6 Functions

Functions are one of the core building blocks of Python, enabling code reuse, organization, and modularity. A function is a block of reusable code designed to perform a specific task. Instead of writing the same code multiple times, a function allows it to be defined once and called whenever needed. Functions help make code more readable, maintainable, and efficient.

A function in Python is defined using the def keyword, followed by a function name and parentheses. It can accept parameters, process data, and return values. Functions may also include **docstrings**, which provide documentation for their usage. Python includes built-in functions that handle common operations, but users can also define their own to customize functionality.

In this chapter, we explore how to define and call functions, different ways to pass arguments, handling return values, scope and lifetime of variables, built-in functions, and best practices for writing clean and efficient functions.

Topics Covered in This Chapter

- Function Syntax
- Function Definition
- Calling Functions
- Function Arguments
- Return Statement
- Scope and Lifetime of Variables
- Built-in Functions
- Docstrings
- Best Practices

Function Syntax

Function Definition

In Python, functions are defined using the def keyword, followed by the function name and parentheses. The function body is indented and contains the statements to be executed when the function is called.

The basic syntax for defining a function is:

```
def function_name():

    # Function body (indented block)

    pass  # Placeholder for future code
```

After defining a function, it does not execute immediately. Instead, it must be **called** by its name elsewhere in the program.

Function Name and Naming Conventions

Function names should be descriptive and follow Python's naming conventions:

- Function names should be **lowercase** with words separated by underscores (snake_case).
- Avoid using Python **reserved keywords** as function names.
- Function names should be meaningful to indicate their purpose.

Examples of well-named functions:

```
def calculate_total():
    pass


def get_user_name():
    pass


def find_maximum_value():
    pass
```

Poorly named functions make code harder to read and maintain, so choosing clear and descriptive names is crucial.

Function Body and Indentation

The function body consists of the statements that define the behavior of the function. In Python, **indentation is mandatory** to indicate the block of code that belongs to the function. All statements within the function must be indented consistently, typically using four spaces.

Example of a properly structured function:

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

If indentation is missing, Python will raise an IndentationError:

```
def greet():
print("Hello!")  # Incorrect indentation (will cause an error)
```

The return Statement for Returning Values

Functions often process data and return a result using the return statement. This allows functions to send values back to the caller for further use.

Example of a function that returns a value:

```
def square(number):
    return number * number


result = square(5)

print(result)  # Output: 25
```

If a function does not include a return statement, it implicitly returns None.

```
def say_hello():

    print("Hello!")


output = say_hello()

print(output)  # Output: Hello! followed by None
```

By using functions, Python programs become more modular and reusable, allowing developers to write efficient, organized, and maintainable code. Functions form the foundation for structuring Python programs, enabling code reuse and logical separation of concerns.

Calling Functions

Once a function is defined, it must be **called** to execute its code. Calling a function runs the statements inside its body, optionally processing input values (arguments) and returning an output. Functions help organize code into reusable components, making programs more modular and efficient.

Syntax for Calling a Function

A function is called by using its name followed by parentheses. If the function expects arguments, they are passed inside the parentheses.

Example of a simple function call:

```
def greet():

    print("Hello, welcome to Python!")


greet()  # Calling the function
```

When the greet() function is called, it executes the print statement, displaying the message. If parentheses are omitted, the function will not be executed, and instead, its reference will be returned:

```
print(greet)  # Output: <function greet at 0x...>
```

The parentheses are essential to invoke the function.

Passing Arguments to Functions

Functions often require input values to work with. These values, called **arguments**, are passed into the function when it is called. A function defines **parameters** in its signature, which act as placeholders for arguments.

Distinction Between Parameters and Arguments

- **Parameters** are variables listed inside the function definition. They define what inputs the function expects.
- **Arguments** are actual values passed to the function when it is called. They replace the parameters.

Example:

```
def greet_user(name):  # 'name' is a parameter

    print(f"Hello, {name}!")


greet_user("Alice")  # 'Alice' is an argument

greet_user("Bob")    # 'Bob' is an argument
```

Here, name is a parameter, and "Alice" and "Bob" are arguments that get passed when the function is called.

