FUNDAMENTALS OF PROGRAMMING

Programming is the process of writing, testing, and maintaining instructions (code) that a computer can execute to perform specific tasks. It involves using programming languages such as Python, Java, C++, and JavaScript to create software, applications, and systems.

In simple terms, programming is like giving step-by-step instructions to a computer to make it perform a task, just like writing a recipe for a robot chef.

1. Basic Concepts

a) Algorithms

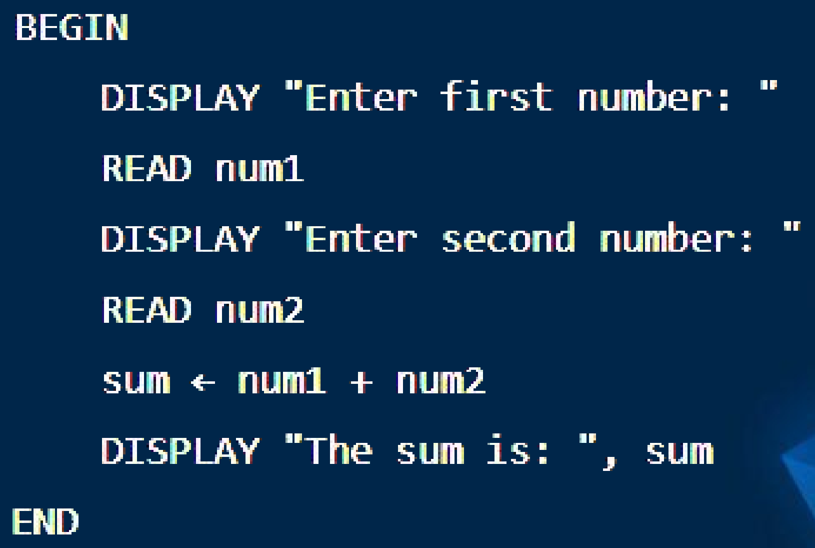
Algorithm: A step-by-step procedure for solving a problem. It can be represented as:

- **Pseudocode:** A human-readable description of the logic. It is not meant to be executed by a computer but serves as a blueprint for coding. It typically follows a structured format similar to actual programming but avoids strict syntax rules.

Example of Pseudocode

Find the sum of two numbers entered by the user.

- The program asks the user to input two numbers.
- It reads (stores) the numbers in num1 and num2.
- It calculates the sum and stores it in sum.
- Finally, it displays the sum.



```
BEGIN
    DISPLAY "Enter first number: "
    READ num1
    DISPLAY "Enter second number: "
    READ num2
    sum ← num1 + num2
    DISPLAY "The sum is: ", sum
END
```

- **Flowchart:** A graphical representation using symbols. It helps in understanding the logical flow of a program by showing the steps sequentially. Flowcharts are widely used in programming, system design, and problem-solving.

Common Flowchart Symbols

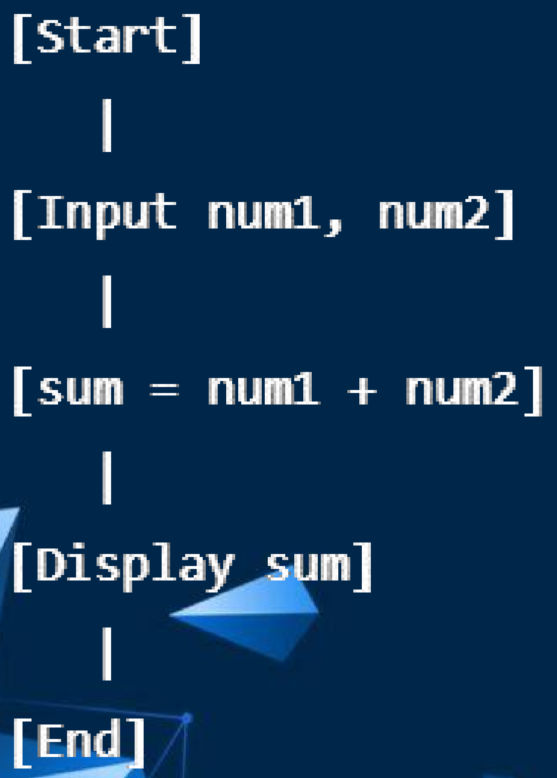
- ***Oval (Terminator):*** Represents the start or end of the flowchart.
- ***Parallelogram (Input/Output):*** Used for input (e.g., user entering data) and output (e.g., displaying results).
- ***Rectangle (Process):*** Represents a process, such as calculations or assignments.
- ***Diamond (Decision):*** Represents a decision-making step (e.g., "Is x > 10?").
- ***Arrow:*** Shows the flow of execution.

Sample Flowchart

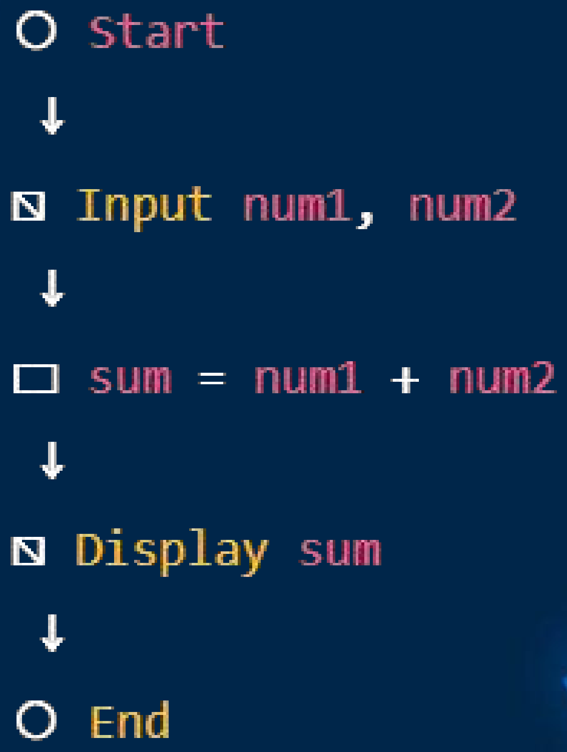
Problem: Create a flowchart to find the sum of two numbers entered by the user.

Graphical Representation:

This flowchart represents the logical sequence of reading two numbers, adding them, and displaying the result.



