وزارة الاتصالات وتكنولوجيا المعلومات

مدارس we للتكنولوجيا التطبيقية

التكنولوجيا التطبيقية
APPLIED TECHNOLOGY
وزارة التربية والتعليم والتعليم الفني

Programming fundamental

WE Applied Technology Schools

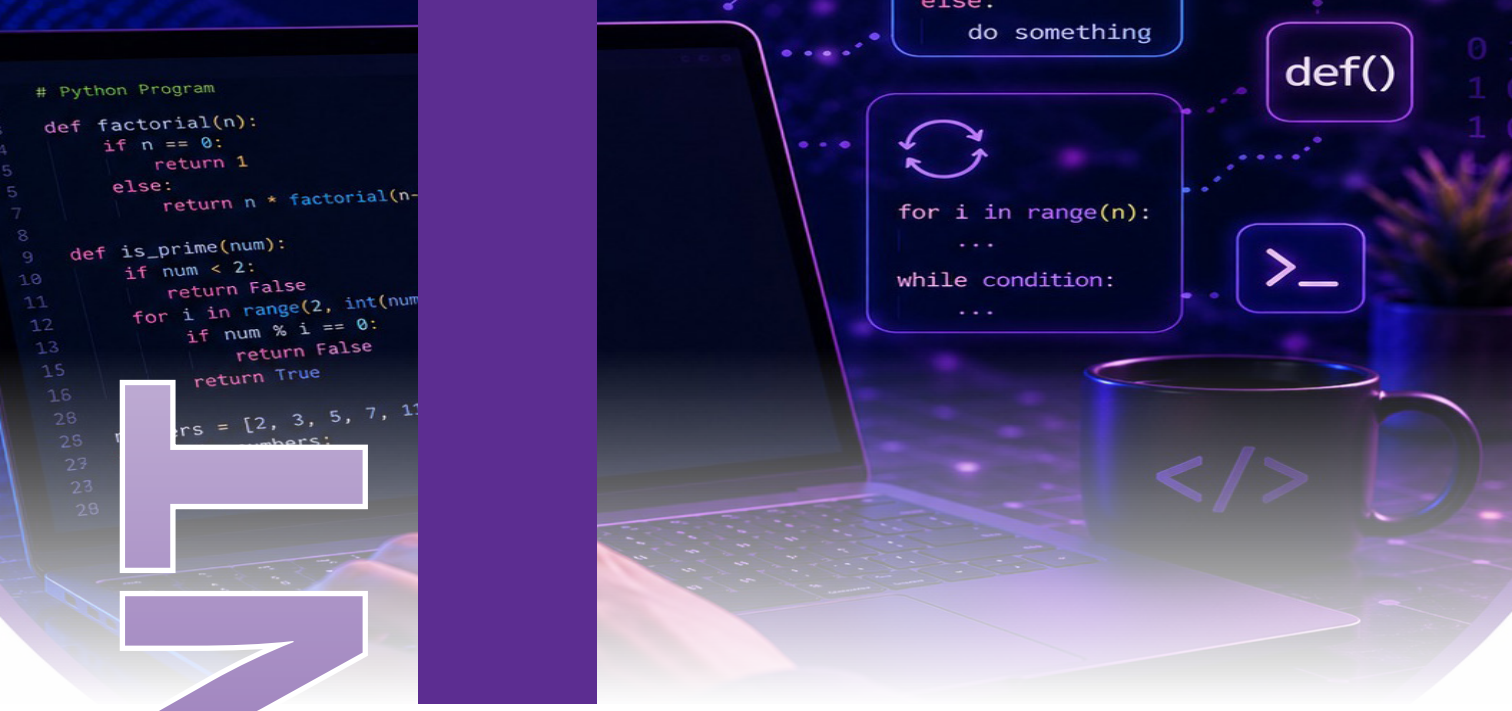
Grade 1
2027



Unit 2

Programming fundamental

WE Applied Technology Schools



CONTENT

01 Introduction to Software and Programming Concepts

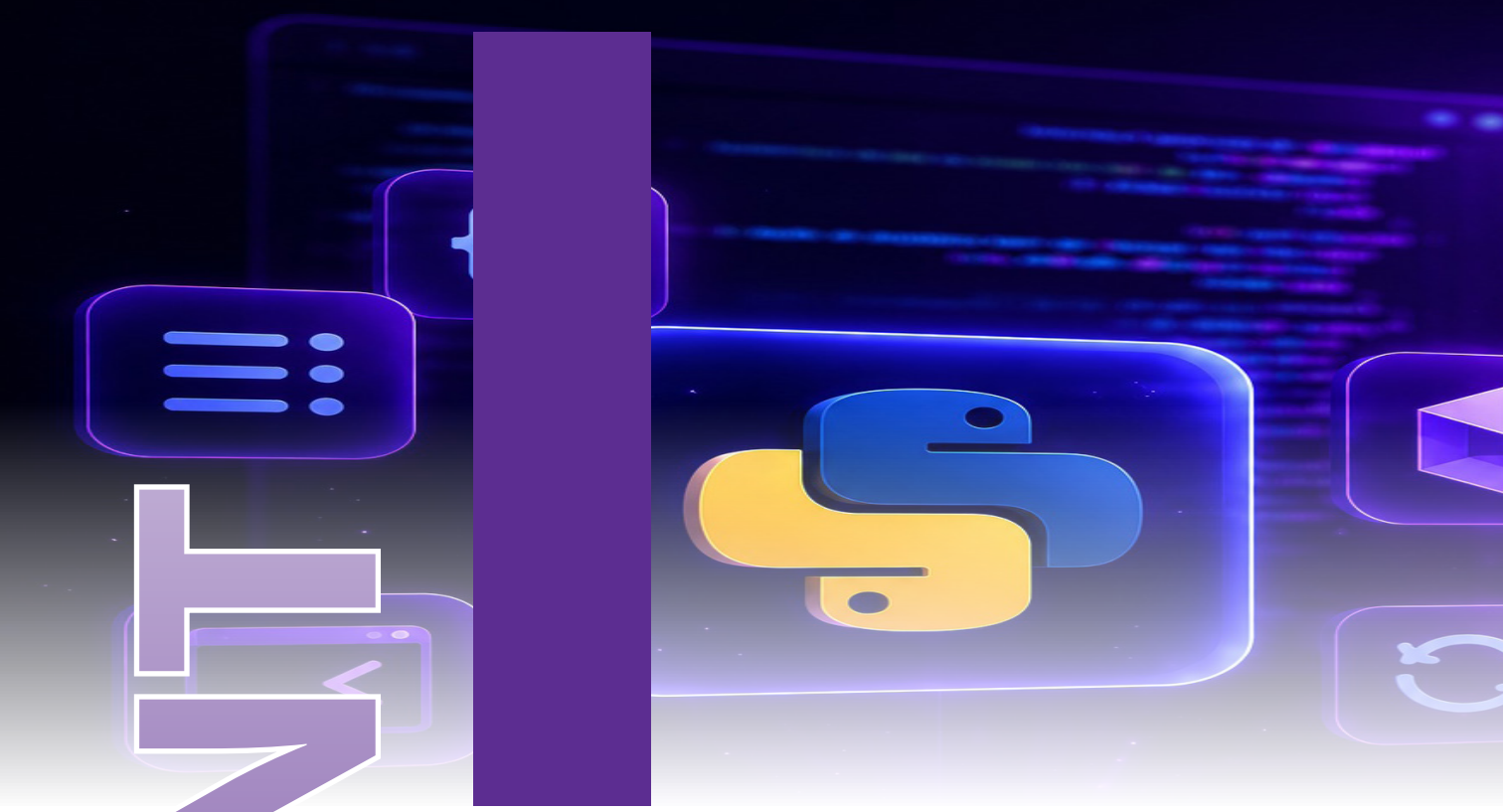
- Item 1 : Programming Languages 10
- Item 2 : Algorithms and Flowchart 12
- Item 3 : Introduction to python 13
- Item 4 : Setting up a python Environment 14
- Item 5 : Python programming structure 18
- Item 6 : Output 21
- Item 7 : User Input 22
- Lesson 1 Assessment 24

02 Python Fundamentals and Core Programming Logic

- item 1: Data Type 27
- item 2: Python Operators 30
- Lesson 2 Assessment 38

03 Control Structures and Program Flow in Python

- item 1:Conditions 41
- item 2: Loops 45
- item 3: Loop Control Statements: “break” 49
- Item 4: Loop Control Statements: “continue” 50
- Lesson 3 Assessment 51



CONTENT

04 Data Structures and Functions in Python

- Item 1 : List 57
- Item 2 : Tuples 63
- Item 3 : Dictionaries 66
- Item 4 : Function in Python 72
- Lesson 4 Assessment 76

05 Interactive Python Programs and Real-World Applications

- Item 1: Object-Oriented Programming (OOP) 80
- item 2: Functional Programming (FP) 81
- Item 3: From Scripts to Software 81
- Item 4:Types of Errors in Python 84
- Lesson 5 Assessment 88

Unit Purpose

Have you ever wondered how games keep score, how apps respond to your actions, or how computers follow instructions so accurately?

In this unit, you will discover the basics of programming using Python. You will learn how to create programs, work with data, make decisions, repeat tasks, and solve problems step by step.

Before you begin, remember that every great software application starts with a few simple lines of code. By learning the fundamentals of programming, you will be able to build your own programs and bring your ideas to life.

Learning Outcomes

By the end of the unit, you will be able to:

- Explain fundamental Python concepts, including variables, data types, operators, and basic program structure.
- Use input and output statements to collect, display, and process information effectively.
- Apply conditional statements to create programs that make decisions based on different situations.
- Design loops that automate repetitive tasks and improve program efficiency.
- Organize and manage data using lists, tuples, and dictionaries.
- Create reusable functions that simplify code and support problem-solving.
- Combine programming concepts to develop complete, logical, and practical Python applications

Unit Title	Programming Fundamental	
Knowledge	TK02	Create simple interactive programs and explaining the interaction between software, hardware, and networks.
	TK03	apply electrical and ICT safety principles to create a safe and compliant technical working environment.
	TK05	Understand systematic testing and verification methods to identify faults .
Skills	PC1.1	Classify computer hardware components and software types.
	PC1.2	Develop simple interactive programs by applying programming logic concepts .
	PC2.1	Apply safety regulations and workplace rules during electronics, ICT, and telecom activities.
	PC2.3	Configure basic computer, network, and programming solutions to support business and technical functions.

The Smart Assistant Project Roadmap:

To achieve these goals, we will move through five core lessons:

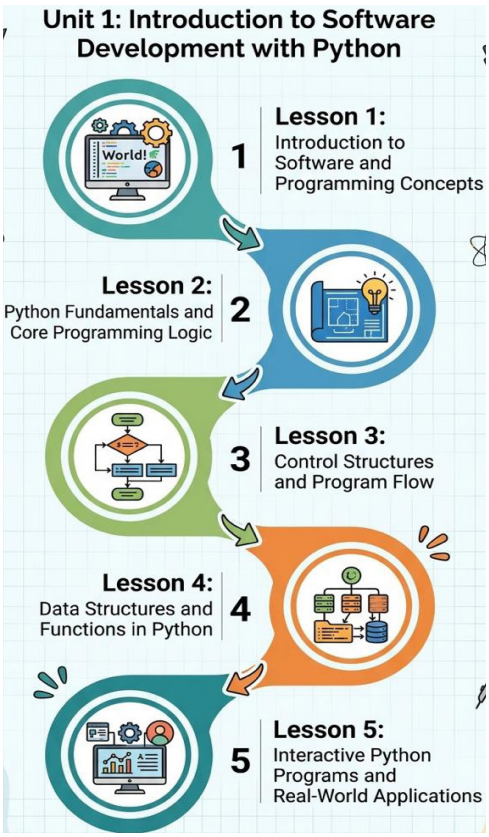
L1: **Python Fundamentals**
Building a strong foundation in programming concepts and Python syntax.

L2: **Variables, Data Types & Operators**
How programs store, process, and manipulate different kinds of data.

L3: **Input, Output & Decision Making**
Creating interactive programs that respond intelligently to user input.

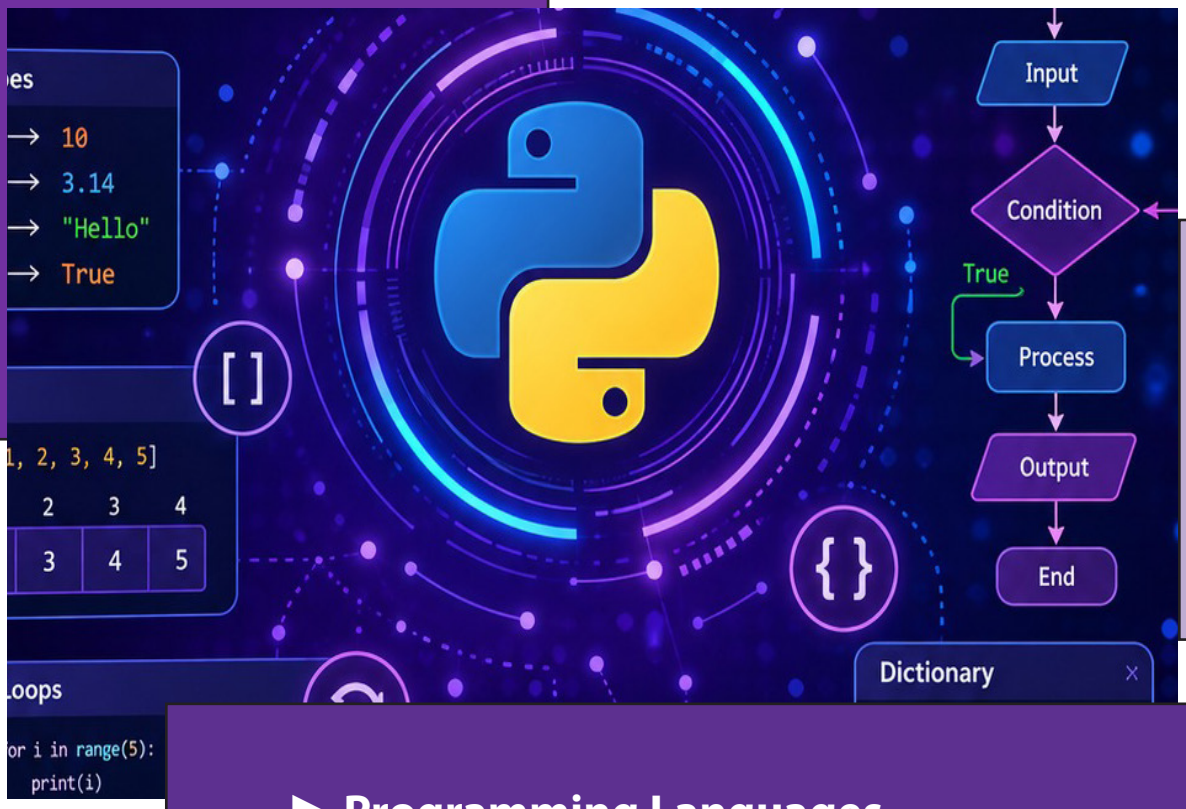
L4: **Loops & Functions**
Automating repetitive tasks and organizing code into reusable components.

