PYTHON NOTES

***Caution**: Please note all these answers are from ChatGPT, can't guarantee for accuracy, but you can refer to this*.

Chapter 1: Python Basics

Q1. Explain the concept of an expression in Python with examples. (CO1) (BTL 2) (Marks: 5)

• Definition of an expression: 2 marks
• Examples of expressions: 2 marks
• Explanation of how Python evaluates expressions: 1 mark

Q1. Concept of an Expression in Python

Definition of an Expression

An **expression** in Python is a combination of values, variables, operators, and function calls that the Python interpreter can evaluate to produce a result. Expressions are the building blocks of Python programs and are used to perform computations, manipulate data, or produce a value.

For example:

- `5 + 3` is an expression that evaluates to `8`.
- `x * y` (where `x` and `y` are variables) is an expression that evaluates to the product of `x` and `y`.

Expressions can range from simple (like a single value) to complex (involving multiple operators and function calls).

Examples of Expressions

1. **Arithmetic Expression**:
   ```
   10 + 20  # Evaluates to 30
   ```
2. **Relational Expression**:
   ```
   5 > 3  # Evaluates to True
   ```
3. **Logical Expression**:
   ```
   True and False  # Evaluates to False
   ```
4. **String Concatenation**:
   ```
   "Hello" + " World!"  # Evaluates to "Hello World!"
   ```
5. **Function Call Expression**:
   ```
   len("Python")  # Evaluates to 6
   ```
6. **Variable Expression**:
   ```
   x = 10
   ```

```
y = x + 5  # Evaluates to 15
```

How Python Evaluates Expressions

1. **Order of Operations**:
   Python uses operator precedence (PEMDAS - Parentheses, Exponents, Multiplication/Division, Addition/Subtraction) to decide the order in which operations are performed. For example:

   ```
   2 + 3 * 4  # Multiplies first, resulting in 2 + 12 = 14
   ```

2. **Left-to-Right Evaluation**:
   Python evaluates operators of the same precedence from left to right. For example:

   ```
   10 - 5 + 2  # Subtracts first, resulting in 5 + 2 = 7
   ```

3. **Function Evaluation**:
   Function calls within an expression are evaluated first, and their results are used in the remaining expression. For example:

   ```
   x = len("abc") * 2  # len("abc") evaluates to 3, and the result is 3 * 2 =
   6
   ```

4. **Short-Circuiting in Logical Expressions**:
   Logical operators like `and` and `or` are evaluated using short-circuiting. For example:

   ```
   False and (5 / 0)  # Python does not evaluate (5 / 0) because False stops
   the `and` operation
   ```

5. **Expression Assignment**:
   If an expression is part of an assignment statement, the evaluated value is stored in the variable:

   ```
   result = 3 ** 2 + 1  # 3 ** 2 evaluates to 9, and 9 + 1 = 10, which is
   assigned to `result`
   ```

Summary

Expressions in Python are fundamental for performing calculations, making decisions, and manipulating data. Understanding how Python evaluates expressions allows developers to write efficient and clear code.

Q2. List and explain the precedence of math operators in Python with an example.(CO1) (BTL 2) (Marks: 5)

• List of math operators and precedence order: 3 marks
• Example with explanation: 2 marks

Q2. Precedence of Math Operators in Python

List of Math Operators and Their Precedence

Math operators in Python have a specific order of precedence, which determines the sequence in which they are evaluated in expressions. Operators with higher precedence are evaluated before those with lower precedence. If operators have the same precedence, they are evaluated from **left to right**.

Operator	Description	Precedence Level
`**`	Exponentiation	Highest
`*`, `/`, `//`, `%`	Multiplication, Division, Floor Division, Modulus	Higher
`+`, `-`	Addition, Subtraction	Lower
`=`	Assignment	Lowest

Example of Precedence

Consider the expression:

```
result = 10 + 20 * 2 ** 2 - 5 / 5
```

Step-by-step evaluation:

1. **Exponentiation (`**`)**:
   `2 ** 2 = 4`
   Now the expression is:
   `result = 10 + 20 * 4 - 5 / 5`
2. **Multiplication and Division (`*`, `/`)**:
   - `20 * 4 = 80`
   - `5 / 5 = 1.0`
     Now the expression is:
     `result = 10 + 80 - 1.0`
3. **Addition and Subtraction (`+`, `-`)**:
   - `10 + 80 = 90`
   - `90 - 1.0 = 89.0`
     Final value of `result` is `89.0`.

Explanation of Precedence

- The exponentiation operator (`**`) has the highest precedence, so it is evaluated first.
- Multiplication (`*`) and division (`/`) have higher precedence than addition (`+`) and subtraction (`-`).
- Operators with the same precedence (like `*` and `/`) are evaluated **left to right**.
- Parentheses can be used to override the precedence and ensure that specific parts of the expression are evaluated first.

Example Using Parentheses

Using parentheses changes the evaluation order:

```
result = (10 + 20) * (2 ** 2) - (5 / 5)
```

Here:

1. `10 + 20 = 30`
2. `2 ** 2 = 4`
3. `30 * 4 = 120`
4. `5 / 5 = 1.0`
5. `120 - 1.0 = 119.0`

Final value of `result` is `119.0`.

Using precedence correctly ensures accurate calculations in Python programs.

Q3. Differentiate between integers, floating-point numbers, and strings in Python.(CO1) (BTL 3) (Marks: 5)
• Definition and examples for integers: 1.5 marks
• Definition and examples for floating-point numbers: 1.5 marks
• Definition and examples for strings: 2 marks

Q3. Difference Between Integers, Floating-Point Numbers, and Strings in Python

1. Integers

- **Definition**:
  Integers in Python represent whole numbers without any fractional or decimal part. They can be positive, negative, or zero.
- **Examples**:

  ```
  x = 5       # Positive integer
  y = -10     # Negative integer
  z = 0       # Zero
  ```

- **Characteristics**:
  - Integers have unlimited precision (no size limit).
  - Common operations include addition (`+`), subtraction (`-`), multiplication (`*`), and modulus (`%`).

2. Floating-Point Numbers

- **Definition**:
  Floating-point numbers, or floats, represent real numbers that include a decimal point. They can also represent scientific notation for very large or small values.
- **Examples**:

  ```
  a = 3.14        # Positive float
  b = -0.001      # Negative float
  c = 2.5e3       # Scientific notation (2.5 × 10³ = 2500.0)
  ```

- **Characteristics**:
    - Floats have a limited precision depending on the system, typically up to 15-17 decimal places.
    - Useful for calculations involving fractions or decimals.
    - Common operations include rounding (`round()`), truncation (`math.floor()`), and division (`/`).

3. Strings

- **Definition**:
  Strings in Python represent sequences of characters enclosed in single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or `"""`). They are used to store text.
- **Examples**:

```
name = "Aliya"                # Double quotes
greeting = 'Hello, world!'   # Single quotes
paragraph = """This is
a multi-line string."""       # Triple quotes
```

- **Characteristics**:
    - Strings are immutable (cannot be changed after creation).
    - Common operations include concatenation (`+`), repetition (`*`), and slicing (`[start:end]`).
    - Strings can store letters, numbers, symbols, and even escape sequences (e.g., `\n` for newline).

Key Differences

Type	Definition	Example
Integer	Whole numbers without a decimal point.	`10`, `-5`, `0`
Floating-Point	Real numbers with a decimal point.	`3.14`, `-2.71`
String	Sequence of characters (text).	`"Hello"`, `'123'`

By understanding these types, developers can select the appropriate data type for their variables and operations in Python.

Q4. Write a Python program to take a user's name and age as input and display a personalized message along with the length of their name. (CO1) (BTL 4) (Marks: 5)

• Input functionality with comments: 2 marks
• String concatenation and len() usage: 2 marks
• Output message correctness: 1 mark

Python Program to Take User's Name and Age as Input

```
# Input functionality with comments
# Taking the user's name as input
name = input("Enter your name: ")  # Input the user's name as a string
```

```
# Taking the user's age as input
age = input("Enter your age: ")  # Input the user's age as a string (will be
converted to an integer if needed)

# String concatenation and len() usage
# Calculating the length of the user's name
name_length = len(name)  # Using len() to get the number of characters in the
name

# Creating a personalized message
message = "Hello, " + name + "! You are " + age + " years old, and your name has
" + str(name_length) + " characters."

# Output message correctness
# Displaying the personalized message
print(message)
```

How the Program Works

1. **Input Functionality**:
   - The program uses the `input()` function to get the user's name and age.
   - The name is stored as a string, and the age is initially stored as a string (it can be converted to an integer if necessary for calculations).
2. **String Concatenation and `len()` Usage**:
   - The `len()` function calculates the number of characters in the name.
   - The personalized message is constructed using string concatenation (`+`) and converting the length to a string using `str()`.
3. **Output Message**:
   - The program displays a personalized message in the format:
     Hello, [Name]! You are [Age] years old, and your name has [Length] characters.

