Function in Python

Premanand S

Assistant Professor
School of Electronics Engineering
Vellore Institute of Technology *john@smith.com*

October 25, 2024

What is a Function?

- A **function** is a block of reusable code that performs a specific task.
- It allows you to organize and reuse code effectively.
- A function can take **inputs** (parameters or arguments) and can return an **output**.

Why are Functions Used in Programming?

- Functions help organize code into smaller, manageable pieces.
- You can avoid repeating code by reusing functions.
- Functions make programs easier to debug and maintain.

Benefits of Using Functions

- **Modularity:** Break down complex tasks into smaller, independent units.
- **Code Reusability:** Reuse functions in different parts of the program or in other programs.
- **Maintainability:** Update functions without affecting other parts of the program.
- **Readability:** Improves the structure and clarity of the code.
- **Abstraction:** Hide implementation details and simplify function usage.

Basic Syntax of a Python Function

- `def`: Keyword to define a function.
- `function_name`: Descriptive name of the function.
- `parameters`: (Optional) Inputs to the function.
- `return`: (Optional) The result the function returns.

Example (Python Code)

```
def function_name(parameters):
    # Function body
    # Code block to perform the task
    return result  # Optional
```

Example of a Simple Function

Example (Python Code)

```
def greet():
    print("Hello, welcome to Python functions!")

# Calling the function
greet()

Output:

Hello, welcome to Python functions!
```

How to Define a Function

- To define a function in Python, use the `def` keyword, followed by the function name, parentheses `()` (which can include parameters), and a colon `:`. The function body is indented under the definition.

Example (Python Code)

```
def function_name(parameters):
    # Function body
    # Code block to perform the task
```

Example of Defining and Calling a Function

- Here's a simple example of defining and calling a function:

Example (Python Code)

```
def greet():
    print("Hello, welcome to Python functions!")

#To call the function, simply write:
greet()
```

The Importance of Indentation

- In Python, indentation is crucial for defining the structure of the code.
- The code block within a function must be indented consistently.
- Indentation defines the scope of the function.
- All code within the function must be indented to be recognized as part of that function.
- Incorrect indentation will lead to **IndentationError** or unexpected behavior.

Example (Python Code)

```
def greet():
print("Hello, welcome to Python functions!")
```

Understanding Parameters and Arguments

- **Parameters** are the variables listed in a function's definition.
- **Arguments** are the values that are passed to the function when it is called.
- Functions can take different types of parameters, which affect how arguments can be passed.

Types of Parameters

- Positional Arguments: Passed based on their position in the function call.
- Keyword Arguments: Passed by explicitly stating the parameter name and its value.
- Default Parameters: Parameters that have a default value if no argument is provided.
- Arbitrary Arguments: Allow passing a variable number of arguments (e.g., using ‘*args‘ and ‘**kwargs‘).

Positional Arguments

- Arguments can be passed to functions by position. The order in which the arguments are passed to the function matters, as they are assigned to the corresponding parameters in the same order

Example (Python Code)

```
def add(a, b):
    return a + b

# Calling the function with positional arguments
result = add(3, 5)  # 3 is assigned to a, and 5 to b

The output will be:
result = 8
```

Keyword Arguments

- Keyword arguments allow you to pass arguments to a function using the names of the parameters.
- This enables you to specify which parameter corresponds to which value, regardless of the order.

Example (Python Code)

```
def describe_pet(pet_name, animal_type='dog'):
    print(f"I have a {animal_type} named {pet_name}.")

# Calling the function with keyword arguments
describe_pet(pet_name='Buddy', animal_type='cat')

The output will be:
I have a cat named Buddy.
```

Parameters with Default Values

- Parameters can be assigned default values. If an argument for that parameter is not provided when the function is called, the default value is used.

Example (Python Code)

```
def greet(name, greeting='Hello'):
    print(f"{greeting}, {name}!")

# Calling the function with default parameter
greet('Python')  # Uses the default greeting

The output will be:
Hello, Python
You can also specify the greeting:
greet('Prem', greeting='Hi')
Hi, Prem!
```

Arbitrary Arguments

- Use ‘*args‘ to pass a variable number of positional arguments.
- Use ‘**kwargs‘ to pass a variable number of keyword arguments

Example (Python Code)

```
def collect_args(*args):
    return args
print(collect_args(1, 2, 3))

def collect_kwargs(**kwargs):
    return kwargs
print(collect_kwargs(name='anand', age=38))
```

Introduction to Variable Scope

- **Scope**: Defines the accessibility of variables in different parts of a program.
- Scope is crucial to avoid errors and unintended behavior.
- Python has three main types of variable scopes: Local, Global, and Nonlocal.

Local Scope

- A variable declared inside a function is in **local scope**.
- It is accessible only within the function.
- Local variables are destroyed when the function ends.

Example (Python Code)

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

my_function()
# Output: 10
```

Global Scope

- Variables declared outside all functions are in **global scope**.
- These variables can be accessed anywhere in the program.

Example (Python Code)

```
x = 5  # Global variable

def my_function():
    print(x)

my_function()
# Output: 5
```

Global Scope

- Variables declared outside all functions are in **global scope**.
- These variables can be accessed anywhere in the program.

Example (Python Code)

```
x = 5  # Global variable

def my_function():
    print(x)

my_function()
# Output: 5
```

Modifying Global Variables

- To modify a global variable inside a function, use the `global` keyword.
- This tells Python to use the global variable, not create a new local one.

Example (Python Code)

```
x = 5
def modify_global():
    global x
    x = 10

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

Global vs Local Variables (Shadowing)

- If a local variable has the same name as a global variable, the local variable **shadows** the global one inside the function.

