# Question 1 : In Python, what is the difference between a built-in function and a user-defined function?

Ans :

**Built-in Function**:

* These are pre-defined functions provided by Python and are always available to use without needing to define them yourself.
* They perform common operations such as mathematical calculations, string manipulations, file handling, etc.
* Examples include print(), len(), sum(), max(), min(), input(), etc.
* Built-in functions are optimized and written in C for performance, making them faster than most user-defined functions.

**User-Defined Function**:

* These are functions that are defined by the programmer using the def keyword. You write the function to perform a specific task based on your requirements.
* They allow you to encapsulate logic into reusable blocks of code.
* Example:

def greet(name):

return f"Hello, {name}!"

* User-defined functions can take parameters and return values as needed, and can be defined as needed for custom behavior.

# Question 2 : How can you pass arguments to a function in Python? Explain the difference between positional arguments and keyword arguments.

In Python, you can pass arguments to a function in several ways. The main types of argument passing are **positional arguments**, **keyword arguments**, **default arguments**, **variable-length arguments**, and **keyword-only arguments**. Let's break down the differences between **positional** and **keyword** arguments, along with other types.

**1. Positional Arguments:**

* Positional arguments are passed to a function in the same order that the parameters are defined in the function.
* The first argument passed is assigned to the first parameter, the second to the second, and so on.
* This is the most common way to pass arguments to functions.

**Example**:

python

Copy code

def greet(name, age):

print(f"Hello {name}, you are {age} years old.")

greet("Alice", 30) # Here, "Alice" is passed to 'name' and 30 to 'age'

In the above example, "Alice" is assigned to name, and 30 is assigned to age based on their position in the function call.

**2. Keyword Arguments:**

* Keyword arguments are passed to a function using the name of the parameter along with its value.
* You can pass arguments in any order when using keyword arguments, as long as the parameter names are correct.
* This method is useful when you have a function with many parameters, and you want to make the code more readable by explicitly stating the parameter names.

**Example**:

python

Copy code

def greet(name, age):

print(f"Hello {name}, you are {age} years old.")

greet(age=30, name="Alice") # Here, 'age' and 'name' are passed as keyword arguments

Here, the arguments are passed in a different order compared to the function definition, but they still work because they're matched by keyword.

**Differences Between Positional and Keyword Arguments:**

* **Order**: Positional arguments must follow the order defined in the function signature, while keyword arguments can be passed in any order.
* **Flexibility**: Keyword arguments allow you to pass only the arguments you need, which makes the code more flexible and readable.

**Other Types of Arguments:**

1. **Default Arguments**:
   * You can assign default values to parameters in a function. If a value is not passed for a parameter when calling the function, the default value is used.

**Example**:

python

Copy code

def greet(name, age=25):

print(f"Hello {name}, you are {age} years old.")

greet("Bob") # Uses the default age value of 25

1. **Variable-Length Arguments**:
   * If you don't know how many arguments you’ll pass to a function, you can use \*args (for non-keyword arguments) or \*\*kwargs (for keyword arguments) to pass a variable number of arguments.

**Example using \*args**:

python

Copy code

def sum\_numbers(\*args):

return sum(args)

print(sum\_numbers(1, 2, 3, 4)) # Outputs: 10

**Example using \*\*kwargs**:

python

Copy code

def greet(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

greet(name="Alice", age=30) # Outputs: name: Alice, age: 30

1. **Keyword-only Arguments**:
   * In some cases, you might want to force a user to pass an argument by keyword (not by position). You can specify this by placing a \* in the function signature.

**Example**:

python

Copy code

def greet(name, \*, age):

print(f"Hello {name}, you are {age} years old.")

greet("Alice", age=30) # Correct usage

greet("Alice", 30) # This will raise a TypeError

**Summary:**

* **Positional arguments** are passed by the order in which they are defined in the function.
* **Keyword arguments** are passed by explicitly stating the parameter names, allowing the order to be flexible.
* Other types like **default arguments**, **variable-length arguments**, and **keyword-only arguments** provide additional flexibility depending on the needs of your function.

# Question 3. What is the purpose of the return statement in a function? Can a function have multiple return

# statements? Explain with an example.

**Purpose of the return Statement in a Function:**

The return statement in Python is used to exit a function and optionally send a result or value back to the caller. When a function is called, the return statement allows the function to **output a value**, which can be used in further computations, printed, or assigned to a variable.

* The return statement ends the function's execution and provides the result to the calling code.
* If no return statement is present, the function will return None by default.