A function can take multiple parameters, allowing multiple arguments to be passed:

```
def add_numbers(a, b):

    return a + b


result = add_numbers(5, 3)

print(result)  # Output: 8
```

Python also supports **default parameter values**, where a function assigns a default value if no argument is provided:

```
def greet(name="Guest"):

    print(f"Hello, {name}!")


greet()          # Output: Hello, Guest!

greet("Alice")   # Output: Hello, Alice!
```

Functions That Return Values

Functions that return values produce an output that can be stored in variables, used in expressions, or passed to other functions:

```
def multiply(a, b):
    return a * b


result = multiply(4, 5)

print(result)  # Output: 20
```

Since this function returns a value, its result can be assigned to a variable and used elsewhere.

Functions That Do Not Return Values

Some functions perform actions, such as printing a message, without returning a value. If no return statement is used, Python implicitly returns None.

```
def print_message():

    print("This function only prints a message.")


output = print_message()

print(output)  # Output: None
```

Even though the function executes successfully, it does not return a meaningful value that can be assigned or used later.

Understanding the difference between returning and non-returning functions helps in designing better programs by deciding whether a function should just perform an action or provide a result for further computation. Functions play a crucial role in structuring Python programs, making them reusable and efficient.

Function Arguments

When calling a function, arguments can be passed in different ways, giving flexibility in how values are assigned to parameters. Python provides multiple methods for passing arguments, including **positional arguments**, **keyword arguments**, **default arguments**, and **variable-length arguments** using *args and **kwargs. Understanding these options allows for more versatile and readable function calls.

Positional Arguments

Positional arguments are the most common way to pass values to a function. The arguments are assigned to parameters **in the order they appear** in the function definition. The number of arguments passed must match the number of parameters.

Example:

```
def describe_person(name, age):

    print(f"{name} is {age} years old.")


describe_person("Alice", 25)  # Output: Alice is 25 years old.
describe_person(25, "Alice")  # Incorrect order leads to unintended
results
```

Here, "Alice" is assigned to name and 25 to age because they match the function's parameter order. If arguments are passed in the wrong order, it can lead to incorrect outputs or errors.

Keyword Arguments

Keyword arguments allow values to be assigned to parameters **by explicitly naming them**, rather than relying on order. This improves clarity and avoids errors due to misplaced values.

Example:

```
describe_person(age=25, name="Alice")  # Output: Alice is 25 years
old.
```

Here, even though the order is reversed, the function correctly assigns "Alice" to name and 25 to age because the arguments are explicitly named.

Using keyword arguments also enhances readability, especially when a function has many parameters:

```
def display_info(name, age, city):

    print(f"{name} is {age} years old and lives in {city}.")


display_info(name="Bob", age=30, city="New York")
```

Default Arguments

Default arguments allow a function to assign a **predefined value** to a parameter if no argument is provided when calling the function. This makes certain parameters optional.

Example:

```
def greet(name="Guest"):

    print(f"Hello, {name}!")


greet()          # Output: Hello, Guest!

greet("Alice")  # Output: Hello, Alice!
```

Here, if no argument is provided for name, it defaults to "Guest". However, if an argument is passed, it overrides the default value.

Default arguments must always come **after required parameters** in the function definition, or Python will raise a SyntaxError:

```
def example(a=10, b):  # Incorrect! Default parameter must be last

    pass
```

To fix this, ensure default parameters are at the end:

```
def example(b, a=10):  # Correct
```

```
    pass
```

Variable-Length Arguments

Python provides special syntax to allow functions to accept a **variable number of arguments**, making them more flexible.

***args for Non-Keyword Arguments**

Using *args, a function can accept **any number of positional arguments**, which are collected into a tuple.

Example:

```
def sum_numbers(*args):
    total = sum(args)
    print(f"Sum: {total}")

sum_numbers(1, 2, 3)          # Output: Sum: 6
sum_numbers(5, 10, 15, 20)    # Output: Sum: 50
```

The function can take any number of arguments, and they are automatically packed into args as a tuple.

****kwargs for Keyword Arguments**

Using **kwargs, a function can accept **any number of keyword arguments**, which are collected into a dictionary.

Example:

```
def display_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_details(name="Alice", age=25, city="Paris")
```

Output:

```
name: Alice
age: 25
city: Paris
```

Since **kwargs stores arguments as key-value pairs in a dictionary, the function can handle dynamic sets of named inputs.

Combining *args and **kwargs

Both *args and **kwargs can be used together in a function definition. *args captures positional arguments, while **kwargs collects keyword arguments.

