ASSIGNMENT 8

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

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

1**. Built-in Function:**

Origin: Built-in functions are functions that are included as part of the Python programming language. They are available for use without the need to define or import them separately.

Purpose: Built-in functions serve common and essential tasks and operations in Python, such as mathematical operations, string manipulation, input/output, and data type conversion.

Example of a built-in function:

```
                                        # Using the built-in len() function to get the length of a string
text = "Hello, World!"
length = len(text)
print(length)                         # Output: 13
```

2. User-Defined Function:

Origin: User-defined functions are functions that programmers create themselves. These functions are defined by the user to perform specific tasks or encapsulate a set of statements into a reusable block of code.

Purpose: User-defined functions are used to modularize code, improve readability, and promote code reusability by breaking down a program into smaller, manageable functions.

Example of a user-defined function:

```
                              # Defining a user-defined function to calculate the square of a number
def square(num):
    return num * num
                              # Using the user-defined function to calculate the square of 5
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.**

In Python, we can pass arguments to a function when we call it. There are two primary ways to pass arguments to a function: positional arguments and keyword arguments

1. **Positional Arguments:**

- Positional arguments are the most common way to pass arguments to a function.

- When you use positional arguments, the values you pass are matched to the function's parameters based on their position or order.
- The order and number of positional arguments must match the order and number of parameters defined in the function.

Example of using positional arguments:

```
def add(x, y):
    return x + y

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

In this example, 3 is the first positional argument, which corresponds to the x parameter in the function, and 5 is the second positional argument, which corresponds to the y parameter.

2. Keyword Arguments:
- Keyword arguments allow you to specify which argument corresponds to which parameter by using the parameter's name.
- This method is more explicit and flexible, as it allows you to pass arguments in any order and skip optional parameters.

Example of using keyword arguments:

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

result = greet(message="How are you?", name="Alice")
print(result)                                          # Output: "Hello, Alice! How are you?"
```

In this example, we use keyword arguments to specify the parameter names (message and name) along with the values we're passing to the function. The order of the arguments doesn't matter.
We can also mix both positional and keyword arguments, but positional arguments must come before keyword arguments:

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

result = greet("Bob", message="How's it going?")
print(result)  # Output: "Hello, Bob! How's it going?"
```

The key difference between positional and keyword arguments is how you pass them to a function. Positional arguments rely on the order, while keyword arguments use parameter names for explicit assignment. The choice between them depends on specific use case and code readability.

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

The return statement in a function serves the purpose of specifying the value or values that the function should produce as output. When a return statement is encountered in a function, it immediately exits the function and returns the specified value(s) to the caller. In Python, a function can have one or more return statements, but only one of them will be executed during the function's execution.

Here's an explanation of the purpose of the return statement and an example demonstrating multiple return statements:

Purpose of the return statement**:**
- To send a value or a set of values back to the calling code or program.
- To terminate the execution of the function.
- It allows you to produce a result or outcome based on the function's calculations or operations.

Example with multiple return statements:

```
def divide(x, y):
    if y == 0:
        return "Division by zero is not allowed."           # First return statement
    else:
        result = x / y
        return result                                        # Second return statement

                                        # Calling the function with different inputs
result1 = divide(10, 2)
print(result1)                          # Output: 5.0

result2 = divide(8, 0)
print(result2)                          # Output: "Division by zero is not allowed."
```

In this example, the divide function has two return statements. The first return statement is executed if y is equal to 0, which returns a string message indicating that division by zero is not allowed. The second return statement is executed if y is not equal to 0, which calculates and returns the result of the division operation.

When call the function with divide(10, 2), the second return statement is executed, and 5.0 is returned as the result. When you call the function with divide(8, 0), the first return statement is executed, and the division by zero error message is returned.

a function can have multiple return statements, but only one of them will be executed based on the condition or logic within the function. The purpose of the return statement is to provide a result or outcome from the function's calculations or operations.

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

Lambda functions in Python are small, anonymous, and inline functions that are defined using the lambda keyword. They are also known as lambda expressions or anonymous functions because they do not have a name like regular functions defined with the def keyword. Lambda functions are typically used for simple operations and are often used in situations where you need a quick, throwaway function.

Here's the basic syntax of a lambda function:

lambda arguments: expression

Lambda functions can have one or more arguments but can only contain a single expression. The result of evaluating the expression is returned as the result of the lambda function. Lambda functions are often used as arguments to higher-order functions like map(), filter(), and reduce().

Here's an example of a lambda function that calculates the square of a number:

```
square = lambda x: x * x
result = square(5)
print(result)                    # Output: 25
```

In this example, lambda x: x * x defines a lambda function that takes one argument x and returns x * x. The square variable is assigned this lambda function, and we can then call square(5) to calculate the square of 5.

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

In Python, the concept of "scope" refers to the region of a program where a particular variable is accessible and can be used. Python has two primary types of scopes: local scope and global scope, and the scope of a variable determines where it can be accessed and modified within your code.

1. Local Scope:

- Local scope, as the name suggests, is the most restricted scope. Variables defined within a function are considered to have local scope.
- These variables are only accessible and visible within the function where they are defined. They are essentially confined to the block of code where they were created.
- Variables in local scope are often referred to as "local variables."
- Once the function finishes executing, local variables go out of scope and are typically destroyed, making them inaccessible from outside the function.