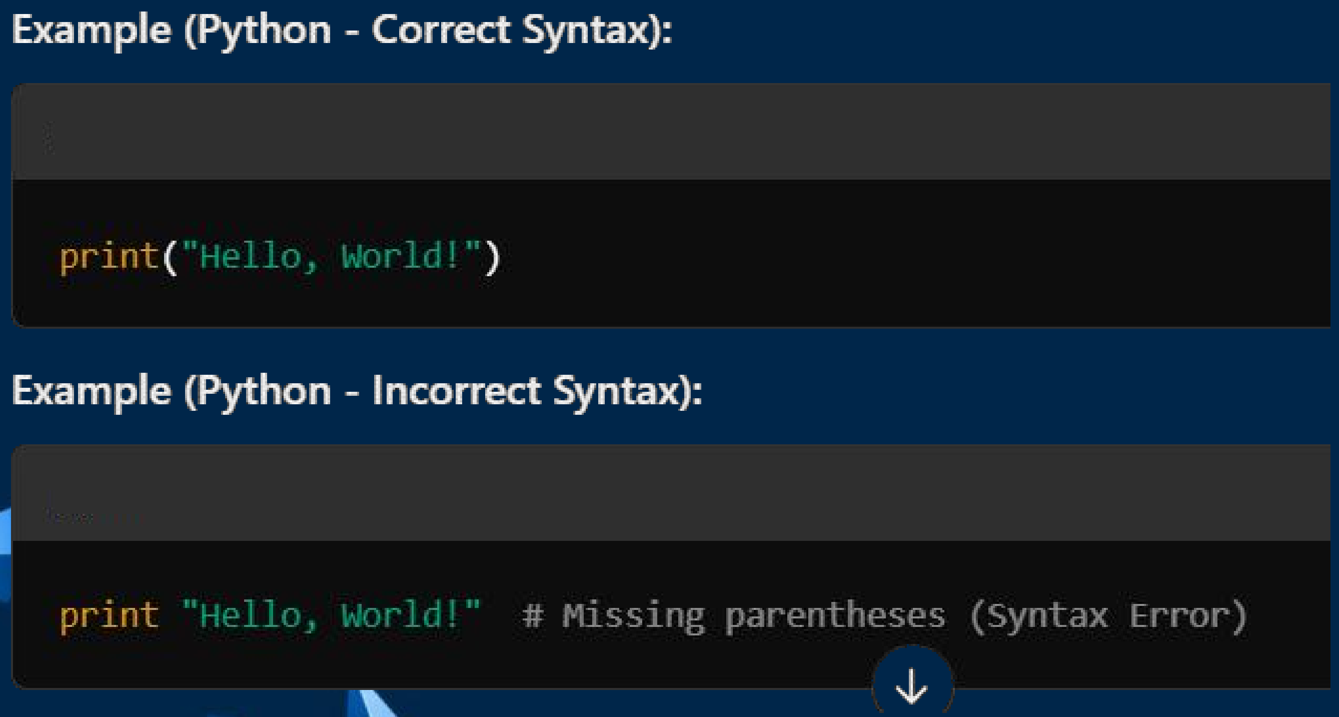
```
[Start]
   |
[Input num1, num2]
   |
[sum = num1 + num2]
   |
[Display sum]
   |
[End]
```



```
○ Start
↓
▧ Input num1, num2
↓
▭ sum = num1 + num2
↓
▧ Display sum
↓
○ End
```

b) Syntax and Semantics

Syntax refers to the set of rules that define how statements must be written in a programming language. It focuses on the structure and format of the code. If syntax rules are not followed, the program will produce errors



Example (Python - Correct Syntax):

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

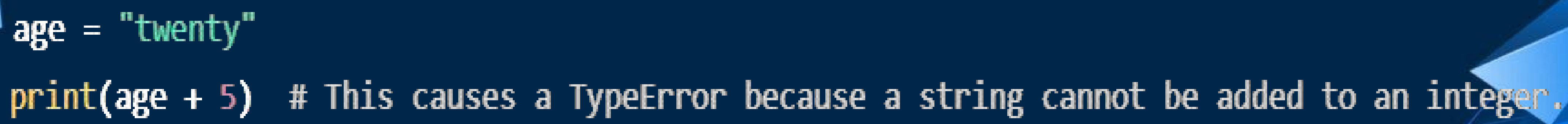
Example (Python - Incorrect Syntax):

```python
print "Hello, World!"  # Missing parentheses (Syntax Error)
```

Semantics refers to the meaning of the code.

Even if the syntax is correct, the code must make logical sense to produce the expected result. A program with semantic errors may run but produce unintended results

Example (Semantic Error in Python):



```
age = "twenty"
print(age + 5)  # This causes a TypeError because a string cannot be added to an integer.
```

- ✓ ***Syntax is about "how" you write code (grammar and structure).***
- ✓ ***Semantics is about "what" your code means and whether it does what you intend.***

c) Data Types and Variables

A **data type** defines the kind of data a variable can hold in a programming language.

Common data types include:

Integer (int): Whole numbers (e.g., 10, -5)

Float (float): Decimal numbers (e.g., 3.14, -0.99)

String (str): Text (e.g., "Hello", 'Python')

Boolean (bool): True/False values (e.g., True, False)

List, Tuple, Dictionary, Set: Collections of values

A **variable** is a named storage location in memory that holds a value.

Variables can change values during program execution.

2. Control Structures

Control structures are programming constructs that determine the flow of execution in a program. They help in decision-making, looping, and function execution.

Three main types of control structures:

Sequential Control: Code executes line by line in order.

Selection (Decision-Making): Code executes different blocks based on conditions (e.g., if-else). Uses conditional statements (if, if-else, if-elif-else) to choose between different execution paths.

Iteration (Looping): Code repeats a block multiple times (e.g., for, while loops).

Sequential Control

```
print("Start")
x = 5
y = 10
sum = x + y
print("Sum:", sum)  # Executes step by step
```

Iteration (Looping) - for Loop

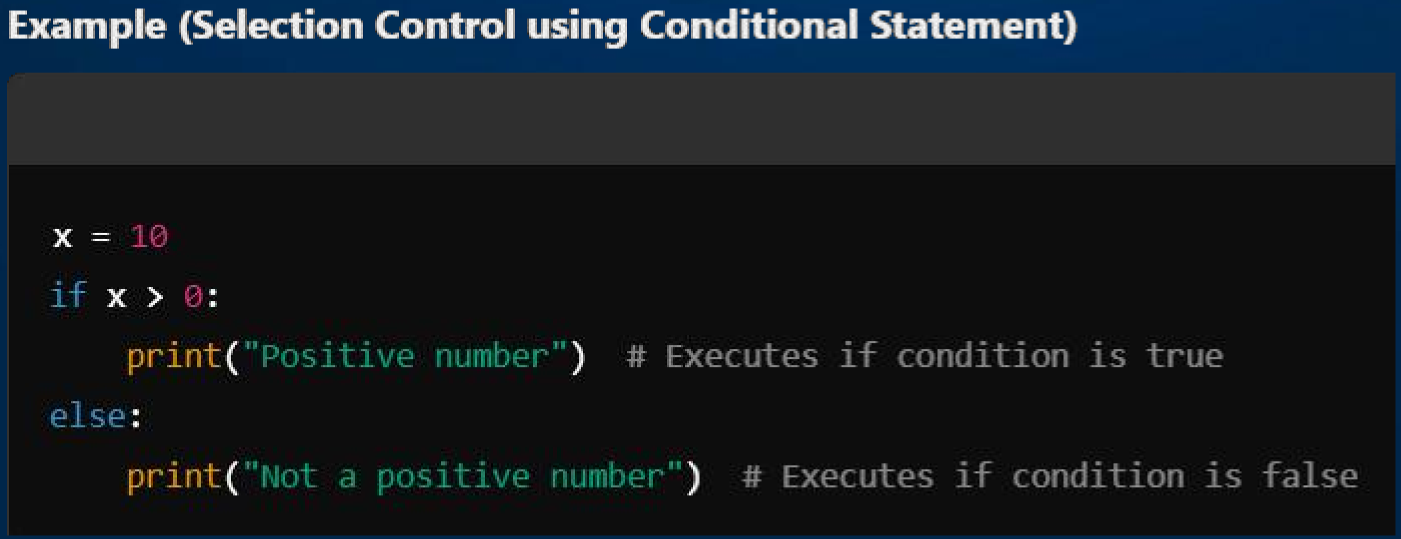
```
for i in range(1, 6):
    print("Count:", i)  # Prints numbers from 1 to 5
```

Selection (Decision-Making) - if-else

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Iteration (Looping) - while Loop

```
x = 1
while x <= 5:
    print("Number:", x)
    x += 1
```