**Can a Function Have Multiple return Statements?**

Yes, a function can have **multiple return statements**. However, a function will exit as soon as the first return statement is encountered, and no code after that will be executed. This can be useful in situations where different conditions in the function lead to different results or actions.

**Example with Multiple return Statements:**

python

Copy code

def check\_number(num):

if num > 0:

return "Positive"

elif num < 0:

return "Negative"

else:

return "Zero"

# Example calls:

print(check\_number(5)) # Output: Positive

print(check\_number(-3)) # Output: Negative

print(check\_number(0)) # Output: Zero

In this example:

* The function check\_number checks if the number is positive, negative, or zero.
* It uses different return statements to exit the function and return the corresponding result for each condition.
* As soon as one return statement is encountered, the function exits, and no further return statements are executed.

# Question 4. What are lambda functions in Python? How are they different from regular functions? Provide an

# example where a lambda function can be useful.

A **lambda function** in Python is a small, anonymous function that is defined using the lambda keyword. It can have any number of arguments but only one expression. The value of the expression is automatically returned, so you do not need to explicitly use the return keyword.

The basic syntax of a lambda function is:

python

Copy code

lambda arguments: expression

**Key Characteristics of Lambda Functions:**

1. **Anonymous**: Lambda functions don't have a name, unlike regular functions defined with the def keyword.
2. **One Expression**: A lambda function can only contain one expression. This expression is evaluated and returned automatically.
3. **Shorter Syntax**: Lambda functions are typically more concise and are used for small tasks where defining a full function using def would be overkill.

**Difference Between Lambda Functions and Regular Functions:**

1. **Syntax**:
   * **Lambda Function**: Defined using lambda with no def keyword and has no name.
   * **Regular Function**: Defined using the def keyword and can have multiple expressions and statements.
2. **Return Value**:
   * **Lambda Function**: Implicitly returns the result of the expression.
   * **Regular Function**: Requires an explicit return statement to return a value.
3. **Use Case**:
   * **Lambda Function**: Typically used for short, one-off functions that are used in places where a function is required temporarily (like in map(), filter(), or sorted()).
   * **Regular Function**: More appropriate for complex operations with multiple expressions or logic that may require multiple lines of code.

**Example of a Lambda Function:**

A lambda function is useful when you need a short function for a one-time operation, like sorting or filtering data.

python

Copy code

# Regular function to add two numbers

def add(x, y):

return x + y

# Lambda function to add two numbers

add\_lambda = lambda x, y: x + y

# Both functions give the same result

print(add(3, 5)) # Output: 8

print(add\_lambda(3, 5)) # Output: 8

**Example Where a Lambda Function is Useful:**

Lambda functions are particularly useful when you need to pass a simple function as an argument to higher-order functions like map(), filter(), or sorted(). Here’s an example using sorted() with a lambda function to sort a list of tuples based on the second element:

python

Copy code

# List of tuples

data = [(1, 'apple'), (2, 'banana'), (3, 'cherry')]

# Sort the list by the second element of each tuple (the string)

sorted\_data = sorted(data, key=lambda x: x[1])

print(sorted\_data)

# Output: [(1, 'apple'), (2, 'banana'), (3, 'cherry')]

In this example:

* The lambda function lambda x: x[1] takes each tuple x and returns the second element (x[1]), which is then used as the sorting key.
* This simplifies the code since you don't need to define a separate function to extract the sorting key.

**Summary:**

* **Lambda Functions**: Short, anonymous functions with a single expression and no need for a return keyword.
* **Regular Functions**: More flexible, can have multiple expressions, and require an explicit return.
* **Usefulness**: Lambda functions are ideal for short, temporary functions that are passed as arguments to other functions like map(), filter(), or sorted().

# Question 5. How does the concept of "scope" apply to functions in Python? Explain the difference between local

# scope and global scope.

**Scope in Python:**

In Python, **scope** refers to the region of the program where a particular variable is accessible. When you define a variable, Python determines the **scope** in which the variable exists and can be used. The two most common types of scopes in Python are **local scope** and **global scope**.

**1. Local Scope:**

* A **local scope** is the area within a function where a variable is defined. Variables created inside a function are local to that function, meaning they are only accessible within that function. Once the function exits, the variable is no longer accessible.
* Local variables can only be accessed within the function or block of code in which they are defined.