Example Run

Input:

```
Enter your name: Aliya
Enter your age: 21
```

Output:

```
Hello, Aliya! You are 21 years old, and your name has 5 characters.
```

Q5. What are variables in Python? How are they declared and overwritten? Explain with examples. (CO1) (BTL 2) (Marks: 5)

• Definition and declaration of variables: 2 marks
• Explanation of overwriting variables: 2 marks
• Example demonstrating the concept: 1 mark

Q5. Variables in Python: Definition, Declaration, and Overwriting

---

1. Definition and Declaration of Variables

- **Definition**:
  A variable in Python is a symbolic name that acts as a reference or container for storing data. It allows the program to access and manipulate data. Variables in Python are dynamically typed, meaning you don't need to declare their type explicitly; it is inferred based on the value assigned.
- **Declaration**:
  In Python, variables are created when a value is assigned to them using the = operator.

  ```
  variable_name = value
  ```

---

2. Explanation of Overwriting Variables

- **Overwriting**:
  In Python, you can reassign a new value to an existing variable. This effectively overwrites the previous value. A variable can even change its type when overwritten because Python is dynamically typed.
- **Example of Overwriting**:

  ```
  x = 10           # x is an integer with a value of 10
  x = 3.14         # x is now overwritten as a float with a value of 3.14
  x = "Hello!"     # x is now overwritten as a string with the value
  "Hello!"
  ```

- The previous value of the variable is no longer accessible after it is overwritten, unless it was explicitly stored elsewhere.

---

3. Example Demonstrating the Concept

```
# Declaration of variables
name = "Aliya"  # Variable 'name' stores a string
age = 21        # Variable 'age' stores an integer

# Printing initial values
print("Name:", name)  # Output: Name: Aliya
print("Age:", age)    # Output: Age: 21

# Overwriting variables
name = "Aisha"  # Changing the value of 'name'
age = age + 1   # Incrementing the value of 'age'

# Printing updated values
print("Updated Name:", name)  # Output: Updated Name: Aisha
print("Updated Age:", age)    # Output: Updated Age: 22
```

Key Points

1. **Variable Declaration**:
   - Variables are declared by assigning a value using `=`.
   - Example: `x = 100`.
2. **Overwriting Variables**:
   - Reassigning a new value overwrites the old one.
   - Example: `x = 50` overwrites `x = 100`.
3. **Dynamic Typing**:
   - Variables can change their type when overwritten.
   - Example: `x = 10` (integer) → `x = "Python"` (string).

This flexibility in Python makes working with variables simple and intuitive.

Q6. Identify and explain the three rules for naming variables in Python. (CO1) (BTL 2) (Marks: 5)

• Each naming rule with examples: 5 marks

Q6. Rules for Naming Variables in Python

Variable names in Python must follow specific rules to ensure they are valid and do not cause errors during execution. These rules are:

---

1. Rule 1: Variable names must start with a letter or an underscore (_)

- A variable name cannot begin with a digit. It must start with:
  - A letter (`a` to `z` or `A` to `Z`), or
  - An underscore (`_`).
- **Examples**:

```
name = "Aliya"      # Valid: Starts with a letter
_age = 21           # Valid: Starts with an underscore
1st_place = "Gold"  # Invalid: Starts with a digit
```

---

2. Rule 2: Variable names can only contain letters, digits, and underscores

- Variable names cannot include spaces, special characters (`@`, `#`, `$`, etc.), or symbols (`!`, `?`, `*`, etc.).
- **Examples**:

```
user_name = "Aliya"    # Valid: Contains letters and underscores
user123 = 45           # Valid: Contains letters and digits
```

```
user-name = "Aisha"    # Invalid: Contains a hyphen
user$name = "Riya"     # Invalid: Contains a special character
```

---

3. Rule 3: Variable names are case-sensitive

- Python treats uppercase and lowercase letters as distinct. As a result, `age`, `Age`, and `AGE` are considered different variables.
- **Examples**:

```
name = "Aliya"    # Variable 'name'
Name = "Aisha"    # Different variable 'Name'
NAME = "Riya"     # Different variable 'NAME'
```

Additional Notes

- Variable names should be descriptive and meaningful for better readability. For example:
  - Use `total_marks` instead of `x`.
  - Use `student_name` instead of `a`.
- Avoid using Python **keywords** (like `if`, `else`, `for`, etc.) or built-in function names (like `print`, `len`, etc.) as variable names.

Summary Table

Rule	Valid Example	Invalid Example
Starts with a letter/underscore	`_name`, `age`	`1name`, `9variable`
Contains only letters, digits, underscores	`user_name`, `age123`	`user-name`, `user@age`
Case-sensitive	`age`, `Age`, `AGE`	N/A

By following these rules, you can write clear and error-free Python programs.

Q7. Explain the usage of the str(), int(), and float() functions in Python. Provide examples for each. (CO1) (BTL 2) (Marks: 5)

• Explanation of str() function with example: 2 marks
• Explanation of int() function with example: 2 marks
• Explanation of float() function with example: 1 mark

Q7. Usage of `str()`, `int()`, and `float()` Functions in Python

The functions `str()`, `int()`, and `float()` are used to convert data types in Python. Each function has its own specific purpose and behavior.

---

1. The `str()` Function

- **Purpose**:
  Converts a value into a string (sequence of characters). It is used when non-string data types (like integers or floats) need to be represented as text.
- **Example**:

  ```
  num = 25            # Integer
  text = str(num)      # Convert to string
  print(text)          # Output: "25"
  print(type(text))    # Output: <class 'str'>
  ```

- **Usage**:
  Commonly used in concatenating strings with other data types.

  ```
  age = 21
  message = "I am " + str(age) + " years old."
  print(message)  # Output: I am 21 years old.
  ```

---

2. The `int()` Function

- **Purpose**:
  Converts a value into an integer. It removes any fractional part of a number (truncates) and converts strings to integers if possible.
- **Example**:

  ```
  # Float to integer
  num = 3.14
  whole_number = int(num)
  print(whole_number)   # Output: 3

  # String to integer
  text = "100"
  number = int(text)
  print(number)         # Output: 100
  print(type(number))   # Output: <class 'int'>
  ```

- **Notes**:
  - The string must represent a valid integer, or it will raise a `ValueError`.
  - Example of an invalid conversion:

    ```
    int("3.14")  # Raises ValueError
    ```

---

3. The `float()` Function

- **Purpose**:
  Converts a value into a floating-point number. It is used when precise representation of numbers with decimals is required.

- **Example**:

  ```
  # Integer to float
  num = 5
  decimal = float(num)
  print(decimal)        # Output: 5.0

  # String to float
  text = "3.14"
  number = float(text)
  print(number)         # Output: 3.14
  print(type(number))   # Output: <class 'float'>
  ```

Summary Table

Function	Purpose	Example	Output
`str()`	Converts to string	`str(25)`	`"25"`
`int()`	Converts to integer	`int(3.14)`	`3`
`float()`	Converts to floating-point number	`float("100")`	`100.0`

These functions are fundamental for type conversions and are widely used in Python programming for input handling, data processing, and arithmetic operations.

Q8. Why does the expression 'I have eaten ' + 99 + ' burritos.' cause an error? How can you fix it? (CO1) (BTL 3) (Marks: 5)
• Explanation of the error: 3 marks
• Corrected expression: 2 marks

Q8. Understanding the Error in the Expression

---

1. Explanation of the Error

The expression:

```
'I have eaten ' + 99 + ' burritos.'
```

causes an error because Python does not allow the concatenation of a string and an integer directly.

- **Why?**
  - The `+` operator, when used with strings, performs string concatenation.
  - Here, `'I have eaten '` and `' burritos.'` are strings, but `99` is an integer.
  - Concatenation requires all operands to be strings. Since `99` is not a string, Python raises a `TypeError`.
- **Error Message**:

  ```
  TypeError: can only concatenate str (not "int") to str
  ```

---

2. Corrected Expression

To fix the error, convert the integer `99` into a string using the `str()` function.
The corrected expression is:

```
'I have eaten ' + str(99) + ' burritos.'
```

- **Explanation of Fix**:
  - The `str()` function converts the integer `99` into the string `'99'`.
  - Now, all parts of the expression are strings, allowing successful concatenation.

---

3. Complete Correct Code Example

```
# Corrected expression
message = 'I have eaten ' + str(99) + ' burritos.'
print(message)  # Output: I have eaten 99 burritos.
```

Alternative Solution

You can also use **f-strings** (introduced in Python 3.6) or the `format()` method for more readable and flexible formatting:

```
# Using f-strings
message = f"I have eaten {99} burritos."
print(message)  # Output: I have eaten 99 burritos.

# Using the format() method
message = 'I have eaten {} burritos.'.format(99)
print(message)  # Output: I have eaten 99 burritos.
```

Summary

1. **Cause of Error**: Mismatched data types (`string` and `integer`) during concatenation.
2. **Fix**: Convert the integer to a string using `str()`, or use f-strings/`format()`.
3. **Corrected Output**:

   ```
   I have eaten 99 burritos.
   ```

*Q9. Write a program to evaluate the following expression step-by-step: (5 - 1) * ((7 + 1) / (3 - 1)). Display the result.(CO1) (BTL 4) (Marks: 5)*

• Breaking down the expression step-by-step: 3 marks
• Correct implementation in Python: 2 marks

Q9. Evaluating an Expression Step-by-Step

---

1. Expression to Evaluate

The given expression is:

(5−1)×((7+1)(3−1))(5 - 1) \times \left(\frac{(7 + 1)}{(3 - 1)} \right)

We will evaluate this step-by-step.

---

2. Breaking Down the Expression

1. **Evaluate `(5 - 1)`**:

   5−1=45 - 1 = 4
2. **Evaluate `(7 + 1)`**:

   7+1=87 + 1 = 8
3. **Evaluate `(3 - 1)`**:

   3−1=23 - 1 = 2
4. **Calculate the Division `((7 + 1) / (3 - 1))`**:

   82=4\frac{8}{2} = 4
5. **Multiply `(5 - 1)` by `((7 + 1) / (3 - 1))`**:

   4×4=164 \times 4 = 16

---

3. Python Implementation

```
# Step-by-step evaluation
# Step 1: Calculate (5 - 1)
step1 = 5 - 1

# Step 2: Calculate (7 + 1)
step2 = 7 + 1

# Step 3: Calculate (3 - 1)
step3 = 3 - 1

# Step 4: Calculate ((7 + 1) / (3 - 1))
step4 = step2 / step3

# Step 5: Calculate the final result
result = step1 * step4

# Displaying the result
print("The result of the expression is:", result)
```

---

4. Output

When you run the above program, the output will be:

```
The result of the expression is: 16.0
```

---

5. Key Points

- **Step-by-step Breakdown**: The program calculates each part of the expression in separate steps to improve clarity and debugging.
- **Result**: The final result is `16.0` because Python performs division (`/`) with floating-point precision by default.

