Functions

1. Defining a Function:
    - Functions are defined using the `def` keyword followed by the function name and parentheses containing optional parameters.
    - The function body is indented and contains the code to be executed when the function is called.
    - Example:

```
def greet(name):
    print("Hello, " + name)
```

2. Function Signatures and Type Hinting:
    - Function signatures include the function name, parameters (including optional default values), and return type (optional).
    - Type hinting allows you to specify the expected types of function parameters and return values.
    - Example:

```
def add(a: int, b: int) -> int:
    return a + b
```

3. Calling a Function:
    - Functions are called by using the function name followed by parentheses containing any required arguments.
    - Example:

```
greet("John")            # Output: "Hello, John"
result = add(2, 3)
print(result)            # Output: 5
```

4. Function Scope:
    - Scope refers to the accessibility and visibility of variables within different parts of a program.
    - Functions have their own local scope, which means variables defined within a function are only accessible within that function unless specified otherwise.
    - Example:

```
def multiply(a, b):
    result = a * b       # local variable within the function
    return result

def calculate():
    x = 2
    y = 3
    z = multiply(x, y)  # Accessing the multiply function within the
calculate function
    print(z)            # Output: 6

calculate()
```

- In the example above, the variable `result` defined within the `multiply` function is local to that function and cannot be accessed outside of it. Similarly, the variables `x`, `y`, and `z` defined within the `calculate` function are local to that function.

Function signatures and type hinting help provide clarity and documentation about the expected inputs and outputs of a function. Functions can be called by using the function name and passing the required arguments. The scope of a variable determines where it can be accessed, and functions have their own local scope unless specified otherwise.

Variable Shadowing

Variable shadowing occurs when a variable declared in a particular scope has the same name as a variable declared in an outer scope. This causes the inner variable to "shadow" or hide the outer variable within its own scope.

When variable shadowing occurs, the inner variable takes precedence over the outer variable within its scope. This means that any references to the variable name within the inner scope will refer to the inner variable, effectively hiding the outer variable.

Here's a brief explanation with an example:

```
x = 10    # Outer variable

def my_function():
    x = 20    # Inner variable, shadows the outer variable
    print(x)  # Output: 20

my_function()

print(x)    # Output: 10 (outer variable is not affected)
```

In this example, we have an outer variable `x` with a value of 10. Inside the `my_function()` function, there is an inner variable `x` with a value of 20, which shadows the outer variable. When we print `x` within the function, it refers to the inner variable, so the output is 20. However, outside the function, when we print `x`, it refers to the outer variable, so the output is 10.

Variable shadowing can lead to confusion and bugs in code, especially when the intention is to use the outer variable but it gets accidentally shadowed. It's generally recommended to avoid variable shadowing and use different names for variables in different scopes to maintain clarity and prevent unexpected behavior.
Unclassified