Example of local scope:

```
def my_function():
    x = 10                       # x has a local scope within my_function
    print(x)
my_function()                    # Output: 10
print(x)                         # Raises a NameError because x is not defined in the global scope
```

2. Global Scope:

- Global scope encompasses the entire Python program or module. Variables defined outside of any function or class have global scope.
- Global variables can be accessed from anywhere in the code, both inside and outside functions.
- However, if you want to modify a global variable from within a function, you need to explicitly declare it as "global" using the global keyword. Otherwise, Python will create a new local variable with the same name inside the function's scope, leaving the global variable unchanged.

Example of global scope:

```
y = 20  # y has global scope
def another_function():
    global y  # Declare y as global to modify the global variable
    y = 30
another_function()
print(y)  # Output: 30 (global variable y has been modified)
```

In summary, "scope" in Python determines where a variable is visible and accessible. Local scope applies to variables defined within functions, making them accessible only within that function. Global scope applies to variables defined outside functions, making them accessible from anywhere in the program, with the option to modify them using the `global` keyword within functions. Understanding scope is crucial for managing variables and avoiding naming conflicts in your Python code.

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

In Python, you can use the `return` statement in a function to return multiple values by packaging them into a data structure such as a tuple, list, or dictionary. Here are several approaches to achieve this:

1. Using a Tuple:

You can return multiple values as a tuple. Simply separate the values with commas in the `return` statement, and Python will automatically pack them into a tuple.

```
def multiple_values():
    x = 10
    y = 20
    z = 30
    return x, y, z
result = multiple_values()
print(result)  # Output: (10, 20, 30)
```

2. Using a List:

Similar to using a tuple, you can also return multiple values as a list.

```
def multiple_values():
    x = 10
    y = 20
    z = 30
    return [x, y, z]
result = multiple_values()
print(result)  # Output: [10, 20, 30]
```

3. Using a Dictionary:

If you want to return multiple values with labels or names, you can use a dictionary to store and return them.

```
def multiple_values():
    x = 10
    y = 20
    z = 30
    return {'x': x, 'y': y, 'z': z}
result = multiple_values()
print(result)  # Output: {'x': 10, 'y': 20, 'z': 30}
```

4. Unpacking Values:

When you call the function, you can unpack the returned tuple or list into individual variables.

```
def multiple_values():
    x = 10
    y = 20
    z = 30
    return x, y, z
a, b, c = multiple_values()
print(a)  # Output: 10
print(b)  # Output: 20
print(c)  # Output: 30
```

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, function arguments are neither strictly "pass by value" nor "pass by reference" in the same way as some other programming languages like C++ or Java. Instead, Python uses a different mechanism called "pass by object reference" or "call by object reference." This means that when you pass an argument to a function in Python, you are passing a reference to the object (value), not the actual object itself. However, the behavior can be a bit different depending on whether the object is mutable or immutable.

pass by value :Immutable Objects (e.g., integers, strings, tuples): When pass an immutable object as an argument to a function in Python, we are essentially passing a copy of the object's reference. Inside the function, if reassign the parameter to a new value, it does not affect the original object outside the function. This behavior is often described as "pass by value" because you cannot modify the original object.

```
def modify_immutable(value):
    value = 42                    # This reassigns the local reference, doesn't affect the original value
```

```
x = 10
modify_immutable(x)
print(x)                    # Output: 10 (unchanged)
```

pass by Reference-Mutable Objects (e.g., lists, dictionaries, sets): When you pass a mutable object as an argument to a function, you are still passing a reference to the object. However, any modifications made to the object inside the function are reflected in the original object outside the function. This behavior is similar to "pass by reference" because changes to the object are visible in both the calling and called functions.

```
def modify_mutable(my_list):
    my_list.append(42)  # This modifies the original list

my_numbers = [1, 2, 3]
modify_mutable(my_numbers)
print(my_numbers)                    # Output: [1, 2, 3, 42] (modified)
```

In summary, the key distinction is not strictly "pass by value" or "pass by reference" but rather whether the object being passed is mutable or immutable:

- Immutable objects are effectively passed by value because changes inside the function do not affect the original object.
- Mutable objects are effectively passed by reference because changes inside the function are reflected in the original object.

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**

```python
import math
def math_operations(value):
    flag=0
    try:
        value = float(value)                # Convert the input to a float to handle
both integers and decimals
    except  Exception as ex:
        print( "Invalid input. Please provide a numeric value.",ex)
        flag=1
        return flag
    result = {
        "Logarithmic (log x)": math.log(value),
        "Exponential (exp(x))": math.exp(value),
        "Power with base 2 (2^x)": math.pow(2, value),
        "Square Root (√x)": math.sqrt(value)
    }
    return result
input_value = 10                                        # You can change this
to any numeric value
operations_result = math_operations(input_value)
if operations_result!=1:
    for operation, result in operations_result.items():
        print(f"{operation}: {result}")
```

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

```python
def first_last(name):

    list_name=name.split()
    if len(list_name)>=2:
        return list_name[0]+" "+list_name[-1]
    else:
        return list_name[0]


name=input("enter your name: ")
only_first_last=first_last(name)
print(only_first_last)
```