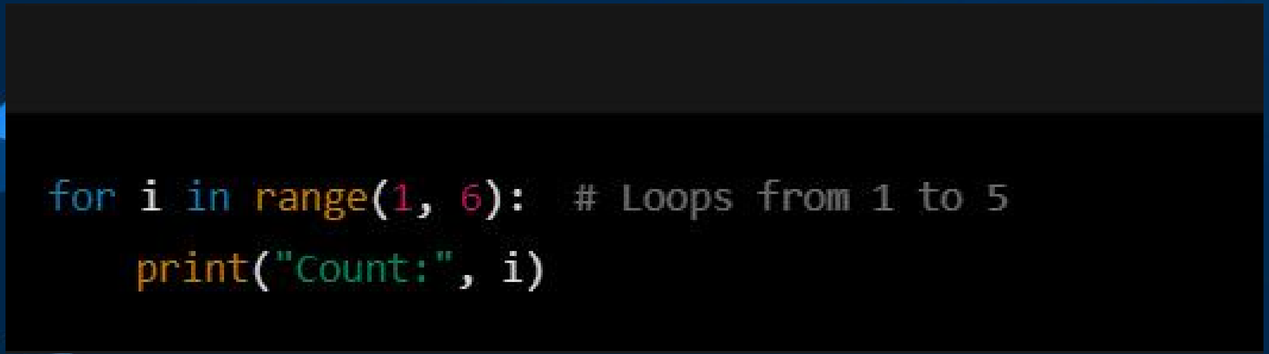


Example (Selection Control using Conditional Statement)

```
x = 10
if x > 0:
    print("Positive number")  # Executes if condition is true
else:
    print("Not a positive number")  # Executes if condition is false
```

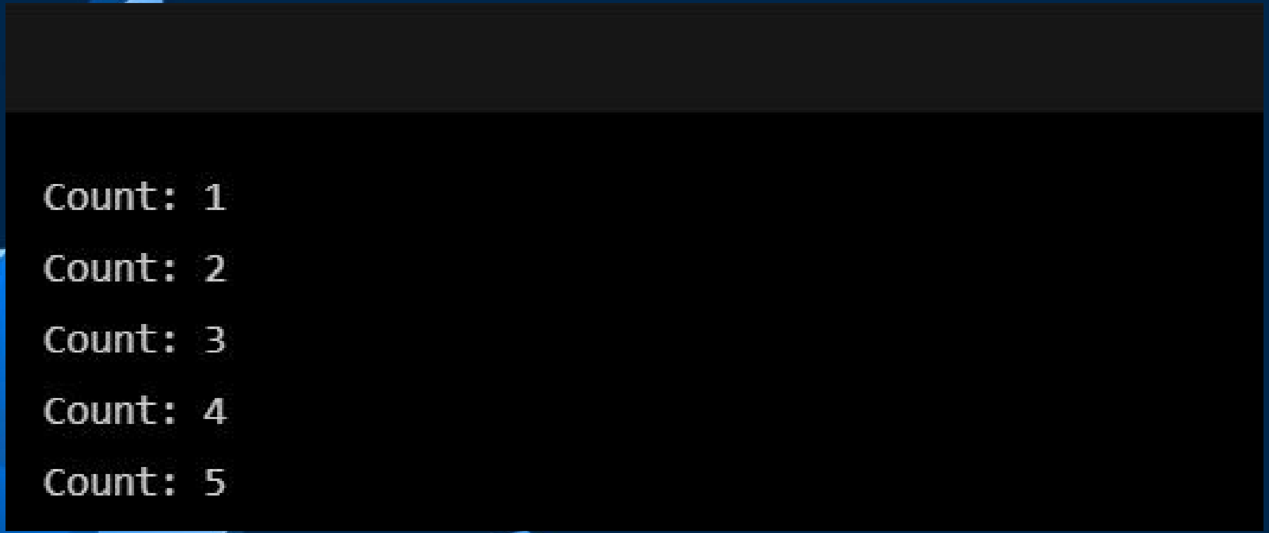
- ❑ Sequential: Executes line by line.
- ❑ Selection: Chooses a path based on conditions.
- ❑ Iteration: Repeats a block of code multiple times.

I. For Loop (Iterating through a range)



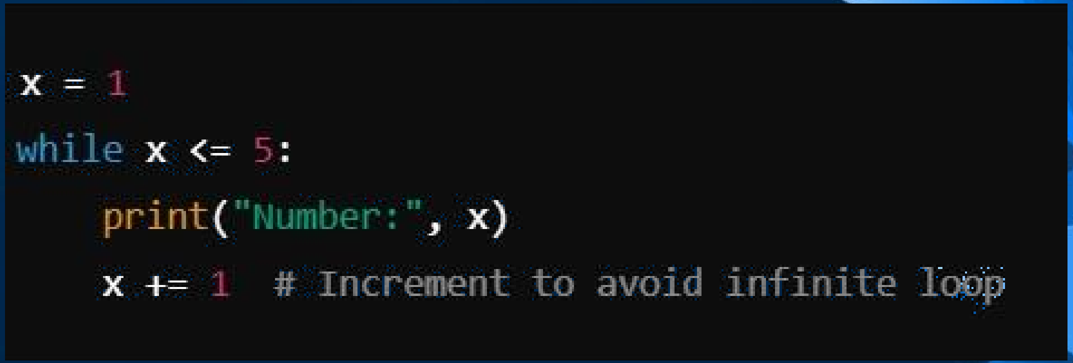
```python
for i in range(1, 6):  # Loops from 1 to 5
    print("Count:", i)
```

OUTPUT



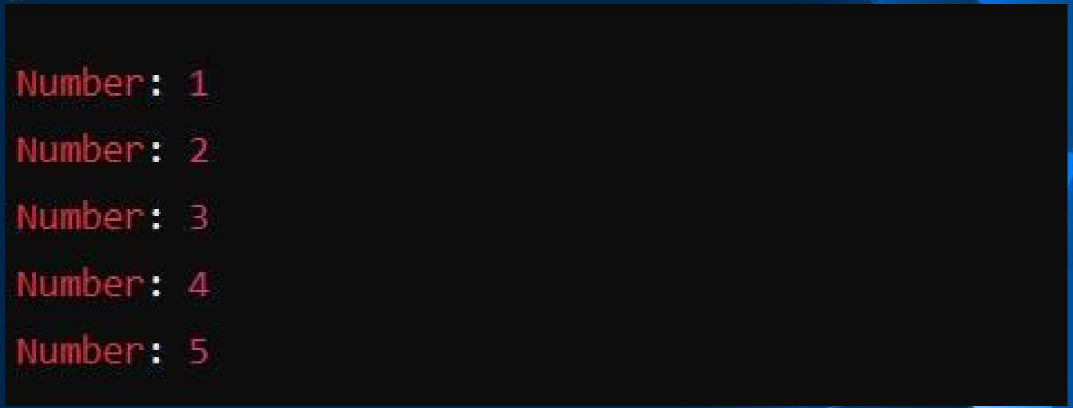
```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

While Loop (Executing until a condition is false)



```python
x = 1
while x <= 5:
    print("Number:", x)
    x += 1  # Increment to avoid infinite loop
```

OUTPUT



```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

3. Function

A **function** is a block of reusable code that performs a specific task. Functions help in organizing code, improving readability, and reducing repetition.

Key Features of Functions:

Reusability: Write once, use multiple times.