Example:

```
def mixed_function(a, b, *args, **kwargs):

    print(f"a: {a}, b: {b}")

    print(f"Additional positional args: {args}")

    print(f"Additional keyword args: {kwargs}")


mixed_function(1, 2, 3, 4, 5, name="Alice", age=30)
```

Output:

```
a: 1, b: 2

Additional positional args: (3, 4, 5)

Additional keyword args: {'name': 'Alice', 'age': 30}
```

The function first assigns values to a and b, then collects extra positional arguments into args and additional keyword arguments into kwargs.

Python offers several ways to pass arguments to functions, providing flexibility and clarity. Positional arguments are assigned in order, while keyword arguments explicitly define values. Default parameters allow functions to have optional values, reducing the need for unnecessary arguments. *args and **kwargs enable handling dynamic numbers of inputs, making functions more adaptable. Understanding these concepts allows for writing more robust, reusable, and maintainable code.

Return Statement

The return statement in Python is used to send a value (or multiple values) from a function back to the caller. Functions can either return a single value, multiple values, or return nothing at all, in which case Python implicitly returns None. Understanding how return works is crucial for writing efficient and reusable functions.

Returning Single Values

A function can return a single value using the return statement. Once return is executed, the function exits immediately, and the specified value is sent back to the caller.

Example:

```
def square(number):

    return number * number


result = square(4)

print(result)  # Output: 16
```

Here, the function square takes a number as input, calculates its square, and returns the result. The returned value is then assigned to result and printed.

Returning values allows functions to be more versatile because the output can be stored, used in further calculations, or passed to other functions.

Returning Multiple Values (Tuples)

Python functions can return multiple values by returning them as a **tuple**. Since tuples are immutable sequences, they are a convenient way to package multiple return values into a single object.

Example:

```
def get_coordinates():
    return 10, 20  # Returns a tuple (10, 20)


x, y = get_coordinates()
print(x, y)  # Output: 10 20
```

Here, get_coordinates() returns two values, which are automatically packed into a tuple. The values are then unpacked into x and y.

Since functions return a tuple when multiple values are returned, it can also be assigned to a single variable:

```
coordinates = get_coordinates()
print(coordinates)  # Output: (10, 20)
```

Returning multiple values is useful when a function needs to provide multiple pieces of information at once, such as processing data and returning both a result and a status message.

Example with additional logic:

```
def divide_numbers(a, b):
    if b == 0:
        return None, "Error: Division by zero"
    return a / b, "Success"


result, status = divide_numbers(10, 2)
print(result, status)  # Output: 5.0 Success
```

Here, the function returns both the division result and a status message. If an error occurs, it returns None and an error message.

The Significance of None for Functions Without an Explicit Return

If a function does not include a return statement, or if return is written without a value, Python automatically returns None.

Example:

```
def say_hello():
    print("Hello!")


result = say_hello()
print(result)  # Output: Hello! followed by None
```

Although say_hello() prints a message, it does not explicitly return anything, so Python implicitly returns None. This is why print(result) outputs None after the greeting.

Explicitly returning None can be useful when signaling that a function does not need to return a meaningful value:

```
def log_message(message):
    print(f"Log: {message}")
    return None  # Explicitly returning None


output = log_message("System started")
print(output)  # Output: Log: System started, followed by None
```

Returning None is also useful in functions that perform an action but do not compute a result, such as logging events, updating databases, or modifying global variables.

The return statement is a powerful tool for sending values from a function to the caller. Functions can return a **single value**, **multiple values as a tuple**, or **nothing at all**, in which case Python returns None by default. Returning values makes functions more useful, allowing them to be integrated into larger calculations and processes. Understanding how to use return effectively helps in writing clean, reusable, and efficient Python programs.

Scope and Lifetime of Variables

In Python, variables have a **scope**, which defines where they can be accessed, and a **lifetime**, which determines how long they exist in memory. Understanding scope and lifetime is crucial for writing functions that behave predictably and efficiently. Python primarily distinguishes between **local** and **global** variables, and it provides the global and nonlocal keywords to control variable access in different scopes.

Local vs. Global Variables

A **local variable** is declared inside a function and is accessible **only within that function**. It is created when the function is called and destroyed when the function execution ends.

Example of a local variable:

```
def greet():
    message = "Hello, world!"  # Local variable
    print(message)
```

```
greet() # Output: Hello, world!


print(message)  # Error: message is not defined outside the function
```

Since message is defined inside greet(), it is not accessible outside the function. Trying to access it elsewhere results in a NameError.

A **global variable**, on the other hand, is defined outside any function and can be accessed **throughout the program**.