L5: **Data Collections in Python**
Managing and organizing information using lists, tuples, and dictionaries



Lesson 1

Introduction to Software and Programming Concepts



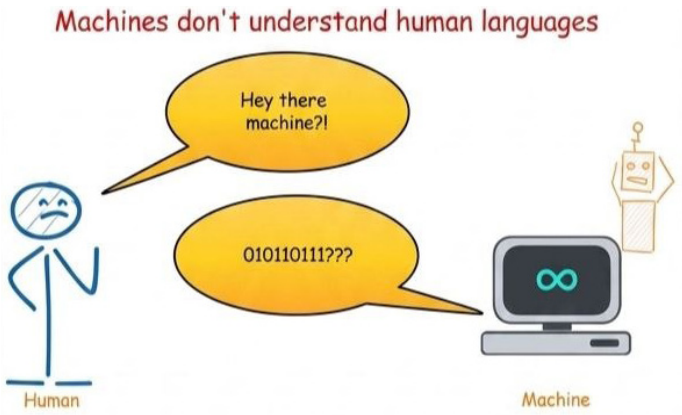
- Programming Languages
- Algorithms and Flowchart
- Introduction to python
- Setting up a python Environment
- Python programming structure
- Output
- User Input

Introduction:

Programming is the process of creating instructions that tell a computer how to perform tasks and solve problems. These instructions are written using programming languages such as Python, which is one of the most popular and beginner-friendly programming languages in the world. Every application, website, game, and smart system we use today is built using programming. From mobile apps and robots to artificial intelligence systems, programming plays an important role in modern technology and daily life.



A computer cannot understand human language directly. Instead, programmers must write clear and organized instructions that the computer can follow step by step. This requires logical thinking, problem-solving skills, and attention to detail.



What Is a Program?

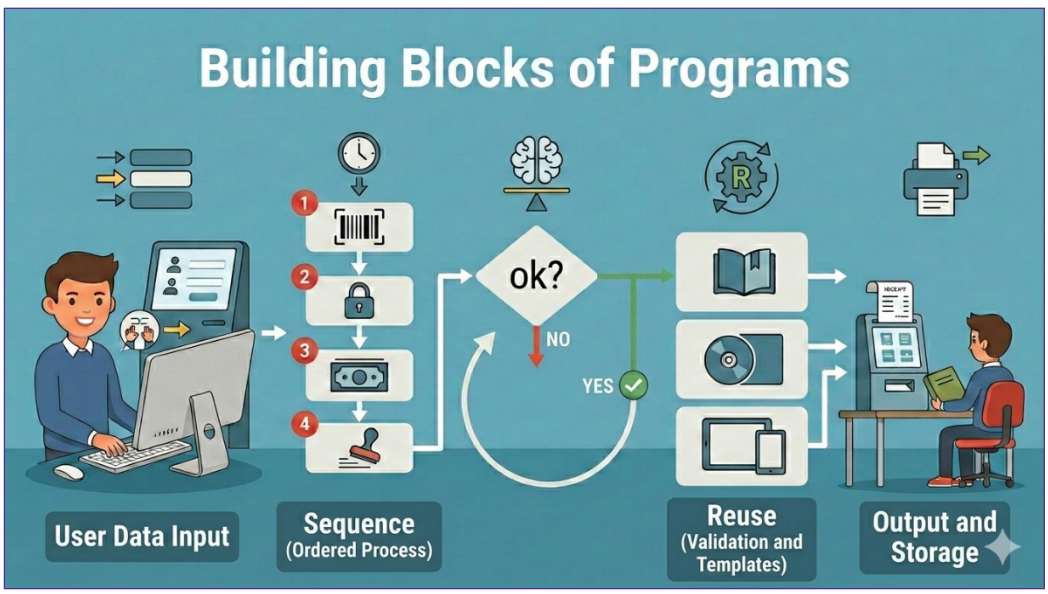
A program is a set of instructions written in a programming language that tells a computer what to do.
Programs help us solve problems and perform tasks automatically.
For example, instead of manually counting the most common word in thousands of social media posts, a program can do it quickly and accurately.
In simple terms, a program is a solution to a problem that a computer can understand and execute.

The Building Blocks of Programs

All programs are built using a few basic concepts:

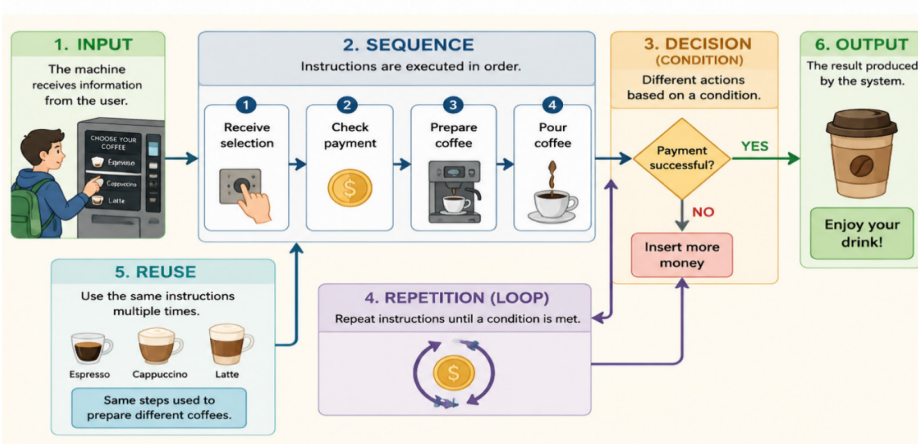
- **Input:** Receiving data from the user, a file, or another source.
- **Output:** Displaying results on the screen or saving them somewhere.
- **Sequence:** Executing instructions one after another in order.
- **Decision (Condition):** Choosing different actions based on a condition.
- **Repetition (Loop):** Repeating a set of instructions multiple times.
- **Reuse:** Writing code once and using it whenever needed.

By combining these building blocks, programmers can create applications that solve real-world problems efficiently.



Example :

A coffee vending machine is a good example of how programs work. When you choose a coffee type, the machine receives your choice (**input**). It then follows a series of steps in order (**sequence**). The machine checks if enough money has been inserted (**decision**). If not, it keeps asking for more money (**loop**). The same process is used to make different types of coffee (**reuse**). Finally, the machine delivers your drink (**output**). This simple example shows the basic building blocks used in all programs.



1.1 Programming Languages

Programming languages are like giving instructions to a computer, just like following steps in a recipe to bake a cake or using directions to build something.

Think about playing video games. Each game has different buttons and rules, right? Programming languages work the same way, each one has its own way of telling the computer what to do.

There are a lot of programming languages in the world, as shown in figure each of them with its features that make it different from the others. These programming languages such as C, Java, C++, Python, JavaScript, Rust, Golang, etc.



Programming languages are mainly divided into two types:

High-Level Languages : A high-level language (HLL) is a programming language designed to be easy for humans to read and write. It hides complex hardware details so programmers can focus on solving problems and building applications.

- Simple and easier to learn and use.
- Helps developers create programs faster.
- Easier to read, understand, and fix errors.
- Can run on different computers with few changes.
- Provides many built-in tools and features.
- Commonly used for websites, apps, games, and software development.
- Usually slower than low-level languages because code must be translated for the computer.

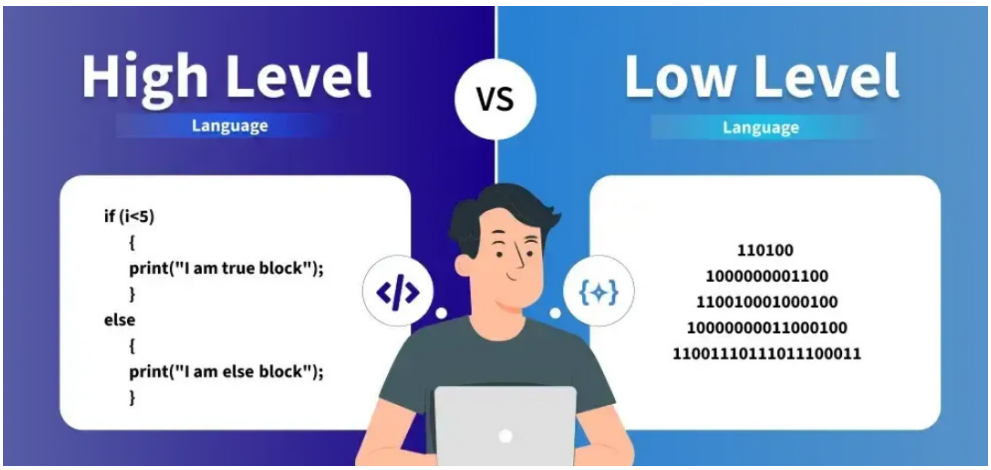
Examples: Python, JavaScript, PHP, Ruby, and Swift.

Low-Level Languages : A low-level language is a programming language that works very close to the computer's hardware. It gives programmers direct control over memory and system operations, making programs fast and efficient.

- Provides direct communication with computer hardware.
- Requires a strong understanding of how computers work.
- More difficult to read, write, and debug.
- Less flexible for use on different types of computers.
- Used for tasks that need high speed and precise control.

• Usually faster and uses fewer system resources than high-level languages.

Examples: Machine Language (Binary) and Assembly Language such as MIPS and ARM.



1.2 Algorithms and Flowcharts

Before writing a program, it is important to understand the steps needed to solve the problem. As problems become more complex, programmers use **algorithms** and **flowcharts** to plan and organize their solutions.
An **algorithm** is a step-by-step list of instructions used to solve a problem.
A **flowchart** is a diagram that shows these steps using symbols and arrows.
Both tools help programmers understand the logic of a solution before writing code.

Most programming tasks use three basic types of instruction:

- **Sequential Instructions** – Steps that are executed one after another. Example: Enter two numbers → Add them → Display the result.
- **Conditional Instructions** – Actions that depend on a condition. Example: If a number is even, display "Even"; otherwise, display "Odd".
- **Iterative Instructions (Loops)** – Steps that are repeated multiple times.Example: Print the numbers from 1 to 100.

Example: Write An algorithm to calculate the factorial of a number entered by **the user.**

Answer :

Step 1: Start
Step 2: Read n
Step 3: Initialize factorial =1 and i=1
Step 4: Repeat the steps until i = n
factorial = factorial*i
i = i+1
Step 5: Print factorial
Step 6: Stop

Example : An algorithm to check whether a given number is a prime number .
Answer :
Step 1: Start
Step 2: Read n
Step 3: Initialize flag = 1 and i = 2
Step 4: Repeat the following steps until i <=(n/2)
If remainder of n/i equals 0
flag = 0
go to step 5
i= i+1
Step 5: If flag == 0
Print n is not prime
else
Print n is prime
Step 6: Stop

1.3 Introduction to python:

Python is one of the most popular programming languages in the world. It is easy to learn, simple to read, and can be used to create many types of applications, including websites, games, data analysis tools, artificial intelligence systems, and scientific software.



Python syntax is clear and similar to the English language, making it an excellent choice for beginners. Python is also open-source, cross-platform, and supported

by a large collection of libraries that help programmers build applications quickly and efficiently.

Technical Strengths of Python

1. Portability

Python programs can run on different operating systems, such as Windows, Linux, and macOS, with little or no modification. This makes Python a portable and flexible programming language.



2. Object-Oriented Programming

Python supports object-oriented programming (OOP), which helps developers organize code into reusable objects and classes. This makes programs easier to build, maintain, and expand.

3. Strong Community Support

Python has a large and active global community. Developers can easily find tutorials, documentation, libraries, and solutions to programming problems through community resources.

4. Advanced Features

Python provides many powerful features that simplify software development, such as built-in libraries, automation tools, and support for different programming styles. As an open-source language, Python is free to use and continuously improved by developers around the world.

These strengths make Python one of the most popular and widely used programming languages today.

1.4 Setting up a python Environment:

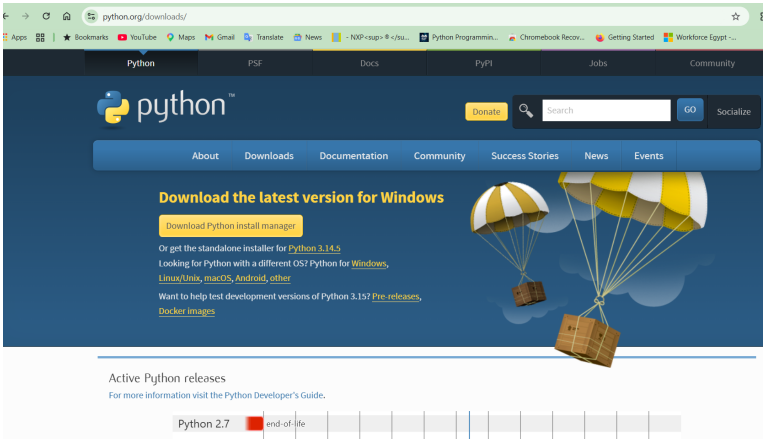
Before you start writing Python code, you need to set up your development environment

Her are the essential steps to get started:

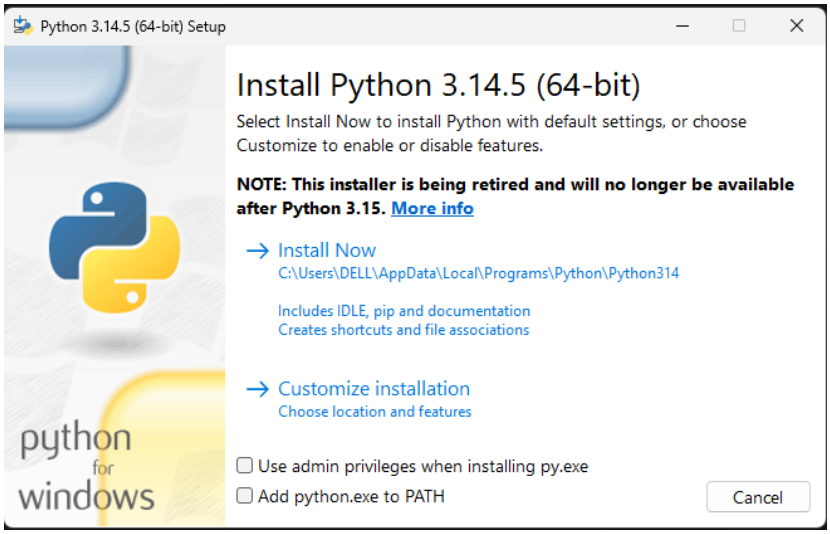
Installing Python

Step 1 : To install Python on your computer, follow these steps:

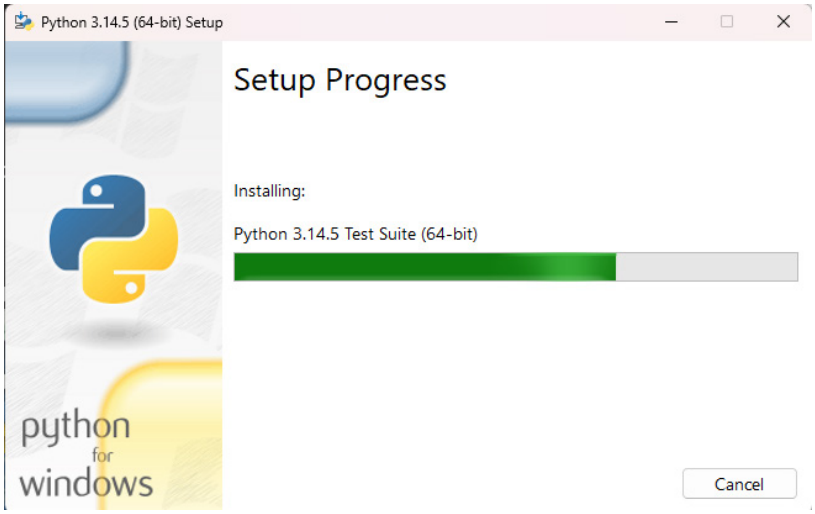
- Visit the official Python website at python.org.
- Download the latest Python version for your operating system (e.g., Windows, macOS, or Linux).
- Run the installer and follow the installation instructions.