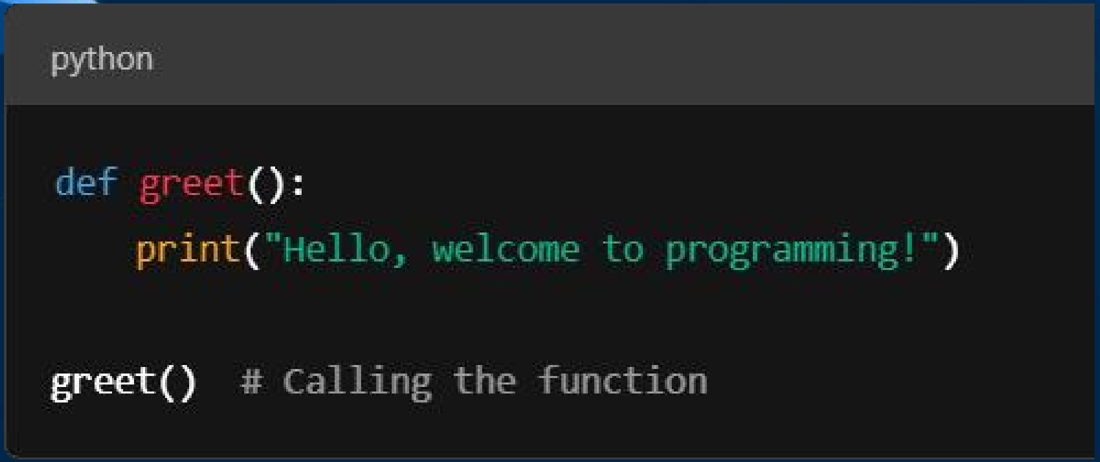
Modularity: Breaks a program into smaller, manageable parts.

Parameters: Accepts input values.

Return Value: Outputs a result.

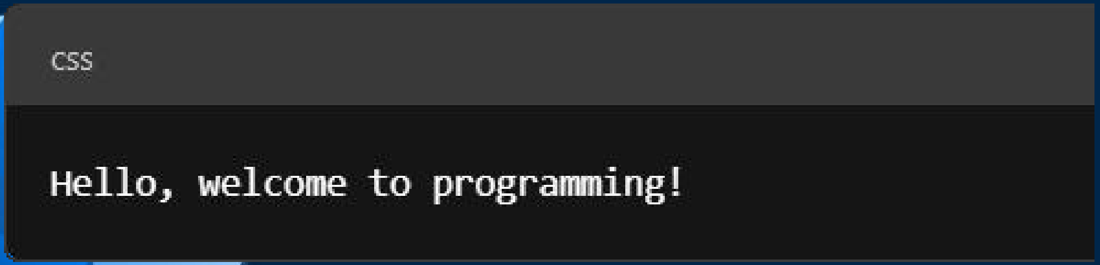
EXAMPLE OF FUNCTION

1. Function Without Parameters



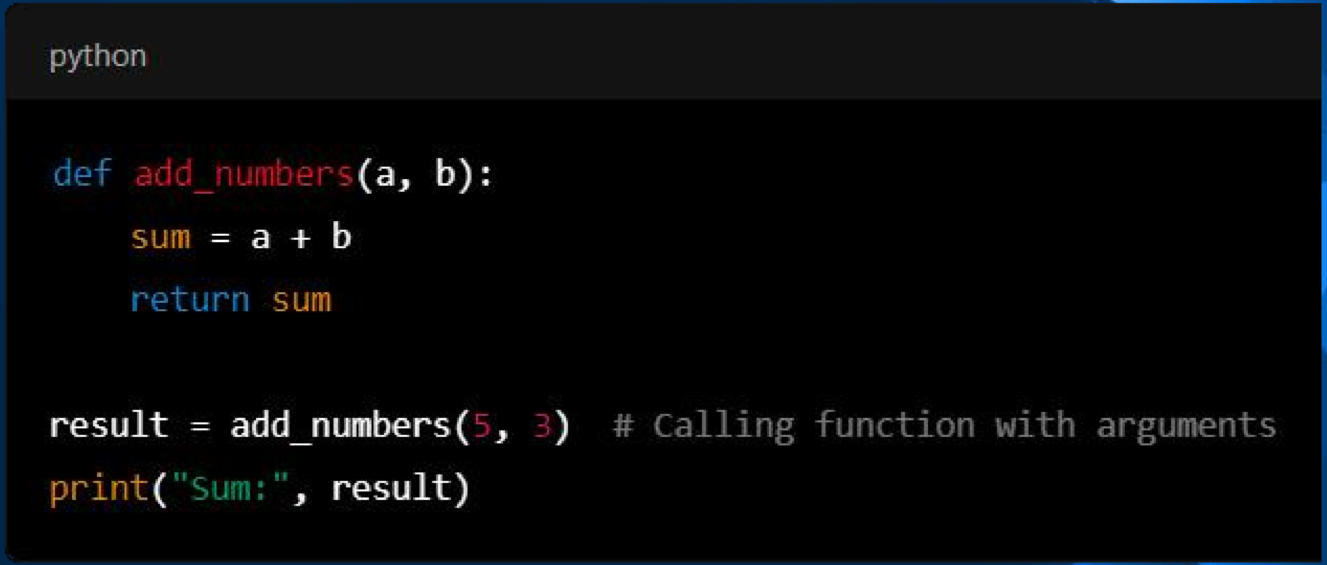
```python
def greet():
    print("Hello, welcome to programming!")

greet()  # Calling the function
```

Output:



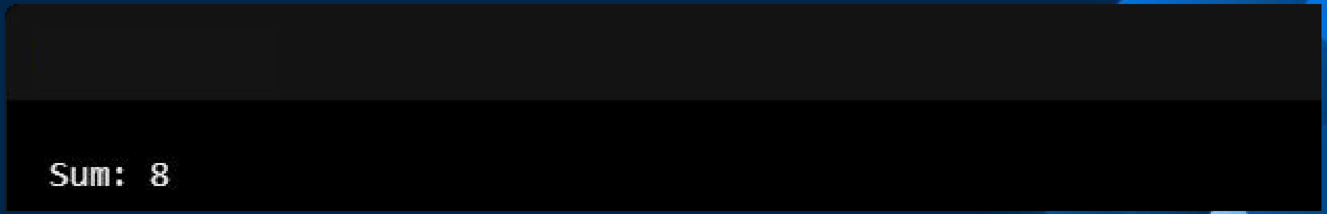
```
Hello, welcome to programming!
```

2. Function With Parameters



```python
def add_numbers(a, b):
    sum = a + b
    return sum

result = add_numbers(5, 3)  # Calling function with arguments
print("Sum:", result)
```

Output:



```
Sum: 8
```

- ❑Functions improve code organization and reusability.
- ❑Parameters allow functions to accept input values.
- ❑Return statements provide output from functions.
- ❑Functions make programming efficient and modular.

4. Object-Oriented Programming (OOP)

A programming paradigm based on objects and classes.

Key concept:

1. Class: A blueprint for creating objects.
2. Object: An instance of a class.
3. Encapsulation: Hiding data within an object.
4. Inheritance: Deriving a new class from an existing one.
5. Polymorphism: Methods behaving differently for different objects.

