**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, both built-in functions and user-defined functions are used to perform specific tasks or operations. However, there are key differences between these two types of functions.

Built-in Functions:

Built-in functions are pre-defined functions that come with Python. These functions are readily available and can be used without the need for any special definitions or imports. Python provides a wide range of built-in functions that cover various aspects of programming, such as mathematical calculations, data manipulation, string operations, etc.

Here's an example of a built-in function:

my\_list = [2, 4, 6, 8, 10]

length = len(my\_list)

print("Length of the list:", length)

In this example, len() is a built-in function that calculates the length of the list my\_list.

User-Defined Functions:

User-defined functions, as the name suggests, are functions created by the programmer. These functions are defined using the def keyword, and they allow you to encapsulate a block of code to perform a specific task. User-defined functions can be customized to fit the requirements of a program, and they help in organizing and modularizing code.

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

def calculate\_square(x):

return x \* x

number = 5

square = calculate\_square(number)

print("Square of", number, "is", square)

In this example, calculate\_square() is a user-defined function that calculates the square of a number.

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

**arguments and keyword arguments.**

ANS; In Python, you can pass arguments to a function in order to provide input data or values that the function operates on. There are two primary ways to pass arguments to a function: positional arguments and keyword arguments.

Positional Arguments:

Positional arguments are passed to a function based on their position or order. The order in which you provide the arguments matters, as they are matched to the function's parameters in the same order. These arguments are directly associated with the function's parameters based on their positions.

Here's an example of using positional arguments:

def add\_numbers(a, b):

return a + b

result = add\_numbers(3, 5)

print("Result:", result)

Keyword Arguments:

Keyword arguments allow you to pass arguments to a function using parameter names as keys. This way, the order of the arguments doesn't matter, and you explicitly specify which value corresponds to which parameter. This can make the code more readable and less prone to errors, especially when dealing with functions that have multiple parameters.

Here's an example of using keyword arguments:

def divide\_numbers(dividend, divisor):

return dividend / divisor

result = divide\_numbers(dividend=10, divisor=2)

print("Result:", result)

You can mix positional and keyword arguments, but positional arguments must always come before keyword arguments.

result = divide\_numbers(10, divisor=2)

**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 produce as its output. When a return statement is encountered in a function, the function execution immediately stops, and the value provided in the return statement is sent back as the result of the function call. The purpose of the return statement is to provide the computed or processed result to the caller of the function.

A function can indeed have multiple return statements. However, once a return statement is executed, the function exits, and no further code within the function is executed.

Here's an example to illustrate the concept:

def divide\_numbers(dividend, divisor):

if divisor == 0:

return "Cannot divide by zero."

else:

result = dividend / divisor

return result

result1 = divide\_numbers(10, 2)

result2 = divide\_numbers(10, 0)

print(result1

print(result2)

In this example, the divide\_numbers function takes two arguments: dividend and divisor. It checks if the divisor is zero and returns an error message if it is, or it calculates and returns the division result if it's not zero. This example demonstrates how multiple return statements can be used in a function to handle different scenarios.

**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 feature in Python that allows you to create small, inline functions without explicitly defining them using the def keyword. They are primarily used for simple operations and are often used when a function is needed for a short period of time or within a specific context.

Here are the key differences between lambda functions and regular functions:

Lambda Functions:

Lambda functions are defined using the lambda keyword.

They can have any number of arguments but only one expression.

They are often used for simple operations or as arguments to higher-order functions.

They do not require a return statement. The result of the expression is automatically returned.

Regular Functions:

Regular functions are defined using the def keyword.

They can have multiple statements, multiple arguments, and can include complex logic.

They are used for more complex tasks and for code that needs to be reused multiple times.

They require a return statement to return a value.

Here's an example where a lambda function can be useful. Let's say you have a list of numbers and you want to sort them based on their remainder when divided by 3:

numbers = [15, 8, 12, 7, 10, 5, 6]

def remainder\_sort\_key(num):

return num % 3

sorted\_numbers = sorted(numbers, key=remainder\_sort\_key)

print("Sorted numbers using regular function:", sorted\_numbers)

Now, let's achieve the same result using a lambda function:

numbers = [15, 8, 12, 7, 10, 5, 6]

sorted\_numbers\_lambda = sorted(numbers, key=lambda num: num % 3)

print("Sorted numbers using lambda function:", sorted\_numbers\_lambda)

**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 of the code where a variable is accessible. Variables in Python have different levels of visibility based on where they are defined, and this concept of scope plays a crucial role in how variables are used and accessed within functions and the broader code.

There are two main types of scope in Python: local scope and global scope.

