APRIL MAY – 2022
PYTHON PROGRAMMING AND PROBLEM SOLVING -GE3151
SOLVED QUESTION PAPER

1. **Write an algorithm to find the smallest among three numbers.**

Algorithm:**
1. Input three numbers: `a`, `b`, and `c`.
2. Compare `a` with `b` and `c`.
3. If `a` is smaller than both `b` and `c`, then `a` is the smallest.
4. Else if `b` is smaller than both `a` and `c`, then `b` is the smallest.
5. Else, `c` is the smallest.
6. Output the smallest number.

2. **Which is better: iteration or recursion? Justify your answer.**

Answer: Iteration is generally better in terms of memory efficiency because recursion can lead to a high call stack usage, especially for large inputs, causing stack overflow errors. However, recursion can be more elegant and easier to understand for problems that involve repetitive subproblems, like traversing trees or solving factorials.**

3. **List any four built-in data types in Python.**

Answer: The four built-in data types in Python are:**
- `int` (Integer)
- `float` (Floating point number)
- `str` (String)
- `list` (List)

4. **How do you assign a value to a tuple in Python?**

Answer: In Python, tuples are immutable, meaning you cannot assign new values to existing elements. However, you can create a new tuple or reassign the entire tuple. For example:**
```python
my_tuple = (1, 2, 3)
```

5. **What are the purposes of the `pass` statement in Python?**

Answer: The `pass` statement in Python is a placeholder that does nothing when executed. It is used in scenarios where a statement is syntactically required but no action is needed, such as defining an empty function or class.**

6. **What is Linear Search?**

Answer: Linear search is a simple searching algorithm that checks each element in a list sequentially until the target element is found or the end of the list is reached. It has a time complexity of O(n).**

7. **Give examples for mutable and immutable objects.**

Answer:
- Mutable objects (can be changed): `list`, `dict`, `set`
- Immutable objects (cannot be changed): `int`, `float`, `str`, `tuple`

8. **What is the purpose of a dictionary in Python?**

Answer: A dictionary in Python is a data structure that stores data as key-value pairs. It allows for fast retrieval of values based on unique keys, making it useful for organizing and accessing data efficiently.**

9. **List any four file operations.**

Answer: Four common file operations in Python are:
- `open()` - to open a file
- `read()` - to read from a file
- `write()` - to write to a file
- `close()` - to close the file

10. **Write a Python program to count words in a sentence using the `split()` function.**

Answer:
```python
sentence = "This is a sample sentence"
word_count = len(sentence.split())
print("Number of words:", word_count)
```

PART B

1. (a) **List out the control flow statements in Python and explain repetition type in detail with a sample program.**

In Python, control flow statements help direct the flow of execution in a program. The primary control flow statements are:

1. **Conditional Statements**:
   - **if**: Executes a block of code if the condition is true.
   - **if-else**: Executes one block if the condition is true; another block if it is false.
   - **if-elif-else**: Checks multiple conditions sequentially and executes the corresponding block of code for the first true condition.**

2. **Loops**:
   - **for loop**: Used for iterating over a sequence (e.g., list, tuple, dictionary, string).
   - **while loop**: Repeats a block of code as long as a given condition is true.

3. **Control Flow Modifiers**:
   - **break**: Exits the loop immediately when encountered.
   - **continue**: Skips the current iteration and proceeds to the next.
   - **pass**: A placeholder statement that does nothing; used when a statement is syntactically required but no action is needed.

Repetition Types (Loops):

1. ***For Loop**:
   - Used to iterate over items in a sequence or to execute a block of code a specific number of times.
   - **Syntax**:
     ```python
     for item in sequence:
         # Code to execute
     ```
2. ***While Loop**:
   - Repeats a block of code as long as the condition is true.
   - **Syntax**:
     ```python
     while condition:
         # Code to execute
     ```

Example Programs:

1. **For Loop Example**:
   ```python
   print("For Loop Example:")
   for i in range(1, 6):  # Loop from 1 to 5
       print(i)
   ```

2. **While Loop Example**:
   ```python
   print("While Loop Example:")
   count = 1
   while count <= 5:
       print(count)
       count += 1
   ```

Explanation:
- In the `for` loop example, the loop iterates over numbers from 1 to 5, printing each number.
- In the `while` loop example, it keeps printing `count` until `count` is greater than 5.

