

Topic 4: Python Programming

Learning Outcomes:

- (a) Define types of programming languages, programming paradigms and language translators

Introduction to Programming Language

- A programming language is a way to **communicate with a computer**.
- It allows us to **write instructions** that tell the computer what to do.
- These instructions **follow a specific syntax** (structure) and rules.

Introduction to Programming Language

- Programming languages are used to **build** software, games, websites, and more.
- **Examples** of programming languages include Python, Java, C++, and JavaScript.
- Each language has its own strengths and is suited for different tasks.

DEFINITION - Programming Language

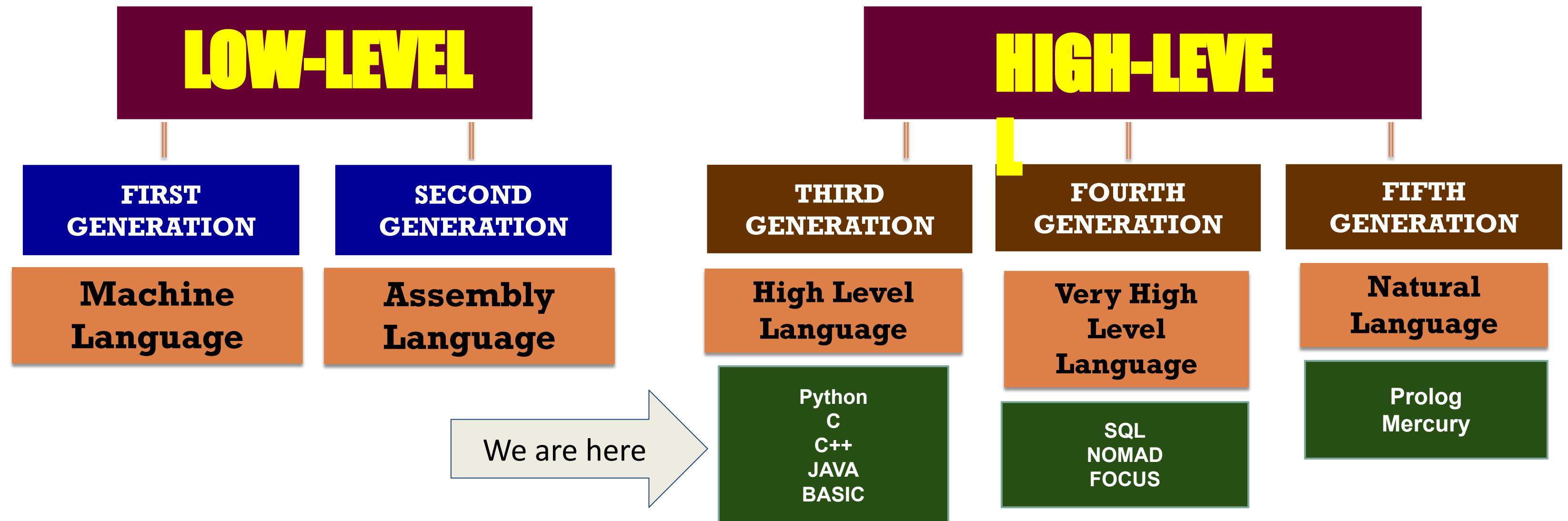
What is **Programming Language**?

- A ***programming language*** is a formal set of rules used to write instructions that a computer can understand and execute.

Source: Britannica, 2024

Types of Programming Language:

TYPES OF PROGRAMMING LANGUAGES



Types of Programming Language:

Low-Level Language

- A **low-level language** is a programming language that is *machine-dependent*.
 - A *machine-dependent language* runs on only one particular type of computer.
 - These programs are not easily portable to other types of computers.

Low-Level Language: Machine Language

- ***Machine language*** is defined as a language that consists of strings of binary digits (1 and 0) to represent instructions to computer.
- It is the natural language of a computer.

To calculate $wages = rates * hours$ in machine language:

100100 010001 // Load (*input rates*)

100110 010010 // Multiply *with hours*

100010 010011 // Store (*result of wages*)

Low-Level Language: Machine Language

Advantages

1. Ready for immediate execution.
2. Instructions are readily understood by the computer.

Disadvantages

1. Machine-dependent.
2. Programs tend to be lengthy.
3. Coding in machine language is tedious and time-consuming.

Low-Level Language: Assembly Language

- **Assembly language** is the second generation of programming languages consists of *English-like abbreviations*.
- Programmer writes instructions using symbolic instruction codes. Symbolic instruction codes are meaningful abbreviations.

Example: x86, arm a32

Example:

ADD for *addition*
CMP for *compare*
LOAD for *load*
MULT for *multiply*

Low-Level Language: Assembly Language

Advantages

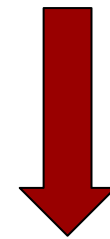
1. Instructions are easier to learn compared to machine language.
2. Easy to understand and use. Assembly language use mnemonics instead of using binary code.

Disadvantages

1. Machine-dependent.
2. Long and tedious to write.

Low-Level Language:

Machine Language



```
10110000 01100001  
10110010 01000010  
00000001  
11100010
```

Assembly Language



```
MOV AL, 61h  
MOV DL, 42h  
ADD AL, 1  
LOOP
```

Types of Programming Language:

High-Level Language

- A **high-level language** is a programming language that is closer to *natural language* and easier to work with than a low-level language.
 - A high-level language is a programming language that is *machine-independent*.
 - A *machine-independent* language can run on many different types of computers and operating systems.

High-Level Language

The equation : $wages = rate \times hours$ can be written in Python as:

```
wages = rate * hours
```

Examples of high-level language:

Java, C, C++, BASIC, Pascal, FORTRAN (for Scientific),
COBOL (for Business)

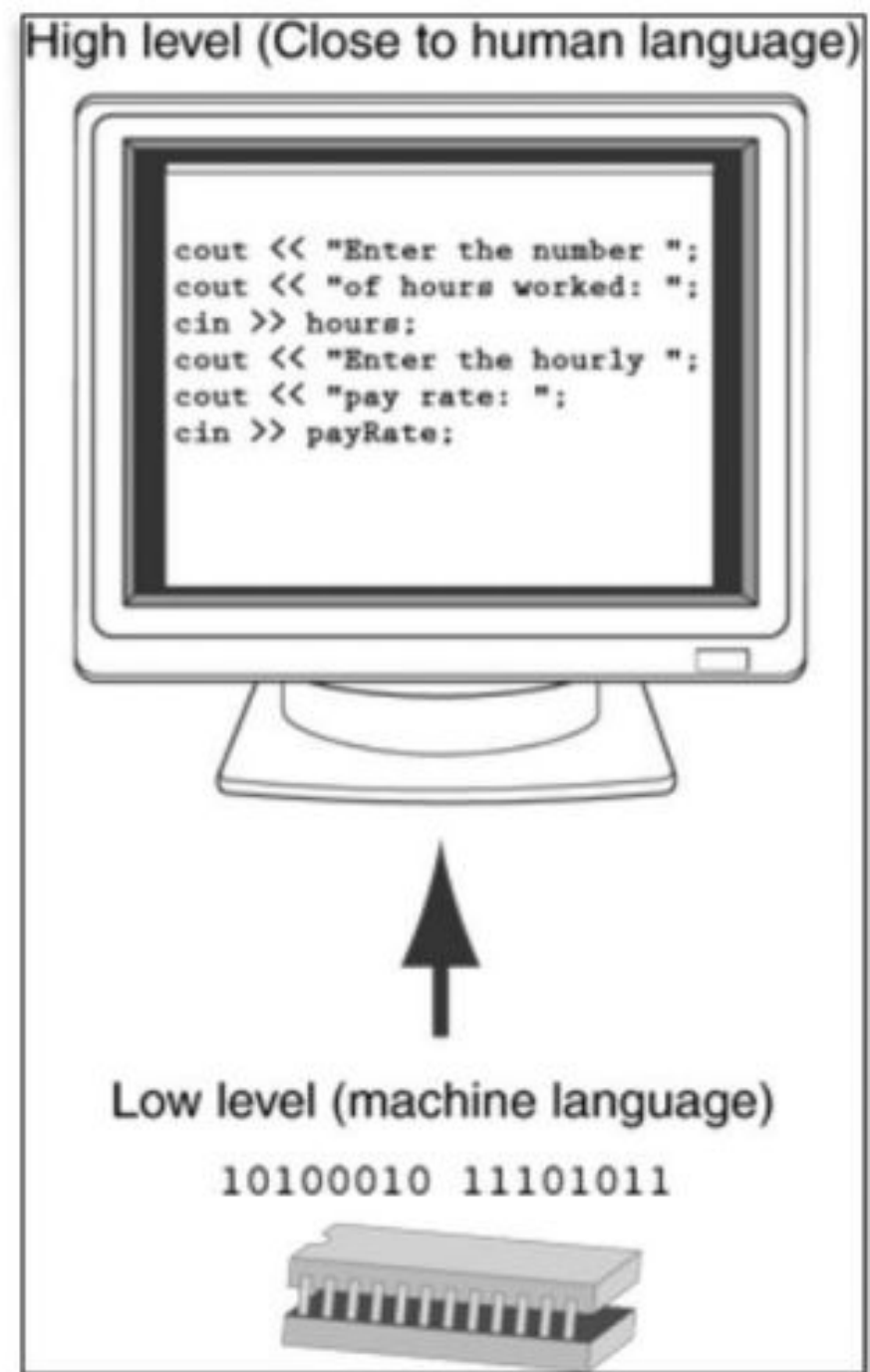
High-Level Language:

Advantages

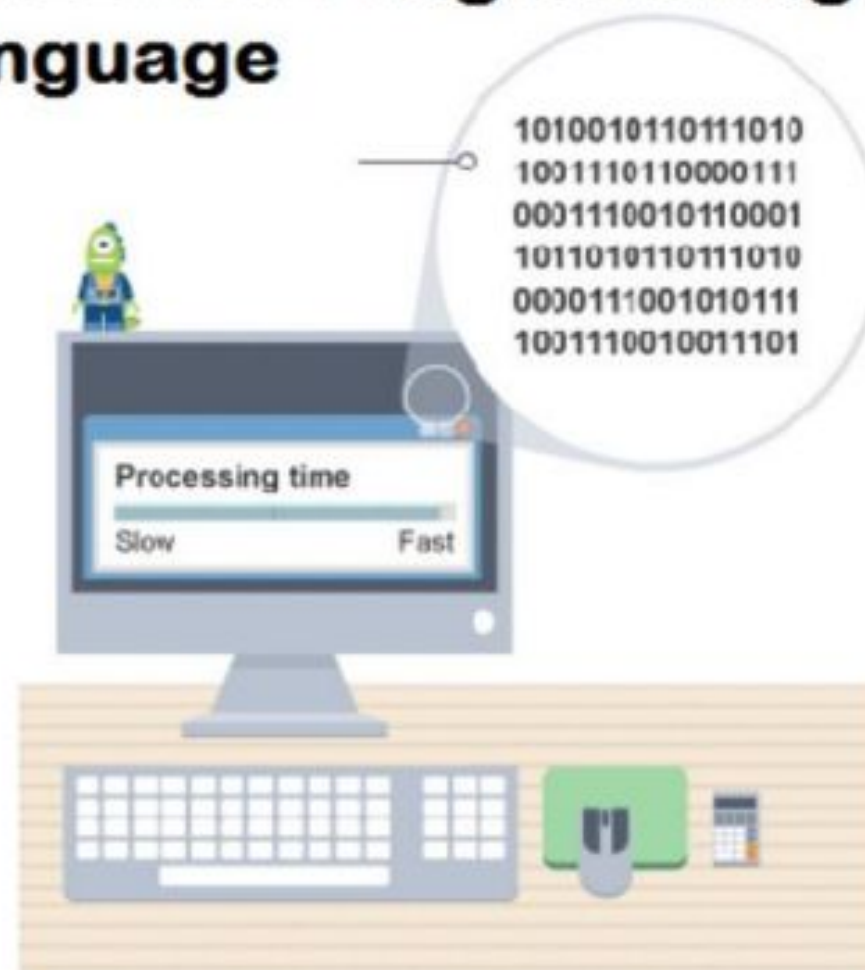
1. The English-like instructions are easier to learn compare to low-level language.
2. The instructions can run on many different types of computers

Disadvantages

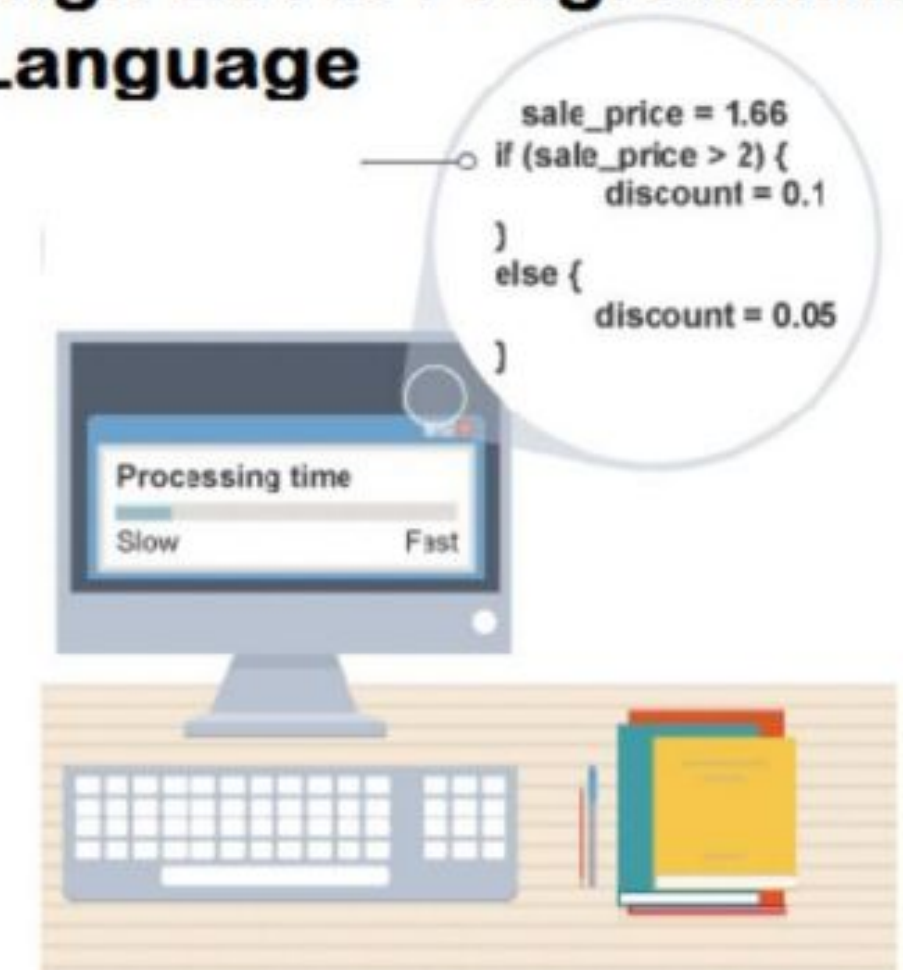
1. Not efficient as low level languages.
2. Program generally run slower. (Need to be translated / compile)



Low Level Programming Language



High Level Programming Language



Low-Level Language	High-Level Language
Consists of binary digits or English-like abbreviations to form instructions.	Consists of English-like words to form instructions.
Machine Language does not need translator.	Need translator to convert to low-level language.
Machine dependant.	Machine independent.
Difficult to learn, modify and far from human language.	Easy to learn, modify and close to human language.

QUICK REVIEW

```
# Get the first number from the user
num1 = int(input("Enter the first number: "))




# Get the second number from the user
num2 = int(input("Enter the second number: "))

# Calculate the sum
sum = num1 + num2

# Print the sum
print("The sum of", num1, "and", num2, "is", sum)
```

1. State the type of programming language shown above.
2. Give one benefit of writing code in your answer above.

Introduction to Programming Paradigm

-  **A programming paradigm** is a way of thinking about and writing code — like a coding "style" or approach.
-  Different paradigms (like procedural, object-oriented, and logical) help solve problems in different ways.
-  Programming languages often support one or more paradigms to give developers flexibility when building software.

DEFINITION - Programming Paradigm

What is **Programming Paradigm**?

- A ***programming paradigm*** is a fundamental style or approach to programming that guides how code is written, structured, and executed.

Source: Britannica, 2024

Types of programming paradigm:

PROCEDURAL

OBJECT-ORIENTED

LOGIC

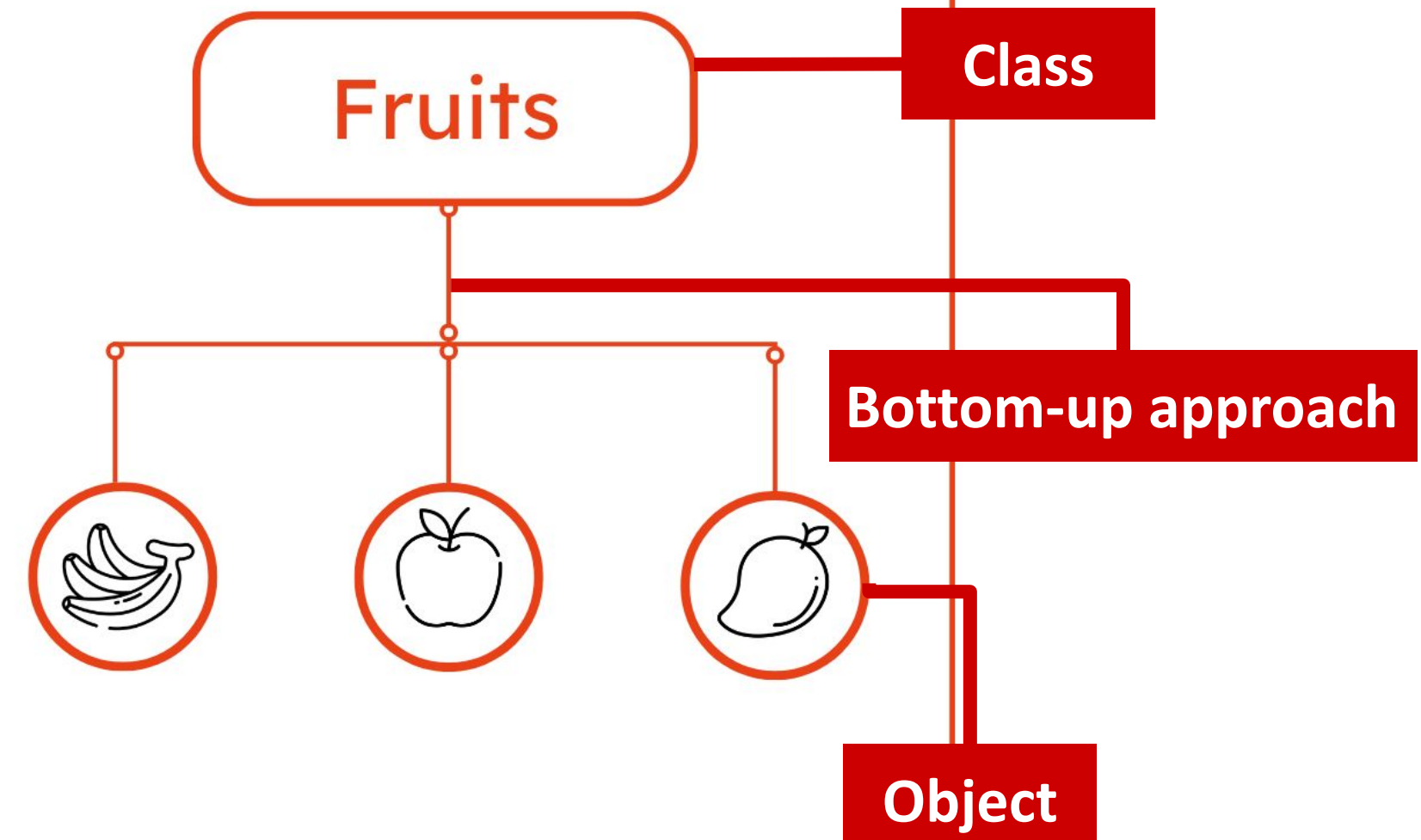
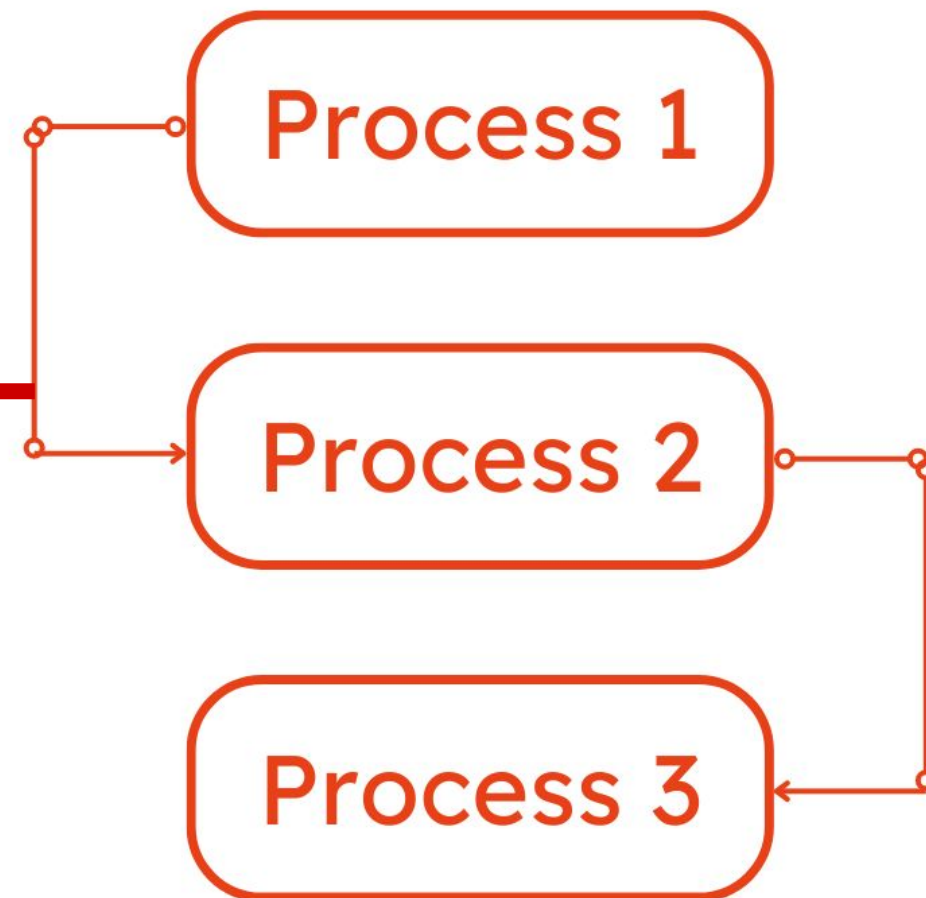
Procedural Programming vs Object Oriented Programming

Procedural Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called functions .	In object-oriented programming, the program is divided into small parts called objects .
Procedural programming follows a top-down approach .	Object-oriented programming follows a bottom-up approach .
Examples: C, FORTRAN, Pascal, Basic, Python etc.	Examples: C++, Java, Python, C#, etc.




Procedural

Object-Oriented

Top-down approach



Introduction to Language Translator

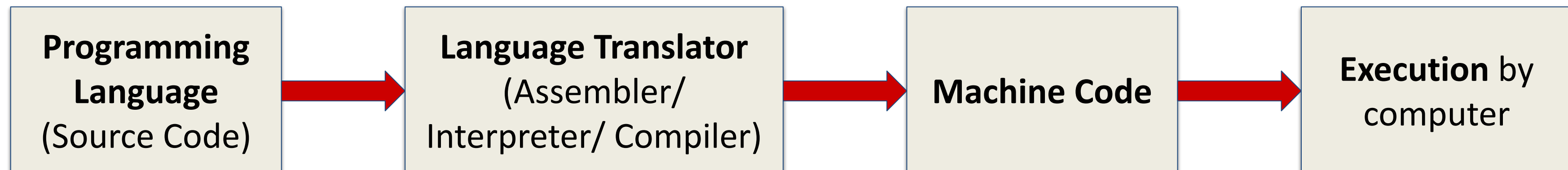
-  A language translator is a program that converts code written in one programming language into another (usually machine language).
-  It helps computers understand and execute code written by humans in high-level languages like Python, Java, or C++.
-  There are three main types: Assembler, Interpreter, and Compiler, each translating in different ways.

DEFINITION - Language translator

What is **Language Translator**?

- A ***language translator*** is a program that converts instructions written in one programming language into machine code

Source: Computers for Librarians (Third Edition), 2003



Types of Language Translator:

TRANSLATOR

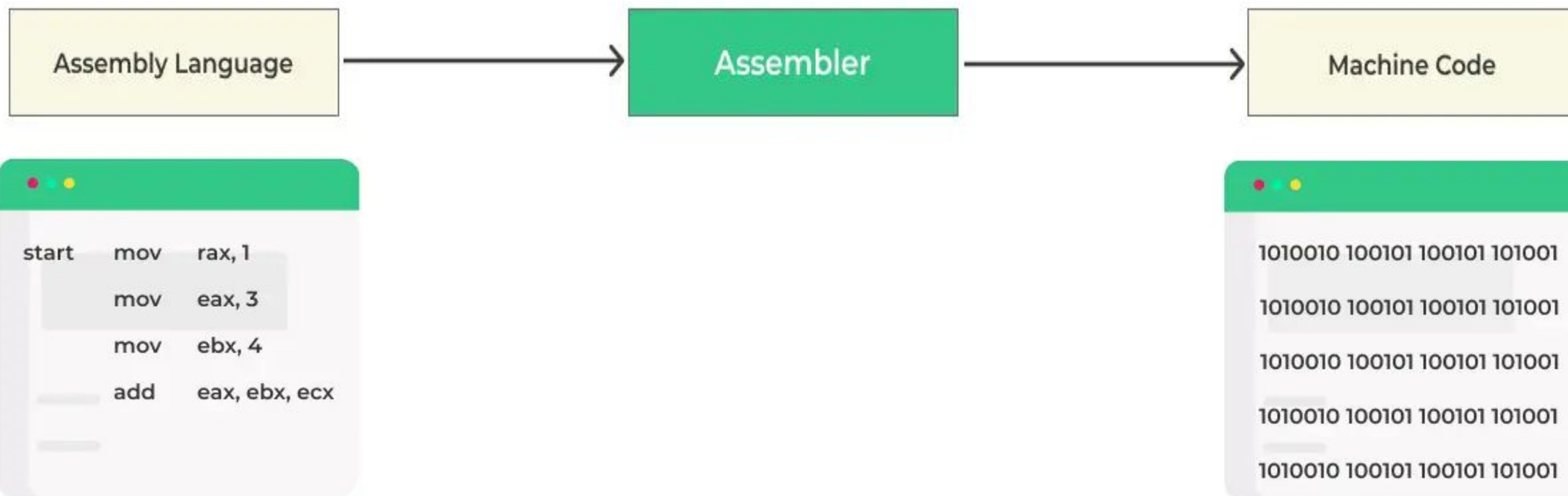
ASSEMBLER

INTERPRETER

COMPILER

1. Assembler

A program that translates **assembly language into machine language.**



1. Assembler

Examples:-

- **NASM** (Netwide Assembler)
- **MASM** (Microsoft Macro Assembler)
- **TASM** (Turbo Assembler)
- **FASM** (Flat Assembler)

2. Interpreter

A program that translates translate the program's high level instructions **line by line** into machine language instructions as the program is running.



