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

Answer

Built-in functions:

Built-in functions are functions that are already defined within the Python programming language. They are readily available for use without requiring any additional coding or importing external modules. These functions are part of Python's standard library and cover a wide range of tasks, such as manipulating data, performing calculations, and interacting with the system.

Example of a built-in function: len()

The len() function is a built-in function used to return the length (the number of items) of an object such as a string, list, or tuple. Here's an example:

```
my_string = "Hello, world!"
length = len(my_string)
print(length)  # Output: 13
```

User-defined functions:

User-defined functions are created by the users themselves to perform specific tasks or operations. These functions are defined using the def keyword, followed by a function name, parentheses for optional parameters, and a colon. The function's logic is defined within the function body using an indented block of code.

Example of a user-defined function: add_numbers()

The add_numbers() function takes two numbers as input and returns their sum. Here's an example:

```
def add_numbers(a, b):
return a + b
result = add_numbers(3, 5)
print(result)  # Output: 8
```

In this example, add_numbers() is a user-defined function that adds two numbers passed as arguments and returns the result.

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

Answer

Positional arguments:

Positional arguments are the most basic type of arguments in Python. They are passed to a function based on their position or order. When calling a function, you provide the arguments in the same order as defined in the function's parameter list.

Example:

```
def greet(name, age):
print(f"Hello, {name}! You are {age} years old.")
greet("Alice", 25)
```

In this example, "Alice" is passed as the name argument, and 25 is passed as the age argument. The order in which the arguments are provided is crucial. Positional arguments are matched to the function parameters based on their position.

Keyword arguments:
Keyword arguments allow you to specify the argument values using their corresponding parameter names. When calling a function, you provide the arguments along with their parameter names, using the format parameter_name=value.
Example:
```
def greet(name, age):
print(f"Hello, {name}! You are {age} years old.")
greet(age=25, name="Alice")
```
In this example, the arguments are provided with their parameter names explicitly. The order of the arguments doesn't matter as long as each argument is assigned to the correct parameter.

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

The return statement in a function is used to specify the value that the function should return after performing its task or computation. It allows a function to produce a result or output that can be used or assigned to a variable in the calling code.
The primary purpose of the return statement is to exit the function and provide a value back to the caller. When a return statement is encountered, the function execution is immediately terminated, and the specified value is passed back to the caller.
A function can have multiple return statements, but only one return statement is executed in a single function call. Once a return statement is executed, the function exits, and the remaining code in the function is not executed.
Here's an example to illustrate multiple return statements:
```
def get_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    else:
        return "F"
grade = get_grade(85)
print(grade)  # Output: "B"
```
In this example, the get_grade() function takes a score parameter and returns the corresponding grade based on the score. It has multiple return statements within different conditional branches. The function evaluates the score and returns the appropriate grade based on the condition that matches. Only one return statement will be executed in a single function call.
When the get_grade() function is called with a score of 85, it evaluates the second condition (score >= 80) as True and executes the return "B" statement, returning the value "B" as the grade. The returned value is then assigned to the grade variable and printed.
It's important to note that once a return statement is executed, the function is immediately exited, and the program control returns to the point of the function call. Therefore, any code or statements after the return statement will not be executed within that function call.

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

Answer

Lambda functions in Python are anonymous functions that are defined using the lambda keyword. Unlike regular functions, lambda functions don't require a function name and are typically used for simple, one-line expressions. They are often used when a small, temporary function is needed without the need for a full function definition.

The syntax for a lambda function is as follows:

lambda arguments: expression

Here's an example to illustrate the use of a lambda function:

```
# Regular function
def multiply(x, y):
    return x * y
# Lambda function
multiply_lambda = lambda x, y: x * y
# Calling regular function
result_regular = multiply(2, 3)
print(result_regular)  # Output: 6
# Calling lambda function
result_lambda = multiply_lambda(2, 3)
print(result_lambda)  # Output: 6
```

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

Answer

Local scope:

Local scope refers to the visibility of variables within a specific function. 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 are destroyed when the function completes its execution.

Example:

```
def my_function():
    x = 10  # Local variable
    print(x)
my_function()  # Output: 10
print(x)  # Raises an error: NameError: name 'x' is not defined
```

In this example, the variable x is defined inside the my_function() function and has local scope. It is accessible and can be used within the function. However, trying to access x outside the function results in a NameError because the variable is not defined in the global scope.