Example of OOP in Python

1. Defining a Class and Creating an Object

python

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Attribute
        self.model = model  # Attribute

    def display_info(self):  # Method
        print(f"Car: {self.brand} {self.model}")

# Creating an object (instance) of the Car class
my_car = Car("Toyota", "Corolla")
my_car.display_info()  # Calling the method
```

Output:

makefile

```
Car: Toyota Corolla
```

2. Inheritance Example

python

```python
class Animal:
    def make_sound(self):
        print("Some generic animal sound")

class Dog(Animal):  # Dog class inherits from Animal class
    def make_sound(self):
        print("Woof! Woof!")

dog = Dog()
dog.make_sound()  # Calls the overridden method
```

Output:

```
Woof! Woof!
```

5. Data Structure

A **data structure** is a way of organizing, storing, and managing data efficiently so that it can be accessed and modified easily. Data structures are fundamental in programming and help optimize algorithms.

Types of Data Structure

Linear Data Structures:

Array: Fixed-size collection of elements of the same type.

List (Dynamic Array): Ordered collection of elements that can grow dynamically.

Stack: Follows LIFO (Last In, First Out) principle.

Queue: Follows FIFO (First In, First Out) principle.

Non-Linear Data Structures:

Tree: Hierarchical structure (e.g., Binary Tree).

Graph: Nodes (vertices) connected by edges (e.g., Social Network Graph).

Examples of Data Structures in Python

1. List (Dynamic Array) - Linear Data Structure

python

```python
fruits = ["Apple", "Banana", "Cherry"]
fruits.append("Orange")  # Adding an element
print(fruits)
```

Output:

css

```css
['Apple', 'Banana', 'Cherry', 'Orange']
```

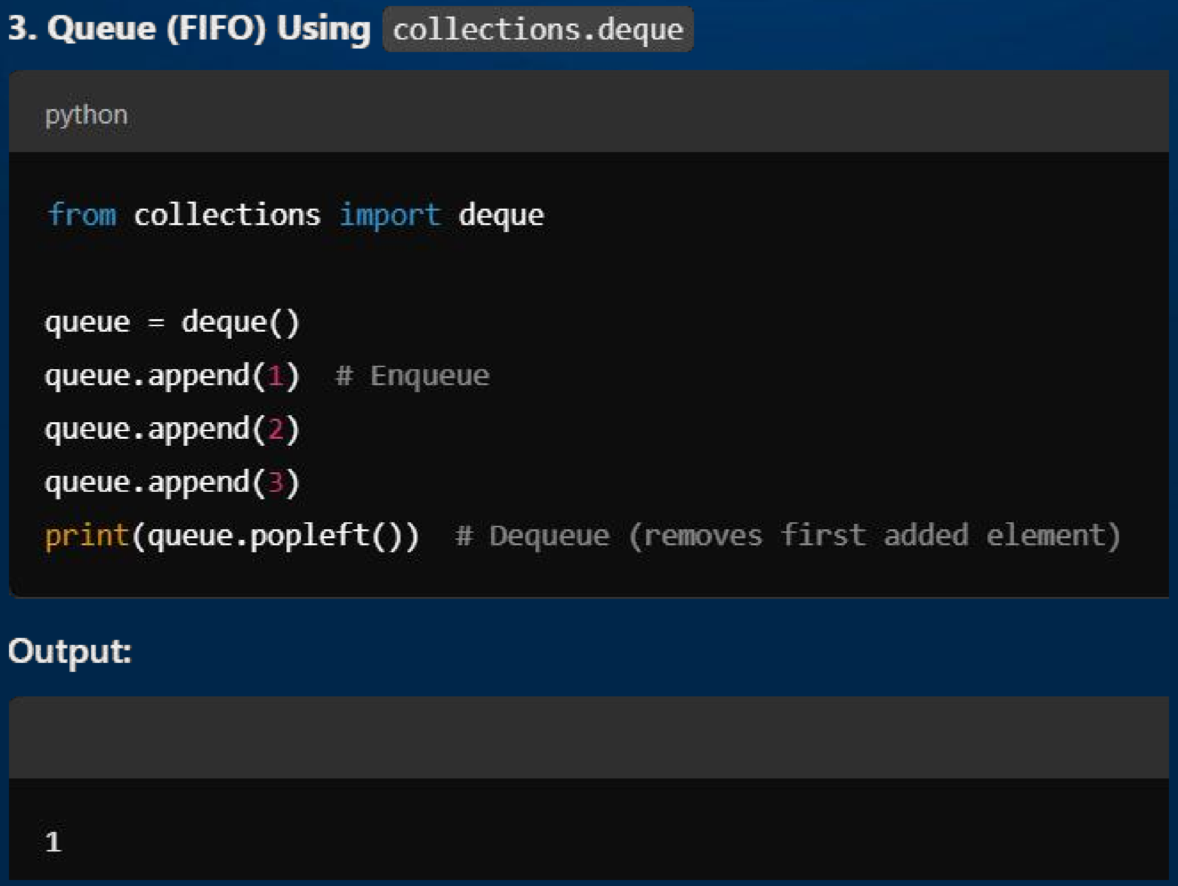
2. Stack (LIFO) Using List

python

```python
stack = []
stack.append(1)  # Push
stack.append(2)
stack.append(3)
print(stack.pop())  # Pop (removes last added element)
```

Output:

```
3
```


3. Queue (FIFO) Using `collections.deque`

python

```python
from collections import deque

queue = deque()
queue.append(1)  # Enqueue
queue.append(2)
queue.append(3)
print(queue.popleft())  # Dequeue (removes first added element)
```

Output:

```
1
```

Data structures organize and manage data efficiently.
Lists, Stacks, and Queues are common linear structures.
Choosing the right data structure improves performance.

6. File Handling

File handling is the process of creating, reading, writing, and managing files in a program. It allows programs to store and retrieve data permanently instead of relying only on temporary memory (RAM).

Types of File Operations:

Opening a File: Using open() function.

Reading a File: Using read(), readline(), or readlines().

Writing to a File: Using write() or writelines().

Appending Data: Using append (a) mode to add data without overwriting.

Closing a File: Using close() to free resources.

Example of File Handling in Python

1. Writing to a File

python

```python
file = open("example.txt", "w")  # Open file in write mode
file.write("Hello, this is a file handling example!")
file.close()  # Close the file
```

2. Reading from a File

python

```python
file = open("example.txt", "r")  # Open file in read mode
content = file.read()  # Read file content
print(content)
file.close()
```

Output:

kotlin

```
Hello, this is a file handling example!
```

3. Appending Data to a File

python