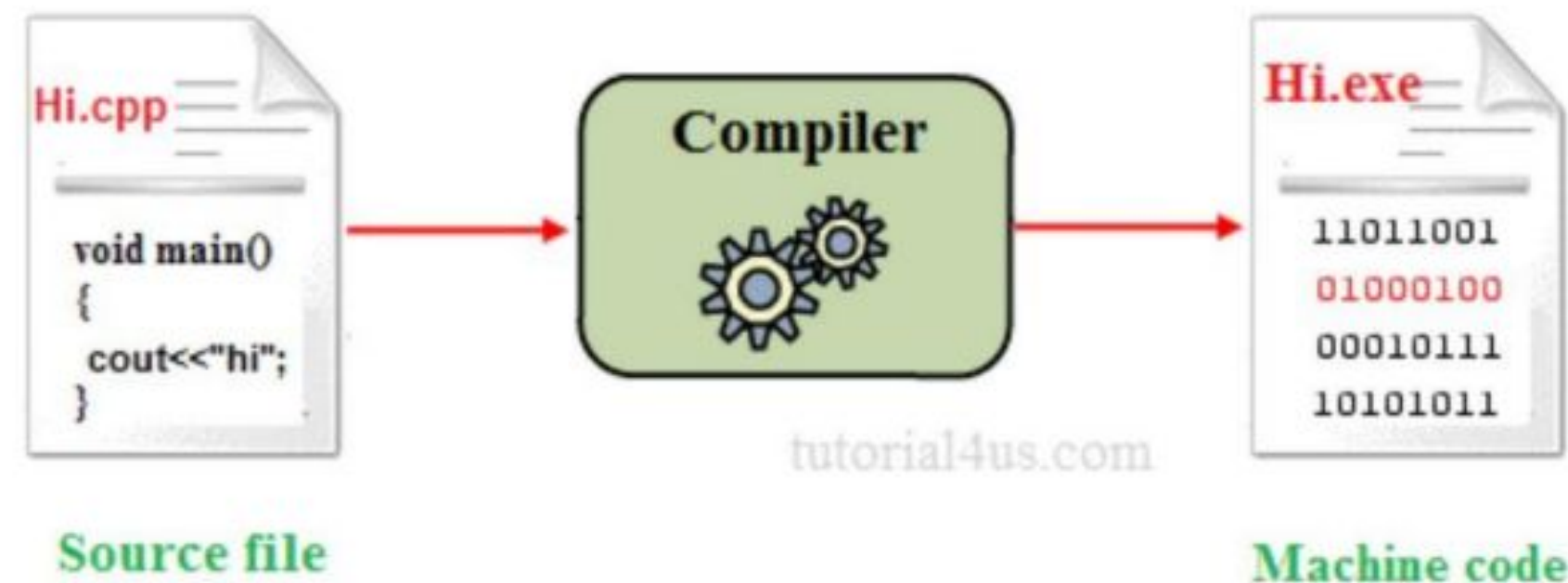
2. Interpreter

Examples interpreted programming language:

- **Python** programming language
- **Ruby** programming language
- **Javascript** programming language

3. Compiler

A program that translate **all of a program's high-level instructions** into machine language instructions before running the program



3. Compiler

Examples

- **Java** programming language
- **C++** programming language
- **C** programming language
- **Fortran** programming language
- **ADA** programming language
- **Pascal** programming language
- **Kotlin** programming language

Topic 4: Python Programming

Learning Outcomes:

- (b) State available platforms for writing Python programs

Requirements to Use Python

The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use. As a partial list, Python is available on:

- Linux and Unix systems
- Microsoft Windows
- Mac OS
- Cloud platform

Types of Python Platforms

There are two main types of Python platforms: **Online-Based** and **Local-Based**.

- **Online-Based platforms** allow you to write and run Python code directly in a web browser, with executions happening remotely on cloud servers.
- **Local-Based platforms** require installation on your local machine, where Python code is executed using your device's own resources.

Online-Based Python Platforms

Your code runs on **remote servers (the cloud)**. You just need a browser and an internet connection. All processing and file storage typically happen online.

Advantages:

- **No setup** needed — just open a link and start coding
- **Accessible anywhere** with an internet connection
- Easy to **share and collaborate** with others in real-time
- Often includes **free computing resources** like GPUs (e.g., Google Colab)

Online-Based Python Platforms

Limitations:

- Requires **stable internet**
- May have **performance limits** (especially with free versions)
- Not ideal for **complex software or offline work**

Online-Based Python Platforms

Google Colab

Google Colab is a **free browser-based Python notebook** provided by Google. It lets you write and execute Python code on cloud servers

Allows you to write code in **cells**, with outputs shown below each cell. Supports uploading datasets and saving notebooks to **Google Drive**



Local-Based Python Platforms

You download and install the software (e.g., PyCharm, Jupyter Notebook). It uses your computer's resources to run the Python interpreter and handle files.

Advantages:

- **No internet required** after installation
- Usually **faster and more powerful**, especially for large projects
- Fully **customizable and extendable** with plugins or libraries
- Better control over **local files, packages, and environment**

Local-Based Python Platforms

Limitations:

- Requires installation and setup
- Takes up disk space
- Can be quite technical for beginners to install

Local-Based Python Platforms

PyCharm

PyCharm is a **professional-grade IDE (Integrated Development Environment)** for Python development, created by JetBrains. It supports full software development workflows.

It is a powerful code editor with intelligent suggestions and refactoring. It also have integrated debugger, version control, and terminal.



Jupyter Notebook

Jupyter Notebook is an **interactive web-based tool** for coding in Python. It lets you write code, add notes, display charts, and run data analysis.

Even though it runs through the browser, it stills run locally on your computer using your computer resources

It split your code into cells for easier execution and experimentation. It also can combine code + text (Markdown) + output (charts, tables).



Platforms Comparison

Platform	Type	Ease of Use	Key Features
Google Colab	Online	Very Easy	No setup required, free access to resources such as CPU and RAM
PyCharm	Local	Moderate (Require setup)	Full-featured IDE with project management and AI coding assistant
Jupyter Notebook	Local	Moderate (Require setup)	Interactive cells with live code and outputs

Use Cases and Suggested Platforms

Running Python Without Installation

- Suitable for trying out Python without setting up a development environment.
- Runs entirely in a web browser with no need for local installation.
- Convenient for quick testing and experimentation.

Suggested Platform: Google Colab

Use Cases and Suggested Platforms

Managing Larger Projects with Multiple Files

- Ideal for structured projects that involve multiple scripts, folders, or modules.
- Supports features like project navigation, debugging tools, and version control.
- Suitable for more advanced or long-term development work.

Suggested Platform: PyCharm

Use Cases and Suggested Platforms

Interactive Coding with Instant Output

- Supports writing and executing code in smaller, organized cells.
- Provides immediate visual output for each section of code.
- Well-suited for data visualization, exploration, and explanation.

Suggested Platform: Jupyter Notebook

Topic 4: Python Programming

Learning Outcomes:

- c) Identify the components of a Python program (identifiers, variables, reserved words/keywords, data types, comments, import statements, input statements, output statements and indentation)



Table of Contents



- ☐ Identifiers
- ☐ Variables
- ☐ Reserved words/keywords
- ☐ Data types
- ☐ Comments
- ☐ Import statements
- ☐ Input statements
- ☐ Output statements
- ☐ Indentation



Definition of Identifier:-

The name used to identify variables, functions, classes, modules, and other objects in Python

Rules for Naming Identifiers:

1. **Can contain letters (a-z, A-Z), digits (0-9), and underscores (_)**
 -  Valid: `my_var`, `speed1`, `Calculate_Area`
 -  Invalid: `my-var` (hyphens are not allowed)

2. **Can use underscores but not special characters (@, \$, %)**
 -  Valid: `_my_variable`, `count_1`
 -  Invalid: `my$var`, `#number`

3. **Cannot start with a number**
 -  Valid: `age1 = 25`, `_value = 50`
 -  Invalid: `1variable = 10` (SyntaxError)

Rules for Naming Identifiers:

4. Case-sensitive

- `Variable` and `variable` are different identifiers.

Example:

```
age = 25
```

```
Age = 30
```

```
print(age)    # Output: 25
```

```
print(Age)    # Output: 30
```

5. Cannot be a reserved word (keyword)

-  Invalid: `class`, `def`, `if`, `while`, `return`



Review Activity : Identifier

Try this !



Exercise 1

Which of the following is a valid Python identifier?

- A. 2name
- B. name_2
- C. For
- D. class-name

Definition of Variable:-

A name that refers to a memory location where data is stored.

- In Python, variables are dynamically typed, □ don't need to declare their type before using them.
- The type is determined at runtime.

