1. In Python, what is the difference between a built-in function and a user-defined function? Provide an example of each.

Answer:-

In Python, the main difference between a built-in function and a user-defined function lies in their origin and availability.

Built-in Function:

A built-in function is a function that comes pre-defined in Python and is readily available for use without the need for any additional setup or import.

Example of using a built-in function:

- print("Hello, World!")
  Output: Hello, World!
- my_list = [1, 2, 3, 4, 5]
  print(len(my_list))
  Output: 5

User-Defined Function:

A user-defined function is a function that is created by the programmer to perform a specific task or set of tasks. These functions are defined using the def keyword followed by a function name, parameters (optional), and the function body.

Example of defining and using a user-defined function:

```
def square(num):
    return num ** 2
result = square(5)
print(result)
# Output: 25
```

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

Answer:-

In Python, you can pass arguments to a function by including them within the parentheses when defining the function and then providing corresponding values when calling the function.

There are two main ways to pass arguments to a function: positional arguments and keyword arguments.

Positional Arguments:

Positional arguments are passed based on their position in the function call. The order of the arguments matters, and they are assigned to the function parameters based on their position. The number of arguments and their positions in the function call must match the number of parameters in the function definition.

Example of using positional arguments:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

# Function call with positional arguments
```

```
greet("Alice", 30)  # Output: Hello, Alice! You are 30 years old.
```
In this example, the function greet() takes two parameters name and age. When we call the function with "Alice" and 30, the first value "Alice" is assigned to the name parameter, and the second value 30 is assigned to the age parameter based on their positions.

Keyword Arguments:
Keyword arguments are passed with the parameter names explicitly specified in the function call. The order of the arguments doesn't matter as long as their names are specified. This allows you to pass arguments in any order and even skip some parameters if their default values are provided in the function definition.

Example of using keyword arguments:
```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

# Function call with keyword arguments
greet(age=25, name="Bob")  # Output: Hello, Bob! You are 25 years old.
```
In this example, we pass the arguments age=25 and name="Bob" to the function greet() using their respective parameter names. The order of the arguments in the function call is different from the order of the parameters in the function definition, but the function correctly assigns the values based on the provided names.

Default Values for Parameters:
You can also provide default values for function parameters, making them optional when calling the function. If an argument is not provided during the function call, the default value will be used.

Example with default values:

python
Copy code
```
def greet(name, age=30):
    print(f"Hello, {name}! You are {age} years old.")

# Function calls with and without providing the age argument
greet("Alice")      # Output: Hello, Alice! You are 30 years old.
greet("Bob", 25)    # Output: Hello, Bob! You are 25 years old.
```
In this example, the age parameter has a default value of 30. If we call the function with only the name argument, the default value 30 is used for the age parameter. If we provide an explicit value for the age parameter, that value is used instead.

3. What is the purpose of the return statement in a function? Can a function have multiple return
statements? Explain with an example.

Answer:-

The return statement in a function serves the purpose of terminating the function's execution and returning a value (or values) back to the caller. When the return statement is encountered, the function exits immediately, and any code below the return statement will not be executed.

Yes, a function can have multiple return statements. However, as soon as the first return statement is executed, the function will terminate, and the value specified after the return keyword will be returned to the caller. Any subsequent return statements in the function will not be executed.

Here's an example of a function with multiple return statements:

```
def find_square_or_cube(number):
    square = number ** 2
    cube = number ** 3

    if number > 0:
        return square
    else:
        return cube

result1 = find_square_or_cube(3)
print("Result 1:", result1)  # Output: Result 1: 9

result2 = find_square_or_cube(-2)
print("Result 2:", result2)  # Output: Result 2: -8
```

4. What are lambda functions in Python? How are they different from regular functions? Provide an example where a lambda function can be useful.

Answer:-

Lambda functions in Python are small, anonymous functions that can have any number of arguments but can only have one expression. They are also known as "anonymous functions" because they don't require a specific name to be defined like regular functions using the def keyword.

The syntax for a lambda function is as follows:

```
lambda arguments: expression
```

Difference between Lambda Functions and Regular Functions:

Syntax: Lambda functions are defined using the lambda keyword, while regular functions are defined using the def keyword.

Name: Lambda functions are anonymous and don't have a specific name, whereas regular functions are given a name when defined.

Body: Lambda functions can only have one expression in their body, while regular functions can have multiple statements and a code block in their body.

Return: Lambda functions implicitly return the result of their expression, while regular functions use the return statement to explicitly return a value.

Using a regular function:

```
def double(x):
    return x * 2

my_list = [1, 2, 3, 4, 5]
result = list(map(double, my_list))
print(result)  # Output: [2, 4, 6, 8, 10]
```

Using a lambda function:

```
my_list = [1, 2, 3, 4, 5]
result = list(map(lambda x: x * 2, my_list))
print(result)  # Output: [2, 4, 6, 8, 10]
```

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

Answer:-

In Python, "scope" refers to the region of the code where a particular variable is accessible and can be referenced. The concept of scope applies to functions, and it determines the visibility and lifetime of variables declared inside or outside functions.

Local variables are accessible only within the function where they are defined, and they have a limited lifetime. Global variables, on the other hand, can be accessed from any part of the code and have a lifetime that spans the entire program execution.

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

Answer:-

In Python, you can use the return statement in a function to return multiple values by returning them as a tuple, list, or any other data structure that can hold multiple elements. When you use return with multiple expressions or variables separated by commas, Python automatically packs them into a single object (such as a tuple) before returning.

7. What is the difference between the "pass by value" and "pass by reference" concepts when it comes to function arguments in Python?

Answer:-

In Python, the concepts of "pass by value" and "pass by reference" are often used to describe how function arguments are treated when passed to a function.

Pass by Value:

In a "pass by value" approach, a copy of the actual value of the argument is passed to the function. This means that any changes made to the function's parameter within the function do not affect the original value of the argument outside the function. Essentially, the function works with a local copy of the argument, and any modifications are isolated to that local copy.

Pass by Reference:

In a "pass by reference" approach, a reference or memory address to the original object is passed to the function. This means that any changes made to the function's parameter will directly affect the

original value of the argument outside the function. In this case, the function works with the actual object, and any modifications are reflected in the original object itself.

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 (2^x)
d. Square root

Answer:-

```
import math

def math_operations(x):
    log_result = math.log(x)
    exp_result = math.exp(x)
    power_result = 2 ** x
    sqrt_result = math.sqrt(x)

    return log_result, exp_result, power_result, sqrt_result

# Test the function with different inputs
integer_input = 4
decimal_input = 2.5

integer_results = math_operations(integer_input)
decimal_results = math_operations(decimal_input)

print("Integer Input:", integer_input)
print("Logarithmic Function (log x):", integer_results[0])
print("Exponential Function (exp(x)):", integer_results[1])
print("Power Function with Base 2 (2^x):", integer_results[2])
print("Square Root:", integer_results[3])

print("\nDecimal Input:", decimal_input)
print("Logarithmic Function (log x):", decimal_results[0])
print("Exponential Function (exp(x)):", decimal_results[1])
print("Power Function with Base 2 (2^x):", decimal_results[2])
print("Square Root:", decimal_results[3])
```

Output:

```
Integer Input: 4
Logarithmic Function (log x): 1.3862943611198906
Exponential Function (exp(x)): 54.598150033144236
Power Function with Base 2 (2^x): 16
Square Root: 2.0

Decimal Input: 2.5
Logarithmic Function (log x): 0.9162907318741551
```

Exponential Function (exp(x)): 12.182493960703473
Power Function with Base 2 (2^x): 5.656854249492381
Square Root: 1.5811388300841898

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

Answer:-

```
def extract_first_last_name(full_name):
    name_parts = full_name.split()

    # Assume the first word is the first name
    first_name = name_parts[0]

    # Assume the last word is the last name
    last_name = name_parts[-1]

    return first_name, last_name

# Test the function
full_name1 = "John Doe"
first_name1, last_name1 = extract_first_last_name(full_name1)
print("First Name:", first_name1)  # Output: First Name: John
print("Last Name:", last_name1)    # Output: Last Name: Doe

full_name2 = "Alice Johnson Smith"
first_name2, last_name2 = extract_first_last_name(full_name2)
print("First Name:", first_name2)  # Output: First Name: Alice
print("Last Name:", last_name2)    # Output: Last Name: Smith
```