Answer 1:-

The difference between built-in functions and user-defined functions is:

1. **Built-in Functions**

- These are pre-defined functions provided by Python.
- They are always available for use without requiring additional definitions.
- Examples: print(), len(), sum(), max(), etc.

Example of a built-in function:-

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)  # Using the built-in function sum()
print(total)  # Output: 15
```

2. **User-defined function**

- These are functions created by the user to perform specific tasks.
- They use the def keyword to define the function.
- They help in modularity and code reusability.

Example of a user-defined function:

```
def add_numbers(a, b):
    return a + b  # Function defined by the user
result = add_numbers(3, 7)
print(result)  # Output: 10
```

Answer 2.

Passing Arguments to a Function in Python:-

In Python, you can pass arguments to a function by defining parameters in the function signature and providing values when calling the function. Arguments can be passed in different ways, primarily as positional arguments or keyword arguments.

1. Positional Arguments

- Arguments are passed in the same order as the function parameters.
- The position of the arguments determines their values.
- If the order is changed, the function may produce incorrect results.

Example:

```
def greet(name, age):
   print(f"Hello, my name is {name} and I am {age} years old.")

greet("Alice", 25)
greet(25, “Alice”)
```

Output:

```
Hello, my name is Alice and I am 25 years old
```

Incorrect order ! output:

```
Hello, my age is 25 and I am Alice years old.
```

Answer 3:-

Purpose of the return Statement in a Function:-

The return statement in Python is used to send a value or result from a function back to the caller. It allows a function to produce output that can be stored in a variable or used in further computations.

Key Points About return:-

- It terminates the function execution immediately.
- It returns a value (or multiple values as a tuple).
- If no return is specified, the function returns None by default.

Can a Function Have Multiple return Statements

Yes, a function can have multiple return statements, typically used within conditional structures like if-else. Once a return statement executes, the function stops running, and no further code inside the function is executed.

Example: Multiple return Statements

```
def check_number(num):
   if num > 0:
      return "Positive"
   elif num < 0:
      return "Negative"
   else:
      return "Zero"
```

```
print(check_number(5))   # Output: Positive
print(check_number(-3))  # Output: Negative
print(check_number(0))   # Output: Zero
```

<u>Answer 4</u>:-

Lambda Functions in Python:-

A lambda function is a small, anonymous function in Python that is defined using the lambda keyword. It can have multiple arguments but only one expression, which is evaluated and returned automatically.

Syntax:

```
lambda arguments: expression
```

Differences Between Lambda Functions and Regular Functions

Lambda Function

- Uses lambda keyword
- Anonymous (unnamed)
- Only one expression
- Short and simple
- Concise but less readable for complex logic

Regular Function

- Uses def keyword
- Has a function name
- Can have multiple expressions
- Can be complex and multi-line
- More readable for complex logic

Example of a Lambda Function

A lambda function can be used for short, simple operations like sorting, filtering, or mapping values.

Using Lambda to Double a Number

```
double = lambda x: x * 2
print(double(5))  # Output: 10
```

Where Can a Lambda Function Be Used

Example: Using Lambda with map()

When applying a function to a list of values, a lambda function is useful for inline operations.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

```
Output: [1, 4, 9, 16, 25]
```

When to Use Lambda Functions?

- When you need a short, throwaway function.
- When using higher-order functions like map(), filter(), or sorted().
- When you want to avoid unnecessary function definitions for small tasks.

However, for complex logic, it's better to use a regular function for better readability and maintainability.

Answer 5:-

scope refers to the region of a program where a particular variable is accessible. The concept of scope is crucial because it determines which variables can be accessed and modified at different points in the code.

Types of Scope in Python:-

1. Global Scope

- A variable declared outside any function or block belongs to the global scope.
- It can be accessed from anywhere in the script, including inside functions (unless shadowed by a local variable).
- However, to modify a global variable inside a function, you need to use the global keyword.

Example:-

```
x = 10  # Global variable

def my_function():
    print(x)  # Accessing the global variable
```

```
my_function()  # Output: 10
```

2. Local Scope

- A variable declared inside a function is local to that function.
- It cannot be accessed outside the function.
- Local variables take precedence over global variables if they have the same name.

Example:-

```
def my_function():
    y = 5  # Local variable
    print(y)

my_function()  # Output: 5
print(y)
```

Modifying a global variable inside a function:

```
x = 10

def update_global():
    global x  # Declaring x as global
    x = 20

update_global()
print(x)  # Output: 20
```

Local variable shadowing a global variable:

```
x = 10

def my_function():
    x = 5  # Local variable (shadows the global x)
    print(x)  # Output: 5

my_function()
print(x)
```

Difference between global and local scope:-

1. Global Scope

- Outside functions

- Available everywhere
- Needs global keyword inside a function
- Lower if a local variable has the same name

2. Local scope

- Inside a function
- Only inside the function
- Can be modified freely within the function
- Higher than global if a variable with the same name exists

Answer 6:-

In Python, you can use the return statement to return multiple values from a function. Python allows returning multiple values as a tuple, list, or dictionary.

How to use return statement:-

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def get_person():
    return Person("Bob", 30)

p = get_person()
print(p.name, p.age)  # Output: Bob 30
```

Answer 7:-

1. Pass by Value (Not exactly in Python)

pass by value means that a copy of the actual value is passed to the function, so changes inside the function do not affect the original variable.

In Python, immutable objects (like int, float, str, tuple) behave similarly to pass by value because any modification results in a new object being created.

Example:-

```
def modify(x):
    x = 10  # Assigning a new value creates a new object
    print("Inside function:", x)
```

```
num = 5
modify(num)
print("Outside function:", num)
```

Here, num remains 5 because integers are immutable, and reassignment inside the function creates a new local variable.

2. Pass by Reference (Not exactly in Python)

In true pass by reference, a function receives a reference to the original variable, so modifying it inside the function affects the original variable.

In Python, mutable objects (like list, dict, set) behave similarly because their contents can be changed inside a function.

```
def modify(lst):
    lst.append(4)  # Modifying the existing list
    print("Inside function:", lst)

my_list = [1, 2, 3]
modify(my_list)
print("Outside function:", my_list)
```

Summary

- Immutable objects (int, float, str, tuple) behave like pass by value because changes inside the function create a new object.
- Mutable objects (list, dict, set) behave like pass by reference because changes affect the original object.
- Reassigning a mutable object inside a function (lst = [...]) does not modify the original variable—it simply binds the local variable to a new object.

Answer 8:-

A. Logarithm function (log x)

Here's a Python function that takes an integer or decimal value and returns its logarithm. By default, it calculates the natural logarithm (base e), but you can specify a different base if needed.

Function:

.

Function:

```
import math

def compute_log(value, base=math.e):
   “””
    Computes the logarithm of the given value.

    Parameters:
    - value (int or float): The number to compute the logarithm for.
    - base (int or float, optional): The base of the logarithm (default is natural log, base e).

    Returns:

   - float: The computed logarithm.
    "””
if value <= 0:
        raise ValueError("Logarithm is undefined for zero or negative numbers.")

    return math.log(value, base)

# Example Usage:
print(compute_log(10))       # Natural log (ln 10)
print(compute_log(100, 10))  # Log base 10 (log_10 100)
```

B. Exponentiation function (exp(x))

```
import math

def compute_exponentiation(value):
    """
    Computes the exponentiation of the given value (e^x).

    Parameters:
    - value (int or float): The exponent to raise e to.

    Returns:
    - float: The computed exponentiation result.
    """
    return math.exp(value)

# Example Usage:

print(compute_exponentiation(1))
print(compute_exponentiation(2.5))
print(compute_exponentiation(-1))
```

C. Power function with base 2:-

```
def power_of_two(value):
    """
    Computes 2 raised to the power of the given value (2^x).

    Parameters:
    - value (int or float): The exponent to raise 2 to.

    Returns:
    - float: The computed power of 2.
    """
    return 2 ** value

# Example Usage:
print(power_of_two(3))   # 2^3 = 8
print(power_of_two(4.5)) # 2^4.5
print(power_of_two(-2))
```

D. **Square root**

```
import math

def compute_square_root(value):
    """
    Computes the square root of the given value.

    Parameters:
    - value (int or float): The number to find the square root of.

    Returns:
    - float: The computed square root.
    """”

if value < 0:
        raise ValueError("Square root is undefined for negative numbers.")

    return math.sqrt(value)

# Example Usage:
print(compute_square_root(9))    # √9 = 3
print(compute_square_root(2.25)) # √2.25 = 1.5
print(compute_square_root(0))    # √0 = 0
```