Example of a global variable:

```
message = "Hello, world!"  # Global variable


def greet():

    print(message)  # Accessible inside the function


greet()

print(message)  # Accessible outside the function
```

Since message is defined globally, it can be accessed both inside and outside functions. However, modifying global variables inside a function requires special handling.

The global Keyword

By default, if a function assigns a value to a global variable, Python treats it as a **new local variable**, which can lead to unexpected behavior.

Example of unintended behavior:

```
counter = 0  # Global variable


def increment():

    counter = counter + 1  # Creates a new local variable (not modifying global)

    print(counter)


increment()  # Error: UnboundLocalError
```

Python raises an UnboundLocalError because it assumes counter is a local variable but finds no initial assignment before modifying it.

To explicitly modify a global variable inside a function, the global keyword is used:

```
counter = 0  # Global variable
```

```
def increment():
    global counter  # Refers to the global variable
    counter += 1
    print(counter)

increment()
print(counter)  # Global counter is updated
```

With global counter, the function modifies the existing global variable instead of creating a new local one. However, excessive use of global can make code harder to debug, as functions can unexpectedly change global state.

The nonlocal Keyword *

The nonlocal keyword is used inside **nested functions** to modify variables in the **enclosing (non-global) scope**. This is useful when working with closures or nested functions.

Example of nonlocal:

```
def outer():
    count = 0  # Variable in enclosing scope

    def inner():
        nonlocal count  # Refers to count in outer()
        count += 1
        print(count)

    inner()
    inner()

outer()
```

Here, count is defined in outer(), and inner() modifies it using nonlocal. Without nonlocal, count inside inner() would be treated as a new local variable, separate from the count in outer().

Function Scope and Nested Functions

Python follows the **LEGB rule** to determine variable scope:

- **Local (L):** Variables defined inside the current function.
- **Enclosing (E):** Variables in any enclosing function (for nested functions).
- **Global (G):** Variables defined at the top level of a script.

- **Built-in (B):** Names in Python's built-in modules (e.g., print, len).

Example demonstrating different scopes:

```
x = "global"  # Global scope


def outer():
    x = "enclosing"  # Enclosing scope


    def inner():
        x = "local"  # Local scope
        print(x)


    inner()


outer()  # Output: local
print(x)  # Output: global (unchanged)
```

Here, inner() prints "local" because it first looks for x in its own scope before checking enclosing or global scopes.

If nonlocal x were used in inner(), it would modify x from outer(), and if global x were used, it would modify the top-level x.

Understanding variable scope is essential for writing clear and predictable Python programs. Local variables exist only within their function, while global variables are accessible throughout the program. The global keyword allows modifying global variables inside functions, while nonlocal is useful for modifying variables in an enclosing function. Nested functions and Python's **LEGB rule** determine how variables are accessed, ensuring structured and maintainable code.

Built-in Functions

Python provides a rich set of **built-in functions** that are available without needing to import any modules. These functions perform common tasks such as handling numbers, strings, lists, and more. Understanding and using built-in functions effectively can make programming more efficient and reduce the need for writing unnecessary custom code.

Overview of Commonly Used Built-in Functions

Python's built-in functions cover a wide range of operations, including mathematical calculations, sequence manipulations, type conversions, and input/output operations. Below are some of the most commonly used functions:

len() – Returns the length of a sequence, such as a list, tuple, or string.

```
fruits = ["apple", "banana", "cherry"]
```

```
print(len(fruits))  # Output: 3
```

max**()** – Returns the largest value from a sequence or a set of numbers.

```
numbers = [10, 20, 5, 30]

print(max(numbers))  # Output: 30
```

min() – Returns the smallest value from a sequence.

```
print(min(numbers))  # Output: 5
```

sum() – Returns the sum of all values in an iterable.

```
print(sum(numbers))  # Output: 65
```

abs() – Returns the absolute value of a number.

```
print(abs(-10))  # Output: 10
```

round**()** – Rounds a number to the nearest integer or to a specified number of decimal places.

```
print(round(3.14159, 2))  # Output: 3.14
```

sorted() – Returns a sorted version of a sequence without modifying the original.

```
nums = [3, 1, 4, 2]

print(sorted(nums))  # Output: [1, 2, 3, 4]
```

type() – Returns the type of an object.

```
print(type(42))  # Output: <class 'int'>
```

isinstance() – Checks if a value belongs to a specified type.

```
print(isinstance(42, int))  # Output: True
```

input() – Reads user input as a string.

```
name = input("Enter your name: ")

print(f"Hello, {name}!")
```

These are just a few examples of Python's built-in functions, which help simplify many programming tasks.

