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

Ans: Built-in Function:

A built-in function is a function that is already available in Python. These functions are part of the Python programming language itself and are ready to use without any additional coding. They are usually implemented in C or Python and are included in the Python standard library. Examples of built-in functions include print(), len(), type(), max(), min(), etc. Here's an example of using the built-in function len() to determine the length of a string:

python

```
string = "Hello, World!"
length = len(string)
print(length)  # Output: 13
```

User-defined Function:

A user-defined function is a function that is created by the user to perform a specific task. These functions are defined using the def keyword, followed by a function name, parentheses for any parameters, and a colon. The function body is then indented and contains the code to be executed when the function is called. Users can define their own functions to encapsulate a block of code and reuse it as needed. Here's an example of a user-defined function that calculates the factorial of a number:

python

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

number = 5
factorial_result = factorial(number)
print(factorial_result)  # Output: 120
```

In the above example, the user-defined function

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

Ans: Positional Arguments:

Positional arguments are passed to a function based on their position or order. The arguments are matched to the function parameters based on their position, and the order in which they are passed matters. When calling a function with positional arguments, the arguments must be provided in the same order as the parameters defined in the function's signature. Here's an example:

python

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

greet("Alice", 25)
```

Keyword Arguments:
Keyword arguments are passed to a function using the name of the parameter along with its corresponding value. The order of the arguments doesn't matter in this case because each argument is explicitly matched to its corresponding parameter. Keyword arguments are useful when you want to skip some arguments or provide them in a different order. Here's an example:
python

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

greet(age=25, name="Alice")
```

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

Ans: The return statement in a function is used to specify the value that the function should return to the caller. It serves two main purposes:

Value Passing: When a function is called, the return statement allows the function to pass a value (or multiple values) back to the caller. The value can be a result of a computation, a processed data structure, or any other information generated within the function. The returned value can then be stored in a variable or used in further computations.

Function Termination: The return statement also acts as a termination point for the function. When the return statement is encountered, it immediately exits the function and returns control to the caller. Any code after the return statement within the function is not executed.

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

Ans: In Python, a lambda function, also known as an anonymous function, is a small and concise function that doesn't require a formal def statement. Lambda functions are defined using the lambda keyword, followed by a list of parameters, a colon, and an expression. The result of the expression is automatically returned by the lambda function.

Lambda functions are different from regular functions in the following ways:

Syntax: Lambda functions have a more compact syntax compared to regular functions. They are typically used for simple, one-line operations.

Nameless: Lambda functions are anonymous, meaning they don't have a name. They are primarily used as inline functions or for passing as arguments to other functions.

Limited functionality: Lambda functions are limited to a single expression and cannot contain multiple statements or complex logic.

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

Ans: In Python, the concept of "scope" refers to the region or context in which a variable or name can be accessed. It determines the visibility and lifetime of variables within a program. When it comes to functions, there are two main scopes to consider: local scope and global scope.

Local Scope:

The local scope, also known as function scope, refers to the region within a function where variables are defined. Variables created inside a function are only accessible within that function's scope. These variables are called local variables. Local variables are created when the function is called and destroyed when the function completes its execution. Local variables cannot be accessed outside the function. Here's an example:

python

```
def my_function():
    x = 10  # Local variable
    print(x)

my_function()  # Output: 10
print(x)  # Error: NameError: name 'x' is not defined
```

In this example, the variable x is defined within the my_function() function. It is a local variable and can only be accessed within the function. Trying to access x outside the function results in a NameError because it is out of scope.

Global Scope:

The global scope refers to the top-level scope of a module or the entire script. Variables defined outside any function or in the global scope are called global variables. Global variables can be accessed from any part of the code, including inside functions. However, if a local variable with the same name is defined within a function, it takes precedence over the global variable within that function. Here's an example:

python

```
x = 10  # Global variable

def my_function():
    x = 20  # Local variable
    print(x)  # Output: 20

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

In this example, we have a global variable x defined outside the function. Inside the my_function() function, a local variable x is defined with a different value. When my_function() is called, it prints the value of the local variable, which is 20. Outside the function, the global variable x retains its original value of 10.

It's important to note that global variables can be accessed within functions, but if you want to modify their values from within a function, you need to use the global keyword to explicitly indicate that the variable is global and not a local variable.