```python
file = open("example.txt", "a")  # Open file in append mode
file.write("\nThis is an appended line.")
file.close()
```

- ❑File handling enables permanent data storage.
- ❑Modes (r, w, a) determine the type of access.
- ❑Always close files after use to free system resources.
- ❑File handling is essential for working with databases, logs, and reports.

7. Error Handling

Error handling is the process of detecting, managing, and responding to errors in a program to prevent crashes and ensure smooth execution. It helps in debugging and improving program stability.

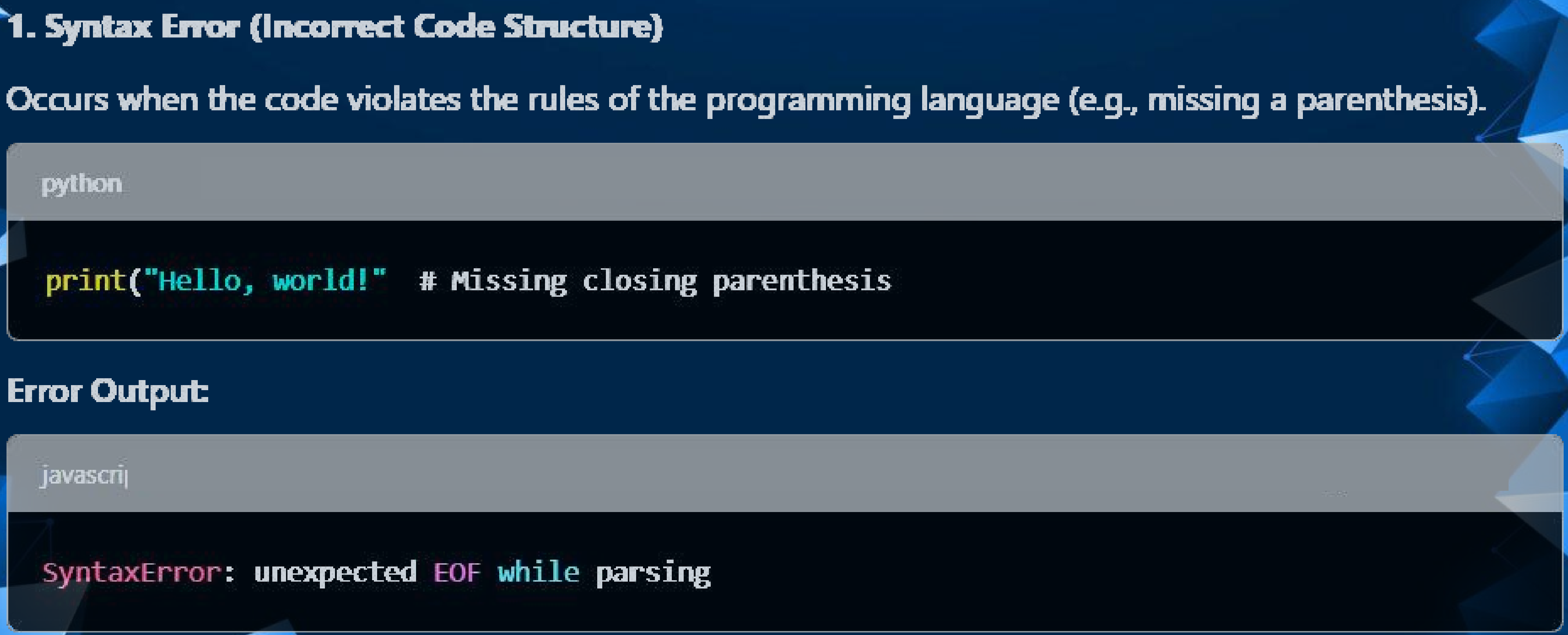
Types of Errors in Programming:

Syntax Errors: Errors in code structure (e.g., missing parentheses).

Runtime Errors: Errors that occur during execution (e.g., division by zero).

Logical Errors: Errors where the program runs but gives incorrect output.

Examples of Different Types of Errors in Python



1. Syntax Error (Incorrect Code Structure)

Occurs when the code violates the rules of the programming language (e.g., missing a parenthesis).

python

```python
print("Hello, world!"  # Missing closing parenthesis
```

Error Output:

javascri

```
SyntaxError: unexpected EOF while parsing
```

2. Runtime Error (Occurs During Execution)

Happens when the code is syntactically correct but encounters an error during execution.

```python
num = 10 / 0  # Division by zero is not allowed
```

Error Output:

```vbnet
ZeroDivisionError: division by zero
```

3. Logical Error (Incorrect Output, No Crash)

The program runs but produces the wrong result due to incorrect logic.

```python
def calculate_area(length, width):
    return length + width  # Incorrect logic, should be length * width

area = calculate_area(5, 4)
print("Area:", area)
```

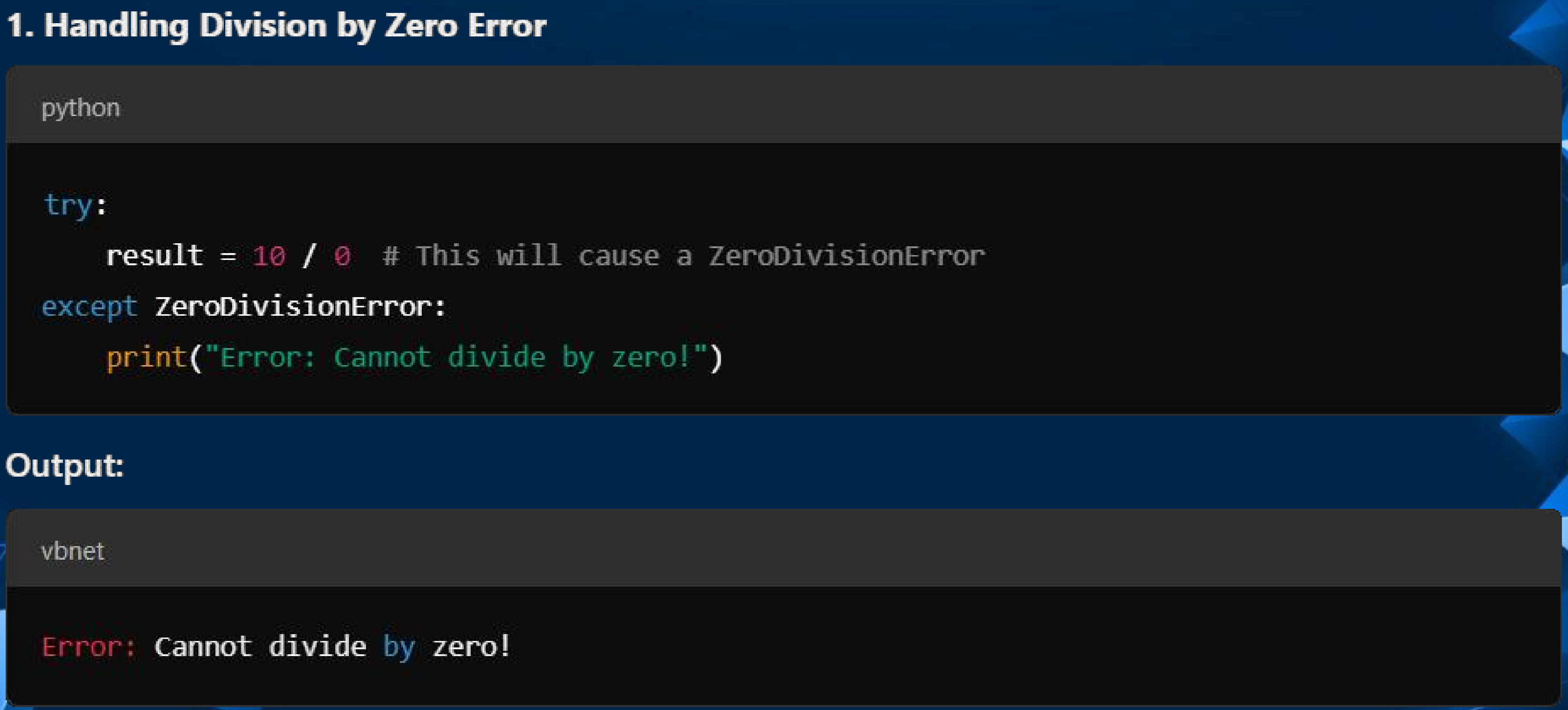
Incorrect Output:

```yaml
Area: 9  # Wrong result (should be 20)
```

(No error message, but incorrect calculation)

- ❑Syntax Errors → Code structure mistakes (e.g., missing :).
- ❑Runtime Errors → Errors during execution (e.g., ZeroDivisionError).
- ❑Logical Errors → Code runs but gives wrong results (e.g., wrong formula).
- ❑Fixing syntax errors requires proper syntax.
- ❑Handling runtime errors needs try-except.
- ❑Avoiding logical errors requires testing and debugging.

Error Handling Using try-except in Python



1. Handling Division by Zero Error

python