**Example of Local Scope**:

python

Copy code

def example\_function():

x = 10 # x is a local variable within example\_function

print(x) # This will print 10

example\_function()

# print(x) # This would raise an error because x is not defined in the global scope.

In the example above:

* The variable x is defined inside the function example\_function(), so it has **local scope**.
* You cannot access x outside the function because it doesn't exist in the global scope.

**2. Global Scope:**

* A **global scope** refers to the area outside of any function or block of code. Variables defined outside any function are considered **global variables**, and they are accessible throughout the program (unless shadowed by local variables with the same name).
* Global variables can be accessed and modified inside functions if explicitly declared as global.

**Example of Global Scope**:

python

Copy code

x = 20 # x is a global variable

def example\_function():

print(x) # x is accessed from the global scope

example\_function() # This will print 20

In this example:

* x is defined outside of any function, so it is in the **global scope** and can be accessed by example\_function().

**Difference Between Local and Global Scope:**

| **Feature** | **Local Scope** | **Global Scope** |
| --- | --- | --- |
| **Definition** | Variables are defined inside a function. | Variables are defined outside of any function. |
| **Visibility** | Only accessible within the function or block. | Accessible from anywhere in the program. |
| **Lifetime** | Exists only as long as the function is executing. | Exists throughout the program’s execution. |
| **Accessing in Functions** | Variables inside a function cannot access global variables without declaration. | Variables are accessible in any function unless shadowed. |

**Using Global Variables Inside Functions:**

To access or modify a global variable inside a function, you must use the global keyword. Without this, Python treats variables inside the function as local, even if they have the same name as a global variable.

**Example of Accessing and Modifying Global Variables**:

python

Copy code

x = 5 # Global variable

def change\_global\_variable():

global x # Declare x as a global variable

x = 10 # Modify the global variable

change\_global\_variable()

print(x) # Output: 10

In this example:

* The global keyword is used inside change\_global\_variable() to indicate that x refers to the global variable.
* Without global, x inside the function would be treated as a local variable.

**LEGB Rule (Local, Enclosing, Global, Built-in):**

Python follows the **LEGB** rule to search for a variable when it's referenced:

1. **Local**: The current function or block of code.
2. **Enclosing**: Any enclosing function (if the function is nested).
3. **Global**: The top-level scope in the current module or file.
4. **Built-in**: The built-in scope, which contains Python's standard library functions and exceptions.

**Example with LEGB:**

python

Copy code

x = 50 # Global variable

def outer\_function():

x = 100 # Enclosing variable

def inner\_function():

x = 200 # Local variable

print(x) # This will print 200 (Local scope is checked first)

inner\_function()

outer\_function()

In this example:

* inner\_function() prints the value 200 because it uses the **local scope** of the variable x.
* Even though x is defined in both the **enclosing** and **global** scopes, the **local scope** in inner\_function() takes precedence.

**Summary:**

* **Local Scope**: Variables defined inside a function or block are only accessible within that function or block.
* **Global Scope**: Variables defined outside of any function are accessible throughout the entire program.
* **global keyword**: Allows modification of a global variable inside a function.
* **LEGB rule**: Python checks the variable's scope in the following order: Local, Enclosing, Global, Built-in.

# Question 6. How can you use the "return" statement in a Python function to return multiple values?

In Python, the return statement can be used to return multiple values from a function. While Python functions traditionally return a single value, you can return **multiple values** as a **tuple**, **list**, or other iterable types. By returning a tuple (which is implicit when you separate values by commas), you can easily return multiple values from a function.

**Returning Multiple Values Using a Tuple:**

A tuple is an ordered, immutable collection of items. In Python, when you separate multiple items by commas, Python automatically packs them into a tuple.

**Example of Returning Multiple Values as a Tuple:**

python

Copy code

def get\_coordinates():

x = 5

y = 10

return x, y # This returns a tuple (x, y)

# Calling the function

coordinates = get\_coordinates()

print(coordinates) # Output: (5, 10)

# You can also unpack the returned tuple into individual variables

x, y = get\_coordinates()

print(f"x: {x}, y: {y}") # Output: x: 5, y: 10

In this example:

* The function get\_coordinates() returns two values: x and y.
* These values are returned as a tuple (5, 10).
* When you call the function, you can either work with the entire tuple or unpack the values into separate variables.

**Returning Multiple Values Using a List:**

You can also return multiple values as a list, which is mutable and can be modified.

