `10`, adds `5`, and stores the new value back in `number`. The Variables panel updates to `number = 15`.

Step 4. The next `print()` statement is executed:

```
print("After Addition:", number)
```

The Terminal displays `After Addition: 15`.

Step 5. The following statement is executed:

```
number = number * 2
```

Python multiplies the current value, `15`, by `2` and stores the result back in `number`. The Variables panel now shows `number = 30`.

Step 6. The final `print()` statement is executed:

```
print("After Multiplication:", number)
```

13.5.2 Code 2: If-Else Statement

```
marks = 65

print("Checking Result...")

if marks >= 50:
    print("Pass")
else:
    print("Fail")

print("Program Ended")
```

Explanation

In this example, we use the Visual Studio Code Debugger to understand how an `if-else` statement works. Instead of running the program all at once, we execute it one line at a time using **F10 (Step Over)** and observe how Python makes a decision based on a condition.

Step-by-Step Execution

Step 1. The first line creates a variable named `marks` and stores the value `65` in it:

```
1  marks = 65
```

The Variables panel now shows `marks = 65`.

Step 2. The first `print()` statement is executed:

```
1  print("Checking Result...")
```

The Terminal displays `Checking Result...`. The value of `marks` remains `65`.

Step 3. The debugger reaches the following condition:

```
1  if marks >= 50:
```

Python checks whether `marks` is greater than or equal to `50`. Since `65 >= 50` is `True`, Python enters the `if` block and skips the `else` block.

Step 4. The following statement is executed:

```
1 print("Pass")
```

The Terminal displays `Pass`.

Step 5. After the `if` block finishes, Python skips the `else` block and moves to the next statement after the `if-else` structure.

Step 6. The final `print()` statement is executed:

```
1 print("Program Ended")
```

The Terminal displays `Program Ended`. The program has now finished executing.

14 Useful VS Code Shortcuts

Editing		Navigation and Running	
`Ctrl + S`	Save file	`Ctrl + F`	Find
`Ctrl + /`	Comment or uncomment	`Ctrl + H`	Replace
`Ctrl + C`	Copy	`Ctrl + N`	New file
`Ctrl + X`	Cut	`Ctrl + O`	Open file
`Ctrl + V`	Paste	`Ctrl + Shift + P`	Command Palette
`Ctrl + Z`	Undo	`F5`	Run with debugging
`Ctrl + Y`	Redo	`Ctrl + F5`	Run without debugging
`Ctrl + A`	Select all	`Alt + Up/Down`	Move line
`Shift + Alt + Down`	Duplicate line		

Remark. *On macOS, many shortcuts use the Command key in place of Ctrl. Some shortcuts may also vary with the operating system or the user's VS Code keymap.*

15 Summary

- `print()` displays text, numbers, variables, and expression results.
- `sleep()` pauses execution; `system()` can clear a platform's console.
- Variables store reusable values, and `=` assigns or replaces those values.
- `input()` reads text; use `int()` when integer arithmetic is required.
- Expressions combine values and operators to produce results.
- Comments, meaningful variable names, and awareness of error types improve code quality.
- Debugging helps identify, understand, correct, and retest unexpected program behavior.