```python
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Output:

vbnet

```
Error: Cannot divide by zero!
```

2. Handling Multiple Errors

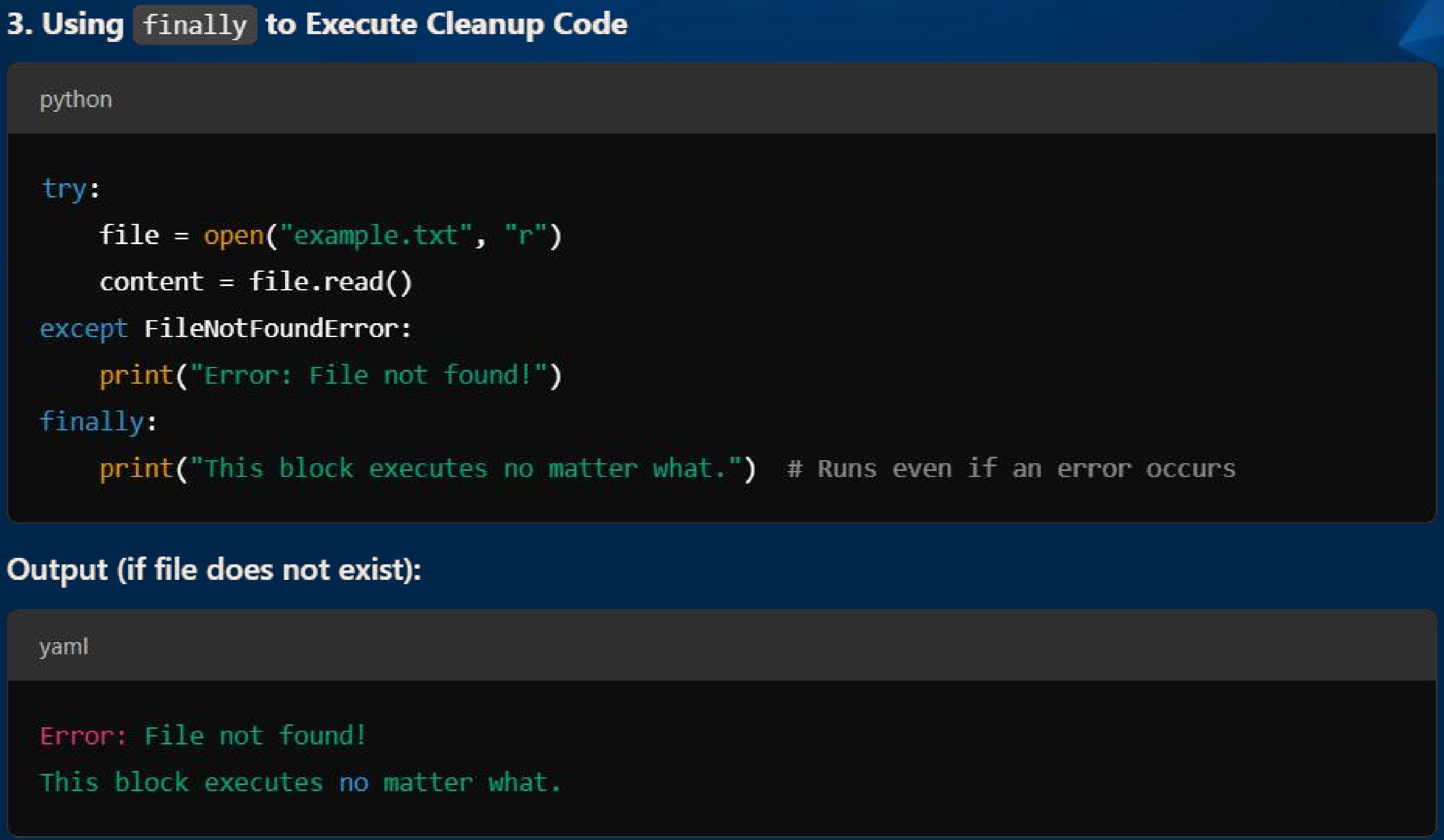
python

```python
try:
    num = int("abc")  # This will cause a ValueError
except ValueError:
    print("Error: Invalid number format!")
```

Output:

typescript

```typescript
Error: Invalid number format!
```



3. Using `finally` to Execute Cleanup Code

python

```python
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("This block executes no matter what.")  # Runs even if an error occurs
```

Output (if file does not exist):

yaml

```yaml
Error: File not found!
This block executes no matter what.
```

-Error handling prevents program crashes.
-try-except helps catch and manage errors.
-finally ensures code execution regardless of errors.
-Good error handling improves user experience and debugging.

8. Recursion

is a programming technique where a function calls itself to solve a problem. It breaks a complex problem into smaller subproblems until a base condition is met.

Key Concepts of Recursion:

Base Case: The condition that stops the recursion.

Recursive Case: The function calls itself with a smaller problem.

1. Base Case (Stopping Condition)

The base case is the condition that ends the recursion.

Without a base case, recursion will continue indefinitely, causing a stack overflow. It usually handles the simplest version of the problem.

Example: In factorial calculation, factorial(1) = 1 is the base case.

2. Recursive Case (Self-Calling Condition)

The recursive case calls the function itself with a smaller problem. This continues until the base case is met.

Example: In factorial, factorial(n) = n * factorial(n-1) is the recursive case.