Q10. Create a Python program that computes the sum of two integers, multiplies the result by 10, and displays the output.(CO1) (BTL 4) (Marks: 5)
• Program structure and comments: 2 marks
• Correct logic implementation: 2 marks
• Display of output: 1 mark

Q10. Python Program to Compute the Sum of Two Integers, Multiply by 10, and Display the Output

---

Python Code

```
# Step 1: Input two integers from the user
# Using input() function and converting input strings to integers
num1 = int(input("Enter the first integer: "))  # First integer
num2 = int(input("Enter the second integer: "))  # Second integer

# Step 2: Compute the sum of the two integers
sum_result = num1 + num2  # Adding the two integers

# Step 3: Multiply the sum by 10
final_result = sum_result * 10  # Multiplying the sum by 10

# Step 4: Display the final result
print("The result after summing and multiplying by 10 is:", final_result)
```

Program Explanation

1. **Input**:
   - The program asks the user to enter two integers using the `input()` function. These inputs are converted to integers using the `int()` function.
2. **Calculation**:
   - The sum of the two integers is computed.
   - The result of the sum is then multiplied by 10.
3. **Output**:
   - The final result is displayed using the `print()` function.

Example Run

Input:

```
Enter the first integer: 5
Enter the second integer: 3
```

Output:

```
The result after summing and multiplying by 10 is: 80
```

This program demonstrates:

- **Input Handling**: Accepting and converting user input.
- **Arithmetic Operations**: Addition and multiplication.
- **Output Display**: Presenting the result in a clear format.

Chapter 2: Flow Control

Q1. What are the Boolean operators in Python? Explain each with a truth table.(CO1) (BTL 2) (Marks: 5)
• Definition of Boolean operators: 1 mark
• Explanation of and with truth table: 1.5 marks
• Explanation of or with truth table: 1.5 marks
• Explanation of not with truth table: 1 mark

Q1. Boolean Operators in Python

In Python, Boolean operators are used to perform logical operations on boolean values (`True` and `False`). These operators allow you to combine conditional statements or expressions and evaluate them based on specific conditions.

The three main Boolean operators in Python are:

1. **`and`**
2. **`or`**
3. **`not`**

---

1. Definition of Boolean Operators (1 mark)

- **Boolean operators** are logical operators that operate on **Boolean values** (`True` and `False`), producing another Boolean value based on the logical relationships between the operands.
    - **`and`**: Returns `True` if both operands are `True`, otherwise returns `False`.
    - **`or`**: Returns `True` if at least one operand is `True`, otherwise returns `False`.
    - **`not`**: Inverts the value of the operand. If the operand is `True`, it returns `False`; if the operand is `False`, it returns `True`.

---

2. `and` Operator with Truth Table (1.5 marks)

The `and` operator returns `True` only if **both** operands are `True`. If either or both operands are `False`, the result is `False`.

Truth Table for `and` Operator

Operand 1	Operand 2	Operand 1 `and` Operand 2
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

Explanation:

- **`True and True`**: Both are `True`, so the result is `True`.
- **`True and False`**: One is `False`, so the result is `False`.
- **`False and True`**: One is `False`, so the result is `False`.
- **`False and False`**: Both are `False`, so the result is `False`.

---

3. `or` Operator with Truth Table (1.5 marks)

The `or` operator returns `True` if **at least one** operand is `True`. The result is `False` only if both operands are `False`.

Truth Table for `or` Operator

Operand 1	Operand 2	Operand 1 `or` Operand 2
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

Explanation:

- **`True or True`**: At least one is `True`, so the result is `True`.
- **`True or False`**: At least one is `True`, so the result is `True`.
- **`False or True`**: At least one is `True`, so the result is `True`.
- **`False or False`**: Both are `False`, so the result is `False`.

---

4. `not` Operator with Truth Table (1 mark)

The `not` operator inverts the Boolean value of its operand. If the operand is `True`, it returns `False`, and if the operand is `False`, it returns `True`.

Truth Table for `not` Operator

Operand	`not` Operand
`True`	`False`
`False`	`True`

Explanation:

- **`not True`**: The result is `False` because `not` inverts the value.
- **`not False`**: The result is `True` because `not` inverts the value.

Summary

- **`and`**: Returns `True` if both operands are `True`; otherwise, `False`.
- **`or`**: Returns `True` if at least one operand is `True`; otherwise, `False`.

- **`not`**: Inverts the Boolean value of its operand. If `True`, returns `False`; if `False`, returns `True`.

Q2. Differentiate between the == (equal to) and = (assignment) operators in Python.(CO1) (BTL 3) (Marks: 5)
• Explanation of == with example: 2 marks
• Explanation of = with example: 2 marks
• Comparison and usage clarity: 1 mark

Q2. Difference Between `==` (Equal to) and `=` (Assignment) Operators in Python

---

1. Explanation of `==` (Equal to) Operator (2 marks)

The `==` operator is used for **comparison**. It checks whether two values are **equal**. The result of using `==` is a Boolean value: `True` if the values are equal, and `False` if they are not.

Example:

```
x = 5
y = 5
z = 10

# Using the == operator for comparison
print(x == y)  # Output: True, because 5 is equal to 5
print(x == z)  # Output: False, because 5 is not equal to 10
```

- **Explanation**:
  - `x == y` evaluates to `True` because both `x` and `y` are `5`.
  - `x == z` evaluates to `False` because `x` is `5` and `z` is `10`.

---

2. Explanation of `=` (Assignment) Operator (2 marks)

The `=` operator is used for **assignment**. It assigns the value on the right-hand side to the variable on the left-hand side.

Example:

```
x = 5  # Assigning 5 to the variable x
y = 10  # Assigning 10 to the variable y

# Printing the values
print(x)  # Output: 5
print(y)  # Output: 10
```

- **Explanation**:
  - `x = 5` assigns the value `5` to the variable `x`.
  - `y = 10` assigns the value `10` to the variable `y`.

---

3. Comparison and Usage Clarity (1 mark)

- **== (Equal to)**: Used to compare values. It checks whether two expressions or variables hold the same value.
  - **Example**: `x == y` is used to check if `x` and `y` are equal.
- **= (Assignment)**: Used to assign a value to a variable.
  - **Example**: `x = 5` assigns the value `5` to the variable `x`.

Comparison:

- == **checks equality** between two values.
- = **assigns** a value to a variable.

Key Difference:

- == is a **comparison operator** used to compare values, while = is an **assignment operator** used to assign values to variables.

Summary

- **`==`**: Used for checking equality. Example: `x == 5`.
- **`=`**: Used for assigning values. Example: `x = 5`.

Q3. What is a flow control statement? Explain with an example of an if-else structure.(CO1) (BTL 2) (Marks: 5)
• Definition of flow control: 2 marks
• Explanation of if-else structure: 2 marks
• Example with output: 1 mark

Q3. Flow Control Statement in Python

---

1. Definition of Flow Control Statement (2 marks)

A **flow control statement** in programming is used to control the order in which the statements are executed based on certain conditions or logic. These statements help the program to make decisions, repeat tasks, or break out of loops.

Flow control statements include conditional statements like `if`, `else`, `elif`, and loops like `for`, `while`, as well as control statements like `break`, `continue`, and `pass`. They guide the flow of execution, enabling more complex logic and decision-making in a program.

---

2. Explanation of `if-else` Structure (2 marks)

The `if-else` statement is a fundamental flow control statement used to make decisions in Python. It checks whether a condition is **True** or **False** and executes different code blocks depending on the result.

- **`if`**: Executes a block of code if the condition is `True`.
- **`else`**: Executes a block of code if the condition is `False`.

Syntax:

```
if condition:
    # Code to execute if condition is True
else:
    # Code to execute if condition is False
```

---

3. Example with Output (1 mark)

Here is an example of using the `if-else` statement to check whether a number is positive or negative:

```
# Taking user input
num = int(input("Enter a number: "))

# If-else structure to check if the number is positive or negative
if num >= 0:
    print("The number is positive or zero.")
else:
    print("The number is negative.")
```

Explanation:

- The program checks if the number entered by the user is greater than or equal to 0 using the `if` condition.
- If the number is positive or zero, it prints "The number is positive or zero".
- If the number is negative, it prints "The number is negative" using the `else` block.

Example Run:

Input:

```
Enter a number: 5
```

Output:

```
The number is positive or zero.
```

Input:

```
Enter a number: -3
```

Output:

```
The number is negative.
```

Summary

- **Flow control statements** manage the flow of execution in a program based on conditions.
- The **`if-else` structure** allows for decision-making by executing different blocks of code based on whether the condition is `True` or `False`.