Using help() to Explore Built-in Functions

Python provides the help() function to get detailed information about built-in functions and modules. This function can be used interactively to understand a function's purpose, parameters, and usage.

Example of using help() on a function:

```
help(len)
```

This command displays documentation for the len() function, including what it does and how to use it.

To explore a specific module’s built-in functions, pass the module name to help():

```
import math

help(math)
```

For a quick lookup of a function’s documentation, you can also use the .__doc__ attribute:

```
print(len.__doc__)
```

This prints the function’s description in a concise format.

Python’s built-in functions provide powerful tools for performing common tasks with minimal effort. Functions like len(), max(), sum(), and sorted() simplify working with sequences, while help() allows users to explore documentation interactively. Leveraging these built-in functions improves efficiency and reduces the need for redundant code, making Python development more productive and streamlined.

Best Practices

Writing well-structured functions is essential for creating maintainable, readable, and efficient Python programs. Functions should be designed to be **clear, reusable, and easy to understand**. Following best practices helps in reducing complexity, making debugging easier, and improving overall code quality.

Writing Clean and Readable Functions

A well-written function should be **easy to read, self-explanatory, and logically structured**. Some key principles to follow include:

- Use **consistent indentation** (Python recommends four spaces per level).
- Write **descriptive docstrings** to explain the function's purpose, parameters, and return values.
- Keep function logic **simple and direct** to make it easier to follow.

Example of a clean function:

```
def calculate_area(length, width):

    """Returns the area of a rectangle given its length and
width."""

    return length * width
```

This function is short, clear, and easy to understand.

Naming Conventions and Meaningful Function Names

Function names should clearly describe what the function does. Follow these conventions for better readability:

- Use **lowercase letters with underscores** (snake_case).
- Choose **meaningful names** that reflect the function’s purpose.
- Avoid **single-letter names**, except for loop counters or temporary variables.

- Use **verbs** or action words for function names, such as calculate_total(), fetch_data(), or validate_input().

Example of good and bad function names:

```
# Good function names
def get_user_info():
    pass


def convert_to_uppercase(text):
    return text.upper()
```

```
# Bad function names
def gu():  # Unclear abbreviation
    pass


def process(t):  # What does "process" mean? What is "t"?
    pass
```

Choosing clear and descriptive names makes code easier to understand and maintain.

Avoiding Excessive Arguments

Functions with too many parameters become difficult to read and use. A good rule of thumb is to **keep parameters to a minimum** (ideally 3–4 or fewer).

Instead of passing numerous arguments, consider:

- **Using default values** for optional parameters.
- **Grouping related arguments** into dictionaries or objects.
- **Using keyword arguments** to improve readability.

Example of a function with too many parameters:

```
def create_user(first_name, last_name, age, email, phone, address):
    pass  # Too many parameters make it hard to remember their order
```

A better approach would be to use a dictionary or class:

```
def create_user(user_info):
    """Creates a user from a dictionary of details."""
    pass


user_data = {
    "first_name": "Alice",
    "last_name": "Smith",
    "age": 30,
    "email": "alice@example.com",
    "phone": "123-456-7890",
    "address": "123 Main St"
}

create_user(user_data)
```

This approach makes function calls **cleaner and more manageable**.

Keeping Functions Short and Focused on a Single Task

A function should follow the **Single Responsibility Principle (SRP)**—it should perform **one well-defined task**. Large, multi-purpose functions are harder to debug and test.

Avoid:

```
def process_data(data):
    # Cleans the data
    cleaned_data = [item.strip() for item in data]

    # Sorts the data
    sorted_data = sorted(cleaned_data)

    # Saves the data to a file
    with open("output.txt", "w") as file:
        file.writelines(sorted_data)

    return sorted_data  # Too many responsibilities!
```

Better approach:

Break the function into smaller, specialized functions:

```
def clean_data(data):
    """Removes extra spaces from each item in the list."""
    return [item.strip() for item in data]
```

```
def sort_data(data):
    """Sorts the cleaned data alphabetically."""
    return sorted(data)


def save_data(data, filename):
    """Writes the sorted data to a file."""
    with open(filename, "w") as file:
        file.writelines(data)
```

# Now, we can use these functions separately or together

```
raw_data = ["  banana", "apple ", " cherry "]
cleaned = clean_data(raw_data)
sorted_data = sort_data(cleaned)
save_data(sorted_data, "output.txt")
```

Each function now performs **one specific task**, making the code more modular, testable, and reusable.

