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

The main difference between a built-in function and a user-defined function lies in their origins and how they are created.

1. Built-in functions: These functions are provided by Python as part of its standard library. They are readily available for use without the need for additional coding or importing external modules. Built-in functions cover a wide range of functionality and are accessible from any Python script or interactive session. Examples of built-in functions include `print()`, `len()`, `range()`, `type()`, and `sum()`.

Here's an example of using the built-in function `print()` to display a message:

print("Hello, world!")

2. User-defined functions: These functions are created by Python programmers to encapsulate a specific set of instructions or a block of code that can be reused multiple times within a program. User-defined functions provide a way to modularize code and improve code readability. They are defined using the `def` keyword followed by a function name, a set of parentheses for parameters (if any), and a colon to indicate the start of the function's code block. Examples of user-defined functions include functions that calculate mathematical operations, perform data processing tasks, or implement custom algorithms.

Here's an example of a user-defined function named `multiply()` that multiplies two numbers:

def multiply(a, b):

return a \* b

result = multiply(3, 5)

print(result) # Output: 15

In the above example, the `multiply()` function takes two parameters (`a` and `b`), multiplies them, and returns the result. The function is then called with the arguments `3` and `5`, and the returned value is stored in the `result` variable. Finally, the result is printed using the `print()` built-in function.

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

You can pass arguments to a function in two ways: positional arguments and keyword arguments.

1. Positional Arguments:

Positional arguments are passed to a function based on their position or order. The function definition specifies the parameters in a specific order, and when calling the function, the arguments are provided in the same order. The number of arguments and their positions must match the function definition.

2. Keyword Arguments:

Keyword arguments are passed to a function using the name of the parameter they correspond to, followed by the `=` sign and the argument's value. With keyword arguments, the order of the arguments does not matter.

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 that the function should produce as its result. It allows the function to pass data back to the caller or to terminate the function's execution prematurely. When a `return` statement is encountered in a function, the function immediately exits, and the value specified in the `return` statement is returned to the caller.

A function can indeed have multiple `return` statements, although only one `return` statement is executed during a single function call. Once a `return` statement is executed, the function terminates, and the value specified in the `return` statement is returned to the caller. Any subsequent `return` statements in the function are ignored.

Here's an example to illustrate the concept:

def absolute\_value(number):

if number >= 0:

return number

else:

return -number

result1 = absolute\_value(5)

print(result1) # Output: 5

result2 = absolute\_value(-8)

print(result2) # Output: 8

In the above example, the `absolute\_value()` function takes a number as an argument and checks whether it is greater than or equal to zero. If the number is non-negative, the function immediately returns the number itself using the `return` statement (`return number`). If the number is negative, the function executes the `else` block and returns the negated value of the number (`return -number`).

When calling the `absolute\_value()` function with `5` as an argument, the function encounters the first `return` statement, and the value `5` is returned. The returned value is then stored in the `result1` variable and printed.

Similarly, when calling the function with `-8` as an argument, the function executes the `else` block and encounters the `return` statement that returns the value `-8`. The returned value is stored in the `result2` variable and printed. Note that once a `return` statement is executed, the function immediately exits, and any subsequent code in the function is not 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.

Lambda functions, also known as anonymous functions, are a way to create small, one-line functions in Python. They are defined using the `lambda` keyword, followed by a set of parameters, a colon, and an expression. Lambda functions are typically used for simple, one-time operations where creating a named function using `def` would be unnecessary.

The main differences between lambda functions and regular functions are:

1. Syntax: Lambda functions are defined using the `lambda` keyword and are written as a single line of code. Regular functions are defined using the `def` keyword and can span multiple lines.

2. Function Name: Lambda functions are anonymous, meaning they do not have a name associated with them. They are usually assigned to a variable or used as an argument within other functions. Regular functions have a name by which they can be called.

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 absolute values:

numbers = [-5, -2, 0, 3, -1, 4]

sorted\_numbers = sorted(numbers, key=lambda x: abs(x))

print(sorted\_numbers) # Output: [0, -1, -2, 3, 4, -5]

In the above example, the `sorted()` function is used to sort the `numbers` list. The `key` parameter specifies a function that calculates the sorting criterion. Instead of defining a regular function using `def`, a lambda function is used as the sorting criterion. The lambda function `lambda x: abs(x)` takes a single parameter `x` and returns the absolute value of `x`. This lambda function is passed as the `key` argument to the `sorted()` function, which sorts the numbers based on their absolute values.

Lambda functions are useful in situations where you need a short, one-time function without the need to define a regular function separately. They are commonly used with higher-order functions like `map()`, `filter()`, and `reduce()`, where a simple function is required as an argument.

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

The concept of "scope" refers to the visibility and accessibility of variables within different parts of a program. The scope determines where a variable can be accessed and how long it remains available for use. In Python, there are two main scopes to consider: local scope and global scope.

1. Local Scope:

Variables defined within a function have local scope. They are accessible only within the specific function where they are defined. Local variables exist and retain their values only during the execution of the function. Once the function execution is complete, local variables are destroyed. Local variables cannot be accessed from outside the function. Here's an example to illustrate local scope:

def my\_function():

x = 10 # Local variable

print(x) # Output: 10

my\_function()

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

In the above example, the variable `x` is defined within the `my\_function()` function. It has local scope and is accessible only within the function. When the function is called, the value of `x` is printed successfully within the function. However, when attempting to access `x` outside the function, a `NameError` occurs because the variable is not defined in the global scope.