Q4. Write a Python program using an elif statement to check and print whether a number is positive, negative, or zero.(CO1) (BTL 4) (Marks: 5)

• Correct elif structure and logic: 2 marks
• Input/output handling: 2 marks
• Code readability: 1 mark

Q4. Python Program to Check if a Number is Positive, Negative, or Zero Using `elif`

Python Program:

```
# Taking user input
num = float(input("Enter a number: "))  # Using float to handle both integers
and floating-point numbers

# Using elif to check the number and print whether it is positive, negative, or
zero
if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Explanation:

1. **Input Handling**:
    - The program uses `input()` to get the user's input. We use `float()` to convert the input into a floating-point number, so it can handle both integers and decimals.
2. **`elif` Structure**:
    - The `if` statement checks if the number is greater than `0` (positive).
    - The `elif` statement checks if the number is less than `0` (negative).
    - The `else` statement captures the case when the number is `0` (neither positive nor negative).
3. **Output Handling**:
    - Based on the value of the input number, the program will print whether it is positive, negative, or zero.

Example Run:

Input 1:

```
Enter a number: 5
```

Output 1:

```
The number is positive.
```

Input 2:

```
Enter a number: -3.7
```

Output 2:

```
The number is negative.
```

Input 3:

```
Enter a number: 0
```

Output 3:

```
The number is zero.
```

Code Readability:

- The program is easy to follow with clear variable names (`num`) and structured logic.
- Each condition is handled using the `if`, `elif`, and `else` statements, making the program flow clear and logical.

Q5. What is the difference between a for loop and a while loop in Python? Illustrate with examples.(CO1) (BTL 2) (Marks: 5)

• Explanation of for loop with example: 2 marks
• Explanation of while loop with example: 2 marks
• Key differences summarized: 1 mark

Q5. Difference Between `for` Loop and `while` Loop in Python

---

1. Explanation of `for` Loop (2 marks)

The `for` loop is used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code for each element in the sequence. It is commonly used when the number of iterations is known beforehand.

Syntax:

```
for variable in sequence:
    # Code to execute for each element in the sequence
```

Example:

```
# Using a for loop to iterate over a list of numbers
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

Output:

```
1
2
3
4
5
```

- **Explanation**:
  The `for` loop iterates over each number in the list `numbers` and prints it.

---

2. Explanation of `while` Loop (2 marks)

The `while` loop executes a block of code as long as a specified condition remains `True`. It is commonly used when the number of iterations is not known in advance and depends on a condition.

Syntax:

```
while condition:
    # Code to execute as long as condition is True
```

Example:

```
# Using a while loop to print numbers from 1 to 5
num = 1
while num <= 5:
    print(num)
    num += 1  # Increment num to avoid infinite loop
```

Output:

```
1
2
3
4
5
```

- **Explanation**:
  The `while` loop continues to execute the block of code as long as `num <= 5`. After printing the number, `num` is incremented by 1.

---

3. Key Differences Between `for` and `while` Loops (1 mark)

Aspect	`for` Loop	`while` Loop
Control Structure	Iterates over a sequence (like a list, range, etc.)	Continues until a condition is no longer true

Aspect	`for` Loop	`while` Loop
Use Case	When the number of iterations is known or fixed	When the number of iterations is unknown or based on a condition
Syntax	`for variable in sequence:`	`while condition:`
Example	Iterating over a list or range	Repeating a block of code as long as a condition holds

---

Summary:

- **`for` loop**: Best used when the number of iterations is known or when iterating over a collection.
- **`while` loop**: Best used when the number of iterations is unknown, and the loop continues as long as a condition is true.

Q6. Explain the usage of the break and continue statements in loops. Provide examples for both.(CO1) (BTL 2) (Marks: 5)
• Definition and usage of break with example: 2.5 marks
• Definition and usage of continue with example: 2.5 marks

Q6. Usage of `break` and `continue` Statements in Loops

---

1. Definition and Usage of `break` with Example (2.5 marks)

The `break` statement is used to **exit a loop** prematurely, meaning the loop stops executing as soon as the `break` statement is encountered, regardless of whether the loop's condition is true or if there are more iterations left.

- **Use Case**: It's typically used when you want to exit the loop as soon as a certain condition is met (e.g., finding a specific item in a list).

Example:

```
# Using break in a loop
for num in range(1, 10):
    if num == 5:
        print("Number 5 found, breaking the loop.")
        break  # Exit the loop when num equals 5
    print(num)
```

Output:

```
1
2
3
4
Number 5 found, breaking the loop.
```

- **Explanation**:
  The `for` loop iterates through numbers from 1 to 9. When the number reaches 5, the `if` condition becomes true, and the `break` statement is executed, which stops the loop, and no further numbers are printed.

---

2. Definition and Usage of `continue` with Example (2.5 marks)

The `continue` statement is used to **skip the current iteration** of the loop and proceed to the next iteration. Unlike `break`, which terminates the loop, `continue` only skips the current iteration and continues executing the loop with the next value.

- **Use Case**: It's useful when you want to skip certain steps or values in a loop based on a condition but still want the loop to continue.

Example:

```
# Using continue in a loop
for num in range(1, 10):
    if num == 5:
        print("Skipping number 5")
        continue  # Skip the current iteration when num equals 5
    print(num)
```

Output:

```
1
2
3
4
Skipping number 5
6
7
8
9
```

- **Explanation**:
  The `for` loop iterates through numbers from 1 to 9. When the number equals 5, the `continue` statement is executed, causing the loop to skip printing the number 5 but continue with the next iteration.

Summary:

- **`break`**: Exits the loop entirely, stopping further iterations.
- **`continue`**: Skips the current iteration and moves to the next iteration of the loop without terminating the loop.

Q7. What are blocks in Python? Identify the blocks in the following code: (CO1) (BTL 3) (Marks: 5)
python Copy code

```
spam = 0
if spam == 10:
  print('eggs')
  if spam > 5:
     print('bacon')
  else:
     print('ham')
  print('spam')
print('spam')
```

• Explanation of blocks: 2 marks
• Identification of blocks in the code: 3 marks

Q7. Explanation of Blocks and Identification in the Given Code

---

1. Explanation of Blocks (2 marks)

In Python, a block is a set of statements that are grouped together and executed as a single unit. Blocks are defined by indentation level; each block of code is indented to show that it is part of the larger statement or structure (like an `if` statement, `for` loop, or function definition). This indentation is required in Python to indicate the scope of the block.

- **Definition**: A block in Python refers to a group of one or more statements that are written together and have a common indentation level. The indentation level determines the nesting and scope of the block.

2. Identification of Blocks in the Given Code (3 marks)

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

Explanation of the Code Blocks:

1. **First Block (Line 3 - `if spam == 10:`):**
    - **Indentation**: Indented by one level (4 spaces or one tab).
    - **Statements**: `print('eggs')`, `if spam > 5:`, and `else:` are included within this block.

    - **Purpose**: This block checks if the variable `spam` is equal to 10. If it is, it prints `'eggs'`.

2. **Second Block (Line 5 - `if spam > 5:`):**
    - **Indentation**: Indented by two levels (8 spaces or two tabs).
    - **Statements**: `print('bacon')` is the statement under this block.
    - **Purpose**: This block checks if `spam` is greater than 5. If `spam` is greater than 5, it prints `'bacon'`.
3. **Third Block (Line 7 - `else:`):**
    - **Indentation**: Indented by two levels (8 spaces or two tabs).
    - **Statements**: `print('ham')` is the statement under this block.
    - **Purpose**: This block is executed if `spam` is not greater than 5. It prints `'ham'`.
4. **Outside of Blocks (Line 9 - `print('spam')`):**
    - **Indentation**: Not indented (zero indentation).
    - **Purpose**: This statement is outside any block and is always executed regardless of the conditions in the `if-else` statements.

Summary:

- **Blocks in Python** are defined by indentation. Each block of code is indented to the same level and is executed as a unit.
- In the provided code:
    - `spam == 10` block contains statements that print `'eggs'`, check if `spam > 5`, and handle the `else` condition.
    - `spam > 5` block prints `'bacon'` if `spam` is greater than 5.
    - `else` block prints `'ham'` if `spam` is not greater than 5.
- The statement `print('spam')` outside any block is always executed.

Q8. Write a program to print the numbers 1 to 10 using a for loop. Then rewrite the same using a while loop. (CO1) (BTL 4) (Marks: 5)
• Correct implementation of for loop: 2 marks
• Correct implementation of while loop: 2 marks
• Equivalence of both outputs: 1 mark

Q8. Program to Print Numbers from 1 to 10 Using `for` Loop and `while` Loop

---

1. Correct Implementation of `for` Loop (2 marks)

In Python, a `for` loop can be used to iterate through a sequence of numbers. The `range()` function is often used to generate a sequence of numbers.

For Loop Implementation:

```
# Using a for loop to print numbers from 1 to 10
for num in range(1, 11):
    print(num)
```

Explanation:

- The `range(1, 11)` generates numbers from 1 to 10.
- The loop iterates through these numbers and prints each number.

---

2. Correct Implementation of `while` Loop (2 marks)

A `while` loop runs as long as the specified condition is `True`. In this case, we will start from 1 and continue until the number reaches 10.

While Loop Implementation:

```
# Using a while loop to print numbers from 1 to 10
num = 1
while num <= 10:
    print(num)
    num += 1  # Increment num by 1 after each iteration
```

Explanation:

- The loop starts with `num = 1` and runs until `num` becomes greater than 10.
- After each iteration, `num` is incremented by 1.

---

3. Equivalence of Both Outputs (1 mark)

Both the `for` loop and the `while` loop will produce the same output, printing numbers from 1 to 10:

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Explanation:

- Both loops print the numbers 1 to 10 in the same order.
- The `for` loop uses a sequence generated by `range()`, while the `while` loop manually increments the variable `num` until it reaches 10.