Factorial Function with Base Case & Recursive Case

python

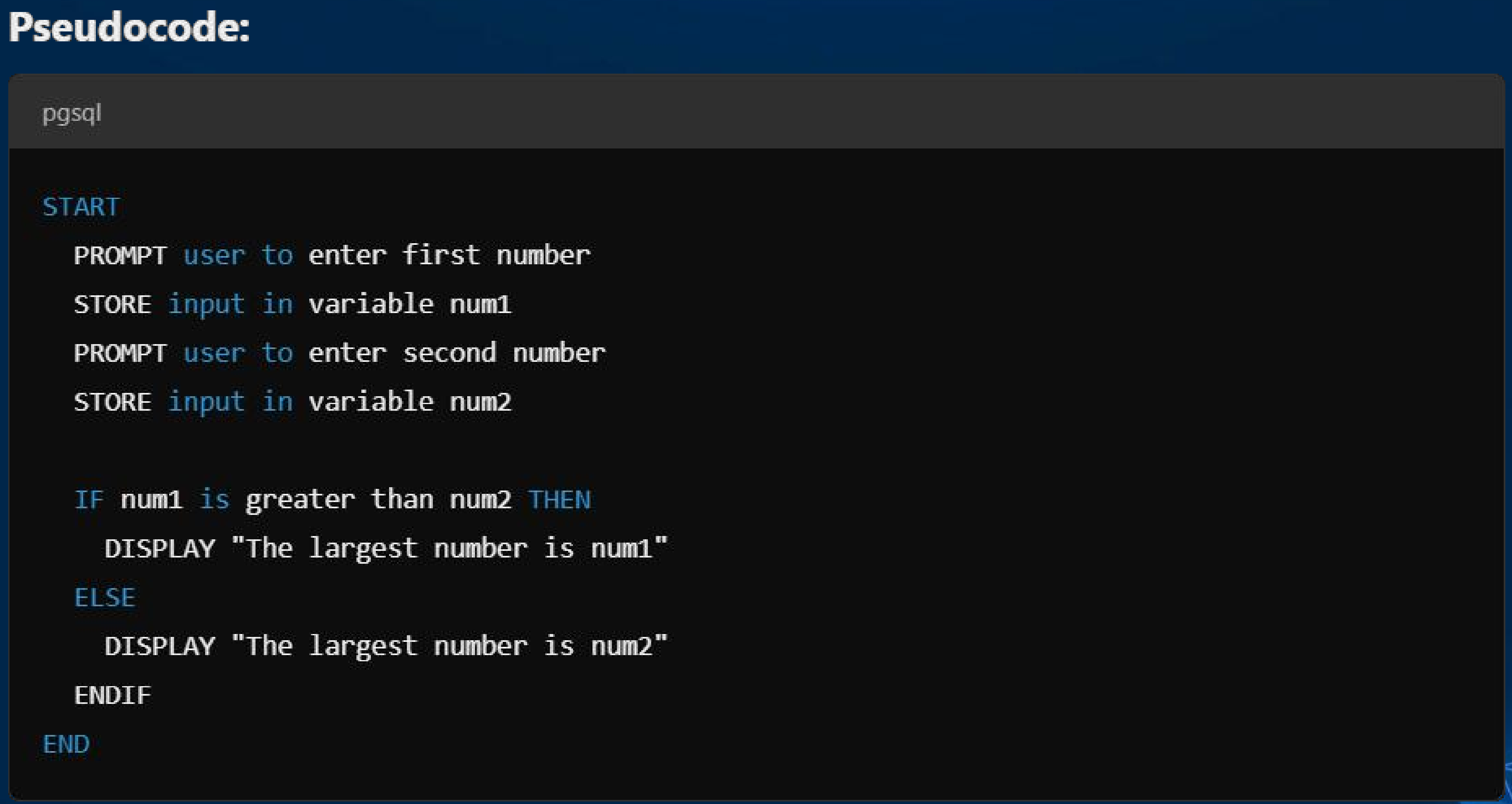
```python
def factorial(n):
    if n == 1:  # Base Case: Stop when n reaches 1
        return 1
    else:
        return n * factorial(n - 1)  # Recursive Case: Function calls itself

print(factorial(5))  # Output: 120
```

- **Breakdown:**

- `factorial(5) = 5 * factorial(4)`
- `factorial(4) = 4 * factorial(3)`
- `factorial(3) = 3 * factorial(2)`
- `factorial(2) = 2 * factorial(1)`
- **Base case:** `factorial(1) = 1` **(Stops recursion)**

- ❑Recursive Case: Calls the function with a smaller problem
- ❑Every recursive function must have a base case.
- ❑It is useful for problems like factorial, Fibonacci, and tree traversals.
- ❑Too much recursion can lead to a stack overflow (infinite recursion).
- ❑Base Case: Stops recursion when a condition is met.
- ❑Both are essential for recursion to work properly.

Simple Task

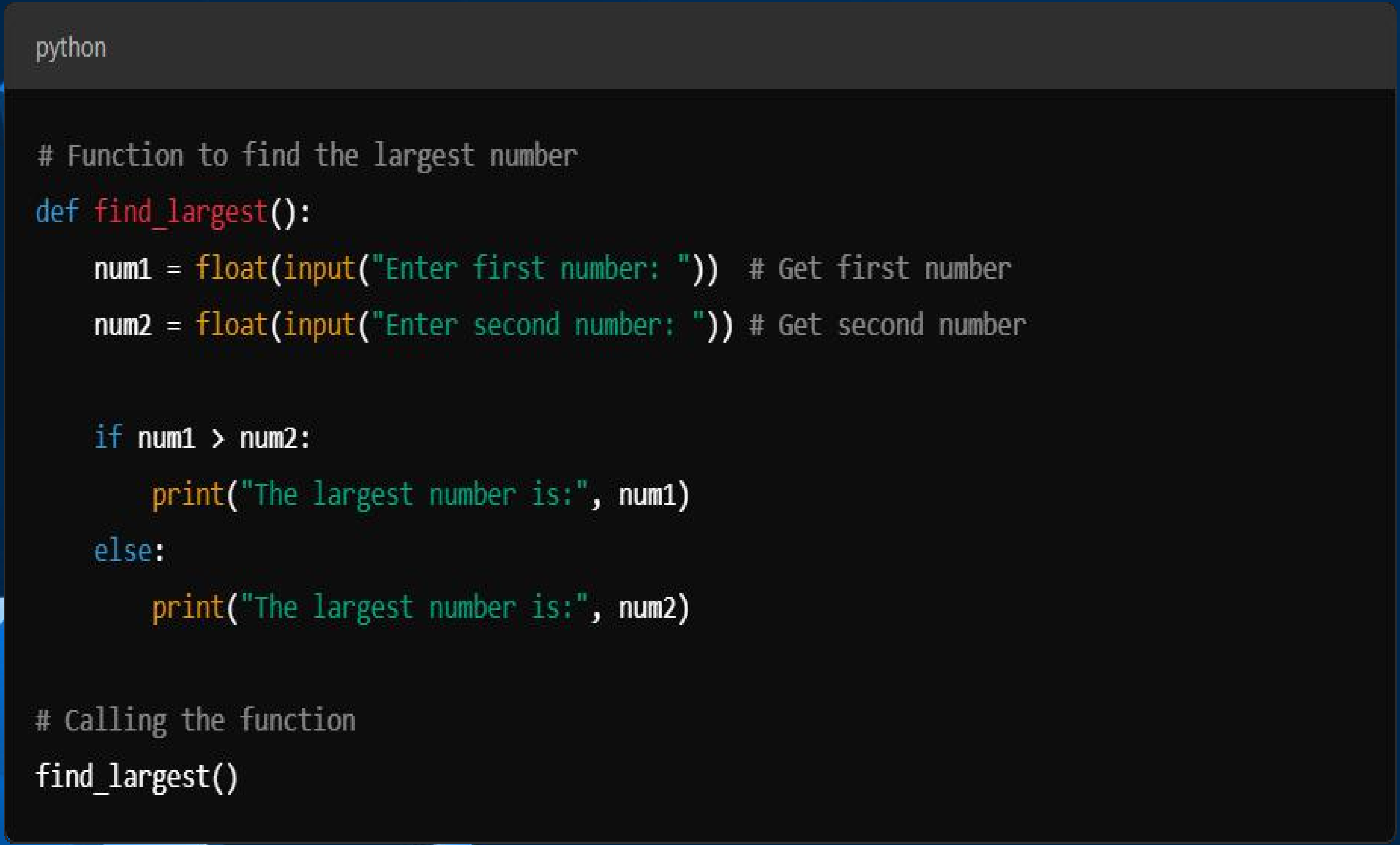


Pseudocode:

pgsql

```
START
  PROMPT user to enter first number
  STORE input in variable num1
  PROMPT user to enter second number
  STORE input in variable num2

  IF num1 is greater than num2 THEN
    DISPLAY "The largest number is num1"
  ELSE
    DISPLAY "The largest number is num2"
  ENDIF
END
```

Python Code Based on the Pseudocode



python

```python
# Function to find the largest number
def find_largest():
    num1 = float(input("Enter first number: "))  # Get first number
    num2 = float(input("Enter second number: ")) # Get second number

    if num1 > num2:
        print("The largest number is:", num1)
    else:
        print("The largest number is:", num2)

# Calling the function
find_largest()
```

Why Use Pseudocode?
-Clarifies logic before coding
-Makes debugging easier
-Helps in structured problem-solving

Using pseudocode ensures that we plan the solution before jumping into writing actual code.