Feature	Identifiers	Variables
<i>Definition</i>	Name of functions, classes, and variables	Name referring to stored data
<i>Purpose</i>	Used to identify elements in code	Stores values that can change during execution
<i>Example</i>	<code>my_function, Student, count</code>	<code>x = 10, name = "John"</code>
<i>Keywords</i>	Cannot use reserved words	Variable names should not be reserved words

Identifiers: my_var, student_name, calculate_total

```
def calculate_total(price, quantity):  
    total = price * quantity    # 'total' is a variable  
    return total
```

Variables: price, quantity, result

```
price = 50  
quantity = 3  
result = calculate_total(price, quantity)
```



Review Activity : Variable

Try this !



Exercise 1

Which of the following is **NOT** a valid variable name in Python?

- a) `my_var`
- b) `total1`
- c) `break`
- d) `dataSet`



Review Activity : Variable

Try this !



Exercise 2

What is the **purpose** of a variable in Python?

- a) To reserve memory for loops
- b) To perform calculations
- c) To store data values
- d) To create comments



Review Activity : Variable

Try this !  Exercise 3

Is `Total_Score` and `total_score` the same variable in Python?

Definition of Reserved Word:-

The special words that have predefined meanings and cannot be used as identifiers (such as variable names, function names, or class names)

- Also known as **keyword**.
- These words are fundamental to the language syntax and serve specific purposes.

Characteristics of Reserved Words:

- **Cannot be used as identifiers** – You cannot name your variables, functions, or classes using reserved words.
- **Predefined meanings** – Each reserved word has a specific role in Python syntax.
- **Case-sensitive** – Python keywords must be written exactly as they are defined (e.g., True is different from true).
- **Fixed list** – The number of reserved words may change with new Python versions.

A list of some common reserved words in Python:

Reserved Word	Meaning
<code>if</code>	Used for conditional statements
<code>else</code>	Defines an alternative condition
<code>elif</code>	Used in multiple condition statements
<code>for</code>	Starts a loop that iterates over a sequence
<code>while</code>	Starts a loop that runs as long as a condition is true
<code>break</code>	Exits the current loop
<code>continue</code>	Skips the current iteration of a loop and moves to the next
<code>def</code>	Defines a function

A list of some common reserved words in Python:

Reserved Word	Meaning
<code>return</code>	Returns a value from a function
<code>class</code>	Defines a class
<code>import</code>	Imports modules into a program
<code>try</code>	Starts a try-except block for handling exceptions
<code>except</code>	Catches exceptions in a try-except block
<code>True</code>	Boolean value for true
<code>False</code>	Boolean value for false
<code>None</code>	Represents a null value



Review Activity : Reserved Word

Try this !



Exercise 1

Which of these is a Python reserved word (keyword)?

- a) name
- b) value
- c) def
- d) var



Review Activity : Reserved Word

Try this !



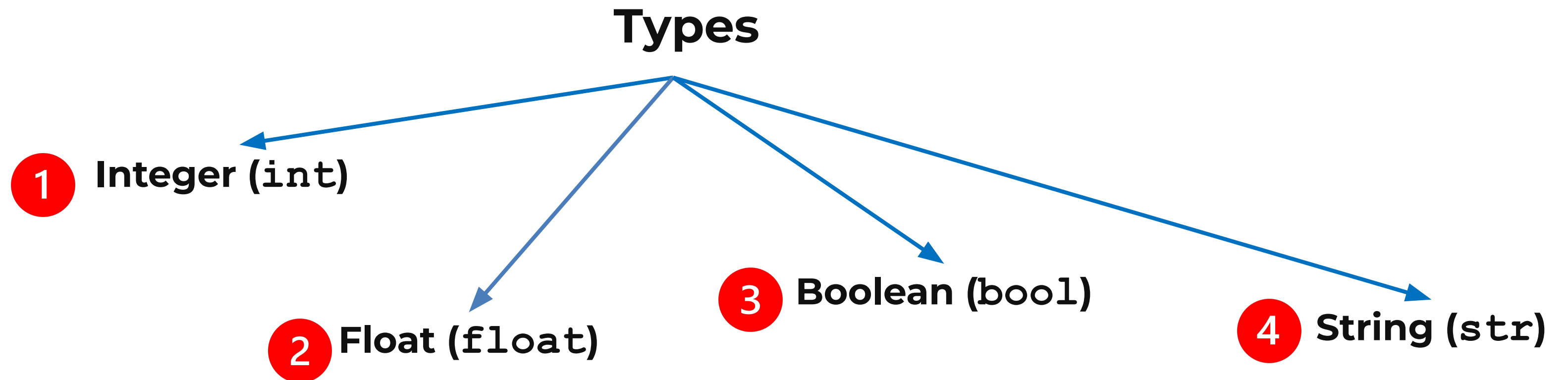
Exercise 2

What will happen if you use a reserved word like if as a variable name?

- a) It will run normally
- b) It will be ignored by Python
- c) It will cause an error
- d) It will change the value of if

Definition of Data Type:-

The kind of values a variable can hold.



1 Integer (`int`)

- Represents whole numbers (positive, negative, or zero) without decimals.
- Maximum size is only limited by the memory of the system.

<i>Syntax</i>	<i>Example</i>
<pre>variable_name = integer_value</pre>	<pre>age = 25 year = 2024 count = -10 print(age, year, count)</pre>

2

Float (float)

- Represents real numbers (numbers with a decimal point).
- Python does not have a double data type like Java or C++, as float in Python already provides double-precision floating points.

Syntax

```
variable_name = float_value
```

Example

```
pi = 3.14159  
price = 19.99  
temperature = -5.5  
print(pi, price, temperature)
```

3 Boolean (bool)

- Represents truth values: `True` or `False` (case-sensitive).
- Commonly used in conditions and comparisons.
- Booleans are essentially integers (`True` = 1, `False` = 0) and can be used in arithmetic operations.

<i>Syntax</i>	<i>Example</i>
<pre>variable_name = True variable_name = False</pre>	<pre>is_raining = True is_sunny = False print(is_raining) print(is_sunny)</pre>

4 String (str)

- Represents a sequence of characters (text data).
- Strings can be enclosed in single ('), double (") or triple (''' ''') quotes
- Strings are immutable (cannot be changed after creation).

Syntax

```
# Double quotes
variable_name = "string_value"
# Single quotes
variable_name = 'string_value'
# Triple quotes
variable_name = """multi-line string"""
```

Example

```
name = "Alice"
message = 'Hello, World!'
paragraph = """This is a
multi-line string."""
print(name)
print(message)
print(paragraph)
```

4 String (`str`)

3 important function or method relate with string

1

`len()` function

2

`upper()` method

3

`lower()` method

4 String (str)

1

`len()` function - Function to Get String Length

- `len()` is a **built-in function** in Python, not a method.
- It returns the number of characters in a string (including spaces and punctuation).

<i>Syntax</i>	<i>Example</i>
<pre>length = len(string_variable)</pre>	<pre>text = "Hello World" print(len(text)) # Output: 11 <i>*(Includes the space between "Hello" and "World")</i></pre>

4 String (str)

2

upper () method - Method to Convert String to Uppercase

- `upper ()` is a **string method** (not a function).
- It converts all lowercase letters in a string to uppercase.
- It does **not** change the original string (strings are immutable in Python).

Syntax

```
uppercase_text = string_variable.upper()
```

Example

```
message = "hello"  
print(message.upper())  
# Output: HELLO
```

4 String (str)

3

lower () method - Method to Convert String to Lowercase

- `lower ()` is also a **string method**.
- It converts all uppercase letters to lowercase.
- Like `upper ()`, it does **not** modify the original string

Syntax

```
lowercase_text = string_variable.lower()
```

Example

```
word = "PYTHON"  
print(word.lower())  
# Output: python
```

4 String (str)

Difference Between `len()` and `upper()` / `lower()`

Feature	<code>len()</code> (Function)	<code>upper()</code> & <code>lower()</code> (Methods)
Type	Built-in / Pre-defined Function	String Methods
Purpose	Counts characters	Modifies case of letters
Applies to	Strings, lists, etc.	Only strings
Return Type	Integer	String

Type Conversion (Casting) Syntax

- To convert between data types, use the following functions:

<code>int_value = int(3.5)</code>	<i># Converts float to int (3)</i>
<code>float_value = float(10)</code>	<i># Converts int to float (10.0)</i>
<code>str_value = str(100)</code>	<i># Converts int to string ("100")</i>
<code>bool_value = bool(1)</code>	<i># Converts 1 to True</i>

Type Conversion (Casting) Syntax

Example

```
num_str = "50"  
num_int = int(num_str)    # Convert string to int  
print(num_int)            # Output: 50
```


exceptional

`type ()` function

- In Python, the built-in `type()` function is used to determine the **type of an object** or to **create a new type (class)** dynamically.

Syntax

```
type (object)
```

- **object** → The variable or value whose type you want to check.
-  Returns the **type/class** of the object.

type () function

- value is **assigned to a variable** first, then checked with type():

```
num = 42  
print(type(num))
```

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

Integer

```
pi = 3.14  
print(type(pi))
```

```
<class 'float'>
```

Float

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

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

String

```
is_ready = True  
print(type(is_ready))
```

```
<class 'bool'>
```

Boolean

Example

- determine the data type of an object

```
print(type(42))  
print(type(3.14))  
print(type("Hello"))  
print(type(True))
```

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



Review Activity : Data Type

Try this !



Exercise 1

Which of the following is a float value?