Q9. Explain the functionality of the range() function in Python. Differentiate between range(10), range(0, 10), and range(0, 10, 1) with examples.(CO1) (BTL 3) (Marks: 5)

• Explanation of range() function: 2 marks
• Differences between variants with examples: 3 marks

Q9. Explanation of `range()` Function and Differences Between Variants

---

1. Explanation of the `range()` Function (2 marks)

The `range()` function in Python is used to generate a sequence of numbers, commonly used in loops such as `for` loops to iterate over a specific range of numbers. The function can take up to three arguments:

- **Syntax**: `range(start, stop, step)`
  - **start** (optional): The value from which the sequence starts (default is 0).
  - **stop** (required): The value at which the sequence ends (this value is not included in the sequence).
  - **step** (optional): The increment between each number in the sequence (default is 1).

The `range()` function does not generate a list directly but instead returns an iterable object, which can be converted into a list using `list()`. It is commonly used for iteration in loops.

---

2. Differences Between Variants with Examples (3 marks)

a. `range(10)`:

- **Explanation**: This creates a sequence of numbers starting from 0 up to, but not including, 10.
- **Syntax**: `range(10)` is equivalent to `range(0, 10, 1)`.
- **Output**: It generates numbers from 0 to 9.

Example:

```
for num in range(10):
    print(num)
```

Output:

```
0
1
2
```

```
3
4
5
6
7
8
9
```

- **Explanation**: The `start` is 0 (default), the `stop` is 10, and the `step` is 1 (default).

---

b. `range(0, 10)`:

- **Explanation**: This is similar to `range(10)`, as it also starts from 0 and ends at 10, not including 10.
- **Syntax**: `range(0, 10)` explicitly specifies the starting point (`0`), the ending point (`10`), and the step (`1`, which is default).
- **Output**: It generates numbers from 0 to 9.

Example:

```
for num in range(0, 10):
    print(num)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

- **Explanation**: The sequence starts from `0` (explicitly defined), ends at `10` (but does not include `10`), and the default step is `1`.

---

c. `range(0, 10, 1)`:

- **Explanation**: This explicitly defines all three parameters. The `start` is 0, the `stop` is 10, and the `step` is 1, which is the default increment between numbers.
- **Syntax**: `range(0, 10, 1)` behaves the same as `range(10)` and `range(0, 10)` but explicitly shows all parameters.
- **Output**: It generates numbers from 0 to 9.

Example:

```
for num in range(0, 10, 1):
    print(num)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

- **Explanation**: The sequence starts from `0`, stops before `10`, and increments by `1`. This is the same as the other two variants, but with all arguments specified.

Summary of Differences:

Variant	Start	Stop	Step	Output
`range(10)`	0	10	1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`range(0, 10)`	0	10	1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`range(0, 10, 1)`	0	10	1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- All three variants will generate the same sequence: 0 through 9.
- The differences lie in how the arguments are specified, but the output remains the same.

Q10. What are the three Boolean operators? Write the truth tables for each and give examples of their usage in Python.(CO1) (BTL 3) (Marks: 5)

• List of Boolean operators: 1 mark
• Truth tables for and, or, and not: 3 marks
• Example for each operator: 1 mark

Q10. The Three Boolean Operators in Python

---

1. List of Boolean Operators (1 mark)

The three Boolean operators in Python are:

- **and**
- **or**
- **not**

These operators are used to combine or negate conditions, typically in `if` statements or loops.

---

2. Truth Tables for Each Boolean Operator (3 marks)

a. `and` Operator:

The `and` operator returns `True` only if **both** conditions are `True`. Otherwise, it returns `False`.

Condition 1	Condition 2	Condition 1 and Condition 2
True	True	True
True	False	False
False	True	False
False	False	False

b. `or` Operator:

The `or` operator returns `True` if **at least one** of the conditions is `True`. It only returns `False` when both conditions are `False`.

Condition 1	Condition 2	Condition 1 or Condition 2
True	True	True
True	False	True
False	True	True
False	False	False

c. `not` Operator:

The `not` operator is a negation operator. It inverts the value of a Boolean expression (i.e., `True` becomes `False`, and `False` becomes `True`).

Condition	not Condition
True	False
False	True

---

3. Example Usage for Each Operator (1 mark)

a. `and` Example:

```
a = 5
b = 10
if a > 3 and b > 5:
    print("Both conditions are true")
```

Explanation: Both conditions `a > 3` and `b > 5` are `True`, so the output will be:

```
Both conditions are true
```

b. `or` Example:

```
a = 5
b = 3
if a > 3 or b > 5:
    print("At least one condition is true")
```

Explanation: `a > 3` is `True`, so the `or` condition evaluates to `True`, and the output will be:

```
At least one condition is true
```

c. `not` Example:

```
a = True
if not a:
    print("a is False")
else:
    print("a is True")
```

Explanation: Since `a` is `True`, the `not` operator negates it to `False`, and the output will be:

```
a is True
```

Summary:

- **`and`**: Returns `True` if both conditions are `True`.
- **`or`**: Returns `True` if at least one condition is `True`.
- **`not`**: Inverts the Boolean value (i.e., `True` becomes `False` and vice versa).

Q11. Write a program that keeps asking the user to type "Python" until they do so. Use a while loop for this. (CO1) (BTL 4) (Marks: 5)
• Correct loop structure: 3 marks
• Proper input and output handling: 2 marks

Q11. Python Program to Keep Asking for "Python" Using a While Loop

---

Program Structure and Explanation:

The program will continuously ask the user to type the word "Python" until they enter it correctly. This can be done using a `while` loop that runs as long as the input is not "Python".

Code Implementation:

```
# Initialize a variable to store user input
user_input = ""

# Start a while loop that continues as long as the user hasn't typed "Python"
while user_input != "Python":
    # Ask the user for input
    user_input = input("Please type 'Python': ")

    # Check if the input is correct
    if user_input != "Python":
        print("That's not right. Try again!")  # Provide feedback if input is
wrong

# Once the user types "Python", the loop will exit and print a success message
print("Correct! You typed 'Python'.")
```

Explanation of the Code:

- **Correct loop structure**: The `while` loop is used to keep asking the user for input until they type "Python".
- **Input handling**: The `input()` function is used to take user input, and it's compared with the string "Python".
- **Output handling**: If the input is incorrect, the program gives feedback and prompts the user to try again. Once the user types "Python", the program prints a success message and exits the loop.

Sample Output:

```
Please type 'Python': Java
That's not right. Try again!
Please type 'Python': python
That's not right. Try again!
Please type 'Python': Python
Correct! You typed 'Python'.
```

This program will continue to ask for "Python" until the correct input is provided, demonstrating a proper use of the `while` loop.

Q12. What is the purpose of the sys.exit() function? Write a Python program to demonstrate its usage. (CO1) (BTL 3) (Marks: 5)
• Explanation of sys.exit() function: 2 marks
• Program demonstrating its use: 3 marks

Q12. Purpose of `sys.exit()` and Demonstration in Python

---

1. Explanation of `sys.exit()` Function (2 marks)

The `sys.exit()` function in Python is used to exit from a program. It is part of the `sys` module, so it requires importing `sys` to use it. When called, `sys.exit()` raises a `SystemExit` exception that terminates the current program. You can optionally pass an argument to `sys.exit()` to indicate an exit status. By default, it exits with a status of `0`, which generally means the program terminated successfully. A non-zero exit status usually indicates an error.

- **Syntax**:

  ```
  sys.exit([status])
  ```

  - **status**: This is an optional argument. A value of `0` indicates successful termination, and any non-zero value indicates an error or abnormal termination.

---

2. Program Demonstrating the Usage of `sys.exit()` (3 marks)

```
import sys

def check_age(age):
    if age < 18:
        print("You must be 18 or older to continue.")
        sys.exit("Exiting program due to age restriction.")  # Exit the program 
with a message
    else:
        print("Access granted! You are eligible to proceed.")

# Asking the user for input
age = int(input("Enter your age: "))

# Call the function to check age
check_age(age)

# This code will not run if age is less than 18 because sys.exit() will 
terminate the program
print("This message won't be printed if the program exits due to age 
restriction.")
```

Explanation of the Program:

- **`sys.exit()` Usage**: In the program, the function `check_age()` checks if the user's age is less than 18. If the age is under 18, the program will print a message and call `sys.exit()` to terminate the program immediately.
- **Input and Output**: The user is prompted to enter their age. If the entered age is below 18, the program exits and displays a message. If the age is 18 or above, the program grants access.
- **Flow Control**: If the user enters a valid age, the program proceeds normally. However, if the age is invalid (less than 18), the program exits and doesn't print further messages.

Sample Output 1 (Age below 18):

```
Enter your age: 15
You must be 18 or older to continue.
Exiting program due to age restriction.
```

Sample Output 2 (Age 18 or above):

```
Enter your age: 20
Access granted! You are eligible to proceed.
This message won't be printed if the program exits due to age restriction.
```

Summary:

- `sys.exit()` is used to terminate the program at any point during execution.
- It is particularly useful when a program encounters an error or a condition that warrants halting further execution, such as user input validation in this example.

Chapter 3: Functions

Q1. What is a function in Python? Explain the advantages of using functions in programming.(CO1) (BTL 2) (Marks: 5)
• Definition of a function: 2 marks
• Advantages of functions (code reuse, modularity, etc.): 3 marks

Q1. What is a function in Python? Explain the advantages of using functions in programming.

---

1. Definition of a Function (2 marks)

A **function** in Python is a block of reusable code that performs a specific task. Functions help break down complex problems into smaller, manageable pieces. Functions can take input (known as parameters or arguments), process it, and return a result.