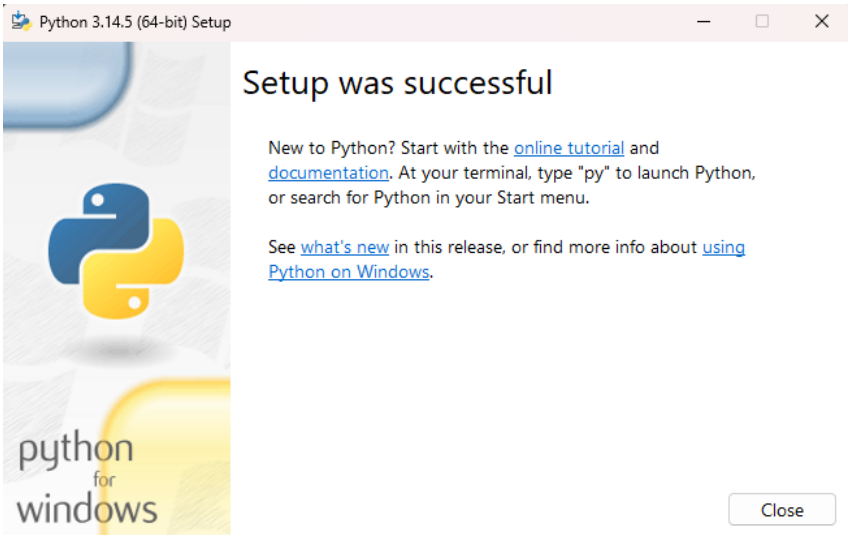
Once your download is complete, click on it to begin the installation. When you do, a window should pop up.



Go ahead and click the **Install Now** button. Then, you'll see this



Wait for the install to finish and the green bar to be full. Once it is done, the final screen should appear, saying that your installation was successful



Make sure to check the ADD PYTHON TO your machine PATH .

Step 2 : Install an IDE

An IDE gives you a more comfortable way to write and run Python code, with features like auto-complete, highlighting errors, and running code with one click. To get your computer ready for Python programming by installing Visual Studio Code (VS Code) for example, a powerful code editor used by developers worldwide.

- Go to the VS Code download page: https://code.visualstudio.com/
- Download the version for your operating system (Windows, macOS, or

Linux).

- Run the installer and follow the instructions to install VS Code.
- When installation is complete, launch VS Code.

Step 3: Set Up Python in VS Code

To write and run Python in VS Code, you need to install the Python extension.

- Open VS Code.
- Click on the Extensions icon (left sidebar, looks like building blocks), or press Ctrl+Shift+X.
- In the search bar, type "Python" and select the extension published by Microsoft.
- Click Install.
- Now VS Code is ready for Python programming!

Step 4: Try It Out

Let's test everything with a simple program.

- Open VS Code.
- Create a new file:
- Go to File > New File
- Save it as hello.py
- Type the following code: print("Hello from VS Code!")
- Save the file.
- To run the code: Right-click wwin the editor and select "Run Python File in Terminal", or click the small ▶ Run button at the top right.

You should see:

Hello from VS Code!

If that worked, you're all set!

Tools for Writing and Running Python Programs

Python programs can be written and executed using a variety of tools, ranging from simple editors to advanced development environments.



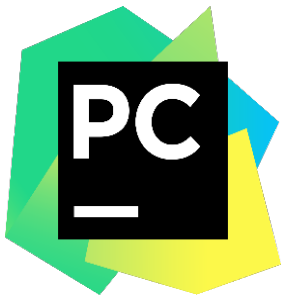
IDLE

IDLE: The default Python editor that comes with Python installation. It is simple and suitable for beginners.



VS Code (Visual Studio Code): A powerful code editor that offers features such as syntax highlighting, code completion, and debugging.

Command Line (CMD) / Terminal: Allows users to run Python programs directly by entering commands.



PyCharm Community Edition: A specialized Python IDE that provides advanced tools for coding, testing, and debugging.



Jupyter Notebook: An interactive environment widely used for learning Python, data analysis, and machine learning. It allows code, text, and visualizations to be combined in one document.

Google Colab: A cloud-based platform that enables users to write and run Python programs directly in a web browser without installing Python on their computers.



Online Python Compilers: Websites such as Replit and Programiz allow users to write and execute Python code online, making them useful for practice and quick testing.

Each tool has its own advantages, and learning to use different Python environments helps programmers become more productive and adaptable.

1.5 Python programming structure:

A Python program consists of a sequence of statements that are executed one by one. Here's a basic program structure

Note

In Python, each statement is typically written on a separate line, and the interpreter understands the end of a statement based on line breaks and proper indentation. There's no need for semicolons in Python

```
A = 42

B = 3.14

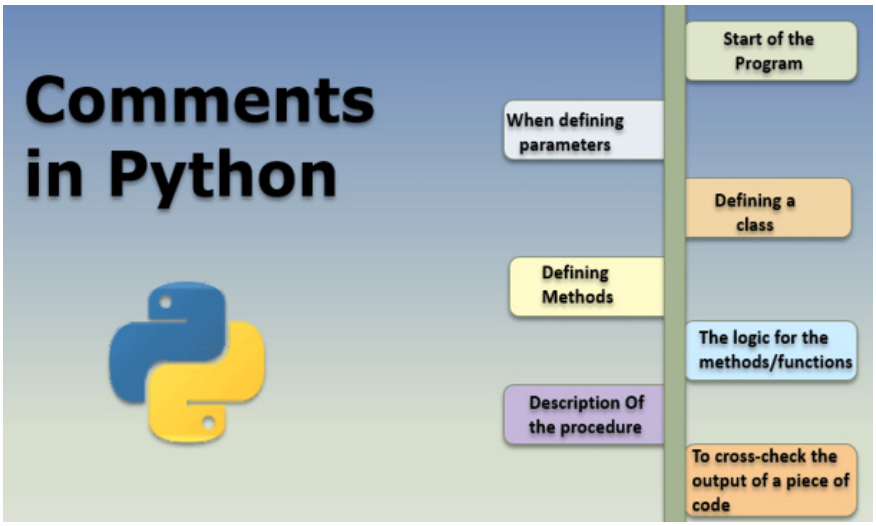
C = «x»
```

Comments:

- Single line comments
- Multi line comments

Comments are essential for documenting your code and making it more understandable. In Python, you can add comments using the `#` symbol:

```
# This is a single-line comment
```



In Figure : A single-line comment is denoted by a `#` symbol. Any text following the `#` symbol on the same line will be ignored by the Python interpreter. You can place single-line comments before or after the code

```
‹›

This is a

Multi line - comment

‹›
```

there are are multi-line comments that can be created using triple-quotes,

either single `('')` or double `("")`. To begin a multi-line comment, you start with three quotes, and to end it, you also use three quotes.

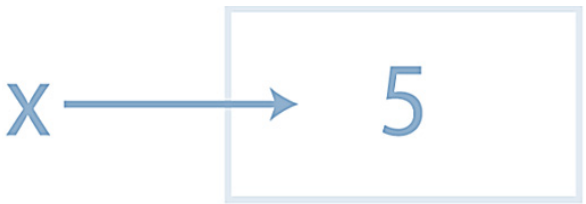
variables:

A variable is a named storage location used to keep data in a program. You can think of a variable as a label attached to a piece of information, making it easy to store, access, and update data when needed.

For example: x = 100

In this example, x is the variable name, and 100 is the value stored in the variable.

Variable Naming Rules



When creating variables in Python, follow these rules:

- Variable names are case-sensitive (age and Age are different variables).
- A variable name can contain letters, numbers, and underscores (_).
- A variable name cannot start with a number.
- Variable names should be meaningful and descriptive, such as student_name or user_age.

Declaring Variables in Python

Creating a variable in Python is simple. You only need:

- A variable name.
- The assignment operator (=).
- A value to store.

Example:

student_name = "Ali"

age = 20

Unlike some programming languages, Python does not require a semicolon (;) at the end of a variable declaration.

task

Determine the valid and invalid variables for each of the following:

string num;	Valid
int _x;	Valid
int ab;	Valid
int 2;	Not valid
int a;	Valid
int ab;	Valid
int float;	Not valid
int a30;	Valid

1.6 Output

Programs not only receive input from users but also display results. In Python, output is usually shown using the **print()** function.
The print() function displays text, numbers, or calculation results on the screen.

Example:

print("Hello World!")	Output	Hello World!

Formatting Output

Python provides several ways to format output:

1. f-Strings : f-strings allow you to insert variables directly into a string.

name = "Ali" print(f"Hello, {name}")	Output: Hello, Ali

2. String Concatenation : You can combine strings using the + operator.

```
name = "Ali"
print("Hello, " + name)
```

3. format() Method

Another way to insert values into a string is using the format() method.

```
name = "Ali"
print("Hello, {}".format(name))
```

Escape Sequences

Escape sequences are special characters that begin with a backslash (\).

They are used to format text output.
Example: print("This is a sentence.\nThis is on a new line.")
Output:
This is a sentence.
This is on a new line.

Common Escape Sequences

Escape Sequence	Meaning
\n	New Line
\t	Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\a	Alert/Beep
\b	Backspace
\f	Form Feed
\r	Carriage Return
\v	Vertical Tab

Exercise

1- Write a program whose output has to be like in figure

```
Hello
    world!
```

Exercise

2- Write a program to print the output in figure. You have to use escaping sequences only.

1.7 User Input

You can interact with users by taking input using the `input()` function:

```
user_input = input(«Enter your name: «)
print(«Hello « + user_input + «!»)
```

Remember that `input()` returns a string, so you may need to convert it to the appropriate data type for calculations or comparisons.

Example:

Read the input from the user and print it on the console.

```
#Get user input for a number
user_input = int(input(«Type a number: «))
#display the user input
print(«Your number is:», user_input)
```

The console output is:

```
Type a number: 22
Your number is: 22
```

When you run this program, it will prompt the user to enter a number. After the user enters a number, the program will display "Your number is:" followed by the entered number.

Lesson 1
Assessment



Part 1: Multiple Choice (MCQ)

1. Which is the correct stage to convert an algorithm into code?
   (a) Design
   (b) Implementation
   (c) Testing
   (d) Maintenance.
2. What is the role of the Python Interpreter?
   (a) To edit files
   (b) To execute code line by line
   (c) To compile to machine code
   (d) To store variwwables.
3. Which of the following is an invalid variable name?
   (a) my_var
   (b) var1
   (c) 1stVar
   (d) _var.
4. What is the correct way to output "Hello" in Python?
   (a) print(Hello)
   (b) output("Hello")
   (c) print("Hello")
   (d) echo "Hello".
5. Which function is used to accept user input?
   (a) get()
   (b) input()
   (c) ask()
   (d) read().
6. What is an algorithm?
   (a) A programming language
   (b) A step-by-step procedure for solving a problem
   (c) A bug in code
   (d) A type of computer.

Lesson 1 Assessment



Part 2: Essay & Application

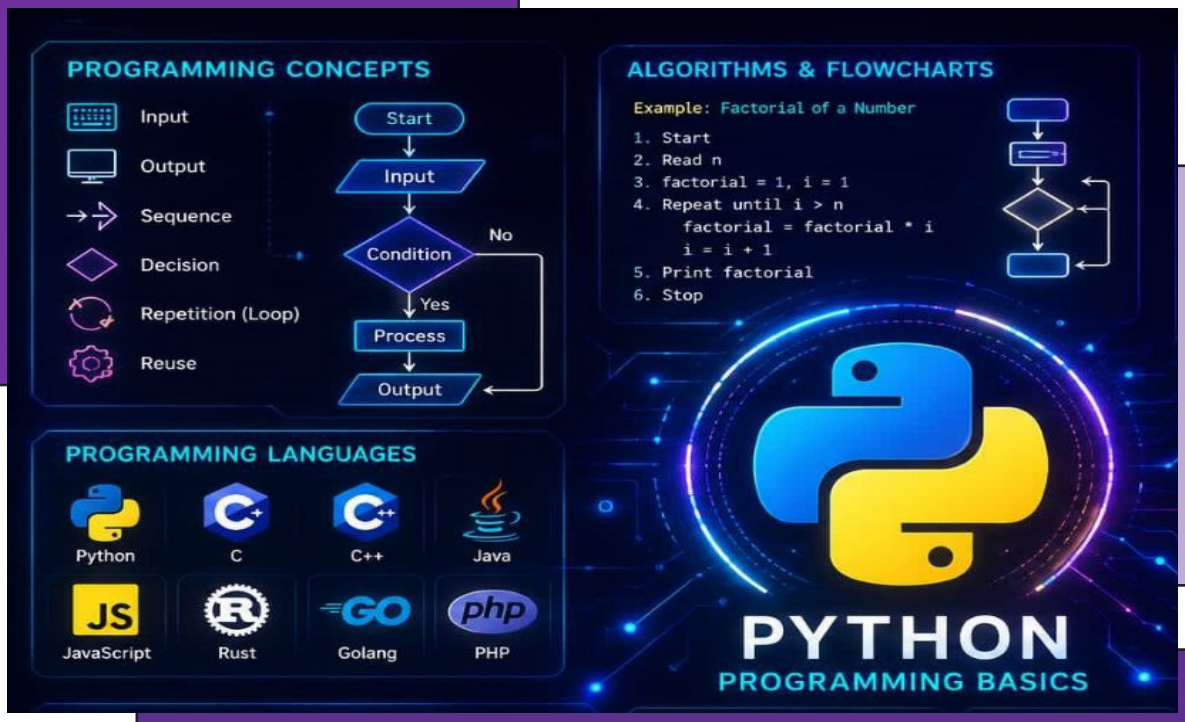
7. **Explain why syntax errors occur in Python.**
8. **Write code to ask for a user's name and favorite color, then print a personalized message.**
9. **What is the importance of "Pseudocode" before writing actual code?**
10. **Why is Python considered "beginner-friendly"?**
11. **Explain the difference between print() and input().**
12. **How do you comment out a line of code in Python, and why is it useful?**
13. **If a computer doesn't understand high-level code, what role does the interpreter play?**
14. **Describe the process of setting up a Python environment.**
15. **Compare "High-level" vs "Low-level" languages.**
16. **Write a snippet of code that demonstrates proper variable naming conventions.**
17. **What happens if you try to use an undefined variable?**
18. **Summarize the "Programming Cycle" (Design -> Write -> Run -> Debug).**

Project Overview

Goal: Initialize the project and store student profiles.
Write a program that asks for a student's Name, Age, and Grade_Level.
Requirement: Store these as variables and print a "Student Profile Card" using formatted strings (f-strings).