Example (Python Code)

```
x = 5  # Global variable

def my_function():
    x = 10  # Local variable
    print(x)

my_function()
# Output: 10
print(x)
# Output: 5
```

Global vs Local Variables (Shadowing)

- If a local variable has the same name as a global variable, the local variable **shadows** the global one inside the function.

Example (Python Code)

```
x = 5  # Global variable

def my_function():
    x = 10  # Local variable
    print(x)

my_function()
# Output: 10
print(x)
# Output: 5
```

Nonlocal Scope in Nested Functions

- **Nonlocal** variables are those from an enclosing function's scope in nested functions.
- They are neither global nor local to the inner function.
- The `nonlocal` keyword is used to modify such variables.

Nonlocal Keyword Example

Example (Python Code)

```
def outer_function():
    x = 5

    def inner_function():
        nonlocal x
        x = 10

    inner_function()
    print(x)

outer_function()
# Output: 10
```

Practical Example: Simple Calculator

- Global and Local Variables in Action:

Example (Python Code)

```
result = 0  # Global variable

def add(a, b):
    global result
    result = a + b

add(2, 3)
print(result)  # Output: 5
```

Closures with Nonlocal Variables

Example (Python Code)

```
def counter():
    count = 0

    def increment():
        nonlocal count
        count += 1
        return count

    return increment

counter_fn = counter()
print(counter_fn())  # Output: 1
print(counter_fn())  # Output: 2
```

Common Mistakes and Best Practices

- Avoid overusing `global` variables; it can lead to unpredictable behavior.
- Use `nonlocal` carefully in closures to maintain clarity.
- Prefer local variables to reduce complexity and enhance code readability.

Conclusion

- Understanding local, global, and nonlocal scope is key to writing clean, efficient Python code.
- Use the `global` and `nonlocal` keywords cautiously to avoid bugs.
- Always aim for clarity and simplicity in variable usage.

Introduction to Recursion

- **Recursion** is a method of solving problems where a function calls itself.
- Useful for breaking down complex problems into simpler, smaller instances.
- Essential for naturally recursive problems, such as trees and graphs.

How Recursive Functions Work

- A recursive function has two parts:
  - **Base Case**: The stopping condition.
  - **Recursive Case**: The function calls itself with a modified argument.
- Example: Calculating factorial of a number.

Example: Factorial Calculation

Example (Python Code)

```
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case

print(factorial(5))  # Output: 120
```

Stack Frames and Call Stack - Factorial

- For factorial(3), the call stack would look like this:
  - factorial(3) calls factorial(2).
  - factorial(2) calls factorial(1).
  - factorial(1) calls factorial(0) (base case).
  - factorial(0) returns 1, and now the frames start popping off the stack:
    - factorial(1) returns 1 * 1 = 1
    - factorial(2) returns 2 * 1 = 2
    - factorial(3) returns 3 * 2 = 6

Stack Frames and Call Stack

- Each recursive call adds a new frame to the **call stack**.
- When the base case is reached, frames are popped off as the function returns.
- Visualizing the call stack helps understand the recursion flow.

Examples of Basic Recursive Functions

- **Factorial**
- **Fibonacci Sequence**
- **Power Calculation**

Example: Factorial Calculation

Example (Python Code)

```
def fibonacci(n):
    if n <= 1:
        return n  # Base case
    else:
        return fibonacci(n-1) + fibonacci(n-2)
        # Recursive case

print(fibonacci(5))  # Output: 5
```

Common Use Cases of Recursion

- **Divide-and-Conquer Algorithms** (e.g., Merge Sort, Quick Sort)
- **Tree and Graph Traversal** (Depth-first search)
- **Dynamic Programming Problems** (e.g., Fibonacci with memoization)

Recursion vs. Iteration

- Recursion can make code simpler and easier to read for certain problems.
- However, recursion uses more memory and can be slower than iteration.
- **When to use Recursion:** Natural recursive problems or when simplicity outweighs performance.

Optimizing Recursive Functions

- **Memoization**: Cache results of previous function calls.
- **Dynamic Programming**: Solve overlapping subproblems efficiently.
- **Tail Recursion**: Reduce memory use by eliminating stack frames.

Example: Memoization

Example (Python Code)

```
memo = {}

def fibonacci(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1) + fibonacci(n-2)
    return memo[n]

print(fibonacci(5))  # Output: 5
```

Common Mistakes in Recursion

- **Missing Base Case**: Leads to infinite recursion and stack overflow.
- **Overlapping Calculations**: Redundant calculations slow down performance.
- **Stack Overflow**: Caused by too deep recursion, exceeding memory limits.

Practical Applications of Recursion

- **Directory Traversal**: Recursively navigating through files and folders.
- **Backtracking Algorithms** (e.g., N-Queens problem).
- **Binary Search**: Recursive search on a sorted list.

Example: Recursive Directory Traversal

Example (Python Code)

```
import os

def list_files(path):
    for item in os.listdir(path):
        full_path = os.path.join(path, item)
        if os.path.isdir(full_path):
            list_files(full_path)
        else:
            print(full_path)

list_files("/path/to/directory")
```

Best Practices for Recursion

- Define a clear base case to avoid infinite recursion.
- Use recursion when it simplifies the code and structure.
- Test thoroughly, especially for edge cases that can lead to deep recursion.

Conclusion

- Recursion is a powerful tool for solving complex problems by breaking them down into simpler subproblems.
- Important to balance readability and performance.
- Use iteration for cases where recursion is too costly.