- **Syntax**:

  ```
  def function_name(parameters):
      # Code to execute
      return result
  ```

  - `def` is used to define a function.
  - `function_name` is the name of the function.
  - `parameters` (optional) are inputs that the function can accept.
  - `return` is used to send back the result.

Example:

```
def greet(name):
    return f"Hello, {name}!"

# Calling the function
print(greet("Aliya"))
```

Output:

```
Hello, Aliya!
```

---

2. Advantages of Using Functions in Programming (3 marks)

Functions offer several benefits in programming, including:

a. Code Reusability (1 mark)

Once a function is defined, it can be reused multiple times throughout the program. This eliminates the need to write the same code over and over, making the code more efficient and reducing errors.

Example:

```
def square(x):
    return x * x

# Reusing the function
print(square(3))  # Output: 9
print(square(5))  # Output: 25
```

In this case, the `square()` function is defined once and used multiple times for different values.

b. Modularity (1 mark)

Functions allow for dividing the program into smaller, more manageable parts. Each function performs a specific task, making the code easier to understand, maintain, and debug.

- You can test and update individual functions without affecting the rest of the program.
- It promotes a **modular** structure where each piece of functionality is encapsulated within a function.

c. Improved Readability and Maintainability (1 mark)

Using functions can make the program more readable because they help break down complex logic into smaller, meaningful components. Functions also enhance maintainability since you can make changes to a specific function without altering the rest of the program.

- For example, if you have a function to calculate tax, you can modify it independently without changing the entire program structure.

Example:

```
def calculate_tax(amount, tax_rate):
    return amount * tax_rate

def calculate_total(amount, tax_rate):
    tax = calculate_tax(amount, tax_rate)
    return amount + tax

# Calling the function
print(calculate_total(100, 0.05))  # Output: 105.0
```

In this case, we have two functions that handle separate tasks, making the code more structured and easier to maintain.

Summary:

- A **function** is a reusable block of code that performs a specific task and can accept inputs and return outputs.
- **Advantages** of using functions include:
  - **Code reusability**: Functions can be called multiple times, reducing repetition.
  - **Modularity**: Functions break the program into smaller, independent modules.
  - **Improved readability and maintainability**: Functions make code easier to understand and modify.

Q2. Differentiate between defining a function and calling a function in Python.(CO1) (BTL 2) (Marks: 5)
• Explanation of function definition: 2 marks
• Explanation of function calling: 2 marks
• Key differences highlighted: 1 mark

Q2. Differentiate Between Defining a Function and Calling a Function in Python

---

1. Explanation of Function Definition (2 marks)

In Python, **defining a function** means creating a function by specifying its name, parameters (optional), and the block of code that it will execute when called. The function is defined using the `def` keyword followed by the function name and parentheses that may include parameters. The function body contains the code that defines the task the function will perform.

- **Syntax**:

  ```
  def function_name(parameters):
      # Code block
      return result  # (optional)
  ```

Example:

```
def greet(name):  # Function definition
    return f"Hello, {name}!"  # Function body that returns a greeting
```

In this example, the function `greet` is defined with a parameter `name`, which will be used to generate a greeting message. The `def` keyword initiates the function definition.

---

2. Explanation of Function Calling (2 marks)

Calling a function means invoking or executing the function that has been defined. When a function is called, Python jumps to the function's definition, executes the code inside it, and optionally returns a result to where it was called.

- **Syntax**:

  ```
  function_name(arguments)
  ```

Example:

```
result = greet("Aliya")  # Function calling
print(result)
```

In this example, the function `greet("Aliya")` is called with the argument `"Aliya"`. Python runs the code inside the `greet` function and returns the greeting message, which is then printed.

---

3. Key Differences Between Defining and Calling a Function (1 mark)

Aspect	Defining a Function	Calling a Function
Purpose	To create a reusable block of code.	To execute or invoke the function to perform its task.
Syntax	Uses the `def` keyword to define the function.	Calls the function by its name followed by parentheses.
Occurs Once	Happens once during the program when the function is defined.	Happens multiple times during the program whenever the function is needed.
Execution	Does not execute; only stores the code for later use.	Executes the function's code and may return a result.

Summary:

- **Function Definition**: The process of creating a function with the `def` keyword, which includes its name, parameters, and body.
- **Function Calling**: The act of invoking a function by using its name and passing any necessary arguments to execute the code within it.

The main difference is that defining a function sets up the functionality, while calling a function actually runs the defined code.

Q3. Write a Python function to check if a number is even or odd and print the result.(CO1) (BTL 4) (Marks: *5)*
• Correct function definition: 2 marks
• Logic to determine even or odd: 2 marks
• Output message clarity: 1 mark

Q3. Write a Python function to check if a number is even or odd and print the result.

Solution:

Here’s a Python function to check if a number is even or odd:

Function Definition (2 marks):

The function will take a number as input and determine whether it is even or odd by using the modulo operator `%`.

- If a number is divisible by 2 (i.e., the remainder when divided by 2 is 0), it is even.
- If a number is not divisible by 2 (i.e., the remainder is 1), it is odd.

Python Code:

```
def check_even_odd(number):  # Function definition
    if number % 2 == 0:  # Logic to check even
```

```
        print(f"{number} is even.")  # Output for even
    else:  # Logic for odd
        print(f"{number} is odd.")  # Output for odd
```

Explanation of Logic (2 marks):

- The `number % 2` expression checks the remainder when the number is divided by 2.
    - If the remainder is `0`, the number is even.
    - If the remainder is `1`, the number is odd.

Output Message Clarity (1 mark):

The function prints clear messages indicating whether the number is even or odd.

Example of Calling the Function:

```
check_even_odd(4)  # Output: 4 is even.
check_even_odd(7)  # Output: 7 is odd.
```

Output:

```
4 is even.
7 is odd.
```

Summary:

- The function `check_even_odd(number)` checks whether a number is even or odd.
- It uses the modulo operator `%` to determine the remainder when divided by 2.
- The output message clearly states if the number is even or odd.

Q4. What is the difference between local and global variables in Python? Illustrate with examples.(CO1) (BTL 3) (Marks: 5)
• Definition of local and global variables: 2 marks
• Example illustrating local and global variables: 3 marks

Q4. What is the difference between local and global variables in Python? Illustrate with examples.

---

1. Definition of Local and Global Variables (2 marks)

- **Local Variables**: A **local variable** is defined within a function or block and is accessible only within that function. It is created when the function is called and destroyed when the function finishes execution. Local variables cannot be accessed outside their function.

- **Global Variables**: A **global variable** is defined outside any function, typically at the top of the program. It can be accessed and modified by any function within the same program. Global variables persist for the lifetime of the program.

---

2. Example Illustrating Local and Global Variables (3 marks)

Example 1: Local Variable

```
def my_function():
    local_var = 10  # This is a local variable
    print("Inside the function, local_var =", local_var)

my_function()  # Output: Inside the function, local_var = 10

# Trying to access local_var outside the function will raise an error
# print(local_var)  # This would raise a NameError
```

- In this example, `local_var` is defined inside the function `my_function`. It can only be accessed within the function. Trying to access it outside of the function will result in a **NameError** because it is a local variable.

Example 2: Global Variable

```
global_var = 20  # This is a global variable

def another_function():
    print("Inside the function, global_var =", global_var)

another_function()  # Output: Inside the function, global_var = 20

# Accessing the global variable outside any function
print("Outside the function, global_var =", global_var)  # Output: Outside the
function, global_var = 20
```

- In this example, `global_var` is defined outside of any function, making it a global variable. It can be accessed both inside the function `another_function()` and outside it.

Example 3: Modifying Global Variables Inside a Function

```
global_var = 20  # Global variable

def modify_global():
    global global_var  # Declare the global variable to modify it
    global_var = 30  # Modify the global variable

modify_global()
print("Modified global_var =", global_var)  # Output: Modified global_var = 30
```

- In this example, the `global` keyword is used to modify the value of a global variable from within a function. Without the `global` keyword, the variable would be treated as a local variable, and the change would not persist outside the function.

---

3. Key Differences Between Local and Global Variables

Aspect	Local Variables	Global Variables
Scope	Limited to the function or block where defined.	Accessible throughout the entire program.
Lifetime	Created when the function is called and destroyed when the function ends.	Exists for the lifetime of the program.
Access	Can only be accessed inside the function.	Can be accessed and modified by any function in the program.
Memory	Takes up memory only while the function is running.	Occupies memory throughout the program's execution.

Summary:

- **Local Variables** are confined to the function or block where they are defined and cannot be accessed outside that scope.
- **Global Variables** are accessible throughout the program and can be modified by any function, making them persistent across function calls.

Q5. Write a Python program using a function to calculate the factorial of a given number.(CO1) (BTL 4) (Marks: 5)

• Function definition and structure: 2 marks
• Correct logic for factorial calculation: 2 marks
• Output format: 1 mark

Q5. Write a Python program using a function to calculate the factorial of a given number.