Lesson 2

Python Fundamentals and Core Programming Logic



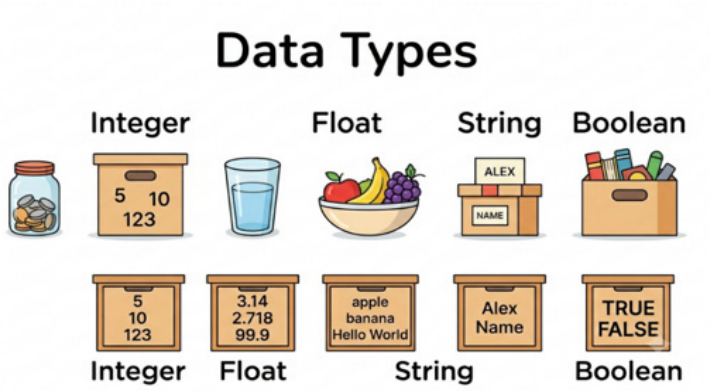
- ► **data Typs**
- ► **Python Operators**

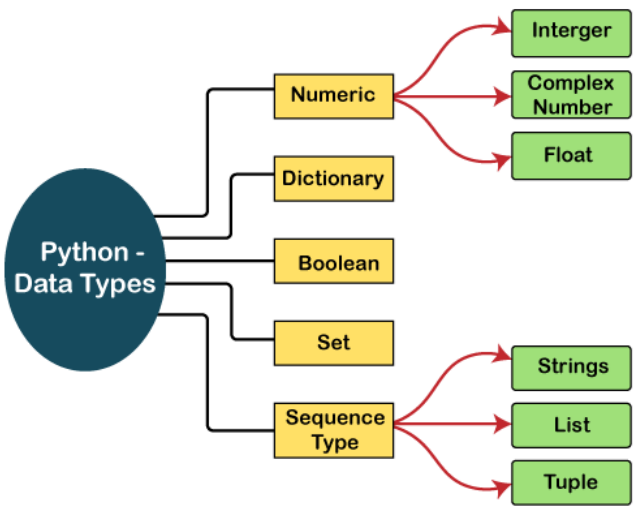
2.1 Data Type

What is a Data Type?

When writing a program, you work with different kinds of information, such as names, ages, scores, and prices. Python needs to know what type of information it is handling so it cwan store and use it correctly. This type of information is called a data type.

Think of data types as different containers. A glass is used for liquids, a bowl is used for food, and a box is used for other items. Each container is designed for a specific purpose. In the same way, Python uses different data types to store different kinds of information. This helps Python manage data correwctly and perform the appropriate operations on it.





Python supports various data types:

Integer (int): Used to store whole numbers without decimal points. Integers can be positive, negative, or zero.

Examples: 0 ,10- ,42

Floating-Point Number (float): Used to store numbers that contain decimal points. Floats are commonly used for measurements, prices, and calculations that require precision.

Examples: 25.75 ,0.5- ,3.14

String (str): Used to store text, including letters, words, sentences, and symbols. Strings are enclosed in single quotes (' ') or double quotes (" ").

Examples: "Hello, World!", 'Python', "Student"

Boolean (bool): Used to represent logical values. A Boolean variable can have only one of two values: True or False. Booleans are often used in decision-making and conditional statements.

Examples: True, False

Python provides the **type()** function to identify the data type of a value or variable.

Unlike some programming languages, Python does not require you to specify a data type when creating a variable. Python automatically determines the appropriate data type based on the value assigned to the variable. This feature is known as **dynamic typing**.

For example, when you assign a number, text, or decimal value to a variable, Python automatically stores it using the correct data type.

Example : Declare some different data types :

- Integer variable a
- Float variable b
- Character variable c

```
A= 42                #Integer

B =3.14              #Float

C = «X»              #String
```

•**"a"** is assigned an integer value.
•**"b"** is assigned a floating-point value.
•**"c"** is assigned a string containing a single character. Python treats single-character strings as regular strings

Example: Define Python variables to store 2 numbers and the summation of these numbers, keeping in mind naming conventions.
As shown in the code below, we need three variables. Two of them are for the numbers, while the third one will be used to store their summation

```
#Display variables to store two numbers and their summation

Frist_number = 10

Second_number = 20

Sum_of_number = Frist_number + Second_number

print(«Frist_number: « , Frist_number)

print(«Second_number: « , Second_number)

print(«Sum_of_number: «, Sum_of_number)
```

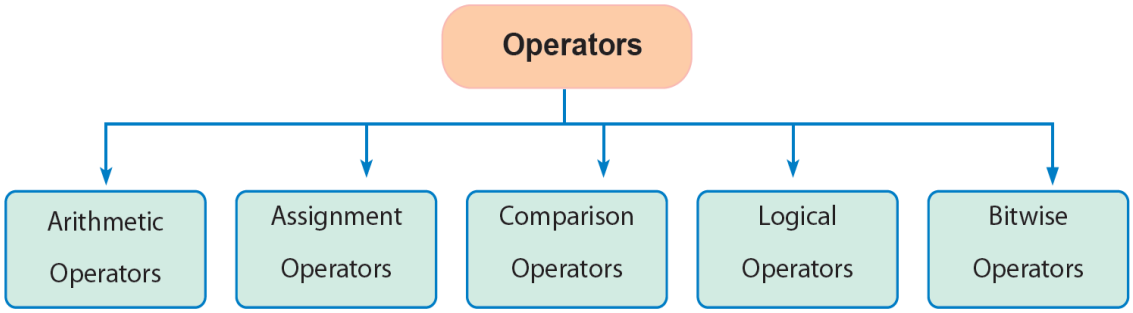
In this Python code:

- We define three variables using snake_case naming conventions: `first_number`, `second_number`, and `sum_of_numbers`.
- We assign values to `first_number` and `second_number`.
- We calculate the sum of the two numbers and store it in the `sum_of_numbers` variable.

Finally, we display the values of all three variables using the `print()` function.

2.2 Python Operators:

The Python programming language has five groups of operators, as shown in figure



Arithmetic Operators:

These operators are responsible for any arithmetic operations such as addition, subtraction, multiplication, division, and so on. Table (7) shows the different Python arithmetic operators.

Operator	Name	Description	Example
+	Addition	It adds together two values.	x + y
-	Subtraction	It subtracts one value from another.	x - y
*	Multiplication	It multiplies two values.	x * y
/	Division	It divides one value by another.	x / y
%	Modulus	It returns the division remainder.	x % y
//	Floor Division	It divides the left operand by the right operand and returns the integer part of the result.	x//y
--	Exponentiation	It raises the left operand to the power of the right operand.	x**y

Example:

```
A = 10
B =3
Sum_AB = A + B
Difference_AB = A – B
Product_AB = A * B
Division_AB =A / B
Floor_div_AB = A // B
Remainder_AB = A % B
```

Example

Write a Python program that applies all the arithmetic operators to two numbers:

```
# Define two numbers
num10 = 1
num4 = 2

# Addition
addition_result = num1 + num2
print(f»{num1} + {num2} = {addition_result}»)

# Subtraction
subtraction_result = num1 - num2
print(f»{num1} - {num2} = {subtraction_
result}»)

# Multiplication
multiplication_result = num1 * num2
print(f»{num1} * {num2} = {multiplication_
result}»)

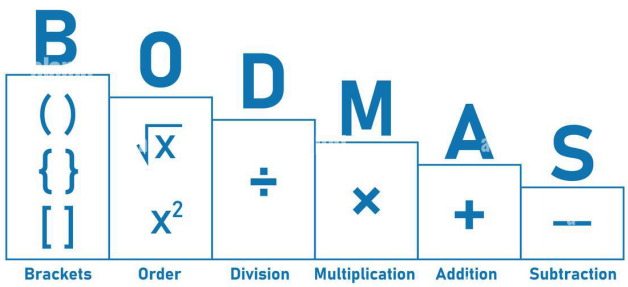
# Division
division_result = num1 / num2
print(f»{num1} / {num2} = {division_result}»)
# Integer Division (Floor Division)
floor_division_result = num1 // num2
print(f»{num1} // {num2} = {floor_division_
result}»)
```

```
# Modulus (Remainder)
modulus_result = num1 % num2
print(f»{num1} % {num2} = {modulus_result}»)

# Exponentiation
exponentiation_result = num1 ** num2
print(f»{num1} ** {num2} = {exponentiation_
result}»)
```

When you run this program, it will perform addition, subtraction, multiplication, division, integer division, modulus (remainder), and exponentiation on "num1" and `num2` and print the results.

When an arithmetic expression contains multiple operations, Python follows a specific order to determine which operation should be performed first. This order is known as the Order of Operations and helps ensure that expressions are evaluated correctly.



Python follows the same rules used in mathematics:

- Brackets ( ) – Evaluate expressions inside parentheses first.
- Orders – Calculate powers and exponents.
- Multiplication and Division – Perform multiplication and division from left to right.
- Addition and Subtraction – Perform addition and subtraction from left to right.

Assignment Operators

The assignment operators are responsible for assigning values to the left-hand side of these operators. The basic operators are the first six rows in table

Operator	Example	Equals To
=	a = 10	a = 10
+=	a += 10	a = a+10
-=	a -= 10	a = a-10
*=	a *= 10	a = a*10
/=	a /= 10	a = a / 10
%=	a %= 10	a = a % 10
//=	a //= 10	a = a // 10
**=	a **= 10	a = a ** 10
&=	a &= 10	a = a & 10
|=	a |= 10	a = a |10
^=	a ^= 10	a = a ^10
>>=	a >>= 10	a = a >> 10
<<=	a <<= 10	a = a << 10

Note

To increment a variable by 1, you should use += 1, and to decrement it by 1, you should use - = 1

Example:

```
# Initialize a variable
count = 0

# Increment the variable by 1
count += 1

# Display the result
print(«Incremented count:», count)

# Decrement the variable by 1
count -= 1

# Display the result
print(«Decremented count:», count)
```

Comparison Operators:

The assignment operators are responsible for assigning values to the left-hand side of these operators. The basic operators are the first six rows in table

Operator	Example	Same as
==	Equal to	x ==y
=!	Not equal	x ! = y
<	Greater than	x > y
>	Less than	x < y
=<	Greater than or equal to	x >= y
=>	Less than or equal to	x <= y

Example:

```
5 = x
10 = y
is_equal = x == y  # False
is_not_equal = x != y  # True
is_less_than = x < y  # True
is_greater_than = x > y  # False
is_less_or_equal = x <= y  # True
is_greater_or_equal = x >= y  # False
```

Logical Operators:

Logical operators are responsible for the logical comparison, like in logic gates and, or, and not.

Example:

```
p = True
q = False
result_and = p and q  # False
result_or = p or q  # True
result_not = not p  # False
```

Bitwise Operators:

Bitwise operators are used to perform operations on the binary (bit-level) representation of integers. They work directly with the individual bits of a number, making them useful in low-level programming, data processing, and optimization tasks.

Operator	Description
&	Bitwise AND Operator
|	Bitwise OR Operator
^	Bitwise XOR Operator
~	Bitwise Complement Operator
>>	Bitwise Shift Left Operator
<<	Bitwise Shift Right Operator

Example:

Write a Python program that reads two numbers from the user and then displays the following:

- The summation of the two numbers.
- The difference between the first number and the second number.
- The difference between the second number and the first number.
- The multiplication between the two numbers.
- The division between the first number and the second number (the first divided by the second).
- The division between the second number and the first number (the second divided by the first).
- The remainder of the first number is divided by the second number.
- The remainder of the second number is divided by the first number.
- The increment of the first number.
- The decrement of the first number.
- The increment of the second number.
- The decrement of the second number.

```
# Input two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
# Perform calculations
summation = num1 + num2
diff1 = num1 - num2
diff2 = num2 - num1
multiplication = num1 * num2
# Check if the second number is not zero before performing division
if num0 =! 2:
    division1 = num1 / num2
else:
    division1 = "Cannot divide by zero"
# Check if the first number is not zero before performing division
if num0 =! 1:
    division2 = num2 / num1
else:
    division2 = "Cannot divide by zero"
# Check if the second number is not zero before performing remainder operation
if num0 =! 2:
    remainder1 = num1 % num2
else:
    remainder1 = "Cannot divide by zero"
```

```
# Check if the first number is not zero before performing remainder operation
if num0 =! 1:
    remainder2 = num2 % num1
else:
    remainder2 = «Cannot divide by zero»
# Increment and decrement operations
increment_num1 = num1 + 1
decrement_num1 = num1 - 1
increment_num2 = num1 + 2
decrement_num2 = num1 - 2
# Display results
print(f»1. Summation: {summation}»)
print(f»2. Difference (1st - 2nd): {diff1}»)
print(f»3. Difference (2nd - 1st): {diff2}»)
print(f»4. Multiplication: {multiplication}»)
print(f»5. Division (1st / 2nd): {division1}»)
print(f»6. Division (2nd / 1st): {division2}»)
print(f»7. Remainder (1st % 2nd): {remainder1}»)
print(f»8. Remainder (2nd % 1st): {remainder2}»)
print(f»9. Increment of 1st number: {increment_num1}»)
print(f»10. Decrement of 1st number: {decrement_num1}»)
print(f»11. Increment of 2nd number: {increment_num2}»)
print(f»12. Decrement of 2nd number: {decrement_num2}»)
```

Lesson 2

Assessment

Part 1: Multiple Choice (MCQ)

1. What is the data type of the value 3.14 in Python?
   - A) int
   - B) str
   - C) float
   - D) bool
2. What is the result of 10 // 3 in Python?
   - A) 3.33
   - B) 3
   - C) 4
   - D) 1
3. Which operator is used to find the remainder of a division?
   - A) /
   - B) //
   - C) %
   - D) **
4. What is the output of 2 ** 4?
   - A) 8
   - B) 6
   - C) 16
   - D) 24
5. What is the result of (5 > 3) and (2 < 1)?
   - A) True
   - B) False
   - C) Error
   - D) None
6. How do you increment a variable x by 1 in Python?
   - A) x++
   - B) ++x
   - C) x += 1
   - D) x

Lesson 2
Assessment



Part 2: Essay & Application

19. Identify the data type of each of the following values:

Value	Data Type
42	
"Hello"	
True	
9.5	
"100"	

20. Explain the importance of Type Casting when dealing with input().

21. Explain the "Order of Operations" (BODMAS) in Python.

22. Write a code to calculate the area of a rectangle using user input.

23. Differentiate between / (division) and // (floor division).

24. Why do we use variables instead of hard-coding values?

25. Explain the difference between int and float.

26. Write a code that calculates the remainder of 17 divided by 4.

27. What are logical operators? (and, or, not).

28. Create a snippet demonstrating dynamic typing.

29. Explain why 5 + "5" causes an error.

30. How can you find the type of a variable?

31. What is the power operator in Python? Write a code for 2 raised to 5.

32. Discuss the importance of descriptive variable naming.

33. If you need to store a large number with decimal points, which data type should you choose

Project Overview

Task 2: The Calculation Engine

Goal: Integrate arithmetic logic and process multiple scores.
Add input fields for three different subjects (e.g., Math, Science, English).
Requirement: Use arithmetic operators to calculate the Total, Average, and Percentage.