**Example of Returning Multiple Values as a List:**

python

Copy code

def get\_fruits():

return ['apple', 'banana', 'cherry'] # Returning a list of fruits

# Calling the function

fruits = get\_fruits()

print(fruits) # Output: ['apple', 'banana', 'cherry']

**Returning Multiple Values Using a Dictionary:**

If you want to return values with specific names or labels, using a **dictionary** is a good option. Each key can represent a label, and the value can represent the corresponding data.

**Example of Returning Multiple Values Using a Dictionary:**

python

Copy code

def get\_person\_info():

return {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Calling the function

person\_info = get\_person\_info()

print(person\_info) # Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Accessing values by their keys

print(person\_info['name']) # Output: Alice

**Summary:**

* You can return multiple values from a function by returning them as a **tuple**, **list**, or **dictionary**.
* **Tuple** is the most common method (by default, Python returns multiple values as a tuple).
* You can **unpack** the values when you return them in a tuple or list.

4o mini

# Question 7. What is the difference between the "pass by value" and "pass by reference" concepts when it

# comes to function arguments in Python?

In Python, the concepts of **"pass by value"** and **"pass by reference"** are often discussed, but Python handles function arguments in a slightly different way. Python uses a mechanism called **"pass by object reference"** (or sometimes referred to as **pass by assignment**). This means that the behavior you observe when passing arguments to a function depends on the type of the object being passed (whether it is mutable or immutable).

Let's break down the difference between **pass by value** and **pass by reference**, and explain how they apply in Python:

**1. Pass by Value:**

* **Pass by value** means that a **copy** of the argument is passed to the function. Modifications to the argument within the function do not affect the original variable in the calling scope.
* In **languages like C**, when a function is called with a variable, it receives a copy of the value, and any changes made to the variable inside the function do not affect the original variable outside the function.

**Example of Pass by Value (in concept)**:

python

Copy code

def modify\_value(a):

a = 10 # This only modifies the local copy of a

print(f"Inside function: a = {a}")

x = 5

modify\_value(x)

print(f"Outside function: x = {x}") # x is still 5

Here:

* a inside the function is a **local variable** that holds a **copy** of x.
* Modifying a inside the function does not affect the original variable x outside the function.

**2. Pass by Reference:**

* **Pass by reference** means that the function receives a reference (or memory address) to the **original object**, not a copy. This means that changes made to the argument inside the function will affect the original object outside the function.
* In **languages like C++**, when you pass a reference to a function, any changes to the argument within the function affect the original variable in the calling scope.

**Example of Pass by Reference (in concept)**:

python

Copy code

def modify\_list(lst):

lst.append(4) # Modifies the original list

print(f"Inside function: lst = {lst}")

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(f"Outside function: my\_list = {my\_list}") # my\_list is modified

Here:

* lst inside the function is a **reference** to the **original list** my\_list.
* Modifying lst inside the function directly affects the original my\_list outside the function, because both refer to the same object in memory.

**Python's Approach: "Pass by Object Reference" (or Pass by Assignment):**

* In Python, **all arguments are passed by object reference**, which means that the function receives a reference to the object, not a copy of it.
* However, whether changes to the argument inside the function affect the original object depends on whether the object is **mutable** or **immutable**.

**How Python's Object Reference Works:**

* **Immutable Objects**: If you pass an **immutable object** (like an integer, string, or tuple) to a function, you cannot modify the original object. Modifying it inside the function creates a **new object**. Hence, it behaves similarly to **pass by value**.

**Example with Immutable Object**:

python

Copy code

def modify\_number(n):

n = 10 # This creates a new object, it does not affect the original variable

print(f"Inside function: n = {n}")

x = 5

modify\_number(x)

print(f"Outside function: x = {x}") # x is still 5

* + Here, n = 10 creates a new integer object, and the original x is unchanged.
* **Mutable Objects**: If you pass a **mutable object** (like a list or dictionary) to a function, any modifications to the object inside the function will affect the original object. Hence, it behaves more like **pass by reference**.

**Example with Mutable Object**:

python

Copy code

def modify\_list(lst):

lst.append(4) # Modifies the original list

print(f"Inside function: lst = {lst}")

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(f"Outside function: my\_list = {my\_list}") # my\_list is modified

* + Here, lst.append(4) modifies the original list my\_list.