1. (b) **What is recursion? Write and explain a Python program to find the factorial of a number using recursion.**

****Recursion**:**
Recursion is a programming technique in which a function calls itself in order to solve smaller sub-problems of the original problem. It is commonly used to solve problems that can be divided into similar sub-problems, such as calculating factorials, generating Fibonacci sequences, etc.

****Key Concepts**:**
- **Base Case: The condition under which the recursive function stops calling itself to prevent infinite loops.**
- **Recursive Case: The part of the function where it calls itself with modified arguments, gradually working towards the base case.**

Example: Finding Factorial Using Recursion

****Factorial Definition**:**
The factorial of a number \(n \) (denoted as \(n! \)) is the product of all positive integers from 1 to \(n \):
- **n! = n × (n-1) × (n-2) × ... × 1**

For instance, \(5! = 5 × 4 × 3 × 2 × 1 = 120 \).

****Recursive Formula**:**
- **n! = n × (n-1)!**
- **0! = 1 (base case)**

Python Program for Factorial Calculation

```python
def factorial(n):
    # Base case
    if n == 0:
        return 1
    # Recursive case
    else:
        return n * factorial(n - 1)

# Example usage
number = 5
result = factorial(number)
print(f"The factorial of {number} is: {result}")
```

****Explanation**:**
1. The `factorial` function takes an integer `n` as input.

2. **Base Case**: If `n` is 0, it returns 1 (since \(0! = 1 \)).
3. **Recursive Case**: If `n` is greater than 0, it calls itself with `n-1` and multiplies the result by `n`.
4. For example, `factorial(5)` will proceed as follows:
   - `5 * factorial(4)` → `5 * 4 * factorial(3)` → `5 * 4 * 3 * factorial(2)` → `5 * 4 * 3 * 2 * factorial(1)` → `5 * 4 * 3 * 2 * 1 * factorial(0) = 120`.

12.A Why Python is Called an Interpreted and Object-Oriented Programming Language

1. Interpreted Language:

Python is called an interpreted language because its code is executed line by line by an interpreter, rather than being compiled into machine code (binary) all at once. Here's what makes Python an interpreted language:

- **Execution by Interpreter**: In languages like C++ or Java, the code is compiled into machine language before running, which makes it faster but less flexible for certain tasks. Python, however, uses an interpreter, which reads and executes the code line by line. This makes Python flexible and easy to debug, as errors are reported immediately after each line is executed.

- **Platform Independence**: Python's interpreted nature makes it platform-independent at the code level. The Python interpreter can run on any operating system, like Windows, MacOS, or Linux, without modifying the code.

- **Dynamic Typing**: Since Python is interpreted, variable types do not need to be declared explicitly. The interpreter assigns the data type at runtime, allowing for a simpler and more flexible coding style. For example, a variable can hold an integer, and later in the program, it can be assigned a string value.

Advantages of Being Interpreted:
- **Ease of Debugging**: Errors are caught line-by-line, which makes it easier to identify and fix issues.
- **Flexibility**: You can test and modify Python code quickly, which is useful for prototyping and scripting tasks.

Example:

```python
x = 5
print(x)  # Outputs: 5
x = "Hello"
print(x)  # Outputs: Hello
```

In this example, `x` initially holds an integer but is reassigned to a string, showcasing Python's flexibility due to its interpreted nature.

2. Object-Oriented Language:

Python is also an object-oriented language because it allows for object-oriented programming (OOP) principles like inheritance, encapsulation, polymorphism, and

abstraction. These concepts are fundamental to managing complex applications in an organized manner. Here's how Python supports OOP:

- **Classes and Objects: Python enables creating classes (blueprints) and objects (instances of classes), making it easy to model real-world entities. Classes can have attributes (data) and methods (functions).**

- **Inheritance: Python supports inheritance, where a class can inherit attributes and methods from another class. This promotes code reuse and hierarchical organization of classes.**

- **Encapsulation: Python supports encapsulation by allowing private and public access modifiers for data and methods, protecting the integrity of data within objects.**

- **Polymorphism: Polymorphism in Python allows different classes to be treated as instances of the same class through shared methods, enabling flexibility in code design.**

- **Abstraction: Python uses abstract classes and interfaces to define a set of methods that must be created within any child classes built from the abstract class.**

****Advantages of Being Object-Oriented**:**
- **Modular Code: OOP allows you to break down large problems into smaller, manageable parts.**
- **Code Reusability: Inheritance allows you to reuse code across multiple classes.**
- **Flexibility and Scalability: OOP makes it easier to scale and extend programs.**

****Example**:**
```python
class Animal:
**   def speak(self):**
**   pass**

class Dog(Animal):
**   def speak(self):**
**   return "Woof!"**

class Cat(Animal):
**   def speak(self):**
**   return "Meow!"**

dog = Dog()
cat = Cat()
print(dog.speak()) # Outputs: Woof!
print(cat.speak()) # Outputs: Meow!
```
In this example, `Dog` and `Cat` inherit from the `Animal` class and implement the `speak` method, demonstrating polymorphism and inheritance in Python.

