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

**Solution 1.**

In Python, a built-in function is a function that is provided by the Python programming language itself. These functions are readily available and can be used without the need for any additional setup or import statements. Examples of built-in functions in Python include print(), len(), type(), and range().

On the other hand, a user-defined function is a function that is defined by the user to perform a specific task or set of tasks. These functions are created using the def keyword, and the user can define their own custom functionality. User-defined functions can be reused and called multiple times within a program. They provide modularity and allow for code organization and reusability.

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

**Solution 2.**

In Python, arguments can be passed to a function in two ways: positional arguments and keyword arguments.

**Positional Arguments:**

Positional arguments are passed to a function based on their position or order. The values are matched with the function parameters in the same order. The number of arguments and their positions must match the function definition.

Here's an example:

def greet(name, age):

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

greet("Alice", 25) # Positional arguments

**Keyword Arguments:**

Keyword arguments are passed to a function using the argument names along with their values. The order of the arguments doesn't matter in this case because they are explicitly assigned based on their names.

Here's an example:

def greet(name, age):

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

greet(age=25, name="Bob") # Keyword arguments

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

**Solution 3**

The purpose of the return statement in a function is to specify the value that the function should return when it is called. The return statement allows the function to send a value back to the caller, which can be stored in a variable or used in further computations.

A function can have multiple return statements. When a return statement is encountered in a function, the execution of the function is immediately stopped, and the specified value is returned. Having multiple return statements can be useful in scenarios where different conditions may lead to different return values.

Here's an example to illustrate the concept:

def check\_even\_odd(num):

if num % 2 == 0:

return "Even"

else:

return "Odd"

result1 = check\_even\_odd(4)

print(result1) # Output: Even

result2 = check\_even\_odd(7)

print(result2) # Output: Odd

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

**Solution 4.**

Lambda functions in Python are small anonymous functions that are defined without a name. They are also known as lambda expressions. Lambda functions are typically used when we need a simple function for a short period of time and don't want to define a separate named function.

The syntax for a lambda function is:

lambda arguments: expression

Lambda functions can take any number of arguments, but they can only have a single expression. The result of the expression is automatically returned when the lambda function is called.

Lambda functions are particularly useful in situations where we need to pass a function as an argument to another function, such as in the case of sorting or filtering operations. They can also be used in combination with built-in functions like map(), filter(), or reduce().

For example, consider the following code that uses the map() function and a lambda function to compute the squares of a list of numbers:

numbers = [1, 2, 3, 4, 5]

squared\_numbers = list(map(lambda x: x \* x, numbers))

print(squared\_numbers) # Output: [1, 4, 9, 16, 25]

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

**Solution 5.**

In Python, the concept of "scope" refers to the region of a program where a particular variable is defined and can be accessed. It determines the visibility and lifetime of variables within a program.

When it comes to functions in Python, each function creates its own local scope. This means that variables defined within a function are only accessible within that function's scope. These variables are called local variables. Local variables have a limited lifespan and are created when the function is called and destroyed when the function finishes executing.

On the other hand, global scope refers to the outermost scope of a program, outside of any function. Variables defined in the global scope are accessible from anywhere in the program, including within functions. These variables are called global variables. Global variables have a longer lifespan and are created when the program starts and destroyed when the program ends.

The key difference between local scope and global scope is the visibility and accessibility of variables. Local variables are only visible within the function that defines them, while global variables are visible to all functions and statements within the program.

Here's an example to illustrate the concept of scope:

def my\_function():

local\_var = 10 # local variable

print(local\_var)

global\_var = 20 # global variable

my\_function() # Output: 10

print(global\_var) # Output: 20

print(local\_var) # Error: NameError: name 'local\_var' is not defined

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

**Solution 6.**

In Python, the return statement in a function is typically used to return a single value. However, you can also use it to return multiple values by returning them as a collection, such as a tuple or a list. Here's an example:

def get\_values():

value1 = 10

value2 = 'Hello'

value3 = [1, 2, 3]

return [value1, value2, value3]

result = get\_values()

print(result) # Output: [10, 'Hello', [1, 2, 3]]

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

**Solution 7.**

In Python, the concept of "pass by value" and "pass by reference" can be a bit different compared to other programming languages. Python uses a "pass by object reference" model, which can be a combination of both pass by value and pass by reference.

In Python, when a function is called and arguments are passed to it, the values of the arguments are actually references to objects. These references are passed to the function, which means that changes made to the objects within the function can affect the objects outside the function. However, how these changes are reflected depends on the nature of the objects being passed.

Immutable Objects (Pass by Value-like behavior):

Immutable objects such as numbers, strings, and tuples cannot be modified once created. When these objects are passed as arguments to a function, a new reference to the object is created within the function's local scope. Any modifications made to the reference within the function do not affect the original object outside the function.

def modify\_value(num):

num += 10 # Changes made to the reference within the function

# do not affect the original object

value = 5

modify\_value(value)

print(value) # Output: 5

Mutable Objects (Pass by Reference-like behavior):

Mutable objects such as lists, dictionaries, and custom objects can be modified. When these objects are passed as arguments to a function, a reference to the original object is passed. Modifications made to the object within the function will affect the original object outside the function.

def modify\_list(lst):

lst.append(4) # Changes made to the reference affect the original list

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Output: [1, 2, 3, 4]

1. Create a function that can intake integer or decimal value and do following operations:
   1. Logarithmic function (log x)
   2. Exponential function (exp(x))
   3. Power function with base 2 (2x)
   4. Square root

**Solution 8.**

Here's an example of a function that can perform the logarithmic, exponential, power (with base 2), and square root operations on an integer or decimal value:

import math

def math\_operations(x):

log\_result = math.log(x)

exp\_result = math.exp(x)

power\_result = math.pow(2, x)

sqrt\_result = math.sqrt(x)

return log\_result, exp\_result, power\_result, sqrt\_result

# Test the function

value = 4.0

logarithm, exponential, power, square\_root = math\_operations(value)

print("Logarithm of", value, ":", logarithm)

print("Exponential of", value, ":", exponential)

print("Power of 2 with exponent", value, ":", power)

print("Square root of", value, ":", square\_root)

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

**Solution 9.**

Here's an example of a function that takes a full name as an argument and returns the first name and last name:

def get\_first\_last\_name(full\_name):

# Split the full name into a list of names

names = full\_name.split()

# Extract the first name and last name

first\_name = names[0]

last\_name = names[-1]

return first\_name, last\_name

# Test the function

full\_name = "John Doe"

first\_name, last\_name = get\_first\_last\_name(full\_name)

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

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