***ASSIGNMENT\_8***

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

***ANS***

In Python, a built-in function is one that is already defined by Python itself. These functions are always available for use without needing to be imported or defined by the user. An example of a built-in function is len(), which returns the length (the number of items) of an object.

Here's an example of how to use len():

***list = [1, 2, 3, 4, 5]***

***print(len(list)) # This will output: 5***

a user-defined function is a function that is defined by the user themselves. These functions are created to perform a specific task that might not be covered by the built-in functions, or to simplify a complex task into smaller, more manageable parts.

Here's an example of a user-defined function:

***def add\_numbers(a, b):***

***return a + b***

***print(add\_numbers(3, 4)) # This will output: 7***

In this example, add\_numbers is a function defined by the user to add two numbers together. The function takes two parameters, a and b, and returns their sum.

Remember, while built-in functions are always available for use, user-defined functions need to be defined in the code before they can be used.

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

***ANS***

Positional Arguments: These are arguments that need to be passed in the correct positional order in which they are defined in the function. The function associates the values according to their position***.***

**def greet(name, greeting):**

**print(f"{greeting}, {name}!")**

**greet("Alice", "Hello") # Outputs: Hello, Alice*!***

Keyword Arguments: These are arguments that are passed to a function by explicitly naming each one along with its value. This allows you to specify arguments in any order without regard to their position in the function definition.

**def greet(name, greeting):**

**print(f"{greeting}, {name}!")**

**greet(greeting="Hello", name="Alice") # Outputs: Hello, Alice!**

In this example, we're using keyword arguments to specify the values for name and greeting. Even though we've listed greeting before name, the function still produces the correct output because we've explicitly named each argument.

It's also worth noting that you can mix positional and keyword arguments in a function call, but positional arguments must always come before keyword arguments.

**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 Python function serves the purpose of ending the execution of the function and "returning" a result to the caller. If the return statement includes an expression (like a variable or a value), the function will return that value to the caller. If there's no expression, or the return statement itself is absent, the function will return None.

Yes, a function can have multiple return statements. When a function encounters a return statement, it will immediately exit the function, regardless of any code that comes after. This can be useful in certain scenarios, such as when you want to exit a function early based on a certain condition.

Here's an example:

**def compare\_to\_five(n):**

**if n > 5:**

**return "Greater than 5"**

**elif n < 5:**

**return "Less than 5"**

**else:**

**return "Equal to 5"**

**print(compare\_to\_five(7)) # Outputs: Greater than 5**

**print(compare\_to\_five(3)) # Outputs: Less than 5**

**print(compare\_to\_five(5)) # Outputs: Equal to 5**

In this example, the function compare\_to\_five has three return statements. Depending on the input value, it will return a different string. As soon as one of the return statements is executed, the function ends, and no further code within the function is 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:**

Lambda functions, also known as anonymous functions, are a way of creating small, one-time, unnamed functions in Python. They are defined using the lambda keyword, hence the name.

The main differences between lambda functions and regular functions are:

Definition: Regular functions are defined using the def keyword, while lambda functions are defined using the lambda keyword.

Name: Regular functions have a name, while lambda functions are anonymous.

Size: Lambda functions are typically much shorter than regular functions. They are limited to a single expression and cannot include complex logic.

Return: Lambda functions automatically return the result of their expression, while regular functions need a return statement to return a value.

Here's an example of a lambda function:

**square = lambda x: x \*\* 2**

**print(square(5)) # Outputs: 25**

In this example, square is a lambda function that takes one argument x and returns x squared. It's equivalent to the following regular function:

**def square(x):**

**return x \*\* 2**

Lambda functions can be useful in many situations, especially when you need a small function for a short period of time, and you don't want to define a full function. A common use case is with functions like map(), filter(), and sort(), which take a function as an argument.

For example, if you want to sort a list of strings by their length, you could use a lambda function:

**words = ['apple', 'banana', 'cherry', 'date']**

**words.sort(key=lambda word: len(word))**

**print(words) # Outputs: ['date', 'apple', 'cherry', 'banana']**

In this example, the lambda function takes a word and returns its length. The sort() method uses this function to determine the order of the words.

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

In Python, "scope" refers to the region of the program where a variable is defined and can be accessed. Scope helps to determine the visibility of a variable in the lines of code. There are two types of scopes in Python: local and global.

Local Scope: A variable that's defined inside a function has a local scope. It's only accessible within that function, not outside it. Each function has its own local scope, so a local variable defined in one function is not accessible from another function.