Solution:

To calculate the factorial of a given number, we will define a function that uses recursion or an iterative approach to compute the result. The factorial of a non-negative integer `n` is the product of all positive integers less than or equal to `n`, and it is denoted as `n!`. Specifically:

- `n! = n * (n-1) * (n-2) * ... * 1`
- Special case: `0! = 1`

Python Code:

```
# Function to calculate the factorial of a number
def calculate_factorial(n):  # Function definition
    if n == 0:  # Base case: factorial of 0 is 1
        return 1
    else:
        return n * calculate_factorial(n - 1)  # Recursive call
```

```
# Input from the user
number = int(input("Enter a number to calculate its factorial: "))

# Calculating and displaying the factorial
factorial = calculate_factorial(number)
print(f"The factorial of {number} is {factorial}.")  # Output format
```

Explanation of the Code:

- **Function Definition and Structure (2 marks)**:
  - The function `calculate_factorial(n)` is defined to take one parameter `n`, which is the number for which we need to calculate the factorial.
  - If `n` is 0, it returns `1` because `0! = 1` (base case).
  - For any number greater than 0, it recursively calls itself with `n-1`, multiplying the result by `n`.
- **Correct Logic for Factorial Calculation (2 marks)**:
  - The logic works through recursion: the factorial of `n` is `n * (n-1)!`. The base case ensures that the recursion stops when `n` reaches 0, which returns 1.
- **Output Format (1 mark)**:
  - The program prints the factorial in the format: `The factorial of {number} is {factorial}.`

Example Output:

```
Enter a number to calculate its factorial: 5
The factorial of 5 is 120.
```

For `5!`, the calculation steps would be:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Summary:

- The program defines a recursive function `calculate_factorial(n)` to compute the factorial of a given number.
- It prompts the user for input and displays the result in a clear and correct format.

Q6. Explain the purpose of the return statement in Python. How is it different from print()?(CO1) (BTL 2) (Marks: 5)

• Definition and purpose of return: 2 marks
• Comparison with print(): 2 marks
• Example highlighting the difference: 1 mark

Q6. Explain the purpose of the return statement in Python. How is it different from print()?

---

1. Definition and Purpose of `return` (2 marks)

- The `return` statement in Python is used to **exit a function** and **return a value** to the caller. When a function executes a `return` statement, it immediately ends the function and passes the specified value back to the place where the function was called.
- **Purpose of `return`**:
    - To provide a value from a function to the calling code (e.g., the main program or another function).
    - To end the execution of a function early and pass control back to the caller.

  Syntax:

  ```
  return value
  ```

---

2. Comparison with `print()` (2 marks)

- **`return`**:
    - Returns a value from the function to the caller, allowing it to be used in further operations.
    - The returned value can be stored in a variable or used in expressions.
    - It ends the function's execution and passes control back to the caller.
- **`print()`**:
    - Outputs a value to the console or standard output. It does **not return any value** from the function.
    - It is primarily used for displaying information to the user or debugging.
    - Does **not affect** the flow of the function; the function continues executing after a `print()` statement.

---

3. Example Highlighting the Difference (1 mark)

```
# Example using return
def add_numbers(x, y):
    return x + y  # returns the sum

result = add_numbers(5, 3)
print(result)  # Output: 8

# Example using print()
def add_and_print(x, y):
    print(x + y)  # prints the sum, but doesn't return anything

add_and_print(5, 3)  # Output: 8
```

Explanation:

- In the first example, the function `add_numbers()` returns the sum, and the result is stored in a variable `result` and then printed. The returned value can be used in further calculations.
- In the second example, `add_and_print()` prints the sum directly, but does not return any value, so we cannot store or reuse the sum for further calculations.

Summary:

- **`return`** is used to return a value from a function, which can be stored or used further in the program. It also ends the function's execution.
- **`print()`** is used to display information to the user or console but does not return any value and does not affect the flow of the function.

Q7. What is the purpose of the try and except statements in Python? Write a program to handle a division by zero error.(CO1) (BTL 3) (Marks: 5)

• Explanation of try and except: 2 marks

• Program implementation with error handling: 3 marks

Q7. What is the purpose of the try and except statements in Python? Write a program to handle a division by zero error.

---

1. Explanation of `try` and `except` (2 marks)

- **Purpose of `try` and `except`**:
  - The `try` and `except` statements in Python are used for **exception handling**. These statements allow you to handle errors (exceptions) that may occur during the execution of the program, preventing the program from crashing.
  - The `try` block contains the code that might cause an exception. If an exception occurs, the program immediately moves to the corresponding `except` block where the error is handled.

  Syntax:

  ```
  try:
      # Code that may raise an exception
  except SomeException:
      # Code that handles the exception
  ```

- **How It Works**:
  - The code inside the `try` block is executed first.
  - If no error occurs, the `except` block is skipped.

    - If an error occurs, the `except` block catches the exception and allows you to handle it gracefully, such as printing an error message, logging the error, or even continuing the program.

---

2. Program Implementation with Error Handling (3 marks)

We will write a Python program that asks the user for two numbers and attempts to divide them. If the user enters `0` as the denominator, the program will handle the "division by zero" error using `try` and `except`.

```
# Program to handle division by zero error

# Function to perform division
def divide_numbers():
    try:
        # Input from the user
        num1 = float(input("Enter the numerator: "))
        num2 = float(input("Enter the denominator: "))

        # Division operation
        result = num1 / num2
        print(f"The result of {num1} divided by {num2} is: {result}")

    except ZeroDivisionError:
        # Handling division by zero error
        print("Error: Cannot divide by zero!")
    except ValueError:
        # Handling invalid input error (non-numeric input)
        print("Error: Please enter valid numbers.")

# Call the function
divide_numbers()
```

Explanation of the Program:

- **`try` block**:
    - This contains the code for taking input from the user and performing the division. The program tries to execute the division operation `num1 / num2`.
- **`except ZeroDivisionError`**:
    - This block handles the case where the user tries to divide by zero. If `num2` is `0`, a `ZeroDivisionError` is raised, and the message "Error: Cannot divide by zero!" is printed.
- **`except ValueError`**:
    - This block catches cases where the user enters non-numeric values (e.g., letters or symbols). A `ValueError` will be raised if the input cannot be converted to a float, and the program prints an error message "Error: Please enter valid numbers."

Example Output:

1. **Valid input**:

```
Enter the numerator: 10
Enter the denominator: 2
The result of 10.0 divided by 2.0 is: 5.0
```

2. **Division by zero**:

```
Enter the numerator: 10
Enter the denominator: 0
Error: Cannot divide by zero!
```

3. **Invalid input**:

```
Enter the numerator: 10
Enter the denominator: abc
Error: Please enter valid numbers.
```

Summary:

- The `try` block is used to write code that may cause an error.
- The `except` block is used to handle errors if they occur. In the example, it catches the "division by zero" error and the "invalid input" error.
- The program continues to run smoothly even when an error occurs, instead of crashing.

Q8. What are the default return value of a function in Python and the data type of None?(CO1) (BTL 2) (Marks: 5)

• Default return value explanation: 2 marks
• Explanation of None and its type: 2 marks
• Example illustrating the concept: 1 mark

Q8. What are the default return value of a function in Python and the data type of None?

---

1. Default Return Value Explanation (2 marks)

- In Python, if a function does not explicitly return a value using the `return` statement, it **implicitly returns `None`** by default. This means that if you call a function that has no `return` statement, Python automatically returns `None`.

  Example:

  ```
  def my_function():
      print("Hello, world!")

  result = my_function()  # The function doesn't return anything
  print(result)  # This will print "None" since the function returns nothing
  ```

  - In this example, `my_function()` does not have a `return` statement, so when you call the function, Python returns `None` by default.

---

2. Explanation of `None` and Its Type (2 marks)

- `None` is a special constant in Python used to represent the absence of a value or a null value. It is often used to indicate that a function does not return anything or that a variable is not yet assigned a value.
- The data type of `None` is **`NoneType`**. It is the only value of this type.

  Example:

  ```
  x = None
  print(type(x))  # Output: <class 'NoneType'>
  ```

  - In this case, `None` is assigned to the variable `x`, and when you check its type using `type(x)`, it returns `<class 'NoneType'>`.

---

3. Example Illustrating the Concept (1 mark)

Here's an example program that demonstrates the default return value (`None`) and the type of `None`:

```
# Function without a return statement
def greet():
    print("Hello, user!")

# Calling the function
result = greet()  # This function does not return anything
print("Return value of greet():", result)  # This will print "None"

# Checking the type of None
none_value = None
print("Type of None:", type(none_value))  # This will print "<class 'NoneType'>"
```

Explanation:

- The function `greet()` does not have a `return` statement, so calling it returns `None` by default. This is printed when we print `result`.
- The `type()` function is used to show that the type of `None` is `NoneType`.

Summary:

- **Default return value**: If no return statement is provided, a Python function returns `None` by default.
- **Type of `None`**: The type of `None` is `NoneType`, which is a special type in Python.

Q9. Write a program to demonstrate the use of keyword arguments in a function.(CO1) (BTL 4) (Marks: 5)

• Correct function definition and structure: 2 marks
• Keyword arguments used effectively: 2 marks
• Program output clarity: 1 mark

Q9. Write a program to demonstrate the use of keyword arguments in a function.

Program Explanation:

In Python, **keyword arguments** allow us to pass arguments to a function by explicitly specifying the parameter names along with their values. This improves readability and allows the arguments to be passed in any order.

Program Code:

```
# Function demonstrating keyword arguments
def greet(name, age, city):
    print(f"Hello {name}! You are {age} years old and live in {city}.")

# Using keyword arguments to call the function
greet(age=25, name="Alice", city="New York")
```

Explanation:

- The function `greet` takes three parameters: `name`, `age`, and `city`.
- When calling the function, we use **keyword arguments** to specify the values for these parameters: `age=25`, `name="Alice"`, and `city="New York"`.
    - The arguments are passed in a different order than they are defined in the function, but since we are using the keyword argument syntax, it still works correctly.
- The `print` statement inside the function outputs a message using these values.

Expected Output:

```
Hello Alice! You are 25 years old and live in New York.
```

Program Structure:

- **Function definition**: The function `greet` is defined with three parameters.
- **Keyword arguments**: During the function call, the values are passed using their parameter names (`age`, `name`, `city`), which ensures clarity and flexibility.
- **Output clarity**: The output clearly displays the personalized message based on the provided arguments.