2. Global Scope:

Variables defined outside of any function, at the top level of a module, have global scope. They can be accessed and modified from anywhere within the module, including inside functions. Global variables persist throughout the entire execution of the program and can be accessed by any function or code block within the module.

Here's an example to illustrate global scope:

x = 10 # Global variable

def my\_function():

print(x) # Output: 10

my\_function()

print(x) # Output: 10

In the above example, `x` is defined outside of any function, making it a global variable. It can be accessed and printed both inside the `my\_function()` function and outside of it.

It's important to note that if a variable is assigned a value within a function without being explicitly declared as global, it will be treated as a local variable within that function, even if a variable with the same name exists in the global scope.

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

In Python, the `return` statement in a function can be used to return multiple values by combining them into a data structure like a tuple, list, or dictionary. This allows you to package and return multiple values as a single entity. Here are a few ways to use the `return` statement to return multiple values:

1. Using a Tuple:

You can use a tuple to group multiple values and return them as a single tuple object. The values can be accessed individually when the function is called.

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 returns three values (`a`, `b`, and `c`) as a tuple. When calling the function, the returned tuple is assigned to the `result` variable. Individual values can be accessed using indexing, such as `result[0]`, `result[1]`, and `result[2]`.

2. Using a List:

Similar to using a tuple, you can also use a list to group multiple values and return them as a single list object.

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 this example, the `get\_values()` function returns the values as a list instead of a tuple. The usage and access of the returned list are similar to the previous example.

3. Using a Dictionary:

You can also use a dictionary to return multiple values, where each value is associated with a specific key.

def get\_values():

a = 10

b = 20

c = 30

return {"value\_a": a, "value\_b": b, "value\_c": c}

result = get\_values()

print(result) # Output: {'value\_a': 10, 'value\_b': 20, 'value\_c': 30}

print(result["value\_a"]) # Output: 10

print(result["value\_b"]) # Output: 20

print(result["value\_c"]) # Output: 30

In this example, the `get\_values()` function returns the values as a dictionary, where each value is associated with a specific key. The returned dictionary can be accessed using the respective keys, such as `result["value\_a"]`, `result["value\_b"]`, and `result["value\_c"]`.

By returning multiple values from a function, you can conveniently capture and use multiple results in a single assignment, enabling more flexible and expressive code.

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

The concepts of "pass by value" and "pass by reference" are not directly applicable because the underlying mechanism is slightly different. Python uses a combination of both concepts, which can sometimes lead to confusion.

1. Pass by Value:

In "pass by value," the value of an argument is copied and passed to the function. Any modifications made to the parameter within the function do not affect the original argument.

2. Pass by Reference:

In "pass by reference," a reference to the memory location of an argument is passed to the function. Modifying the parameter within the function affects the original argument since both point to the same memory location.

However, in Python, the actual behavior depends on the type of the object being passed as an argument:

- Immutable Objects (like numbers, strings, and tuples): Immutable objects are passed by value in Python. A copy of the value is made, and any changes made to the parameter within the function do not affect the original object.

- Mutable Objects (like lists, dictionaries, and user-defined objects): Mutable objects are passed by reference in Python. A reference to the object is passed, and modifications made to the parameter within the function affect the original object.

Let's look at an example to understand this behavior:

def modify\_value(num, lst):

num += 10

lst.append(4)

number = 5

my\_list = [1, 2, 3]

modify\_value(number, my\_list)

print(number) # Output: 5 (unchanged)

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

In the example above, `number` is an immutable object (integer), and `my\_list` is a mutable object (list). When `number` is passed to the `modify\_value()` function, it is passed by value, so modifications made to `num` within the function do not affect the original `number`. On the other hand, `my\_list` is passed by reference, so any modifications made to `lst` within the function affect the original `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

Here's a function that can perform the requested operations on an input integer or decimal value:

import math

def perform\_operations(number):

# Logarithmic function (log x)

logarithm = math.log(number)

# Exponential function (exp(x))

exponential = math.exp(number)

# Power function with base 2 (2^x)

power = math.pow(2, number)

# Square root

square\_root = math.sqrt(number)

return logarithm, exponential, power, square\_root

# Example usage

result = perform\_operations(3.5)

print(result)

In the above example, the function `perform\_operations` takes a number as input and performs the following operations on it:

a. Logarithmic function: The `math.log()` function is used to calculate the natural logarithm of the number.

b. Exponential function: The `math.exp()` function is used to calculate the exponential value of the number.

c. Power function with base 2: The `math.pow()` function is used to calculate the power of 2 with the input number as the exponent.

d. Square root: The `math.sqrt()` function is used to calculate the square root of the number.

The function then returns the values of all the calculated operations as a tuple.

In the example usage, the function is called with the input value `3.5`. The resulting values of the operations are stored in the `result` variable and then printed.

You can pass both integer and decimal values to the function, and it will perform the operations accordingly.

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

Here's 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):

names = full\_name.split()

first\_name = names[0]

last\_name = names[-1]

return first\_name, last\_name

# Example usage

name = "Subash Kedia"

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

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

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