12)B) Python Program to Swap Two Numbers (With and Without Temporary Variables)

Swapping two numbers means exchanging their values. Let's discuss two methods to achieve this in Python: using a temporary variable and without using a temporary variable. Each method has its advantages, and knowing both is helpful.

1. Swapping with a Temporary Variable

This is the most straightforward method. We use an extra variable to temporarily hold one of the values during the swapping process.

****Program:****
```python
# Swapping two numbers using a temporary variable
x = 10
y = 20

print("Before Swapping:")
print("x =", x, "y =", y)

# Using a temporary variable
temp = x
x = y
y = temp

print("After Swapping (with temporary variable):")
print("x =", x, "y =", y)
```

****Explanation:****
1. Initially, `x` is assigned the value `10`, and `y` is assigned the value `20`.
2. We introduce a temporary variable `temp` to hold the value of `x`. Now, `temp = 10`.
3. We then assign the value of `y` to `x`, making `x = 20`.
4. Finally, we assign the value of `temp` (which holds the original value of `x`) to `y`, making `y = 10`.
5. After this process, the values of `x` and `y` are swapped.

****Output:****
```
Before Swapping:
x = 10 y = 20
After Swapping (with temporary variable):
x = 20 y = 10
```

****Advantages:****

- Simple and easy to understand.
- Safe to use as it doesn’t alter the original variables directly.

****Disadvantages:****
- Requires an extra variable (`temp`), which consumes additional memory, albeit a small amount.

Swapping Without a Temporary Variable

In Python, we can swap two numbers without using a temporary variable by using either arithmetic operations or tuple unpacking.

(a) Using Arithmetic Operations

****Program:****
```python
 # Swapping two numbers without a temporary variable using arithmetic
x = 10
y = 20

print("Before Swapping:")
print("x =", x, "y =", y)

 # Swapping without a temporary variable
x = x + y # Now x is 30 (10 + 20)
y = x - y # Now y is 10 (30 - 20)
x = x - y # Now x is 20 (30 - 10)

print("After Swapping (without temporary variable - arithmetic):")
print("x =", x, "y =", y)
```

****Explanation:****
1. We start with `x = 10` and `y = 20`.
2. We add `x` and `y` and store the result in `x`, so `x` becomes `30` (`10 + 20`).
3. We then subtract the new value of `x` (which is `30`) from `y`, giving `y = 10` (`30 - 20`).
4. Finally, we subtract the new value of `y` (which is `10`) from `x`, making `x = 20` (`30 - 10`).

****Output:****
```
Before Swapping:
x = 10 y = 20
After Swapping (without temporary variable - arithmetic):
x = 20 y = 10
```

****Advantages:****

- Does not require an extra variable, which saves memory.

****Disadvantages:****
- This method can cause overflow issues in some languages (not in Python, but this could be an issue in languages with limited integer ranges).
- Slightly less readable compared to the method using a temporary variable.

13, A) what is the difference between break and continue in python explain with suitable examples

1. `break` Statement

The `break` statement is used to **exit a loop prematurely when a specific condition is met. Once the `break` statement is executed, the loop terminates, and the program control moves to the first line after the loop.**

Example of `break`:
Let's say we have a loop that prints numbers from 1 to 10, but we want to stop printing if we encounter the number 5.

```python
for i in range(1, 11):  # Loop from 1 to 10
    if i == 5:
        break  # Exit the loop when i is 5
    print(i)
```

****Output:****
```
1
2
3
4
```

****Explanation:****
- The loop starts printing numbers from 1 to 10.
- When `i` becomes 5, the `break` statement is triggered, and the loop is exited immediately.
- As a result, the numbers after 4 are not printed.

****Use Case:****
The `break` statement is commonly used to terminate a loop when a desired condition is met, such as finding a specific item in a list and then stopping further search.