- A) "3.5"
- B) 3.5
- C) 3
- D) '3'



Review Activity : Data Type

Try this !



Exercise 2

What will be the output of this code?

```
x = 10
```

```
print(type(x))
```

A) `<class 'float'>`

B) `<class 'int'>`

C) `<class 'bool'>`

D) `<class 'str'>`



Review Activity : Data Type

Try this !  Exercise 3

What will be the output of this code?

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

- A) 2
- B) 3
- C) 4
- D) 5



Review Activity : Data Type

Try this !  Exercise 4

```
name = "Aina"  
age = 17  
height = 160.5  
is_student = True  
  
print("Name:", name.upper())  
print("Length of name:", len(name))  
print("Age:", age)  
print("Height:", height)  
print("Is a student:", is_student)
```

- a) State the data type of the following variables:
- i. name
 - ii. age
 - iii. height
 - iv. is_student



Review Activity : Data Type

Try this !  Exercise 4

```
name = "Aina"  
age = 17  
height = 160.5  
is_student = True  
  
print("Name:", name.upper())  
print("Length of name:", len(name))  
print("Age:", age)  
print("Height:", height)  
print("Is a student:", is_student)
```

b) What is the output of `name.upper()` ?



Review Activity : Data Type

Try this !  Exercise 4

```
name = "Aina"  
age = 17  
height = 160.5  
is_student = True  
  
print("Name:", name.upper())  
print("Length of name:", len(name))  
print("Age:", age)  
print("Height:", height)  
print("Is a student:", is_student)
```

c) What is the output of `len(name)` ?

Definition of Comment:-

A line of text within the code that explain what the code is doing and is ignored by the interpreter during execution.

When writing comments:

- The `#` character should be followed by a single space. Ex: `# End of menu` is easier to read than `#End of menu`.
- Comments should explain the purpose of the code, not just repeat the code itself. Ex: `# Get the user's preferences` is more descriptive than `# Input item1 and item2`

Purposes of Comment:-

1. **Code Documentation** – Helps explain the purpose of code for other developers.
2. **Debugging Assistance** – Comments can be used to temporarily disable lines of code for testing.
3. **Increasing Readability** – Well-commented code is easier to understand and maintain.
4. **Collaboration** – Makes it easier for multiple developers to work on the same project.
5. **Docstrings for APIs** – Helps generate documentation for libraries and frameworks.

LECTURE NOTES

Comments can be written

- 1 before a line of code

This is a comment

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

- 2 can be placed at the end of a line, and Python will ignore the rest of the line

```
print("Hello, World!") # This is a comment
```


Python has two main types of comments:

1 Single-line Comments

- Uses the **#** symbol.
- Everything after # on that line is ignored by the interpreter.
- Used for short explanations or notes.

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

```
x = 10 # Assigning value to x
```

```
print(x) # Printing x
```

Python has two main types of comments:

2 Multi-line (Block) Comments

- Uses triple quotes (' ' ' or " " ") as multi-line strings, which is often called **docstrings**.

```
"""
```

```
This is a multi-line comment.
```

```
It is written using triple quotes.
```

```
This program is to print
```

```
Hello, World!
```

```
"""
```

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



Review Activity : Comment

Try this !  Exercise 1

The main purpose of writing comments is to _____.

- a. avoid writing syntax errors
- b. explain what the code does
- c. make the code run faster



Review Activity : Comment

Try this !  Exercise 2

Which symbol is used for comments in Python?

Table of Contents

- ☒ Identifiers
- ☒ Variables
- ☒ Reserved words/keywords
- ☒ Data types
- ☒ Comments
- ☐ Import statements
- ☐ Input statements
- ☐ Output statements
- ☐ Indentation

Topic 4: Python Programming

Learning Outcomes:

- c) Identify the components of a Python program (identifiers, variables, reserved words/keywords, data types, comments, import statements, input statements, output statements, indentation)

Table of Contents

- ☒ Identifiers
- ☒ Variables
- ☒ Reserved words/keywords
- ☒ Data types
- ☒ Comments
- ☐ Import statements
- ☐ Input statements
- ☐ Output statements
- ☐ Indentation

Purpose of Import Statement:-

to bring in modules and their functionalities into the program.

- This allows you to use pre-defined functions and constants without having to define them yourself.

In the case of mathematical operations, Python provides the math module, which includes important constants and functions such as:

- 1 `math.pi` → *The mathematical constant π (pi).*
- 2 `math.pow(x, y)` → *Raises x to the power of y .*
- 3 `math.sqrt(x)` → *Computes the square root of x .*

Importing the math Module

Syntax

```
import math
```

1 `math.pi`

- `math.pi` is a mathematical constant representing the value of π (pi), approximately 3.141592653589793.

Import the Entire math Module

Syntax

```
import math

print(math.pi) # Using pi
```

Example

```
import math # Import the entire math module

radius = 7

# Using math.pi to calculate circumference
circumference = 2 * math.pi * radius

print("Circumference of the circle:",
      circumference)
```

2 `math.pow(x, y)`

- It always returns a **floating-point** number.

Import the Entire math

Syntax	Module	Example
<pre>import math print(math.pow(x, y)) # Using pow()</pre>		<pre>import math # Import the entire math module base = 3 exponent = 4 # Using math.pow() to calculate 3^4 result = math.pow(base, exponent) # Using string concatenation (+) print(base, "raised to the power of", exponent, "is:", result)</pre>

Answer: 3 raised to the power of 4 is:

81.0

3 `math.sqrt(x)`

- The argument `x` must be **non-negative** (≥ 0).
- The result is always a **floating-point** number.

Import the Entire math

Syntax	Module	Example
<pre>import math print(math.sqrt(x)) # Using sqrt()</pre>		<pre>import math # Import the entire math module number = 64 # Using math.sqrt() to find the square root of 64 square_root = math.sqrt(number) # Displaying the result print("The square root of", number, "is:", square_root)</pre>

Answer : The square root of 64 is: 8.0

`input()` function

allows your Python program to **interact with the user** by accepting **keyboard input to display text, numbers, or other data types on the screen.**

Syntax

```
variable = input("prompt message")
```

Syntax

```
variable = input("prompt message")
```

- input() is a **built-in function**.
- The "Prompt message" is **optional**, but commonly used to guide the user.
- The user input is always returned as a **string** (data type: str).

```
Enter your name:
```

- The value typed by the user is returned as a **string**:

Example

```
name = "Aina"
```

- if the user types a number like 25, the result is:

Example

```
age = "25"    # still a string
```

check input data type

Convert the Input String to the Correct Data Type

- **Convert to Integer (casting input value from str to int)**

Example

```
age = int(input("Enter your age: "))    # user types 25 → age = 25 (int)
```

- **Convert to Float**

Example

```
price = float(input("Enter the price: "))    # user types 19.99 → price =  
19.99 (float)
```

Common Mistakes

Mistake	Explanation
Forgetting to convert input	All input is a string, so "5" + "3" becomes "53" instead of 8.
Not handling invalid input	If user types text instead of a number, <code>int(input())</code> will crash with <code>ValueError</code> .
No prompt message	User doesn't know what to type. Always give a clear prompt.

Summary

Feature	Description
Function name	<code>input()</code>
Returns	Always returns <code>str</code>
Used for	Accepting keyboard input from user
Prompt	Optional message inside the function
Conversion	Use <code>int()</code> , <code>float()</code> for numeric values
Blocking	Program waits until user presses Enter

Function of Output Statement:-

to display data or messages to the user.

- The most commonly used output function in Python is the `print()` function.

`print()` function

to display text, numbers, or other data types on the screen.

Syntax

```
print(value1, value2, ..., sep=' ', end='\n')
```

`print ()` function

Components of the `print ()` function

1. **value1, value2, ...** → The values to be printed. These can be strings, numbers, variables, or expressions.
2. **sep=' '** (optional) → Specifies the separator between multiple values (default is a space " ").
3. **end='\n'** (optional) → Specifies what to print at the end of the statement (default is a newline '\n').

`print()` function

Example 1 : Printing a Simple String

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

Output

```
Hello, World!
```

`print()` function

Example 2 : Printing Multiple Value

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

Output

```
Hello Python World
```


`print()` function

Example 3 : Using `sep` Parameter (separator)

```
print("Hello", "Python", "World", sep="-")
```

Output

```
Hello-Python-World
```

`print()` function

Example 5 : Printing Variables

```
name = "Alice"  
age = 25  
print("Name: ", name, " Age: ", age)
```

Output

```
Name: Alice Age: 25
```

`print()` function

Example 6: Printing with `f`-strings (formatted strings)

```
name = "Alice"  
age = 25  
print(f"My name is {name} and I am {age} years old.")
```

Output

```
My name is Alice and I am 25 years old.
```

Output statements

<code>print("Hello World")</code>	Hello World
<code>print("Hello", "how are you?", sep="---")</code>	Hello---how are you?
<code>print(25+25)</code>	50
<code>x = ("apple", "banana", "cherry")</code> <code>print(x)</code>	('apple', 'banana', 'cherry')

Definition of Indentation:-

use of whitespace (spaces or tabs) at the beginning of a line to define the structure of the code.

- Unlike other programming languages (such as Java or C++) that use curly braces {} to define blocks of code, Python relies on indentation to determine the grouping of statements.