Key Points:

- **Keyword arguments** make the function calls more readable and flexible, especially when a function has multiple parameters.
- They allow us to pass arguments in any order and provide default values if needed.

Q10. What are the four rules for determining if a variable is in the global or local scope?(CO1) (BTL 2) (Marks: 5)

• List of all four rules: 5 marks

Q10. What are the four rules for determining if a variable is in the global or local scope?

In Python, the scope of a variable refers to where it is accessible. A variable can either be in the **global scope** (accessible throughout the program) or in the **local scope** (accessible only within a specific function). The following four rules, based on Python's LEGB (Local, Enclosing, Global, Built-in) scope resolution model, help determine whether a variable is in the global or local scope:

---

1. Rule 1: Local Scope (L)

- A variable is considered to be in the **local scope** if it is defined within the current function or block of code.
- It is only accessible within that function or block.
- If a variable is assigned inside a function, that variable will be in the local scope of that function.

Example:

```
def my_function():
    x = 10  # 'x' is local to my_function
    print(x)

my_function()  # Output: 10
```

- In this case, `x` is local to the function `my_function()` and is not accessible outside it.

---

2. Rule 2: Enclosing Scope (E)

- If a variable is not found in the **local scope**, Python checks the **enclosing scope**, which refers to the scope of any enclosing functions.
- This applies to nested functions, where an inner function can access variables of its enclosing (outer) function.

Example:

```
def outer_function():
    y = 5  # 'y' is in the enclosing scope for inner_function
    def inner_function():
        print(y)  # 'y' is accessed from the enclosing scope
    inner_function()

outer_function()  # Output: 5
```

- Here, `y` is in the enclosing scope of the inner function `inner_function()`.

---

3. Rule 3: Global Scope (G)

- If the variable is not found in the local or enclosing scopes, Python checks the **global scope**, which refers to variables defined at the top level of the program (outside of any functions).
- Global variables are accessible anywhere in the code, but their value can be modified within functions using the `global` keyword.

Example:

```
x = 20  # 'x' is in the global scope

def my_function():
    print(x)  # Accesses global variable 'x'

my_function()  # Output: 20
```

- In this case, `x` is defined globally, so it is accessible inside `my_function()`.

---

4. Rule 4: Built-in Scope (B)

- If a variable is not found in the local, enclosing, or global scopes, Python checks the **built-in scope**. This scope contains all the standard functions and objects that Python provides, such as `print()`, `len()`, etc.
- These variables are always accessible.

Example:

```
print("Hello, World!")  # 'print' is a built-in function
```

- In this case, `print()` is accessed from Python's built-in scope.

Summary of the LEGB Rule (Local, Enclosing, Global, Built-in):

- **Local (L)**: The variable is found within the current function or block.
- **Enclosing (E)**: The variable is found in an enclosing function's scope (relevant for nested functions).
- **Global (G)**: The variable is defined at the top level of the program or module.
- **Built-in (B)**: The variable is part of Python's built-in functions and objects.

These rules help Python determine which variable to use when there are multiple variables with the same name in different scopes.

Q11. Write a program to create a guessing game where the user guesses a number between 1 and 10.(CO1) (BTL 4) (Marks: 5)

• Random number generation and user input: 2 marks
• Loop and condition handling: 2 marks
• Correct program output: 1 mark

Q11. Write a program to create a guessing game where the user guesses a number between 1 and 10.

Here's a Python program that implements a guessing game where the user has to guess a randomly generated number between 1 and 10:

```
import random  # Importing the random module to generate random numbers

def guessing_game():
    # Randomly generate a number between 1 and 10
    secret_number = random.randint(1, 10)

    # Initialize a variable to keep track of the number of attempts
    attempts = 0

    # Loop until the user guesses the correct number
    while True:
        # Get the user's guess
        guess = int(input("Guess a number between 1 and 10: "))

        # Increment the attempt counter
        attempts += 1

        # Check if the guess is correct
        if guess == secret_number:
            print(f"Congratulations! You've guessed the correct number 
{secret_number} in {attempts} attempts.")
            break  # Exit the loop when the correct number is guessed
        elif guess < secret_number:
            print("Too low! Try again.")
        else:
            print("Too high! Try again.")

# Call the function to start the game
guessing_game()
```

Explanation:

- **Random number generation**: The `random.randint(1, 10)` function generates a random integer between 1 and 10 (inclusive).
- **User input**: The program asks the user to input their guess using `input()`. The guess is converted to an integer with `int()`.
- **Loop and condition handling**: The program uses a `while` loop to repeatedly ask the user for a guess until they guess the correct number. It checks if the guess is too low or too high and gives appropriate feedback.
- **Correct output**: Once the user guesses the correct number, the program displays a congratulatory message with the number of attempts it took.

Example Output:

```
Guess a number between 1 and 10: 4
Too low! Try again.
Guess a number between 1 and 10: 7
Too high! Try again.
Guess a number between 1 and 10: 5
Congratulations! You've guessed the correct number 5 in 3 attempts.
```

Q12. What is the purpose of the global statement in Python? Explain with an example. (CO1) (BTL 2) (Marks: 5)

• Explanation of global statement: 2 marks
• Example demonstrating its usage: 3 marks

Q12. What is the purpose of the global statement in Python? Explain with an example.

Explanation of the `global` statement:

In Python, the `global` statement is used to declare that a variable is global, meaning it is defined outside of any function or block and can be accessed or modified inside functions. Without the `global` statement, Python will treat variables defined inside functions as local to that function. The `global` keyword allows you to modify a global variable within a function.

Usage of `global`:

When a variable is declared as global inside a function using the `global` keyword, any changes made to that variable inside the function will affect the global variable, and the changes will persist outside the function as well.

Example demonstrating the usage of `global`:

```
# Global variable
counter = 0

def increment():
    global counter  # Declare that we are using the global 'counter' variable
    counter += 1    # Increment the global 'counter' variable

def decrement():
    global counter  # Declare that we are using the global 'counter' variable
    counter -= 1    # Decrement the global 'counter' variable

# Calling the functions
increment()
increment()
print("Counter after increments:", counter)  # Output: Counter after increments:
2

decrement()
print("Counter after decrement:", counter)  # Output: Counter after decrement: 1
```

Explanation of the example:

1. **Global variable**: The variable `counter` is declared outside of any function, making it a global variable.
2. **Global statement**: Inside the `increment()` and `decrement()` functions, we use the `global` keyword to refer to the global `counter` variable, allowing us to modify it inside the functions.

3. **Function calls**: Each function modifies the global `counter` variable, and the changes are reflected outside the functions.
4. **Output**: The `counter` variable keeps its updated value even after the function execution, showing the effect of the `global` statement.

Example Output:

```
Counter after increments: 2
Counter after decrement: 1
```

Q13. Explain the role of try and except in handling exceptions. Demonstrate their use in preventing a program crash.(CO1) (BTL 3) (Marks: 5)

• Role of try and except in exception handling: 2 marks
• Program implementation to handle exceptions: 3 marks

Q13. Explain the role of `try` and `except` in handling exceptions. Demonstrate their use in preventing a program crash.

Role of `try` and `except` in exception handling:

In Python, exceptions are runtime errors that can cause the program to crash if they are not handled properly. The `try` and `except` block is used for handling these exceptions to prevent the program from crashing and to manage errors more gracefully.

- **`try` block**: This block contains the code that might raise an exception. The code inside this block is executed normally unless an error occurs.
- **`except` block**: This block catches the exception raised in the `try` block. If an exception occurs in the `try` block, the code inside the `except` block is executed. This allows the program to continue running instead of terminating.

By using `try` and `except`, we can manage unexpected errors (like division by zero, invalid input, etc.) and ensure that the program doesn't stop abruptly.

Program implementation to handle exceptions:

```
def divide_numbers():
    try:
        # Get user input for two numbers and try dividing them
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        # Try performing the division
        result = num1 / num2
        print(f"The result of {num1} divided by {num2} is {result}")

    except ZeroDivisionError:
        # Handle the case where division by zero is attempted
        print("Error: Cannot divide by zero!")

    except ValueError:
        # Handle the case where user inputs a non-numeric value
        print("Error: Please enter valid numbers!")

    except Exception as e:
        # Catch any other general exceptions
        print(f"An unexpected error occurred: {e}")

    finally:
        print("Program execution complete.")

# Call the function
divide_numbers()
```

Explanation of the program:

1. **`try` block**:
   - The program asks the user to input two numbers and attempts to perform the division of the first number by the second.
2. **`except` blocks**:
   - **`ZeroDivisionError`**: If the user attempts to divide by zero, this block will catch the error and display an appropriate message.
   - **`ValueError`**: If the user enters a non-numeric value, this block catches the error and informs the user to input valid numbers.
   - **`Exception`**: A general `Exception` block is included to catch any other unexpected errors.
3. **`finally` block**:
   - This block always executes at the end, regardless of whether an exception occurred or not, ensuring that the program finishes cleanly.

Example Output:

Scenario 1: User enters valid numbers

```
Enter the first number: 10
Enter the second number: 2
The result of 10.0 divided by 2.0 is 5.0
Program execution complete.
```

Scenario 2: User tries to divide by zero

```
Enter the first number: 10
Enter the second number: 0
Error: Cannot divide by zero!
Program execution complete.
```

Scenario 3: User enters a non-numeric value

```
Enter the first number: ten
Error: Please enter valid numbers!
Program execution complete.
```

Conclusion:

The `try` and `except` blocks effectively prevent the program from crashing when exceptions occur, allowing for better error handling and user experience