Lesson 3

Control Stuctures and Program Flow in Python



- ▶ **Conditios**
- ▶ **Loops**
- ▶ **Loop Control Statments : “break”**
- ▶ **Loop Control Statments: “continue”**

3.1Conditions

Think of conditions like real-life decisions. If it's raining, you take an umbrella.
If it's sunny, you wear sunglasses.
In programming, conditions help our programs make decisions based on information
Conditional statements allow you to make decisions in your Python programs. In this chapter, we'll explore how to use `if`, `elif`, and `else` statements for decision-making.
Using `if` Statements for Decision-Making:
The `if` statement is used to execute a block of code if a condition is true.

Conditional Statements

In programming, a **condition** is a test that evaluates to either **True** or **False**. Conditions allow programs to make decisions and perform different actions based on the result.
We use conditional statements when we want the program to choose between different options.



For example, if a student passes an exam, the program can display a congratulatory message; otherwise, it can display a different message.
Python uses **if**, **elif**, and **else** statements to implement decision-making:

- **if** checks whether a condition is true.
- **elif** checks additional conditions if the previous condition is false.
- **else** executes when none of the previous conditions are true.

These statements help make programs more interactive and capable of responding to different situations.
Here's the basic structure:

```
if condition:
    # Code to execute if the condition is true
```

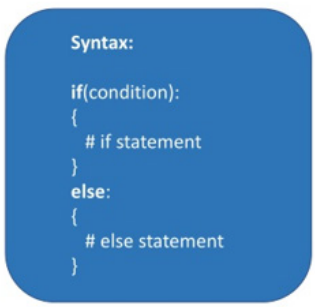
Example:

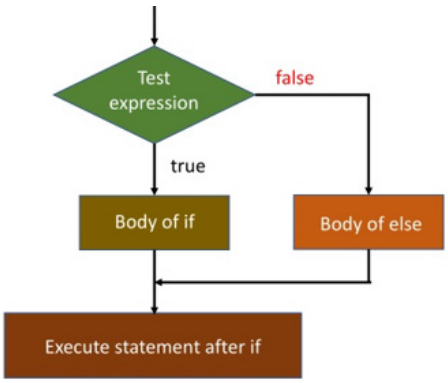
```
age = 16
if age >= 18:
    print(«You are an adult.»)
else:
    print(«You are a minor.»)
```

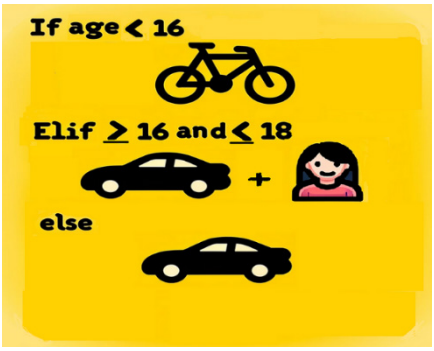
In this example, the code inside the `if` block is executed because the condition
"age >= 18" is true.

Python uses these three keywords to make decisions:

- **if –** Checks a condition and runs the code if it's true.
- **elif** (else if) – Adds another condition to check if the first one is false.
- **else** – Runs when none of the conditions are true.







Adding Multiple Conditions with `elif`:

You can use the `elif` (short for "else if") statement to specify additional conditions to check `if` the initial if condition is false. Here's how it works:

```
if condition1:
    # Code to execute if condition1 is true
elif condition2:
    # Code to execute if condition1 is false and condition2 is true pass
```

Example:

```
# Nested if statement
num = 10

if num > 0:
    print(«The number is positive.»)
    if num % 2 == 0:
        print(«The number is even.»)
    else:
        print(«The number is odd.»)
else:
    print(«The number is not positive.»)
```

In this example, the program checks multiple conditions and prints the corresponding grade based on the grade variable

Handling Cases with `else`:

The `else` statement is used to execute a block of code when none of the preceding conditions are true. It provides a fallback option. Here's how it's used:

```
if condition1:
    # Code to execute if condition1 is true
else:
    # Code to execute if condition1 is false
```

Example:

```
age = 15
if age >= 18:
   print(«You are an adult.»)
else:
      print(«You are not an adult.»)
```

In this example, if the `age` is less than 18, the code inside the `else` block is executed.

Example:

Write a program to show a message depending on the temperature degree, given that the temperature is 19°C. The output message should be "It's a cold weather".

When the degree is less than 20°C, otherwise, it should be "It's a nice weather".

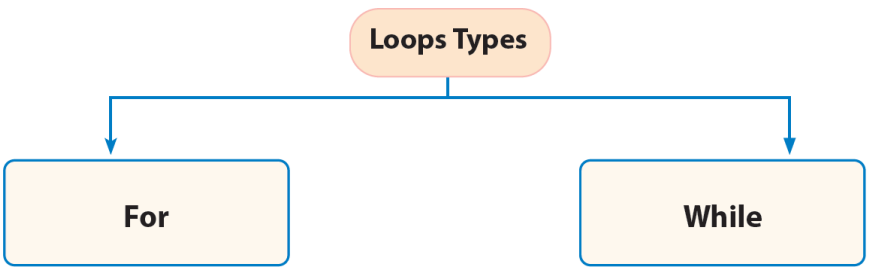
```
# Input the temperature in Celsius
temperature = float(input(«Enter the temperature in Celsius: «))


# Check the temperature and display a message
if temperature < 20:
   print(«It›s a cold weather»)
else:
   print(«It›s a nice weather»)
```

Example:

```
# Boolean variables
is_logged_in  = True
has_passed    = False
# Booleans from comparisons
score = 85
print(score >= 60)   # True
print(score == 100)  # False
print(score != 100)  # True
# Boolean operators
print(True and False) # False
print(True or False)  # True
print(not True)       # False
```

3.2 Loops



A **loop** in Python is a way to repeat a block of code multiple times without writing the same code again and again.
Imagine you want to print the word "Hello" five times. Without a loop, you would write Print statement five times .

This works, but it is repetitive. A loop lets us do the same thing in a shorter and cleaner way.

```
for i in range (5):
    print(«Hello»)
```

This code prints "Hello" five times

For Loop:

A for loop is used to repeat a block of code multiple times. It is useful when you want to perform the same action for a sequence of values. It's like telling your friend to take steps forward and count each step until they reach a certain number.

```
for item in sequence:
    # Code to execute for each item
```

Example:

```
fruits = [«apple», «banana», «cherry»]


for fruit in fruits:
    print(fruit)
```

In this example, the `for` loop iterates through the `fruits` list and prints each item.

Example:

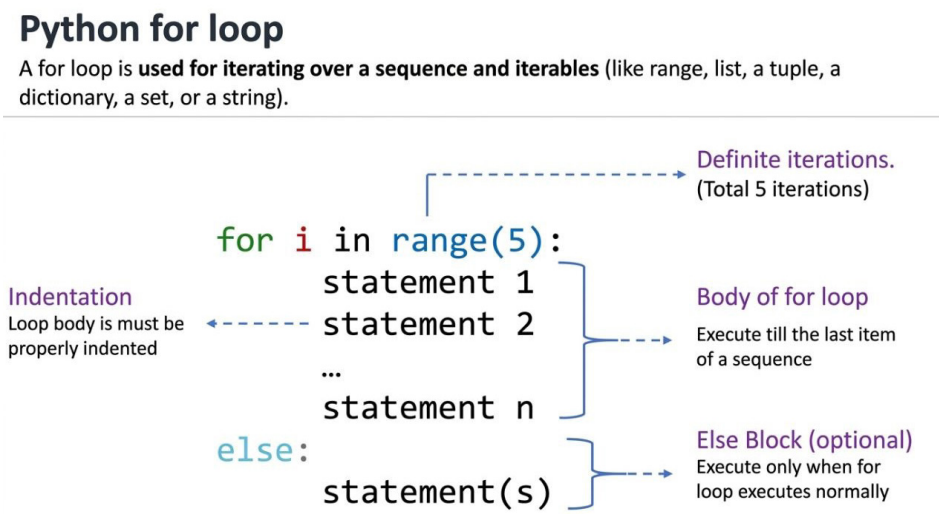
Write a Python program to print numbers from 1 to 100 using the `for` loop.

```
# Using a for loop to print numbers from 1 to 100
for number in range(1, 101):
    print(number)
```

for, in, range

A for loop consists of three main parts:

- **for:** Starts the loop and indicates that the code will be repeated.
- **in:** Connects the loop variable to the sequence of values being processed.
- **range():** Generates a sequence of numbers that controls how many times the loop runs.



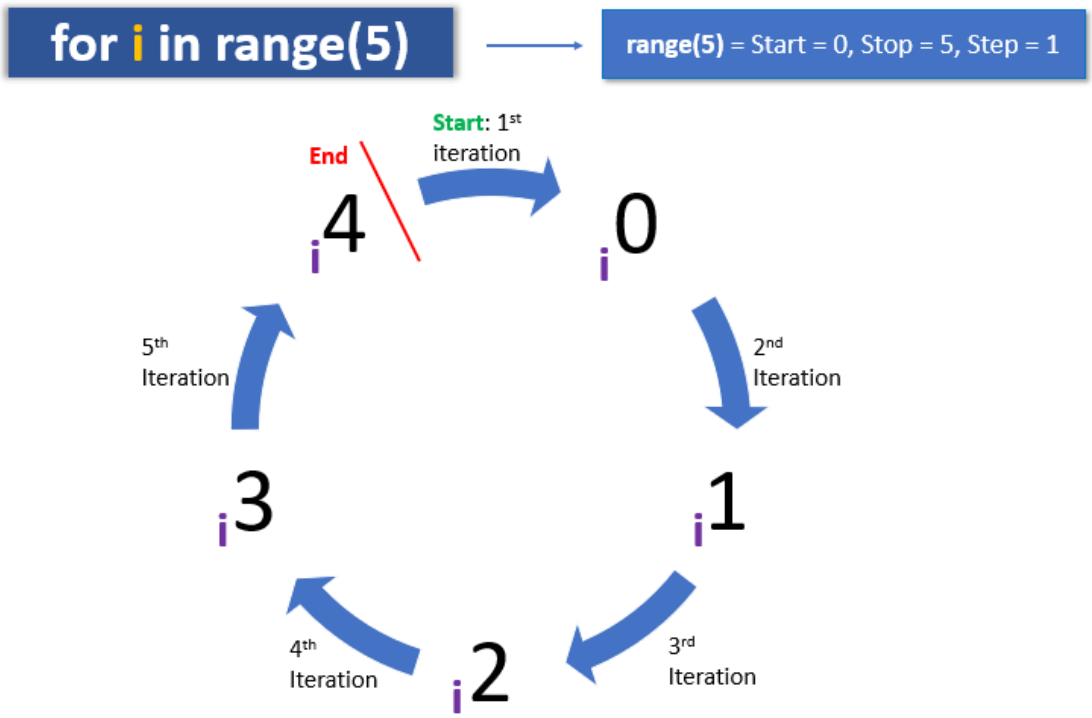
The range() function can use up to three parameters:

Start: The first number in the sequence. If omitted, Python starts from 0.

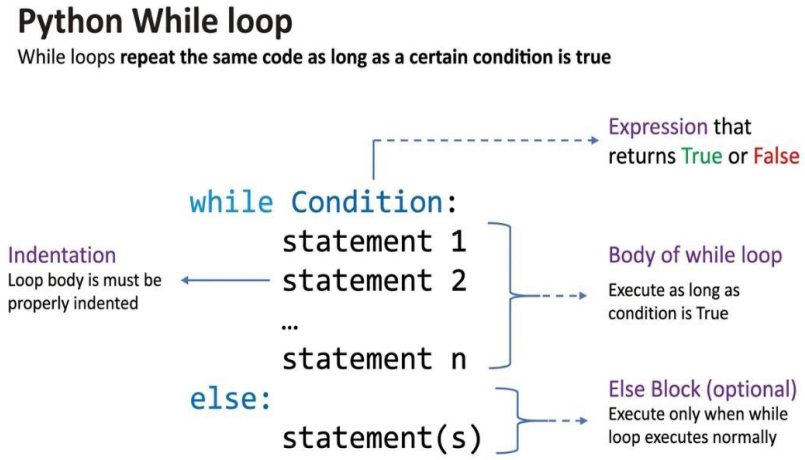
Stop: The value where the sequence ends. This value is not included in the sequence.

Step: The amount by which the value increases each time. If omitted, the default step is 1.

By combining for, in, and range(), Python can efficiently repeat instructions a specified number of times.



While loop:



Example: Write a program to print numbers from 0 to 99 using the **`while`** loop

imagine you're in an forest looking for flowers. You'll keep looking as long as you haven't found 5 flowers yet. Here's how you could use a "while loop" for your quest:

Note

If there is no termination condition, a while loop in Python will continue to execute indefinitely.

Exercise

Write a program that reads a number from the user and then prints the numbers from 1 to the entered number using a while loop

Answer:

```
# Read a number from the user
number = int(input(«Enter a number: «))

# Initialize a variable to start at 1
count = 1

# Use a while loop to print numbers from 1 to the entered number
while count <= number:
    print(count)
    count += 1
```

3.3 Loop Control Statements: «break»

The `**break**` statement is used to exit a loop prematurely, typically when a certain condition is met. It allows you to terminate the loop and continue with the rest of the program.