**Summary of Differences:**

| **Concept** | **Pass by Value** | **Pass by Reference** | **Python's Behavior** |
| --- | --- | --- | --- |
| **Definition** | A copy of the value is passed to the function. Changes inside the function do not affect the original value. | A reference to the original variable is passed to the function. Changes inside the function affect the original variable. | Python passes arguments by object reference (pass by assignment). |
| **Effect on Immutable Objects** | No effect on the original object, behaves like pass by value. | N/A for immutable objects. | Changes do not affect the original object (e.g., integers, strings). |
| **Effect on Mutable Objects** | N/A for mutable objects. | Changes affect the original object. | Changes to mutable objects (e.g., lists, dictionaries) will affect the original object. |
| **Example** | Integers, Strings, Tuples | Lists, Dictionaries | Mixed behavior depending on whether the object is mutable or immutable. |

**Conclusion:**

* **Python** uses a concept called **pass by object reference** or **pass by assignment**, which behaves differently based on whether the object being passed is mutable or immutable.
* For **immutable objects**, it behaves like **pass by value** because any modification creates a new object.
* For **mutable objects**, it behaves like **pass by reference** because the original object can be modified within the function.

# Question 8. Create a function that can intake integer or decimal value and do following operations:

# a. Logarithmic function (log x)

# b. Exponential function (exp(x))

# c. Power function with base 2 (2x)

# d. Square root

You can create a Python function that takes an integer or decimal value and performs the following operations:

1. **Logarithmic function (log x)**: Using the math.log() function.
2. **Exponential function (exp(x))**: Using the math.exp() function.
3. **Power function with base 2 (2^x)**: Using the exponentiation operator (\*\*).
4. **Square root**: Using math.sqrt().

Here’s a function that does all the operations:

python

Copy code

import math

def perform\_operations(value):

if value <= 0:

print("Logarithm and square root are not defined for zero or negative numbers.")

return

# Logarithmic function

log\_value = math.log(value)

# Exponential function

exp\_value = math.exp(value)

# Power function with base 2 (2^x)

power\_value = 2 \*\* value

# Square root

sqrt\_value = math.sqrt(value)

# Display the results

print(f"Logarithm (log {value}): {log\_value}")

print(f"Exponential (exp {value}): {exp\_value}")

print(f"Power with base 2 (2^{value}): {power\_value}")

print(f"Square root (sqrt {value}): {sqrt\_value}")

# Example usage

perform\_operations(4)

**Explanation:**

* **Logarithmic function (log x)**: The function math.log(value) calculates the natural logarithm (base e) of the input value. If you want a logarithm with a different base, you can pass the base as a second argument, like math.log(value, base).
* **Exponential function (exp(x))**: The function math.exp(value) calculates the exponential of value, i.e., exe^{x}ex.
* **Power function with base 2 (2^x)**: We use the \*\* operator to calculate 2x2^x2x.
* **Square root (sqrt(x))**: The function math.sqrt(value) computes the square root of the input value.

**Example Output for perform\_operations(4):**

c

Copy code

Logarithm (log 4): 1.3862943611198906

Exponential (exp 4): 54.598150033144236

Power with base 2 (2^4): 16

Square root (sqrt 4): 2.0

**Notes:**

* The function checks if the input value is less than or equal to 0 to avoid errors with the logarithmic function and square root, which are not defined for non-positive numbers.

# Question 9. Create a function that takes a full name as an argument and returns first name and last name.

You can create a function that takes a full name as an argument, splits it into the first name and last name, and returns them. Here's a simple implementation:

python

Copy code

def split\_name(full\_name):

# Split the full name into parts

name\_parts = full\_name.split()

# Check if the name has at least two parts (first and last name)

if len(name\_parts) < 2:

return "Please provide both first and last names."

# Extract the first and last name

first\_name = name\_parts[0]

last\_name = name\_parts[-1] # In case there are middle names, it returns the last part as the last name

return first\_name, last\_name

# Example usage

full\_name = "John Doe"

first\_name, last\_name = split\_name(full\_name)

print(f"First Name: {first\_name}")

print(f"Last Name: {last\_name}")

**Explanation:**

* The split\_name() function splits the full\_name string into a list of parts using the split() method, which separates the string by whitespace.
* If the full name contains at least two parts (a first name and a last name), it assigns the first part to first\_name and the last part to last\_name.
* If the name contains more than two parts (e.g., a middle name), it will assign the last part to last\_name.
* If the full name contains fewer than two parts (e.g., just a first name), it returns a message asking for both the first and last names.

**Example Output for split\_name("John Doe"):**

mathematica

Copy code

First Name: John

Last Name: Doe

**Example Output for split\_name("John Michael Doe"):**

mathematica

Copy code

First Name: John

Last Name: Doe

**Example Output for split\_name("John"):**

sql

Copy code

Please provide both first and last names.