Importance of Indentation

1 Defines Code Blocks

- In Python, indentation is used to indicate blocks of code, such as loops, functions, and conditionals.

Example

```
if True:
    print("This is inside the if statement")
print("This is outside the if statement")
```

Importance of Indentation

2 Ensures Readability

- Proper indentation makes code more readable and easier to understand.

3 Maintains Consistency

- Python enforces a consistent indentation style, which helps developers write cleaner and more structured code.

Importance of Indentation

4 Avoids Syntax Errors

- Since indentation is mandatory in Python, incorrect indentation leads to IndentationError.

Example of incorrect indentation:

```
if True:
print("Hello")    # This will cause an IndentationError
```


Examples of Improper Indentation

1 *Example of Indentation Error in an if Statement*

```
age = 18
if age >= 18:
print("You are allowed to vote")    # No indentation, causes IndentationError
```



No Indentation

Examples of Improper Indentation

2 *Example of Indentation Error in a `for..` Loop*

```
for i in range(5):  
print(i)
```

No indentation, causes IndentationError

No Indentation



Examples of Improper Indentation

3 *Example of Indentation Error in a Function*

```
def greet():  
print("Hello, World!")  
  
greet()
```

No indentation, causes IndentationError

No Indentation



Examples of Improper Indentation

4 *Example of Mixing Spaces and Tabs (Bad Practice)*

```
def add(a, b):  
    result = a + b    # Indented using spaces  
    result += 1        # Indented using a tab (causes error)  
    return result
```

Python Indentation Rules

1. Use **4 spaces per indentation level** (recommended by PEP 8).
2. Avoid mixing spaces and tabs to prevent errors.
3. Ensure all statements in a block have the same indentation.

Table of Contents

- ✓ Identifiers
- ✓ Variables
- ✓ Reserved words/keywords
- ✓ Data types
- ✓ Comments
- ✓ Import statements
- ✓ Input statements
- ✓ Output statements
- ✓ Indentation

Topic 4: Python Programming

Learning Outcomes:

(e) Identify the use of assignment and arithmetic operators. (1st hour)

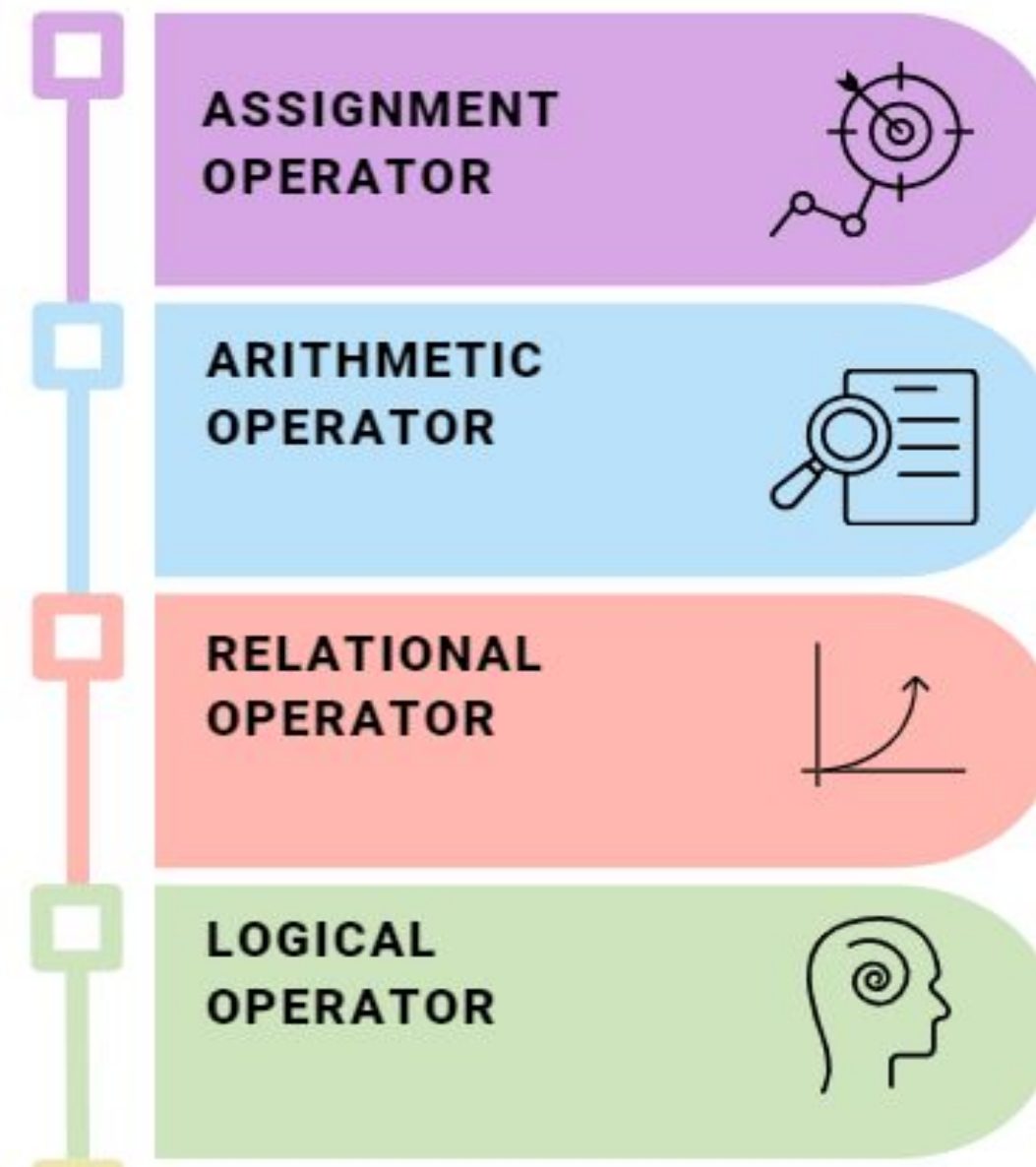
What is an **Operator**?

A symbol of the programming language, which is able to operate on the values.

What is an **Expression**?

The combination of variables, literals, operators, and parentheses.

- Here's a list of different types of Python operators.



Assignment Operators

- The assignment operation is one of the most important operations.
- Assignment operators in Python are used to store or assign a value to a variable for later use.
- They allow programmers to perform different types of operations and store the results in variables.
- A statement can set a variable to a value using the **assignment operator** (=).

*** Note that this is different from the equal sign of mathematics.*

Examples:

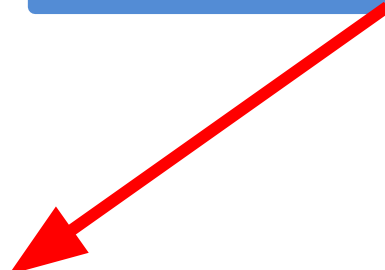
```
age = 6
```

```
birth = "May 15"
```

- The left side of the assignment statement is a **variable**, and
- the right side is the **value the variable is assigned**.

- The simplest form of an assignment statement is the following:

```
variable_name = value
```



Any Python variable name,
such as `totalIncome` or
`taxRate`.



Any Python expression, such as
`" " * 10 + "Python"`

Example

Here, `=` is an assignment operators that assigns 5 to `x`.

```
# assign 5 to x  
var x = 5
```

`x` is the variable name

5 is the value being stored in `x`

IMPORTANT NOTES !

- One equal sign (=) is used for assignment
- Do not confuse it with == which is used for comparison

```
x = 5    # This assigns 5 to x
```

```
x == 5   # This checks if x is equal to 5 (gives True or False)
```

IMPORTANT NOTES !

- You can assign different types of data

```
x = 10                # Integer
pi = 3.14             # Float / Double
name = "Mira"         # String
is_student = True     # Boolean
```


IMPORTANT NOTES !

- Variables can be reassigned

```
x = 5  
  
x = 8    # x now holds 8, not 5 anymore  
  
print(x) # Output: 8
```

How??

- A variable is created the first time it is assigned a value.
- Assigning a value to an existing variable replaces the previously stored value.

IMPORTANT NOTES !

- Multiples assignments are also allowed :
 - a. Assign the same value to multiple variables

```
a = b = c = 0  
print(a, b, c)
```

Output :

0	0	0
---	---	---

- b. Assign multiple values to multiples variables

```
x, y, z = 1, 2, 3  
print(x, y, z)
```

Output :

1	2	3
---	---	---

- An **arithmetic operators** are used to perform mathematical operations.