2. `continue` Statement

The `continue` statement is used to **skip the current iteration of the loop and move to the next iteration. When the `continue` statement is executed, the remaining code inside the loop for that iteration is skipped, and the loop continues with the next iteration.**

Example of `continue`:

Let's modify our previous example to print numbers from 1 to 10 but skip the number 5.

```python
for i in range(1, 11):  # Loop from 1 to 10
    if i == 5:
        continue  # Skip the current iteration when i is 5
    print(i)
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Explanation:

- The loop starts printing numbers from 1 to 10.

- When `i` is 5, the `continue` statement is triggered, skipping the print statement for that iteration.

- The loop then continues to the next iteration, printing 6 to 10 but skipping 5.

Use Case:

The `continue` statement is useful when you want to skip certain iterations in a loop without terminating the entire loop, such as processing only certain elements in a list based on a condition.

Summary of Differences

Feature	`break`	`continue`
Function	Exits the loop completely	Skips the current iteration and moves to the next
Loop Behavior	Ends loop execution	Continues loop execution
Use Case	Used to terminate the loop early	Used to skip specific iterations

13)BWHAT IS STRING FUNCTION IN PYTHON , EXPLAIN ANY THREE PYTHON STRING METHODS?

In Python, strings are a sequence of characters, and Python provides many built-in string functions (methods) to perform various operations on strings. String methods help

manipulate and transform strings for different purposes. Here, I'll explain three commonly used Python string methods: `upper()`, `find()`, and `replace()`.

1. `upper()` Method

The `upper()` method converts all lowercase letters in a string to uppercase letters and returns a new string with the changes. It doesn’t modify the original string (strings in Python are immutable), but rather returns a modified copy.

Syntax:
```python
string.upper()
```

Example:
```python
text = "hello world"
uppercase_text = text.upper()
print(uppercase_text)
```

Output:
```
HELLO WORLD
```

Explanation:
In the example above, `upper()` converts "hello world" to "HELLO WORLD".

---

2. `find()` Method

The `find()` method searches for a substring within a string and returns the index of the first occurrence of the substring. If the substring is not found, it returns `-1`.

Syntax:
```python
string.find(substring, start, end)
```

- `substring` is the string to search for.
- `start` (optional) is the starting index to begin the search.
- `end` (optional) is the ending index where the search stops.

Example:
```python
text = "welcome to python programming"
```

index = text.find("python")
print(index)
```

**Output:**
```
11
```

**Explanation:**
**In the example, the `find()` method locates the substring "python" in the string and returns the index `11`, where it first appears.**

### 3. `replace()` Method

**The `replace()` method replaces occurrences of a specified substring within a string with another substring and returns a new string with the replacements.**

**Syntax:**
```python
string.replace(old, new, count)
```

- `old` is the substring to be replaced.
- `new` is the substring that will replace `old`.
- `count` (optional) specifies the maximum number of replacements (if not specified, all occurrences are replaced).

Example:
```python
text = "I like apples, apples are sweet"
modified_text = text.replace("apples", "oranges")
print(modified_text)
```

Output:
```
I like oranges, oranges are sweet
```
Explanation:
In the example above, `replace()` changes every occurrence of "apples" in the string to "oranges".

14)A) Define python list ? how to add element in the list explain with suitable example

In Python, a **list is a built-in data structure used to store multiple items in a single variable. Lists are **ordered**, **mutable**, and can contain elements of different types, such as integers, strings, or even other lists.**
```

Key Concepts:
- **Ordered**: Items in a list are stored in a specific order, and this order is maintained.
- **Mutable**: You can change the content of the list after it is created.
- **Indexed**: Each item in the list has a position (index), starting from 0.

How to Create a List:
You can create a list by placing elements inside square brackets `[]` and separating them with commas.

```python
my_list = [1, 2, 3, 4, 5]
```

Adding Elements to a List:
To add elements to a list, you can use the following methods:
1. **`append()`** – Adds a single element to the end of the list.
2. **`insert()`** – Adds an element at a specific index.
3. **`extend()`** – Adds multiple elements from another iterable (like another list) to the end of the list.

Examples:

1. **Using `append()`**:
```python
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

2. **Using `insert()`**:
```python
my_list = [1, 2, 3]
my_list.insert(1, 10)  # Adds 10 at index 1
print(my_list)  # Output: [1, 10, 2, 3]
```

3. **Using `extend()`**:
```python
my_list = [1, 2, 3]
my_list.extend([4, 5])
print(my_list)  # Output: [1, 2, 3, 4, 5]
```

These methods allow you to modify and add to a list dynamically.