Writing clean and readable functions is crucial for maintainable code. Using **meaningful names**, **limiting the number of arguments**, and **keeping functions short and focused** ensures clarity and reusability. By following these best practices, developers can create Python programs that are easier to understand, debug, and scale over time.

Summary

Functions are an essential part of Python programming, allowing developers to write reusable, modular, and efficient code. By defining a function once and calling it when needed, code duplication is minimized, and program structure improves. In Python, functions are created using the def keyword, followed by a function name and parentheses. Inside the function body, indented statements define its behavior, and the return statement can be used to send values back to the caller.

Calling a function executes its block of code, and arguments can be passed in different ways, including positional arguments, keyword arguments, and default arguments. Python also supports variable-length arguments using *args for arbitrary positional arguments and **kwargs for keyword arguments, providing flexibility in handling inputs. The distinction between returning values and performing actions without returning is important, as functions without a return statement implicitly return None.

Understanding scope and the lifetime of variables is crucial for writing predictable code. Local variables exist only within their function, while global variables are accessible throughout the program. The global keyword allows modifying global variables within a function, whereas nonlocal helps manage variables in nested functions. Python follows the **LEGB rule**, determining variable scope by searching in local, enclosing, global, and built-in scopes.

Python also includes a vast collection of built-in functions such as len(), max(), sum(), and sorted(), which simplify many operations without requiring additional code. The help() function provides documentation on built-in functions and modules, making it easier to explore their usage. Writing well-structured functions involves using **docstrings**, which serve as documentation inside function definitions. These can be accessed using help() or the .__doc__ attribute.

Following best practices ensures that functions remain readable, maintainable, and efficient. Function names should be meaningful and follow the snake_case naming convention. Avoiding excessive arguments improves clarity, and functions should be kept short, each focusing on a single task. Structuring functions properly reduces complexity, making debugging and modifications easier.

By understanding function syntax, argument handling, return values, scope, built-in functions, and best practices, Python developers can write clean and efficient code. Functions play a fundamental role in organizing programs, making them more scalable and easier to manage.

Exercises

1. **Basic Function Definition and Calling**
   Define a function called greet that prints "Hello, welcome to Python!" when called. Then, call the function.
2. **Function with Parameters**
   Write a function called calculate_area that takes two parameters, length and width, and returns the area of a rectangle. Call the function with sample values and print the result.
3. **Using Default Arguments**
   Modify the function greet so that it takes an optional parameter name with a default value of "Guest". If a name is provided, print "Hello, [name]!"; otherwise, print "Hello, Guest!".
4. **Returning Multiple Values**
   Write a function calculate_stats that takes a list of numbers as input and returns the **sum**, **minimum**, and **maximum** of the list as a tuple. Call the function and unpack the values into separate variables.
5. **Using *args for Variable-Length Arguments**
   Create a function sum_numbers that accepts any number of arguments using *args and returns their sum. Call it with different sets of numbers.
6. **Using **kwargs for Named Arguments**
   Write a function display_user_info that takes keyword arguments (**kwargs) for name, age, and city and prints the information in a formatted string. Call the function with different arguments.
7. **Understanding Scope: Local vs. Global Variables**
   Declare a global variable counter and define a function increment_counter that increases the value of counter by 1 using the global keyword. Call the function multiple times and print the updated value of counter.

8. **Recursive Function for Factorial Calculation**
   Write a recursive function called factorial that takes a number n and returns its factorial using recursion. Ensure that factorial(5) returns 120.

9. **Using a Built-in Function**
   Write a function get_sorted_words that takes a sentence as input and returns a list of unique words in alphabetical order. Use Python’s sorted() and set() functions.

10. **Following Best Practices: Modular Function Design**
    Rewrite the following code by breaking it into three smaller functions:

- clean_data(data): Removes extra spaces from each item in a list.
- sort_data(data): Sorts the cleaned data alphabetically.
- print_data(data): Prints each item on a new line.

```
data = ["  banana", "apple ", " cherry "]

cleaned_data = [item.strip() for item in data]

sorted_data = sorted(cleaned_data)

for item in sorted_data:

    print(item)
```