Example:

```
numbers = [1, 2, 3, 4, 5]


for number in numbers:
    if number == 3:
        break
    print(number)
```

In this example, when the loop encounters the number `3`, the `break`

statement is executed, and the loop terminates.

3.4 Loop Control Statements:«continue»

The `**continue**` statement is used to skip the current iteration of a loop and proceed to the next iteration. It allows you to skip specific items or conditions within the loop.

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    if number == 3:
        continue
    print(number)
```

In this example, when the loop encounters the number `3`, the `continue` statement is executed, and the loop skips printing it.

Lesson 3
Assessment



Part 1: Multiple Choice (MCQ)

1. Which keyword starts a condition?
   (a) loop
   (b) if
   (c) for
   (d) check.
2. What does `break` do?
   (a) Skips the current iteration
   (b) Stops the loop entirely
   (c) Restarts the program
   (d) Does nothing.
3. Which loop is best when the number of iterations is known?
   (a) while
   (b) for
   (c) if
   (d) break.
4. What is the correct syntax for an `if-else` block?
   (a) use `else if`
   (b) use `elif`
   (c) use `case`
   (d) use `switch`.
5. What does `continue` do?
   (a) Stops the loop
   (b) Skips the rest of the current iteration
   (c) Goes to the next file
   (d) Ends the program.
6. What is a condition?
   (a) An integer
   (b) A comparison resulting in True or False
   (c) A loop
   (d) A variable

Lesson 3
Assessment



7. What keyword is used in Python to check an additional condition after if?

A) else
B) elif
C) then
D) switch

8. What is the output of the following code?

```
x = 15
 if x > 20:
    print(“Big”)
 elif x > 10:
    print(“Medium”)
 else:
    print(“Small”)
```

A) Big
B) Medium
C) Small
D) Error

9. What does range (1, 6) generate?

A) 6 ,5 ,4 ,3 ,2 ,1
B) 5 ,4 ,3 ,2 ,1 ,0
C) 5 ,4 ,3 ,2 ,1
D) 5 ,3 ,1

10. What does the break statement do inside a loop?

A) Skips the current iteration
B) Exits the loop completely
C) Restarts the loop
D) Pauses the loop

11. What does the continue statement do inside a loop?

A) Exits the loop



B) Skips the current iteration and moves to the next
C) Repeats the current iteration
D) Stops the program

12. How many times will this loop run?

```
for i in range (0, 10, 2):
    print(i)
```

A) 10
B) 5
C) 4
D) 6

Part 2: Essay & Application

13. Explain the difference between break and continue with an example.
14. Write a for loop that prints numbers from 1 to 10.
15. Explain when to use a while loop instead of a for loop.
16. Write code to check if a number is positive, negative, or zero.
17. Trace this code: x = 5; while x > 0: print(x); x -= 1. What is the output?
18. Why do we need "Indentation" in Python control structures?
19. Write a program that asks for a password and stays in a loop until it's correct.
20. What are nested loops? Provide a brief example.
21. Explain how and and or change the logic of an if statement.
22. Can a for loop run infinitely? How?
23. Write a code that sums all numbers from 1 to 100 using a loop.
24. Discuss the logical flaw if a condition is always True.
25. How do you check if a number is even or odd?
26. Explain the role of the else keyword in a while loop (or if).
27. What is the difference between if, elif, and else? Explain in your own words.
28. Write a Python program that asks the user for a number and prints whether it is positive, negative, or zero.

Lesson 3
Assessment



29. What is the output of this code?

```
score = 75
 if score >= 90:
    print("Excellent")
 elif score >= 70:
    print("Good")
 elif score >= 50:
    print("Pass")
 else:
    print("Fail")
```

30. Write a program that asks the user for their age and prints:

- **- "Child" if age < 13**
- **- "Teenager" if age is 13–17**
- **- "Adult" if age >= 18**

31. Write a for loop that prints the numbers from 1 to 10.

32.. Write a while loop that prints the numbers from 5 down to 1
print(count)

33. What is the output of this code?

```
for i in range(1, 6):
   if i == 3:
      continue
   print(i)
```

34. What is the output of this code?

```
for i in range(1, 10):
   if i == 5:
      break
   print(i)
```

Lesson 3
Assessment



35. Write a program that prints the multiplication table of 7 (from 7×1 to 7×10).

36. Explain the difference between a for loop and a while loop. When would you use each?

37. Write a program that calculates the sum of all numbers from 1 to 100 using a for loop.

38. Write a guessing game: The secret number is 7. Keep asking the user to guess until they get it right. Print "Correct!" when they succeed.

39. What is the output of this code?

```
for i in range(0, 15, 3):
  print(i)
```

40. Write a program that prints all even numbers from 2 to 20 using a for loop.

Task 3: The Decision Maker

Automate grading and batch processing.

Implement if-else logic to assign a Letter Grade (A, B, C, F) based on the average.

Requirement: Wrap your code in a while loop that allows the user to enter data for multiple students until they type "exit".

Lesson 4

Data Structures and Functions in Pythone



- List
- tuples
- Dictionaries
- Function in Python

Introduction

In Python, **data structures** are used to store and organize data.
For example, if you want to store student names, numbers, or scores, you need a data structure.
Python has many data structures, but in this lesson, we will study four important ones:

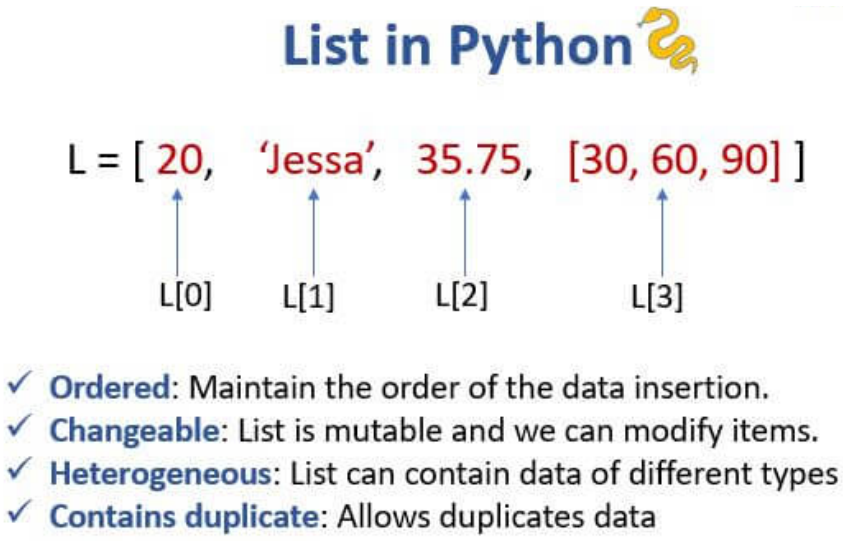
- **List**
- **Tuple**
- **Set**
- **Dictionary**

We will also learn about **functions**, which help us organize our code and reuse it.

4.1List

Lists are one of the fundamental data structures in Python, allowing you to store and manipulate collections of data.

A **list** is a data structure used to store multiple items in a single variable. Lists allow you to keep related data together and access, modify, add, or remove items when needed.



Creating Lists

You can create lists by enclosing a sequence of values in square brackets ` [] `, separated by commas. Lists can contain elements of different data types.

Example:

fruits = [“Apple”, “Banana”, “Orange”]

In this example, the list fruits contains three items: “Apple”, “Banana”, and “Orange”.

Key Features of Lists

- A list can store multiple values in one variable.
- Items in a list are ordered and can be accessed by their position (index).
- Lists are mutable, which means their contents can be changed after creation.
- A list can contain different data types, such as numbers, strings, and Boolean values.

Lists are widely used in Python to organize and manage collections of data efficiently.

Example:

```
fruits = [«apple», «banana», «cherry»]
numbers = [1, 2, 3, 4, 5]
mixed = [«apple», 42, True]
```

Accessing List Elements:

Each item in a Python list has a position called an **index**. Python uses **zero-based indexing**, which means the first item has an index of 0, the second item has an index of 1, and so on.

To access an item in a list, write the list name followed by the index inside square brackets.

Example:

```
fruits = [«apple», «banana», «cherry»]

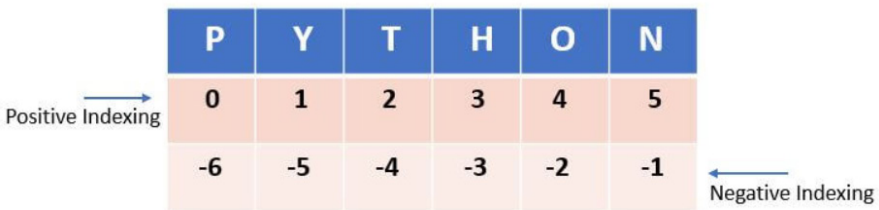

first_fruit = fruits[0]  # «apple»
second_fruit = fruits[1]  # «banana»



last_fruit = fruits[-1]  # «cherry»
```

Python also allows you to access items from the end of the list using negative indexes.

- refers to the last item.
- refers to the second-last item.




```
last_fruit = fruits[-1]  # «cherry»
```

Adding Elements :

The "append()" method is used to add a new element to the end of a list. This is useful when you want to store additional data after the list has been created. Example:

```
fruits = [«apple», «banana», «cherry»]
fruits.append(«orange»)
# fruits is now [«apple», «banana», «cherry», «orange»]
```

Inserting Elements :

The `insert()` method allows you to add an element at a specific position in the list. Unlike append(), which always adds an item to the end of the list, insert() allows you to choose where the new element will be placed.
Example:

```
fruits = [«apple», «banana», «cherry»]
fruits.remove(«banana»)
# fruits is now [«apple», «cherry»]
```

Removing Elements :

The **remove()** method is used to delete a specific element from a list. Instead of using an index, you provide the value of the item you want to remove.

Example:

```
fruits = [«apple», «banana», «cherry»]
fruits.insert(1, «orange»)
# fruits is now [«apple», «orange», «banana», «cherry»]
```

Iterating Through Lists Using Loops:

Iterating through a list means accessing each element in the list one by one. This is commonly done using a **for loop**, which allows you to process all items in a list without accessing them individually.

Example:

```
fruits = [«apple», «banana», «cherry»]


for fruit in fruits:
    print(fruit)
```

This code will print each fruit in the `fruits` list.

Exercise

Create a Python list named `fruits` containing the names of three different fruits. Then, print the second fruit in the list.

Answer:

```
# Create the list of fruits
fruits = [«apple», «banana», «orange»]


# Print the second fruit in the list (index 1)
print(f»The second fruit is: {fruits[1]}»)
```

Exercise

You have a list called `colors` containing the names of colors. Add the color “green” to the end of the list, insert the color “red” at the beginning of the list, and remove the color “blue” if it exists.

Answer:

```
# Initial list of colors
colors = [«blue», «yellow», «orange»]

# Add «green» to the end of the list
colors.append(«green»)

# Insert «red» at the beginning of the list
colors.insert(0, «red»)

# Remove «blue» if it exists
if «blue» in colors:
   colors.remove(«blue»)

# Print the modified list
print(«Updated list of colors:», colors)
```

Exercise

Given a list of `numbers` called numbers, use a `for` loop to print each number in the list on a separate line.

```
# List of numbers
numbers = [10, 20, 30, 40, 50]

# Iterate through the list and print each number
for number in numbers:
    print(number)
```

4.2 Tuples

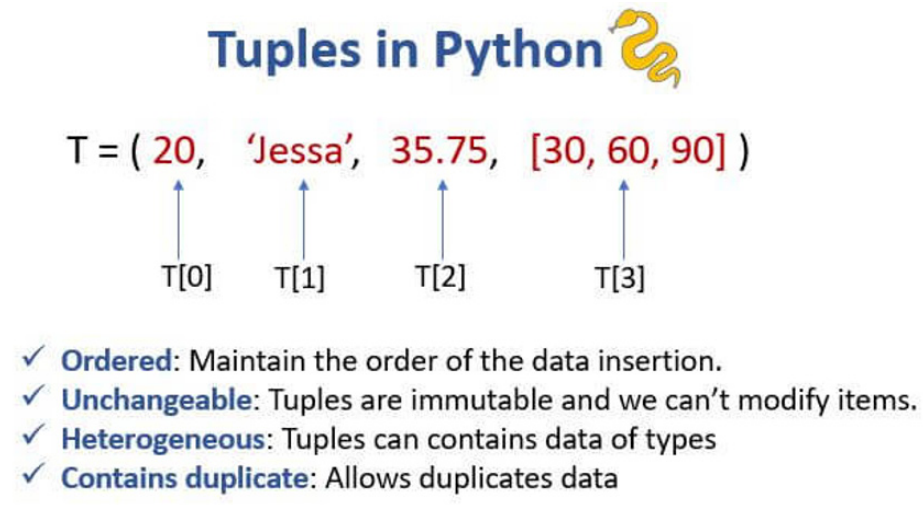
Tuples are ordered collections of heterogeneous data that are unchangeable:

Heterogeneous means tuple can store variables of all types.

Tuple has the following characteristics:

- **Ordered:** Items are stored in a specific order and can be accessed using indexes.
- **Immutable:** Once a tuple is created, its elements cannot be added, removed, or changed.
- **Can Store Different Data Types:** A tuple can contain integers, strings, floats, lists, and other data types.
- **Allows Duplicates:** The same value can appear more than once in a tuple.

These features make tuples useful for storing collections of data that should remain unchanged during program execution.



Creating Tuples:

We can create a tuple using the two ways

- **Using parenthesis ():** A tuple is created by enclosing comma-separated items inside rounded brackets.
- **Using a tuple() constructor:** Create a tuple by passing the comma-separated items inside the tuple().
-

Example:

```
# create a tuple using ()
# number tuple
number_tuple = (10, 20, 25.75)
print(number_tuple)
```

```
# Output (10, 20, 25.75)

# string tuple
string_tuple = (‹Jessa›, ‹Emma›, ‹Kelly›)
print(string_tuple)
# Output (‹Jessa›, ‹Emma›, ‹Kelly›)

# mixed type tuple
sample_tuple = (‹Jessa›, 30, 45.75, [25, 78])
print(sample_tuple)
# Output (‹Jessa›, 30, 45.75, [25, 78])
# create a tuple using tuple() constructor
sample_tuple2 = tuple((‹Jessa›, 30, 45.75, [23, 78]))
print(sample_tuple2)
# Output (‹Jessa›, 30, 45.75, [23, 78])
```

Accessing Tuple Elements:

Since tuples are ordered, each element has an index number starting from **0**. You can access an element by placing its index inside square brackets [].

Example:

```
coordinates = (3, 4)
fruits = ("apple", "banana", "cherry")
```

Tuples are often used to represent collections of related, immutable data. Slicing allows you to access a range of elements from a tuple.

Example:

```
tuple1 = (‹P›, ‹Y›, ‹T›, ‹H›, ‹O›, ‹N›)
print(tuple1[2.0])
```

Adding and Changing Items in a Tuple

Tuples are immutable, which means their elements cannot be changed,

added, or removed after the tuple is created.
Unlike lists, tuples do not have methods for adding or modifying items.

Example:

```
tuple1 = (0, 1, 2, 3, 4, 5)

tuple1[1] = 10
```

Deleting Items from a Tuple

Since tuples are immutable, their elements cannot be removed directly.
Tuples do not support methods such as remove() or pop().
However, items can be removed in two ways:

- Using the del keyword to delete the entire tuple.
- Converting the tuple to a list, modifying the list, and then converting it back to a tuple.

Because tuples are designed to be immutable, they are best used when the stored data should not change.

Example:

```
sampletup1 =(0,1,2,3,4,5,6,7,8,9,10)

del sampletup1


print(sampletup1)
```

Example:

```
tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list
sample_list = list(tuple1)
# reomve 2nd item
sample_list.remove(2)

# converting list back into a tuple
tuple1 = tuple(sample_list)
print(tuple1)
# Output (0, 1, 3, 4, 5)
```

Exercise

Create a tuple named `days_of_week` containing the names of the days of

the week. Attempt to change the name of the second day in the tuple to "Funday." Explain what happens and why.

Answer:

```
#create a tuple of days the week
days_of_week ={«Monday» , «Tuesday» , «Wednesday» , «Thursday» , «Friday» }

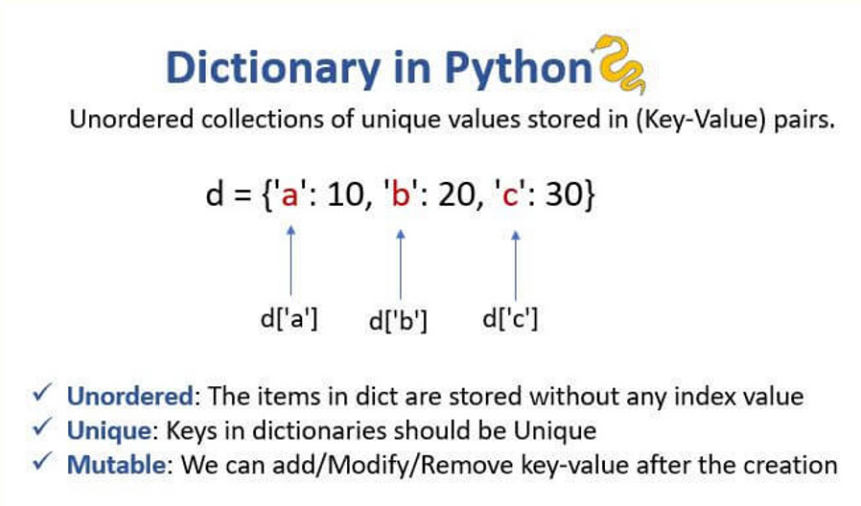
# Try to change the name of second day
# This will result in typeError because tuple are immutable

try:
   days_of_week[1] = «Funday»
except TypeError as e:
   print(f»Error : {e}»)
```

4.3Dictionaries

A dictionary is a collection that stores data in key-value pairs. Each key is unique and is used to access its associated value.

Think of a dictionary like a phone contact list: the name is the key, and the phone number is the value. Using the name, you can quickly find the corresponding phone number



Characteristics of Dictionaries

- **Key-Value Pairs:** Data is stored as a key and its corresponding value.
- **Ordered:** In Python 3.7 and later, items maintain their insertion order.
- **Unique Keys:** Each key must be unique. If the same key is used again, its value is updated.
- **Mutable:** Dictionary items can be added, modified, or removed after creation.

Dictionaries are useful for storing and organizing related data that can be easily accessed using a unique key.

Creating Dictionaries:

There are several ways to create a dictionary in Python:

- **Using Curly Braces {}**: Create a dictionary by placing key-value pairs inside curly braces.
- **Using the dict() Function**: Create a dictionary using the built-in dict() constructor.
- **Using a Sequence of Key-Value Pairs**: Create a dictionary from a list or tuple containing key-value pairs.

Example:

```
# create a dictionary using {}
person = {"name": "Jessa", "country": "USA", "telephone": 1178}
print(person)

# create a dictionary using dict()
person = dict({"name": "Jessa", "country": "USA", "telephone": 1178})
print(person)

# create a dictionary from sequence having each item as a pair
person = dict([("name", "Mark"), ("country", "USA"), ("telephone", 1178)])
print(person)

# create dictionary with mixed keys keys
# first key is string and second is an integer
sample_dict = {"name": "Jessa", 10: "Mobile"}
print(sample_dict)

# create dictionary with value as a list
person = {"name": "Jessa", "telephones": [1178, 2563, 4569]}
print(person)
```

Accessing Dictionary Elements:

You can access values in a dictionary in two common ways:

- **Using Square Brackets []**: Retrieve a value by specifying its key.
- **Using the get() Method**: Retrieve a value by passing the key to the get() method.

Example:

```
# create a dictionary named person
person = {"name": "Jessa", "country": "USA", "telephone": 1178}
# access value using key name in []
print(person['name'])
#  get key value using key name in get()
print(person.get('telephone'))
```

Adding Items:

New items can be added to a dictionary in two common ways:

- **Using Key-Value Assignment**: Assign a value to a new key using the assignment operator (=).
- **Using the update() Method**: Add one or more key-value pairs from another dictionary or a sequence of pairs.

Example:

```
person = {"name": "Jessa", 'country': "USA", "telephone": 1178}
# update dictionary by adding 2 new keys
person["weight"] = 50
person.update({"height": 6})

# print the updated dictionary
print(person)
```

Updating Items:

You can change the value of an existing dictionary item in two ways:

- **Using the Key Name**: Assign a new value directly to an existing key.
- **Using the update() Method**: Update the value by passing a key-value pair to the update() method.

```
person = {"name": "Jessa", "country": "USA"}

# updating the country name
person["country"] = "Canada"
# print the updated country
print(person['country'])
# Output 'Canada'

# updating the country name using update() method
person.update({"country": "USA"})
# print the updated country
print(person['country'])
```

Deleting Items:

Python provides several ways to remove items from a dictionary:

Method	Description
pop(key)	Removes an item using its key and returns its value.
popitem()	Removes and returns the last inserted key-value pair.
del	Deletes a specific item using its key.
clear()	Removes all items from the dictionary.
del dictionary_ name	Deletes the entire dictionary.

Example:

```
person = {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50,
'height': 6}
# Remove last inserted item from the dictionary
deleted_item = person.popitem()
print(deleted_item)  # output ('height', 6)
# display updated dictionary
print(person)
# Remove key 'telephone' from the dictionary
deleted_item = person.pop('telephone')
print(deleted_item)  # output 1178
# display updated dictionary
print(person)
# delete key 'weight'
del person['weight']
# display updated dictionary
print(person)
# remove all item (key-values) from dict
person.clear()
# display updated dictionary
print(person)  # {}
# Delete the entire dictionary
del person
```

Exercise

Create a dictionary named `student_info` containing the following information about a student: "name," "age," and "major." Access and print the student's name from the dictionary.

Answer:

```
# Create a dictionary containing student information
student_info = {
    «name»: «Alice»,
    «age»: 20,
    «major»: «Computer Science»
}

# Access and print the student›s name
print(f»Student›s name: {student_info[‹name›]}»)
```

Exercise

You have a dictionary named `car_info` with information about a car. Add the following information to the dictionary:

- "model": "Toyota Camry"
- "year": 2022

Update the "year" to 2023, and then delete the "color" entry from the dictionary if it exists.

Answer

```
# Initial dictionary containing car information
car_info = {
    «make»: «Toyota»,
    «color»: «Blue»
}

# Add model and year to the dictionary
car_info[«model»] = «Toyota Camry»
car_info[«year»] = 2022

# Update the year to 2023
car_info[«year»] = 2023

# Delete the «color» entry if it exists
if «color» in car_info:
    del car_info[«color»]

# Print the modified dictionary
print(«Updated car information:», car_info)
```

4.4 Function in Python

Imagine having a magic box that can solve a math problem or do a task whenever you ask. In Python, this magic box is called a function!
A function is like a helper in your code. It performs a specific job for you. You can call (ask) it to do the task anytime. It saves time because you don't have to repeat the same code again and again.

A **function** is a named block of code that performs a specific task. Functions help you avoid writing the same code multiple times, making programs easier to read, maintain, and reuse.
Instead of repeating code, you can write it once inside a function and call it whenever needed. This follows the **DRY (Don't Repeat Yourself)** principle.

Benefits of Functions

- **Code Reusability:** Write code once and use it many times.
- **Better Organization:** Break large programs into smaller, manageable parts.

- **Fewer Errors:** Reduce code duplication and make programs easier to maintain.

Types of Functions

Python supports two main types of functions:

Built-in Functions: Functions that come with Python, such as:

- print()
- input()
- type()
- range()

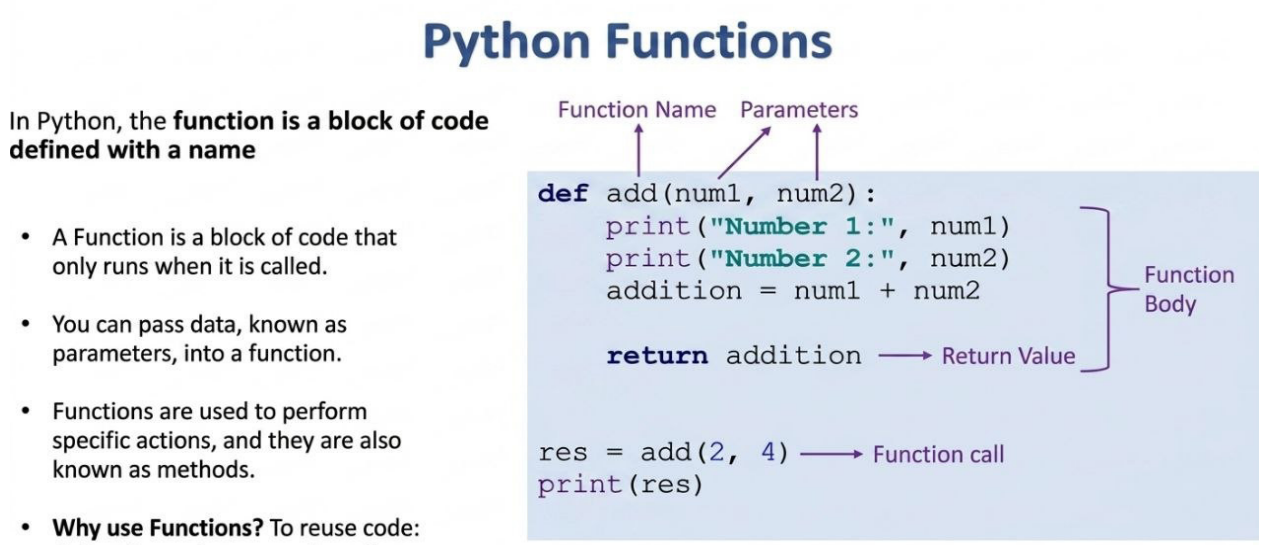
User-Defined Functions: Functions created by programmers to perform specific tasks.

Create a Function

To create a function in Python, use the def keyword followed by the function name and parentheses ().

A function can optionally receive parameters, perform a task using its code block, and return a result using the return statement.

```
def function_name(parameter1, parameter2):
        # function body
        # write some action
return value
```



Function Components

Function Name: The name used to identify and call the function.
Parameters: Optional values passed to the function.
Function Body: The block of code that performs the required task.
Return Value: The result returned by the function (optional).

Example:

```
# function
def message():


   print("Welcome to PYnative")
# call function using its name
message()
```

Note

Python uses **indentation** to define the function body. Unlike some programming languages, Python does not use curly braces {} to group code blocks.

Return Value From a Function

In Python, the return statement is used to send a value back from a function to the caller. This value is the **result (output)** produced by the function after execution.

Functions with Inputs (Parameters)

Functions in Python can accept **parameters** to make them more flexible and reusable. Parameters allow you to pass data into a function so it can operate on different values each time it is called.
When a function is defined with parameters, it expects values (called arguments) to be provided during the function call.

Types of Function Parameters in Python

Python supports different ways of passing parameters to functions:

1. Positional Arguments

Arguments are passed in the same order as defined in the function.

```
def greet(name, age):
    print("Name:", name)
    print("Age:", age)
greet("Ali", 25)
```

2. Keyword Arguments

Arguments are passed using parameter names, so order does not matter.

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

greet(age=25, name="Ali")
```

3. Default Arguments

You can assign default values to parameters. If no value is passed, the default is used

```
def greet(name, age=18):
    print("Name:", name)
    print("Age:", age)

greet("Ali")
```

4. Variable-Length Arguments

Used when you don't know how many arguments will be passed.

```
def add(*numbers):
    total = sum(numbers)
    print("Sum:", total)

add(1, 2, 3, 4)
```

Lesson 4
Assessment



Part 1: Multiple Choice (MCQ)

1. Which of the following creates a list in Python?
   - A) fruits = (3 ,2 ,1)
   - B) fruits = [3 ,2 ,1]
   - C) fruits = {3 ,2 ,1}
   - D) fruits = "3 ,2 ,1"

2. What is the main difference between a list and a tuple?
   - A) Lists use () and tuples use []
   - B) Tuples can be changed, lists cannot
   - C) Lists can be changed, tuples cannot
   - D) There is no difference

3. Q3. How do you access the value associated with the key "name" in a dictionary student?
   - A) student("name")
   - B) student["name"]
   - C) student.name
   - D) student{name}

4. What keyword is used to define a function in Python?
   - A) func
   - B) function
   - C) define
   - D) def

5. What does the return keyword do in a function?
   - A) Prints a value to the screen
   - B) Sends a value back to where the function was called
   - C) Stops the program
   - D) Creates a new variable

Lesson 4 Assessment



6. What is the output of this code?

```
def add(a, b):
    return a + b
result = add(3, 7)
print(result)
```

A) 3
B) 7
C) 10
D) Error

Part 2: Easy Questions

7. Create a list named colors with 4 colors. Print the first and last item.

8. Write a program that creates a list of 5 numbers and uses a for loop to print each number.

9. Given the list fruits = ["apple", "banana", "cherry"], write code to:

- Add "orange" to the end
- Insert "mango" at position 1
- Remove "banana"

10. What is a tuple? Why can't you change its elements? Give an example.

11. Create a dictionary called student with keys: "name", "age", "grade". Print each value.

12. Write code that adds a new key "school" with value "Al Noor Academy" to the student dictionary, then updates "age" to 17.