Global scope:

Global scope refers to the visibility of variables throughout the entire program. Variables defined outside any function or at the top level of a module have global scope and can be accessed from anywhere within the program, including inside functions.

Example:

```
x = 10  # Global variable
def my_function():
    print(x)
my_function()  # Output: 10
```

```
print(x)  # Output: 10
```

In this example, the variable x is defined outside the my_function() function and has global scope. It can be accessed both inside the function and outside the function.
When a variable is accessed within a function, Python first checks the local scope for that variable. If the variable is not found in the local scope, Python then looks for it in the next outer scope, which is typically the global scope. If the variable is still not found, an error is raised.
It's important to note that while a function has access to variables in the global scope, it cannot modify a global variable directly unless explicitly declared using the global keyword inside the function.

Example:

```
x = 10  # Global variable
def my_function():
    global x
    x = 20  # Modifying the global variable
my_function()
print(x)  # Output: 20
```

In this example, the global keyword is used inside the my_function() function to indicate that the variable x refers to the global variable x, and any modifications to it will affect the global variable.

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

   Answer

Returning a tuple:
You can use a tuple to group multiple values together and return them as a single entity. The return statement can be followed by a comma-separated list of values, which will be automatically packed into a tuple.

Example:

```
def get_values():
    a = 10
    b = 20
    c = 30
    return a, b, c
result = get_values()
print(result)  # Output: (10, 20, 30)
```

In this example, the function get_values() returns three values a, b, and c. These values are packed into a tuple, and when the function is called, the tuple is returned and assigned to the result variable.

Returning a list:
Similar to using tuples, you can also use a list to return multiple values from a function.

Example:

```
def get_values():
    a = 10
    b = 20
    c = 30
    return [a, b, c]
result = get_values()
```

```
print(result)  # Output: [10, 20, 30]
```

In this example, the function get_values() returns a list containing the values a, b, and c.

Returning a dictionary:
If you want to return multiple values with associated keys or labels, you can use a dictionary.
Example:

```
def get_values():
    a = 10
    b = 20
    c = 30
    return {"a": a, "b": b, "c": c}
result = get_values()
print(result)  # Output: {'a': 10, 'b': 20, 'c': 30}
```

In this example, the function get_values() returns a dictionary where the values a, b, and c are associated with their respective keys.

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

Pass by value:
In a strict "pass by value" scenario, a copy of the value of the variable is made and passed to the function. Any modifications made to the parameter within the function do not affect the original variable.
Example:

```
def modify_value(x):
    x = x + 1
a = 10
modify_value(a)
print(a)  # Output: 10
```

In this example, the variable a is passed as an argument to the modify_value() function. However, since integers in Python are immutable objects, a copy of the value of a is made and passed to the function. Any modifications made to x within the function do not affect the original variable a.
Pass by reference:
In a strict "pass by reference" scenario, the reference or memory address of the variable is passed to the function. Any modifications made to the parameter within the function affect the original variable.
However, in Python, the behavior is different, and objects are neither strictly "passed by value" nor "passed by reference." Instead, Python passes references to objects by value.

Example:

```
def modify_list(lst):
    lst.append(4)
my_list = [1, 2, 3]
modify_list(my_list)
print(my_list)  # Output: [1, 2, 3, 4]
```

In this example, the list my_list is passed as an argument to the modify_list() function. Lists in Python are mutable objects, so the reference to my_list is passed to the function. The function can modify the list by appending an element, and the change is visible outside the function. However, note that the reference itself is passed by value, meaning that if the parameter lst were reassigned within the function, it would not affect the reference to my_list outside the function.

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

Answer

```
import math
def perform_operations(x):
    logarithm = math.log(x)
    exponential = math.exp(x)
    power = math.pow(2, x)
    square_root = math.sqrt(x)
     return logarithm, exponential, power, square_root
# Example usage
result = perform_operations(5)
print(result)
```

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

Answer

```
class person:
    def __init__(self,fname,lname):
        self.fname=fname
        self.lname=lname
  def display_info(self):
        print(f"first name is : {self.fname}\nlast name is : {self.lname}")
Person=person("Neha","Dhananju")
Person.display_info()
```