14)b) Explain bubble sort algorithm with python program

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated for each element until the list is sorted. The algorithm is called "bubble sort" because smaller elements bubble up to the top of the list.

Key Concepts:

- **Comparing Adjacent Elements**: The algorithm compares adjacent pairs and swaps them if needed.
- **Multiple Passes**: Each pass moves the largest unsorted element to its correct position.
- **Time Complexity**: The worst-case and average time complexity of bubble sort is $O(n^2)$, making it inefficient for large datasets.

Algorithm Steps:

1. Start at the beginning of the list.
2. Compare each pair of adjacent elements.
3. If the current element is greater than the next, swap them.
4. After one full pass, the largest element will be at the end.
5. Repeat the process for the remaining unsorted portion of the list.
6. Stop when no more swaps are needed, indicating the list is sorted.

Python Program:

```python
# Bubble Sort Algorithm
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all elements in the list
    for i in range(n):
        swapped = False  # Flag to optimize the algorithm

        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):

            # Compare adjacent elements and swap if needed
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # If no elements were swapped in the inner loop, the list is sorted
        if not swapped:
            break

# Example usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", arr)

bubble_sort(arr)
```

print("Sorted list:", arr)

Explanation of the Program:

- **`bubble_sort(arr)`: This function sorts the list `arr` using the bubble sort algorithm.**
- **Outer loop (`for i in range(n)`): The outer loop runs for `n` passes, where `n` is the length of the list.**
- **Inner loop (`for j in range(0, n-i-1)`): This loop compares adjacent elements and performs swaps. The range reduces with each pass as the largest elements are sorted at the end.**
- **Swapping: If an element at index `j` is greater than the element at index `j+1`, they are swapped.**
- **Optimization with `swapped`: If no swaps are made during a pass, the list is already sorted, and the algorithm stops early.**

Example Output:

```
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list: [11, 12, 22, 25, 34, 64, 90]
```

This simple algorithm is easy to understand but is inefficient for large datasets due to its time complexity of $O(n^2)$.

15)A) why does python require file handling explain opening files in python with all modes?

****File Handling** in Python is essential for performing operations such as reading from or writing to files. When working with large data sets, storing and retrieving information from files is often more practical than storing everything in memory. File handling in Python provides a way to interact with files (e.g., text files, binary files) on your system.**

Python offers several modes for opening files, depending on what you intend to do (read, write, append, etc.). These modes dictate how the file is opened and how you can interact with it.

Why Python Requires File Handling:

1. **Persistence of Data: File handling allows you to persist data beyond the runtime of a program. Data can be stored in files and retrieved later.**
2. **Data Exchange: Files allow programs to exchange data with other programs or systems, making it easier to share information.**
3. **Working with Large Data: For large data, it is not feasible to store everything in memory. File handling helps read and write data to/from files in chunks.**

Opening Files in Python:

To open a file in Python, you use the built-in `open()` function. The general syntax is:

```python
file_object = open("file_name", "mode")
```

Here, `file_name` is the name of the file you want to work with, and `mode` specifies the file access mode.

File Modes in Python:
1. **`'r'` - **Read Mode**:**
- Opens the file for reading. If the file does not exist, it raises a `FileNotFoundError`.
- **Example:**
```python
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

2. **`'w'` - **Write Mode**:**
- Opens the file for writing. If the file exists, it truncates (deletes) the content of the file. If the file does not exist, it creates a new file.
- **Example:**
```python
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()
```

3. **`'a'` - **Append Mode**:**
- Opens the file for appending. It allows adding content at the end of the file without truncating it. If the file does not exist, it creates a new one.
- **Example:**
```python
file = open("example.txt", "a")
file.write("\nAppending some new content!")
file.close()
```

4. **`'x'` - **Exclusive Creation**:**
- Creates a new file, but if the file already exists, it raises a `FileExistsError`.
- **Example:**
```python
file = open("example.txt", "x")
file.write("This is a new file!")
file.close()
```

5. **`'b'` - **Binary Mode**:**
- This mode is used for reading or writing binary files. It is added to other modes (e.g., `'rb'`, `'wb'`) to open the file in binary mode.
- **Example:**

