**Homework 2**

|  |  |  |
| --- | --- | --- |
| 202283890036 | 黄家睿(Jerry Huang) | Internet of Things |

# Introduction of function parameter

Function parameters are a crucial aspect of programming, serving as the means by which a function receives input data or values. Below is an introduction to function parameters:

## 1.1 Definition

Function parameters are variables specified within the parentheses when defining a function or method. These variables are used to input data or values into the function. They represent the information that the function requires to perform its task.

## 1.2 Characteristics

1. Place of Definition:

Function parameters are defined within the parentheses of the function definition.

1. Purpose:

They serve as placeholders for the actual data or values that will be provided when the function is called.

1. Scope:

The lifetime of a parameter is confined to the function body. Once the function execution completes, the parameter no longer exists.

1. Type Specification:

In strongly typed languages, function parameters may require explicit type specifications to ensure type safety.

## 1.3 Difference from Arguments

It's important to distinguish between parameters and arguments:

Arguments: These are the actual data or values passed to the function when it is called.

Parameters: These are the variables specified in the function definition that represent the expected input.

For example, in the function definition def add(x, y):, x and y are parameters. When the function is called as add(3, 5), 3 and 5 are arguments.

## 1.4 Examples

Here are a few examples to illustrate the concept of function parameters:

1. ****Simple Addition Function****
2. **def** add(a, b):
3. **return** a + b
5. result = add(5, 3)  # Here, 5 and 3 are arguments passed to the function.
6. **print**(result)  # Output: 8
7. ****Function with Multiple Parameters****
8. **def** greet(name, greeting):
9. **return** f"{greeting}, {name}!"
11. message = greet("Alice", "Hello")  # Here, "Alice" and "Hello" are arguments.
12. **print**(message)  # Output: Hello, Alice!
13. ****Function with Default Parameters****
14. **def** multiply(a, b=2):
15. **return** a \* b
17. result1 = multiply(4)  # Here, only one argument is passed, and b defaults to 2.
18. result2 = multiply(4, 3)  # Here, both arguments are passed.
19. **print**(result1)  # Output: 8
20. **print**(result2)  # Output: 12

# The difference between closure and partial function

## Closure

A closure in Python is a function object that retains a reference to its enclosing scope, even after the outer function has returned. This means that the inner function can access variables from the outer function's scope even when the outer function is no longer executing.

Here's an example of a closure in Python:

1. **def** outer\_function(outer\_variable):
2. **def** inner\_function():
3. **print**(outer\_variable)
4. **return** inner\_function
6. closure = outer\_function("Hello, World!")
7. closure()  # Output: Hello, World!

In this example, inner\_function() is a closure because it retains a reference to outer\_variable even after outer\_function has returned.

## Partical function

In Python, the concept of a "partial function" is typically associated with the functools.partial function, which is used to create a new function by partially applying some of the arguments to an existing function. This allows you to "freeze" some of the arguments of a function, leaving the rest to be specified later.

Here's an example of using functools.partial:

1. **from** functools **import** partial
3. **def** greet(greeting, name):
4. **print**(f"{greeting}, {name}!")
6. greet\_hello = partial(greet, greeting="Hello")
7. greet\_hello("Alice")  # Output: Hello, Alice!

In this example, greet\_hello is a new function that is created by partially applying the greeting argument to the greet function. The resulting function can be called with just the name argument.

## Key differences

1. Scope Capture:

A closure captures variables from its enclosing scope, allowing the inner function to access those variables even after the outer function has returned. A partial function does not capture any scope; it simply creates a new function by partially applying some arguments to an existing function.

1. Argument Handling:

A closure can have access to variables from the outer scope, but it does not alter the number or names of the arguments that the inner function takes. A partial function alters the number of required arguments by fixing some of them in advance.

1. Use Case:

Closures are useful for creating functions that maintain a state across multiple calls (e.g., counters, factories). Partial functions are useful for creating specialized versions of functions with predefined arguments (e.g., setting default values for some arguments).

In summary, closures and partial functions in Python serve different purposes and have different mechanisms. Closures are about capturing scope, while partial functions are about partially applying arguments to existing functions.