Local Scope:

Local scope refers to the scope within a function or a code block. Variables defined inside a function are considered to have local scope and are only accessible within that function.

Local variables are created when the function is called and destroyed when the function exits.

Variables defined in a local scope are not accessible outside the function in which they are defined.

Example of local scope:

def my\_function():

x = 10

print(x)

my\_function()

)

Global Scope:

Global scope refers to the scope that covers the entire program or module. Variables defined outside any function or code block are considered to have global scope and can be accessed throughout the code.

Global variables are created when the program starts and persist until the program ends.

Variables defined in the global scope can be accessed and modified from within functions, but you need to use the global keyword to indicate that you are referring to a global variable.

Example of global scope:

y = 20

def another\_function():

print(y)

another\_function()

print(y)

**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 simply separating the values with commas. When you return multiple values, they are actually packed into a tuple, and you can then unpack them when you receive the return value.

Here's how you can use the return statement to return multiple values from a function:

def get\_values():

x = 10

y = 20

z = 30

return x, y, z

result = get\_values()

x\_result, y\_result, z\_result = result

print("x:", x\_result)

print("y:", y\_result)

print("z:", z\_result)

In this example, the get\_values() function returns three variables: x, y, and z. When the function is called and the return values are stored in the result variable, they are automatically packed into a tuple. Then, we unpack the tuple into separate variables x\_result, y\_result, and z\_result.

**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 not quite accurate descriptions of how function arguments are passed. Python uses a different approach, which can be understood as "passing objects by reference."

Pass by Value:

In a "pass by value" scenario, a copy of the value of the argument is passed to the function. This means that changes made to the argument within the function do not affect the original value outside the function.

Pass by Reference:

In a "pass by reference" scenario, a reference to the original variable is passed to the function. Any changes made to the argument within the function directly affect the original value outside the function.

However, in Python, the situation is more nuanced:

Python passes function arguments using a mechanism often called "call by object reference" or "pass by object reference." It means that when you pass an argument to a function, you are passing a reference to the object, not a copy of the object itself. This can lead to behaviors that are similar to "pass by reference," but there are distinctions due to how Python handles different types of objects.

Immutable Objects (e.g., numbers, strings, tuples):

When you pass an immutable object to a function and modify it within the function, a new object is actually created, and the original object remains unchanged outside the function.

Mutable Objects (e.g., lists, dictionaries, custom objects):

When you pass a mutable object to a function and modify it within the function, the changes are reflected in the original object outside the function, because you're working with the same object in memory.

Here's a simple example to illustrate:

def modify\_immutable(x):

x = x + 1

print("Inside function:", x)

num = 10

modify\_immutable(num)

print("Outside function:", num)

def modify\_mutable(lst):

lst.append(4)

print("Inside function:", lst)

my\_list = [1, 2, 3]

modify\_mutable(my\_list)

print("Outside function:", my\_list)

**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**; that takes an input value (integer or decimal) and performs the specified operations:

import math

def math\_operations(value):

logarithmic\_result = math.log(value)

exponential\_result = math.exp(value)

power\_base2\_result = 2 \*\* value

square\_root\_result = math.sqrt(value)

print(f"Input value: {value}")

print(f"Logarithmic function (log x): {logarithmic\_result}")

print(f"Exponential function (exp(x)): {exponential\_result}")

print(f"Power function with base 2 (2^x): {power\_base2\_result}")

print(f"Square root: {square\_root\_result}")

math\_operations(3)

math\_operations(0.5)

In this example, the math\_operations function takes an input value and calculates the logarithmic function, exponential function, power function with base 2, and square root using the math module's functions. The results are then printed for each operation.

Keep in mind that some of these operations might have limitations or undefined results for certain input values, so it's a good practice to handle exceptions or validate the input as needed.

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

ANS; def extract\_first\_last\_name(full\_name):

parts = full\_name.split()

first\_name = parts[0]

last\_name = parts[-1] if len(parts) > 1 else ""

return first\_name, last\_name

full\_name1 = "John Smith"

first\_name1, last\_name1 = extract\_first\_last\_name(full\_name1)

print("First Name:", first\_name1)

print("Last Name:", last\_name1)

full\_name2 = "Mary"

first\_name2, last\_name2 = extract\_first\_last\_name(full\_name2)

print("First Name:", first\_name2)

print("Last Name:", last\_name2)

In this example, the extract\_first\_last\_name function splits the input full name using spaces and extracts the first part as the first name and the last part as the last name (if available). If the full name consists of only one part (no spaces), the function treats it as the first name and returns an empty string as the last name.