**Python Function Part 2**

### Scope | Docstring | Nested function | Anonymous Function | Recursive Function

**6. Scope of Variables:**

**Global vs. Local Scope:**

**- Global Scope:** Variables declared outside of any function have global scope. They can be accessed from anywhere within the program, including inside functions.

- **Local Scope:** Variables declared within a function have local scope. They can only be accessed within the function where they are defined and are not visible outside of it.

**LEGB Rule:**

Python follows the LEGB rule to determine the scope of variables:

**- Local:** Variables defined within the current function.

**- Enclosing:** Variables defined in the enclosing function (for nested functions).

- **Global:** Variables defined at the top level of the module.

- **Built-in:** Variables built into Python (like `print`, `len`, etc.).

**Examples:**

```python

x = 10 # Global variable

def func():

y = 20 # Local variable

print("Inside func():", x, y)

func()

print("Outside func():", x)

# Attempting to access y outside of func() will result in an error

```

**7. Anonymous Functions (Lambda Functions):**

**Definition:**

Anonymous functions, also known as lambda functions, are small, unnamed functions defined using the `lambda` keyword. They are typically used for short, one-time operations where defining a full function using `def` would be overkill.

**Syntax**:

```python

lambda arguments: expression

```

- `lambda`: Keyword used to define a lambda function.

- `arguments`: Parameters passed to the function.

- `expression`: Single expression whose result is returned by the function.

**Example:**

```python

# Lambda function to square a number

square = lambda x: x \*\* 2

print(square(5)) # Output: 25

# Lambda function to add two numbers

add = lambda x, y: x + y

print(add(3, 4)) # Output: 7

```

**Usage:**

- Lambda functions are often used in combination with higher-order functions such as `map()`, `filter()`, and `reduce()`.

- They are useful for writing concise code, especially when the function logic is simple and does not require a separate named function definition.

**8. Recursive Functions:**

**Definition:**

Recursive functions are functions that call themselves directly or indirectly in order to solve a problem. They are particularly useful for solving problems that can be broken down into smaller, similar subproblems.

**Example**:

```python

# Recursive function to calculate factorial

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120

```

**Explanation:**

- In the factorial example, the function calls itself with a smaller value (`n - 1`) until it reaches the base case (`n == 0`), at which point it returns 1.

- Each recursive call builds on the result of the previous call, eventually computing the factorial of the original input.

**Use Cases:**

- Recursive functions are commonly used to solve problems involving tree structures, such as traversing directories or parsing nested data structures.

- They are also useful for problems that can be solved by breaking them down into smaller, identical subproblems.

**9. Pass by Reference vs. Pass by Value:**

**Definition:**

**- Pass by Value:** In pass by value, a copy of the variable's value is passed to the function, leaving the original variable unchanged.

- **Pass by Reference:** In pass by reference, a reference to the original variable is passed to the function, allowing the function to modify the original variable.

**Example**:

```python

**# Pass by value**

def increment(x):

x += 1

print("Inside function:", x)

num = 10

increment(num)

print("Outside function:", num) # Output: 10 (unchanged)

**# Pass by reference**

def append\_item(lst):

lst.append(4)

my\_list = [1, 2, 3]

append\_item(my\_list)

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

```

**Explanation:**

- **In the pass by value example,** the function `increment()` receives a copy of the variable `num`, so any modifications made to `x` inside the function do not affect `num`.

**- In the pass by reference example**, the function `append\_item()` receives a reference to the original list `my\_list`, so modifications made to `lst` inside the function affect the original list.

**Use Cases:**

**- Pass by value** is typically used for immutable types (e.g., integers, strings) where the original value should not be modified.

**- Pass by reference** is commonly used for mutable types (e.g., lists, dictionaries) where modifications to the original value are expected.

**10. Docstrings:**

**Definition:**

Docstrings, short for documentation strings, are string literals used to document Python modules, classes, functions, and methods. They are used to describe the purpose, behavior, and usage of the code they document.

**Syntax**:

```python

def function\_name(parameters):

"""Docstring"""

# Function body

```

- The docstring is enclosed within triple quotes (`""" """`) immediately after the function definition line.

- It can span multiple lines and should provide clear and concise information about the function's purpose, parameters, return values, and any other relevant details.

**Example:**

```python

def add(x, y):

""" Function to add two numbers."""

return x + y

print(add.\_\_doc\_\_) # Output: Function to add two numbers.

```

**Usage:**

- Docstrings serve as a form of self-documentation for Python code, making it easier for developers to understand and use.

- They are particularly useful when generating documentation using tools like Sphinx or when inspecting code using the `help()` function or IDE features.

**11. Function Within Functions (Nested Functions):**

**Definition:**

Nested functions, also known as inner functions, are functions defined within the scope of another function. They have access to variables in the enclosing function's scope and can be used to encapsulate and organize code.

**Example**:

```python

def outer\_function():

"""Outer function"""

def inner\_function():

"""Inner function"""

print("Inside inner function")

inner\_function() # Call inner function from outer function

outer\_function() # Output: Inside inner function

```

**Explanation:**

**- In the example,** `inner\_function()` is defined within the scope of `outer\_function()`.

- `inner\_function()` can access variables from `outer\_function()` but not vice versa, encapsulating its behavior within the outer function's context.

**Use Cases:**

- Nested functions are useful for breaking down complex tasks into smaller, more manageable components.

- They help improve code organization, readability, and maintainability by keeping related functionality together.

**12. Return Statement:**

**Definition:**

The `return` statement is used to exit a function and return a value or expression to the function caller. It can be used to send data back from the function to the calling code.

**Syntax:**

```python

def function\_name(parameters):

"""Docstring"""

# Function body

return expression

```

- The `return` keyword is followed by an expression whose result will be returned to the caller.

- If no `return` statement is present or if `return` is used without an expression, the function returns `None` by default.

**Example:**

```python

def square(x):

"""Function to calculate the square of a number."""

return x \*\* 2

result = square(5)

print(result) # Output: 25

```

**Usage:**

- The `return` statement allows functions to produce output that can be used by other parts of the program.

- It terminates the execution of the function and immediately returns control to the caller.