```python
file = open("image.jpg", "rb")
data = file.read()
file.close()
```

6. **`'t'` - **Text Mode**:**
- This is the default mode. It is used for reading and writing text files. Text files are automatically encoded and decoded based on the system's default encoding.
- **Example:**
```python
file = open("example.txt", "rt")
content = file.read()
file.close()
```

7. **`'r+'` - **Read and Write Mode**:**
- Opens the file for both reading and writing. The file must exist; otherwise, it raises a `FileNotFoundError`.
- **Example:**
```python
file = open("example.txt", "r+")
content = file.read()
print(content)
file.write("New content added")
file.close()
```

8. **`'w+'` - **Write and Read Mode**:**
- Opens the file for both writing and reading. If the file exists, it truncates the content; if the file does not exist, it creates a new one.
- **Example:**
```python
file = open("example.txt", "w+")
file.write("Some new content!")
file.seek(0) # Go to the beginning of the file
content = file.read()
print(content)
file.close()
```

9. **`'a+'` - **Append and Read Mode**:**
- Opens the file for both appending and reading. The file pointer is placed at the end of the file for writing.
- **Example:**
```python
file = open("example.txt", "a+")
file.write("\nAppending more content!")

```
file.seek(0)  # Go to the beginning of the file to read
content = file.read()
print(content)
file.close()
```

15)B) Give a brief notes on python exception handling using try except raise and finally statements

Concept:
Exception handling in Python allows you to manage errors and exceptions that may occur during program execution. It helps to prevent the program from crashing and ensures that the program can continue executing or handle the error appropriately. Python provides several keywords to handle exceptions: `try`, `except`, `raise`, and `finally`.

Key Concepts:
- **Exception**: An error that disrupts the normal flow of a program.
- **Handling Exceptions**: Allows the program to continue running by catching and responding to errors.

Python Exception Handling Statements:

1. **`try`**:
   - Used to enclose the code that may raise an exception.
   - If an exception occurs, the code inside the `except` block will be executed.

   Example:
   ```python
   try:
       result = 10 / 0  # Division by zero will raise an exception
   except ZeroDivisionError:
       print("Cannot divide by zero!")
   ```

2. **`except`**:
   - Catches and handles the exception that occurs in the `try` block.
   - You can specify the type of exception to catch (e.g., `ZeroDivisionError`, `ValueError`, etc.).
   - You can also catch all exceptions using a general `except` block.

   Example:
   ```python
   try:
       num = int(input("Enter a number: "))
   except ValueError:
       print("That's not a valid number!")
   ```

3. **`else`** (Optional):
   - Executed if no exception occurs in the `try` block.
   - Can be used to run code when no errors are encountered.

   Example:
   ```python
   try:
       num = int(input("Enter a number: "))
   except ValueError:
       print("That's not a valid number!")
   else:
       print(f"You entered {num}")
   ```

4. **`finally`**:
   - The code inside the `finally` block is always executed, whether an exception is raised or not.
   - Useful for cleanup operations like closing files, releasing resources, etc.

   Example:
   ```python
   try:
       file = open("example.txt", "r")
       content = file.read()
   except FileNotFoundError:
       print("File not found!")
   finally:
       file.close()  # Always close the file, even if an exception occurred
   ```

5. **`raise`**:
   - Used to explicitly raise an exception in your program.
   - Can be used to throw exceptions when certain conditions are met, or to re-raise caught exceptions.

   Example:
   ```python
   def check_age(age):
       if age < 18:
           raise ValueError("Age must be 18 or older!")
       print(f"Your age is {age}")

   try:
       check_age(16)
   except ValueError as e:
       print(e)
   ```

Summary of Flow:
1. **`try` block**: Code that may raise an exception.
2. **`except` block**: Handles the exception (if any).
3. **`else` block**: Executes if no exception is raised.
4. **`finally` block**: Always executes, regardless of whether an exception occurred or not.
5. **`raise`**: Used to explicitly raise exceptions when needed.

Example Combining All:
```python
def divide(a, b):
    try:
        if b == 0:
            raise ZeroDivisionError("Division by zero is not allowed")
        result = a / b
    except ZeroDivisionError as e:
        print(e)
    else:
        print(f"The result is {result}")
    finally:
        print("Execution completed.")

divide(10, 0)  # Will raise ZeroDivisionError
divide(10, 2)  # Will execute normally
```