**def my\_function():**

**local\_var = "I'm local!"**

**print(local\_var)**

**my\_function() # Outputs: I'm local!**

**print(local\_var) # Raises a NameError**

In this example, local\_var is a local variable. It's defined inside my\_function, so it can be accessed within my\_function. But if we try to access local\_var outside my\_function, we get a NameError because local\_var is not defined in the global scope.

Global Scope: A variable that's defined in the main body of a Python script (not inside a function or a class) is a global variable. It has a global scope, which means it can be accessed from anywhere in the code, including inside functions (unless a local variable with the same name is defined).

**Here's an example:**

**global\_var = "I'm global!"**

**def my\_function():**

**print(global\_var)**

**my\_function() # Outputs: I'm global!**

**print(global\_var) # Outputs: I'm global!**

In this example, global\_var is a global variable. It's defined in the main body of the script, so it can be accessed both inside my\_function and outside it.

Remember, it's generally best to avoid using global variables when possible, as they can make code harder to understand and debug. It's usually better to pass variables as function arguments if you need to use them in a function.

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

Returning Multiple Values from a Function: In Python, you can return multiple values from a function using tuples. When a function needs to return multiple values, you can return them as a tuple, which is an ordered and immutable sequence type. Here's an example:

**def calculate\_numbers(a, b):**

**sum = a + b**

**difference = a - b**

**product = a \* b**

**quotient = a / b**

**return sum, difference, product, quotient**

**result = calculate\_numbers(10, 2)**

**print(result) # Outputs: (12, 8, 20, 5.0)**

In this example, the function calculate\_numbers returns four values. When the function is called, it returns a tuple containing all four values.

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

Pass by Value vs. Pass by Reference: In Python, the distinction between pass by value and pass by reference is a bit different than in some other languages.

Pass by Value: In some languages, pass by value means that a copy of the variable is passed into the function. Changes to the copy do not affect the original variable. However, Python doesn't have a true "pass by value" behavior for its objects.

Pass by Reference: In some languages, pass by reference means that a reference to the variable is passed into the function. Changes to the reference do affect the original variable. In Python, the behavior is somewhat similar. When you pass a mutable object (like a list or a dictionary) to a function, the function gets a reference to the object, so if the function modifies the object, the changes are visible outside the function.

Here's an example:

**def add\_element(my\_list):**

**my\_list.append('New element')**

**my\_list = ['Old element']**

**add\_element(my\_list)**

**print(my\_list) # Outputs: ['Old element', 'New element']**

In this example, my\_list is a mutable object. When we pass it to add\_element, the function modifies my\_list by appending a new element. This change is visible outside the function, which shows that my\_list was passed by reference.

However, if you pass an immutable object (like an integer or a string) to a function, the function can't modify the object itself, but it can reassign the reference to a new object. This won't affect the original object.

Here's an example:

**def reassign\_value(my\_int):**

**my\_int = 10**

**my\_int = 5**

**reassign\_value(my\_int)**

**print(my\_int) # Outputs: 5**

In this example, my\_int is an immutable object. When we pass it to reassign\_value, the function tries to reassign my\_int to a new value. However, this doesn't affect the original my\_int variable, which shows that my\_int was not truly passed by reference.

**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**

**ANS**:

import math

**def perform\_operations(x):**

**# Logarithmic function**

**if x > 0:**

**log\_x = math.log(x)**

**else:**

**log\_x = "Logarithm not defined for non-positive values"**

**# Exponential function**

**exp\_x = math.exp(x)**

**# Power function with base 2**

**power\_x = 2 \*\* x**

**# Square root**

**if x >= 0:**

**sqrt\_x = math.sqrt(x)**

**else:**

**sqrt\_x = "Square root not defined for negative values"**

**return log\_x, exp\_x, power\_x, sqrt\_x**

**# Test the function**

**result = perform\_operations(4)**

**print(result)**

In this function, perform\_operations takes a number x as input. It then calculates the logarithm of x (if x is positive), the exponential of x, 2 to the power of x, and the square root of x (if x is non-negative). The function returns all four results as a tuple.

Please note that the logarithm and square root functions are not defined for non-positive and negative values respectively. In such cases, the function will return a message indicating this.

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

**def split\_name(full\_name):**

**first\_name, last\_name = full\_name.split(' ')**

**return first\_name, last\_name**

**You can use this function like this:**

**first, last = split\_name('John Doe')**

**print('First Name:', first)**

**print('Last Name:', last)**

This function assumes that the full name is composed of two parts: the first name and the last name, separated by a space. If the full name has a middle name or multiple last names, you might need a more complex function to handle those cases.