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

In Python, a built-in function refers to a function that is already provided by the Python programming language. These functions are readily available and can be used without any additional steps or defining them explicitly. On the other hand, a user-defined function is created by the programmer to perform a specific task or set of tasks.

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

# Example of a built-in function: len()

my\_list = [1, 2, 3, 4, 5]

length = len(my\_list)

print(length) # Output: 5

# Example of a user-defined function: greet()

def greet(name):

print(f"Hello, {name}!")

# Calling the user-defined function

greet("Alice") # Output: Hello, Alice!

greet("Bob") # Output: Hello, Bob!

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

We can pass arguments to a function by including them within parentheses after the function name when you call it. There are two types of arguments you can use: 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 based on their respective positions. The order in which you pass the arguments matters, as they are matched with the parameters in the same order.

Ex:

def add\_numbers(a, b):

return a + b

result = add\_numbers(3, 5)

print(result) # Output: 8

Keyword Arguments:

Keyword arguments are passed to a function using the names of the parameters as keys and their corresponding values. The order of the arguments doesn't matter in this case because they are explicitly assigned to specific parameter names.

def calculate\_total(price, quantity):

return price \* quantity

total = calculate\_total(price=10, quantity=3)

print(total) # Output: 30

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

The purpose of the return statement in a function is to specify the value or values that the function should produce as its result. When a return statement is executed, it terminates the execution of the function and sends the specified value(s) back to the caller.

A function can indeed have multiple return statements. However, once a return statement is encountered in a function, the function execution immediately stops, and the specified value is returned. This means that only one return statement will be executed during the function's execution.

Ex:

def get\_grade(score):

if score >= 90:

return "A"

elif score >= 80:

return "B"

elif score >= 70:

return "C"

else:

return "F"

result = get\_grade(85)

print(result) # Output: B

When we call get\_grade(85), the value 85 is passed as an argument. Since 85 is greater than 80, the second return statement "B" is executed, and the grade "B" is returned. The returned value is then assigned to the variable result, and it is printed as the output.

1. **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, also known as anonymous functions, are small, inline functions in Python that are defined without a name. They are typically used for simple, one-line operations and are created using the lambda keyword. Lambda functions can take any number of arguments, but they can only have a single expression as their body.

Here's the general syntax of a lambda function:

lambda arguments: expression

Lambda functions are different from regular functions in a few ways:

Anonymous: Lambda functions are anonymous because they don't have a name. They are defined and used in a single line.

Concise: Lambda functions are typically used for short, simple operations. They can be written concisely without the need for a formal function definition.

Limited functionality: Lambda functions are limited to a single expression as their body. They cannot contain multiple statements or complex logic like regular functions.

Lambda functions are useful in situations where a small, one-time function is needed, especially when it's not necessary to define a named function separately. They are commonly used with built-in functions like map(), filter(), and reduce(), which accept functions as arguments.

Here's an example where a lambda function can be useful with the map() function

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

squared\_numbers = list(map(lambda x: x\*\*2, 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.**

In Python, the concept of "scope" refers to the visibility or accessibility of variables within different parts of a program. Scope determines where and how a variable can be accessed and modified.

Local Scope:

Local scope refers to the scope within a specific function. Variables defined within a function are considered local variables and are only accessible within that function. Local variables have limited visibility and cannot be accessed from outside the function.

Here's an example to illustrate local scope:

def my\_function():

x = 10 # Local variable

print(x)

my\_function() # Output: 10

print(x) # Error: NameError: name 'x' is not defined

In the above example, the variable x is defined within the my\_function() function. It is a local variable and can only be accessed within the function. If we try to access x outside the function, it will result in a NameError because the variable is not defined in the global scope.

Global Scope:

Global scope refers to the outermost scope in a program, outside of any functions. Variables defined outside of functions, at the top level of a module, have global scope. They are accessible from any part of the program, including inside functions.

Here's an example to illustrate global scope:

x = 10 # Global variable

def my\_function():

print(x)

my\_function() # Output: 10

print(x) # Output: 10

1. **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 as a tuple, a list, or any other iterable object. The values are separated by commas within the return statement.

Here's an example of returning multiple values using the return statement:

def get\_values():

a = 10

b = 20

c = 30

return a, b, c

result = get\_values()

print(result) # Output: (10, 20, 30)

print(result[0]) # Output: 10

print(result[1]) # Output: 20

print(result[2]) # Output: 30

In the above example, the get\_values() function defines three variables a, b, and c with respective values. The return statement is used to return these values separated by commas. When we call the function and assign its result to the result variable, we get a tuple (10, 20, 30) as the return value. We can then access individual values of the tuple using indexing, as demonstrated in the subsequent print statements.

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

In Python, the concepts of "pass by value" and "pass by reference" are not directly applicable because the way function arguments are passed behaves differently from traditional definitions in some other programming languages. However, understanding the underlying behavior will help clarify how Python handles function arguments.

In Python, arguments are passed by assigning objects to local variables, which creates references to those objects. The key point to understand is that Python uses a combination of pass-by-object-reference and pass-by-value behavior.

Pass by Object Reference:

When an object is passed as an argument to a function, a reference to the object is passed. This means that changes made to the object within the function will be reflected outside the function as well. The object itself is not copied.

def modify\_list(my\_list):

my\_list.append(4)

numbers = [1, 2, 3]

modify\_list(numbers)

print(numbers) # Output: [1, 2, 3, 4]

Pass by Value (for immutable objects):

For immutable objects like integers, strings, and tuples, a new copy of the value is created when passed as an argument. This means that modifications made to the argument within the function will not affect the original value outside the function.

def modify\_value(x):

x += 1

number = 10

modify\_value(number)

print(number) # Output: 10

1. **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**

import math

def perform\_operations(x):

log\_value = math.log(x)

exp\_value = math.exp(x)

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

square\_root = math.sqrt(x)

return log\_value, exp\_value, power\_value, square\_root

# Example usage

result = perform\_operations(5)

print(result)

In the perform\_operations() function, we take an argument x which can be an integer or a decimal value. Inside the function, we use the math module to perform the requested operations:

Logarithmic function: We use math.log(x) to compute the natural logarithm of x.

Exponential function: We use math.exp(x) to compute e raised to the power of x.

Power function with base 2: We use math.pow(2, x) to calculate 2 raised to the power of x.

Square root: We use math.sqrt(x) to calculate the square root of x.

The values calculated for each operation are then returned as a tuple using the return statement.

In the example usage, we call the perform\_operations() function with 5 as the argument. The returned values are assigned to the variable result, and we print the result.

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

def extract\_name(full\_name):

name\_parts = full\_name.split()

first\_name = name\_parts[0]

last\_name = name\_parts[-1]

return first\_name, last\_name

# Example usage

result = extract\_name("John Doe")

print(result)

In the extract\_name() function, we take a full name as the full\_name argument. We then use the split() method to split the full name into a list of name parts based on whitespace. By default, split() splits the string at whitespace characters.

We assume that the first part in the resulting list is the first name, and the last part is the last name. The first name is accessed using name\_parts[0], and the last name is accessed using name\_parts[-1].

Finally, we return the first name and last name as a tuple using the return statement.

In the example usage, we call the extract\_name() function with the full name "John Doe". The returned values, the first name "John" and the last name "Doe", are assigned to the variable result, and we print the result.