Example:

`+, -, /, *, %, **, //`

- An **arithmetic expression** consists of operands and operators combined in a manner

Example:

```
result = 3 + 4 * 2
```

Arithmetic Operators

Operator	Meaning	Syntax
<code>**</code>	Exponentiation	<code>a ** b</code>
<code>*</code>	Multiplication	<code>a * b</code>
<code>/</code>	Division	<code>a / b</code>
<code>//</code>	Quotient or Floor division	<code>a // b</code>
<code>%</code>	Remainder or modulus	<code>a % b</code>
<code>+</code>	Addition	<code>a + b</code>
<code>-</code>	Subtraction	<code>a - b</code>

Precedence Rules

Operation	Precedence	Associativity	Description
Exponentiation (**)	Highest	Right to Left	Evaluated first
Unary Negation (-value)	High	Right to Left	Evaluated before multiplication and division
Multiplication, Division, Remainder	Medium	Left to Right	
Addition, Subtraction	Low	Left to Right	Evaluated before addition and subtraction
Assignment (=, +=, etc.)	Lowest	Right to Left	
Parentheses ()	Overrides All	—	Evaluated after multiplication/division

Precedence Rules

Operation	Example / Notes
Exponentiation (**)	$2 ** 3 ** 2 \rightarrow 2 ** (3 ** 2)$
Unary Negation (-value)	$-5 * 2 \rightarrow (-5) * 2$
Multiplication, Division, Remainder	$10 * 2 / 5 \% 3$
Addition, Subtraction	$4 + 3 - 1$
Assignment (=, +=, etc.)	$x = y = 10$ assigns 10 to both y and x
Parentheses ()	$(3 + 2) * 4$ forces addition first

- Operations with **equal precedence** are usually **left to right**.
- **Exceptions:** ****** and **=** are **right to left**.
- Use **parentheses** to make evaluation order explicit.

Examples of arithmetic expressions and their values

Expression	Evaluation	Value
<code>5 + 3 * 2</code>	<code>5 + 6</code>	<code>11</code>
<code>(5 + 3) * 2</code>	<code>8 * 2</code>	<code>16</code>
<code>6 % 2</code>	<code>0</code>	<code>0</code>
<code>2 * 3 ** 2</code>	<code>2 * 9</code>	<code>18</code>
<code>-3 ** 2</code>	<code>-(3 ** 2)</code>	<code>-9</code>
<code>(3) ** 2</code>	<code>9</code>	<code>9</code>
<code>2 ** 3 ** 2</code>	<code>2 ** 9</code>	<code>512</code>
<code>(2 ** 3) ** 2</code>	<code>8 ** 2</code>	<code>64</code>
<code>45 / 0</code>	Error: cannot divide by 0	
<code>45 % 0</code>	Error: cannot divide by 0	

- Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.

Example:

If an expression contains both int and float, **the result will always be a float.**

```
a = 5          # int
b = 2.0        # float
```

```
print(a + b)    # 7.0 (int + float → float)
print(a * b)    # 10.0
print(a / b)    # 2.5
print(a // b)   # 2.0 (floor division but result is float)
```


- Strings do not support any arithmetic operations.
- '+' stands for the concatenation of the string. It is not the arithmetic addition.
- For example :

```
greet = "Hello, "  
name = "Jack"  
  
result = greet + name  
print(result)
```

Output :

```
Hello, Jack
```

Example 1

```
val1 = 2
val2 = 3

# using the addition operator
total = val1 + val2
print(total)
print("Total = ", total)
```

Output :

```
5
Total = 5
```

Example 2

```
a = 21.0
b = 10
# Addition
print ("a + b : ", a + b)
# Subtraction
print ("a - b : ", a - b)
# Multiplication
print ("a * b : ", a * b)
# Division
print ("a / b : ", a / b)
# Modulus
print ("a % b : ", a % b)
# Exponent
print ("a ** b : ", a ** b)
# Floor Division
print ("a // b : ", a // b)
```

Output :

```
a + b : 31.0
a - b : 11.0
a * b : 210.0
a / b : 2.1
a % b : 1.0
a ** b : 16679880978201.0
a // b : 2.0
```

Example 3: Multiplication

```
length = 7
```

```
width = 3
```

```
area = length * width
```

```
print(area)    # Output: 21
```


You can also multiply a string by a number (useful in formatting) :

```
print("Hi! " * 3)    # Output: Hi! Hi! Hi!
```

Example 4

```
a=6
b=3

a /= 2 * b

print (a)
print (100 + a * b)
print ((a+b) - (a+b))
```

Output:

```
1.0
103.0
0.0
```



Python always calculates the RIGHT SIDE FIRST, then show or stores the result.

TRY THIS !

```
print (((5 + 4) / 3) * 2)
print((2 ** 4), (2 * 4.), (2 * 4))
print((-2 / 4), (2 / 4), (2 // 4), (-2 // 4))
print((2 % -4), (2 % 4), (2 ** 3 ** 2))
```

Output :

```
6.0
16 8.0 8
-0.5 0.5 0 -1
-2 2 512
```

TRY THIS !

What is the expected output of the following snippet?

(a)

```
a = '1'
b = "1"
print(a + b)
```

(b)

```
print(9 % 6 % 2)

print((2 ** 4), (2 * 4.), (2 * 4))

print((-2 / 4), (2 / 4), (2 // 4), (-2 // 4))

print((2 % -4), (2 % 4), (2 ** 3 ** 2))
```


ANSWER TRY THIS !

What is the expected output of the following snippet?

(a)

```
a = '1'  
b = "1"  
print(a + b)
```

(b)

```
print(9 % 6 % 2)
```

```
print((2 ** 4), (2 * 4.), (2 * 4))
```

```
print((-2 / 4), (2 / 4), (2 // 4), (-2 // 4))
```

```
print((2 % -4), (2 % 4), (2 ** 3 ** 2))
```

Output :

```
11
```

```
1  
16 8.0 8  
-0.5 0.5 0 -1  
-2 2 512
```

LETS CALCULATE THE FINAL EXAM SCORE AVERAGE :

```
math = 90
```

```
english = 85
```

```
science = 80
```

```
average = (math + english + science) / 3
```

```
print("Average Score:", average) # Output: 85.0
```

Topic 4: Python Programming

Learning Outcomes:

(e) Identify the use of assignment and arithmetic operators. (2nd hour)

Combining Assignment and Arithmetic

- In Python, you can combine arithmetic and assignment.
- For example, the instruction

`total += cans` is a shortcut for `total = total + cans`

`total *= 2` □ `total = total * 2`

- Many programmers find this a convenient shortcut especially when incrementing or decrementing by 1:

`count += 1`

A list of different assignment operators available in Python.

Operator	Equivalent Expression	Description
<code>+=</code>	<code>x = x + y</code>	Addition assignment
<code>-=</code>	<code>x = x - y</code>	Subtraction assignment
<code>*=</code>	<code>x = x * y</code>	Multiplication assignment
<code>/=</code>	<code>x = x / y</code>	Division assignment (float)
<code>//=</code>	<code>x = x // y</code>	Floor division assignment
<code>%=</code>	<code>x = x % y</code>	Modulus assignment
<code>**=</code>	<code>x = x ** y</code>	Exponentiation assignment

Example 1 : Assign values and perform basic arithmetic

```
#add
#Assignment Operators
a = 5
b = 2

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

Output :

7

Example 2: Assign values and perform basic arithmetic

```
#subtraction
#Assignment Operators
a = 5
b = 2

# a = a - b
a -= b
print(a)
```

Output :

3

Example 3: Assign values and perform basic arithmetic

```
#multiplication
#Assignment Operators
a = 5
b = 2

# a = a * b
a *= b
print(a)
```

Output :

10

Example 4: Assign values and perform basic arithmetic

```
#division
#Assignment Operators
a = 5
b = 2

# a = a / b
a /= b
print(a)
```

Output :

2.5

Example 5 : Assign values and perform basic arithmetic

```
#exponent
#Assignment Operators
a = 5
b = 2

# a = a ** b
a **= b
print(a)
```

Output :

25

- ✓ Assignment always stores the LATEST value

Example 7 (with multiple operations) :

```
x = 10
x -= 2    # Now x = 8
x *= 3    # Now x = 24
x /= 4    # Now x = 6.0
print(x)
```

Output :

6

Try this :

```
# Add, Subtract, Multiply & Divide
# Assignment Operator

a = 3.0
b = 5

# a = a + b
a += b
print(a)

# a = a - b
a -= b
print(a)

# a = a * b
a *= b
print(a)

# a = a / b
a /= b
print(a)
```

Output :

```
8.0
3.0
15.0
3.0
```

Example 6:

```
# Assign String  
  
f = 'guru99'  
print (f)
```

Output :

guru99

Example 7 :

```
string1 = "hello"  
string2 = "world "  
string_combined = string1+string2  
print(string_combined)
```

Output :

helloworld

Example 8:

```
a = "guru"  
b = 99  
print (a+b)
```

Output :

```
Traceback (most recent call last):  
  File "main.py", line 3, in <module>  
    print (a+b)  
TypeError: can only concatenate str (not "int") to str
```

Correct code :

```
a="Guru"  
b = 99  
print(a+str(b))
```

Output :

```
Guru99
```

- `a = "guru"` → `a` is a **string**
- `b = 99` → `b` is an **integer**
- `str(99)` → converts the number 99 into a **string** "99"
- `a + str(99)` → joins (concatenates) the string "guru" with "99" → result is "guru99"

! In Python, you **CANNOT** directly add a string and a number. You must **convert the number to a string first** using `str()`.

a) What is the output of the following snippet?

```
name = 'Adelea'
age = 12
marks = 87.50

string1 = "Hello, I am " + name
print(string1)
string2 = "My age is "
print(string2 ,age)
string3= "My English mark is " + str(marks)
print(string3)
```

Output :

```
Hello, I am Adelea
My age is 12
My English mark is 87.5
```


SUMMARY

TYPE OF OPERATOR	OPERATORS
Assignment Operators	<code>= += -= /= *= %= **= //</code>
Arithmetic Operators	<code>+ - / * % ** //</code>