In summary, local scope refers to variables defined within a function, accessible only within that function, while global scope refers to variables defined outside any function, accessible throughout the code, including within functions.

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

Ans: In Python, you can use the return statement in a function to return multiple values by separating them with commas. The values are packed into a tuple and returned as a single object. Here's an example:

python

```
def get_person_details():
    name = "Alice"
    age = 25
    city = "New York"
    return name, age, city

person_info = get_person_details()
print(person_info)  # Output: ('Alice', 25, 'New York')
print(person_info[0])  # Output: Alice
print(person_info[1])  # Output: 25
print(person_info[2])  # Output: New York
```

In this example, the get_person_details() function defines three variables: name, age, and city. Instead of returning them individually, the return statement returns all three values separated by commas. When the function is called, it returns a tuple containing the three values.

The returned tuple can be assigned to a single variable (person_info in this case), and individual values can be accessed using indexing (person_info[0], person_info[1], etc.).

If you prefer, you can also unpack the returned values into separate variables when calling the function, like this:

python

```
name, age, city = get_person_details()
print(name)  # Output: Alice
print(age)  # Output: 25
print(city)  # Output: New York
```

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

Ans: In Python, the concepts of "pass by value" and "pass by reference" are often used to describe how arguments are passed to functions. However, it's important to note that Python employs a different mechanism known as "passing by object reference" or "call by object reference."

In Python, when you pass an argument to a function, you are actually passing a reference to the object. This means that changes made to the object within the function can affect the original object outside the function. However, the distinction between "pass by value" and "pass by reference" becomes important when dealing with mutable and immutable objects.

Pass by Value (Immutable Objects):

When you pass an immutable object (e.g., numbers, strings, tuples) as an argument to a function in Python, it behaves like "pass by value." This means that a copy of the value is created, and any modifications made to the argument within the function will not affect the original object. Here's an example:

python

```
def modify_value(num):
    num += 10
    print("Inside function:", num)

value = 5
modify_value(value)
print("Outside function:", value)
```

In this example, the modify_value() function takes an integer argument num. Inside the function, 10 is added to num, but the original value variable remains unchanged outside the function. The output will be:

bash

```
Inside function: 15
Outside function: 5
```

Pass by Reference (Mutable Objects):

When you pass a mutable object (e.g., lists, dictionaries) as an argument to a function in Python, it behaves like "pass by reference." This means that the function operates on the original object itself, and any modifications made to the object within the function will be reflected outside the function. Here's an example:

python

```
def modify_list(my_list):
    my_list.append(4)
    print("Inside function:", my_list)

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

In this example, the modify_list() function takes a list argument my_list. Inside the function, the value 4 is appended to my_list, and this change is reflected in the original my_list variable outside the function. The output will be:

bash

```
Inside function: [1, 2, 3, 4]
Outside function: [1, 2, 3, 4]
```

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

Ans:

```
import math
def math_operations(x):
    logarithmic = math.log(x)
    exponential = math.exp(x)
    power = 2 ** x
    square_root = math.sqrt(x)
    return logarithmic, exponential, power, square_root
```

```
# Example usage:
result = math_operations(4)
print("Logarithmic:", result[0])
print("Exponential:", result[1])
print("Power (2^x):", result[2])
    print("Square Root:", result[3])
```

9.Create a function that takes a full name as an argument and returns first name and last name.
Ans: Certainly! Here's an example of a function that takes a full name as an argument and returns the first name and last name:

python

```
def get_first_and_last_name(full_name):
    names = full_name.split()
    first_name = names[0]
    last_name = names[-1]
    return first_name, last_name

# Example usage:
name = "John Doe"
first_name, last_name = get_first_and_last_name(name)
print("First Name:", first_name)
print("Last Name:", last_name)
```

In this example, the get_first_and_last_name() function takes a single argument full_name, which represents the full name. The split() method is used to split the full name into individual names based on the whitespace delimiter.

The names variable stores a list of names obtained by splitting the full name.
The first_name variable is assigned the first name from the list (names[0]).
The last_name variable is assigned the last name from the list (names[-1]).
Finally, the function returns the first name and last name as a tuple.

In the example usage, the function is called with name set as "John Doe". The returned tuple is unpacked into the variables first_name and last_name. The values are then printed accordingly.

You can call the get_first_and_last_name() function with different full names to extract the first name and last name from them.