13. What is the difference between a list, a tuple, and a dictionary? Fill in the table.

Feature	List	Tuple	Dictionary
Symbol			
Mutable?			
Ordered?			
Access by			

Lesson 4
Assessment



14. Write a function called greet that takes a name as a parameter and prints "Hello, [name]!".
15. Write a function called multiply that takes two numbers and returns their product. Call the function and print the result.
16. What is the difference between print() inside a function and return? Explain with an example.
17. Write a function is_even that takes a number and returns True if it's even, False if it's odd.
18. Write a function calculate_average that takes a list of numbers and returns the average.
19. What is the output of this code?

```
def get_info():
    return {"name": "Khalid", "age": 17}
info = get_info()
print(info["name"])
print(info["age"])
```

20. Write a program that:
    - Creates a list of 5 student names
    - Uses a function to print each name with a greeting
    - Uses a dictionary to store one student's full info (name, age, grade)

Task 4: Data Management

Organize data into structures and use functions.

Action: Instead of variables, store each student's data as a **Dictionary**. Store all student dictionaries inside a **List**.

Requirement: Create a function **add_student()** to handle data entry and a function **print_report()** to display the stored list.

Lesson 5

Interactive Pythone Programs and Real-World APPlications



- Object-Oriented Programming (OOP)
- Functional Programming (FP)
- From Scripts To Software
- Types of Errors in Python

programming paradigm

A programming paradigm is a fundamental style, approach, or blueprint for writing and structuring computer code. It isn't a specific programming language, but rather a way of thinking about how to solve problems and organize logic. Different languages support different paradigms, and some languages (like Python or C#) are "multi-paradigm," meaning they let you mix and match styles.

Think of it like building a house: you could use a modern, minimalist approach, a classic traditional style, or a modular prefabricated method. The goal is the same (a house), but the philosophy and process are completely different.
Here is a breakdown of 3 of the most popular and widely used **programming paradigms, explained simply:**

5.1 Object-Oriented Programming (OOP)

OOP is centered around the concept of **"Objects."** Instead of thinking about code as a series of steps, you think of it as a collection of self-contained entities that have data (attributes) and can perform actions (methods).
It is designed to mimic how we interact with the real world.

Core Idea

Group related data and behaviors together.

Key Concepts

Encapsulation, Inheritance, Polymorphism, and Abstraction.
Real-World Example: If you are building a school system, you would create a `Student` object. This object has data (Name, Age, Grades) and actions it can perform (Study, AttendClass).

Popular Languages

C#, Java, Python, C++.

5.2 Functional Programming (FP)

Functional programming treats computation as the evaluation of mathematical functions and avoids changing state or using mutable (changeable) data.

In FP, you pass data through a sequence of functions. The functions take an input and return an output without altering anything outside the function itself.

Core Idea

Code is built using pure, reusable functions, and data cannot be changed after it is created (Immutability).

Key Concepts

Pure Functions, Immutability, and First-Class Functions.

Real-World Example: Think of an assembly line. Raw materials go into a machine, the machine processes them and outputs a product, but the machine itself doesn't change, and it doesn't modify anything else in the factory.

Popular Languages

Haskell, JavaScript (supports it heavily), Scala, Python.

5.3 From Scripts to Software

you have learned the building blocks of Python: variables, operators, loops, and functions. But now, we are going to bridge the gap between "writing code" and "building an application." We are moving from simple scripts that run once, to Professional Software that is stable, interactive, and smart.

Case 1: Ensuring Data Integrity (Preventing Duplicate IDs)

The Problem: Your school registration system requires every student to have a unique ID. If two students are assigned the same ID, the entire database becomes corrupted. **The Solution:** Use a Set combined with a membership check.

```
# System to track unique student IDs in the registry
registered_ids = {1001, 1002, 1003}

def add_student_id(new_id):
    # Check if the ID already exists in the system
    if new_id in registered_ids:
        print(f»Error: The ID {new_id} is already registered!»)
        return False
    else:
        # Add the unique ID to the set
        registered_ids.add(new_id)
        print(f»Success: Student {new_id} added successfully.»)
        return True

# Testing the robustness
add_student_id(1004) # Valid addition
add_student_id(1001) # Duplicate detected and rejected
print(f»Current Registry: {registered_ids}»)
```

Case 2: Defensive Programming (The Input Guardian)

The Problem: When you ask a user for a grade, they might type “Excellent” or leave it blank, causing your program to crash. **The Solution:** Use a try-except block inside a while loop to create a “gatekeeper” function.

```
def get_valid_grade():
    while True:
        try:
            # Attempt to capture and convert input
            score = input(«Please enter the student›s grade (0-100): «)
            val = int(score)
            if 0 <= val <= 100:
                return val
            else:
                print(«Input error: Grade must be between 0 and 100.»)
        except ValueError:
            # Handle non-integer inputs gracefully
            print(«Type error: Please enter a valid whole number.»)

# The program forces the user to provide correct data
student_grade = get_valid_grade()
print(f»The grade {student_grade} has been recorded safely.»)
```

Case 3: Creating a Fair Random Selection (Lottery System)

The Problem: Selecting a "Student of the Week" manually can seem biased. You need an automated, impartial system. **The Solution:** Use the random module with validation to handle empty class lists.

```
import random

def select_winner(student_list):
    # Ensure the list is not empty before picking a winner
    if len(student_list) == 0:
        print(«System Alert: No students available in the list.»)
        return None

    # Pick a lucky student randomly from the list
    print(«--- Running Lottery System ---»)
    winner = random.choice(student_list)
    print(f»The winner for this week is: {winner}»)
    return winner

# Execute the selection
class_list = [«Omar», «Laila», «Hassan», «Nour»]
select_winner(class_list)
```

Case 4: The Continuous Menu Interface

The Problem: A program that executes one task and then closes is not a "system." The Solution: Use a while True loop as a "Menu Driver" to keep the program alive until the user chooses to exit

```
def main_menu():
    while True:
        print(«\n--- Student Management System ---»)
        print(«1. Add Data | 2. View All | 3. Exit»)
        choice = input(«Select an option (1-3): «)

        # Branching logic based on user interaction
        if choice == ‹1›:
            print(«Accessing database to add student...»)
        elif choice == ‹2›:
            print(«Displaying all student records...»)
        elif choice == ‹3›:
            print(«Exiting system. Goodbye!»)
            break
        else:
            print(«Invalid choice, please try again.»)
# Entry point for the application
main_menu()
```

Case 5: Clean Reporting (Math & Formatting)

The Problem: Raw averages often contain long decimal sequences (e.g., ٨٥,٦٦٦٧) that look unprofessional on reports. **The Solution:** Use math.ceil for professional rounding and f-strings for alignment.

```
import math

def generate_report(name, grades):
    # Calculate average
    total = sum(grades)
    average = total / len(grades)

    # Professional rounding using math module
    final_score = math.ceil(average)

    print(«-» * 30)
    print(f»Student: {name}»)
    print(f»Raw Average: {average:.2f}») # Format to 2 decimals
    print(f»Official Final Score: {final_score}»)
    print(«-» * 30)

# Running the report for a student
generate_report(«Ahmed», [85, 92, 78, 88])
```

4.4Types of Errors in Python

Errors are problems in a program that prevent it from running correctly. Python shows an error message to help programmers find and fix the problem

There are three main types of errors in Python:

- Syntax errors
- Runtime errors
- Logical errors

Syntax Errors

A syntax error happens when Python code does not follow the correct rules (grammar) of the Python language

Python cannot run the program until the syntax error is fixed

Example:

```
print(«Hello»
```

Output:

SyntaxError: '(' was never closed

Common causes:

- Missing parentheses
- Missing quotation marks
- Wrong indentation
- Missing colon

Example:

```
if x > 5
   print(x)
```

Output:

SyntaxError: expected ':'

Runtime Errors (Exceptions):

A runtime error happens while the program is running.

The code is written correctly, but something unexpected happens.

Example:

```
x = 10
y = 0

print(x / y)
```

Output

ZeroDivisionError: division by zero

Example:

```
numbers = [1, 2, 3]


print(numbers[5])
```

Output

IndexError: list index out of range

Common runtime errors:

Error	Meaning
ZeroDivisionError	Dividing by zero
TypeError	Using wrong data types
ValueError	Wrong value
IndexError	Wrong list index
FileNotFoundError	File does not exist

Logical Errors

A logical error happens when the program runs but gives the wrong result.

Python does not show an error message because the code is valid.

Example:

```
length = 5
width = 10
area = length + width
print(area)
```

Output:

15

But the correct area should be:

```
50
```

How to Handle Errors : Python uses try and except to handle exceptions

Example

```
try:
    number = 10 / 0
except ZeroDivisionError:
    print(«Cannot divide by zero»)
```

Output

Cannot divide by zero

Error Type	When it Happens	Example
Syntax Error	Code rules are broken	Missing)
Runtime Error	During execution	Divide by zero
Logical Error	Wrong program result	Wrong calculation

Lesson 5
Assessment



Part 1: Multiple Choice (MCQ)

1. What is an error in Python?
   - A) A feature of Python
   - B) A problem that prevents correct execution
   - C) A type of variable
   - D) A loop

2. Which error happens when Python grammar rules are wrong?
   - A) Runtime Error
   - B) Logical Error
   - C) Syntax Error
   - D) Value Error

3. What error occurs in this code?

```
print("Hello"
```

   - A) IndexError
   - B) SyntaxError
   - C) TypeError
   - D) ValueError

4. What type of error happens here?

```
10 / 0
```

   - A) Syntax Error
   - B) Logical Error
   - C) ZeroDivisionError
   - D) IndexError

5. Which error happens when you access an item that does not exist in a list?
   - A) IndexError
   - B) SyntaxError
   - C) TypeError
   - D) NameError

6. A program runs but gives the wrong answer. What type of error is this?
   - A) Syntax Error
   - B) Runtime Error

Lesson 5
Assessment



C) Logical Error
D) File Error

7. What keyword is used to handle errors in Python?
A) check
B) try
C) error
D) catch

8. What is the output problem in this code?

```
age = "20"

print(age + 5)
```

A) Syntax Error
B) TypeError
C) IndexError
D) Logical Error

9. wWhich statement is correct?
A) Syntax errors happen after the program finishes
B) Runtime errors happen while the program runs
C) Logical errors always show an error message
D) Python cannot have errors

10. Why are errors important to learn?
A) To make programs harder
B) To find and fix problems in code
C) To delete Python
D) To avoid writing programs

11. Which data structure in Python is best suited for storing a collection of unique identifiers where duplicates must be automatically prevented?
a) List
b) Tuple
c) Set
d) Dictionary

12. What is the primary purpose of the 'try-except' block in Python?
a) To speed up loops



b) To prevent crashes due to runtime errors
c) To define functions
d) To convert strings to integers

13. Which module would you import if you need to calculate the ceiling (rounding up) of a number?
a) random
b) math
c) sys
d) os

14. In a menu-driven application, what is the significance of a 'while True' loop combined with a 'break' statement?
a) It restarts the program
b) It creates a continuous loop that ends on a specific user command
c) It crashes the system
d) It ignores errors

15. Why is it considered a 'best practice' to use defensive programming in a student management system?
a) It reduces memory usage
b) It makes code shorter
c) It ensures the system stays stable with invalid input
d) It generates a GUI

16. Which of the following best describes the functionality of `random.choice()`?
a) It returns a random integer
b) It selects one random element from a sequence
c) It shuffles a list
d) It generates a float

Part 2: Essay & Application

17. Explain what errors are in Python and why programmers need to understand different types of errors.

18. Describe the difference between Syntax Errors, Runtime Errors, and Logical Errors with examples.

19. What is a Syntax Error? Explain its causes and write a Python example



that produces a Syntax Error.

20. Explain Runtime Errors (Exceptions) and mention three common types of runtime errors with examples.
21. What is a Logical Error? Why is it considered difficult to find compared to other errors? Give an example.
22. Explain the purpose of using try and except in Python. Write a simple example.
23. Compare between Syntax Error and Runtime Error in terms of when they happen and how Python handles them.
24. A program runs successfully but gives an incorrect output. What type of error is this? Explain how to fix it.
25. Write a short explanation about the following errors:
    - ZeroDivisionError
    - TypeError
    - IndexError
26. Imagine you are debugging a Python program. Describe the steps you would follow to find and fix errors in the code.
27. Explain the difference between using a 'List' and a 'Set' for storing student IDs. When would you choose a Set?
28. Write a code snippet using `try-except` to ask for a student's age. Why is this more professional than a simple `input()`?
29. How does a `while True` loop with `break` logic facilitate a menu-driven program?
30. Discuss the role of built-in modules like 'math' and 'random'. How do they help programmers avoid "reinventing the wheel"?
31. In your Student Management System, why is it important to use functions for specific tasks (add, view, search)?
32. If a user inputs an invalid grade multiple times, how would you modify your `get_valid_grade` function to notify them of their specific error?
33. Define 'Defensive Programming' in your own words. Why is it critical for public software?
34. Describe the logical process for sorting a list of student records by



their grades.

35. **Explain the benefits of using a Dictionary to represent a student's record versus using individual variables.**
36. **Describe a scenario where `math.ceil()` is more appropriate for grade reporting than `math.floor()`.**
37. **Why are 'f-strings' preferred in professional reports over simple comma-separated print statements?**
38. **When creating a menu-driven system, how do you handle an option that has not been implemented yet?**
39. **Briefly outline the five stages of your Capstone project. How does completing Stage 4 (Data Structures) make Stage 5 (Integration) easier?**
40. **How does using a 'Set' optimize a search function compared to checking for existence in a large List?**

Task 3: The Professional Integration

Convert the system into a menu-driven program (using while True and a main_menu function).

Requirements:

Use a Set to store Student_IDs to ensure no duplicates.

Use try-except blocks to protect all numeric inputs (Age/Grades) from crashes.

Add a "Random Student Picker" using the random module.

Glossary

Term	Description
Programming	Writing instructions and commands to make a computer perform tasks.
Python	A programming language used to write programs and solve problems.
Code	A set of instructions written in a programming language.
Syntax	The correct rules for writing commands in a programming language.
Syntax Error	An error that occurs when the code does not follow Python language rules.
Runtime Error	An error that occurs while the program is running after execution starts.
Exception	A problem or error that happens during program execution and needs to be handled.
Program Execution	The process of running a program and performing its instructions.
Debugging	The process of finding and fixing errors in code.
Error Message	A message displayed by Python to explain the type or location of an error.
Type Error	An error that occurs when an operation uses an inappropriate data type.
Value Error	An error that occurs when a function receives an incorrect value.
Zero Division Error	An error that occurs when trying to divide a number by zero.
Input	Data entered into the program by the user.
Output	The result or information displayed by the program.
Exception Handling	A method used to manage errors during program execution using try and except.

Try	A block of code used to test instructions that may cause an error.
Except	A block of code used to handle an error when it occurs
Interpreter	A program that reads and executes programming instructions.
Algorithm	A step-by-step set of instructions used to solve a problem.
Argument	A value passed to a function when it is called.
Debugging	Finding and fixing mistakes in a program.
Indentation	Spaces at the beginning of a line that define code blocks in Python.