**Module 8. Functions**

**Learning Objectives**

* Understand the importance of modularity in software development using functions.
* Learn to define and use functions in Python, including syntax and structure.
* Differentiate between void functions and value-returning functions.
* Utilize various types of function arguments and parameters effectively.
* Apply advanced techniques like yield and nesting function calls for efficient coding.

1. **Why Functions?**

**Modularity in Complex Systems**

Imagine a car: it’s made up of many parts, each with its own job. If a part breaks, you can replace it without worrying if the whole car will stop working. This is because each part is modular, meaning it can work independently while fitting into the whole system.

Similarly, in software development, we need a lot of code to solve big problems. By breaking down the work into smaller parts, we can develop, maintain, and reuse code more efficiently. This is where functions come in handy.

**Divide and Conquer**

Humans solve big problems by breaking them into smaller, manageable parts. This approach, known as "divide and conquer," is a key technique in computer science. We divide a complex problem into simpler ones and solve each one individually.

**The Problem with No Functions**

If you try to build a software system without using functions, you’ll quickly see a lot of unnecessary effort and repeated code. Imagine writing a long essay without using paragraphs. It would be a mess of text with no clear structure, making it hard to read and update.

For example, if you need to calculate the sum of two numbers at multiple places in your code, and you don't use functions, you would have to write the same code snippet over and over again. This repetition is not only tedious but also error-prone. If you need to update the calculation, you’ll have to find and change every instance of that code.

**The Power of Reusability**

Using functions allows you to define a piece of logic once and reuse it wherever needed. This reduces redundancy and the chances of making mistakes. For instance, if you have a function to calculate the sum of two numbers, you can call this function whenever you need to perform this calculation, saving time and reducing errors.

**Benefits of Functions**

1. **Reduces Redundancy**: You write the code once and use it multiple times.
2. **Easier Maintenance**: Updating the function updates the logic everywhere it’s used.
3. **Improves Code Quality**: Less repetitive code means fewer chances for errors.
4. **Organizes Code**: Breaks down complex problems into manageable pieces.

Using functions helps you create cleaner, more efficient, and easier-to-maintain code. They are fundamental in building robust and scalable software systems.

1. **The Syntax of Function Definition**

In Python, the syntax for defining a function follows a specific format. Here are the key elements:

* **Header Line**: The function definition begins with the def keyword, followed by the function name, a list of parameters in parentheses, and a colon (:).
* **Function Body**: This part includes a valid block of code that performs the function's tasks. The body can contain multiple statements and should be indented.
* **Pass Statement**: If you don't want the function to do anything, you can use the pass statement. This makes it a valid block of code that does no work.

**Syntax of Function Definition**

Python functions are defined using the following syntax.

|  |
| --- |
| def function\_name():  statement  statement |

**Function Naming Rules**

Here are the naming rules for Python functions, which are similar to those for variables:

* Start with a letter or underscore (\_), followed by letters, digits, or underscores.
* Avoid using spaces and reserved words.
* Use descriptive, lowercase names with underscores to enhance readability.
* Indicate private functions with a leading underscore (\_).
* Avoid confusing or ambiguous names and opt for verb-noun pairs to clearly indicate functionality.

**Example of a Function**

Here’s a simple example of a function in Python:

|  |
| --- |
| def add\_numbers(a, b):  return a + b |

This function, ***add\_numbers()***, takes two inputs, ***a*** and ***b***, and returns their sum. Instead of writing the addition logic repeatedly, you can just call this function:

|  |
| --- |
| result = add\_numbers(5, 3)  print(result) # Output: 8 |

**Using Functions with New Data**

In programming, functions allow us to perform the same task with different data without having to rewrite the code. Here's an example to illustrate this concept:

|  |
| --- |
| def fun\_function(age, alias\_list):  """print age and aliases for the user"""  print("You're ", age, " years old. Try not to think about it")  print("Should I call you ")  for name in alias\_list:  print(name, " or ...")  print()  def main():  # Set initial data values  monty\_age = 37  alias\_list = ["Rico Suave", "Super Monty", "Lamer"]  # Call the fun\_function  fun\_function(monty\_age, alias\_list)  # Call the main function  main() |

In this example, we have a function ***fun\_function*** that takes two parameters: ***age*** and ***alias\_list***. The function prints the age and a list of aliases.

We then set some initial data:

* monty\_age = 37
* monty\_aliases = ["Rico Suave", "Super Monty", "Lamer"]

When we call the function with these values:

*fun\_function(monty\_age, monty\_aliases)*

Output will be:

You're 37 years old. Try not to think about it

Should I call you

Rico Suave or ...

Super Monty or ...

Lamer or ...

(Explanation)

* **Function Definition**: The function ***fun\_function*** is defined with two parameters, ***age*** and ***alias\_list***. This allows the function to work with any age and any list of aliases provided when it is called.
* **Setting Data**: We set specific values for ***monty\_age*** and ***monty\_aliases***.
* **Calling the Function**: The function is called with ***monty\_age*** and ***monty\_aliases*** as arguments. This is the call statement where the actual parameter values are passed to the function. The call statement is:

*fun\_function(monty\_age, monty\_aliases)*

* **Reusability**: The function can be reused with different data without changing the function itself. For example:

*fun\_function(45, ["The Rock", "Super Hero", "Cool Dude"])*

This call would produce different output, working with the new age and aliases provided.

1. **Void Functions and Value Returning Functions**

In Python, functions can be categorized into two main types: void functions and value-returning functions. Void functions are designed to perform specific tasks without returning any value to the caller. They typically execute a series of statements, such as printing output, modifying global variables, or writing to files, and are ideal for tasks where the outcome is not needed beyond the function's scope.

On the other hand, value-returning functions perform a task and then return a result using the return statement. These functions are essential for operations where the result is needed for further computations or decision-making processes. By returning a value, they enable the calling code to capture and use the result, making them versatile for calculations, data processing, and other operations. Understanding the distinction between these two types of functions is crucial for writing effective and efficient Python code.

**Void Functions**

**Void functions** are functions that perform an action but do not return any value. They might print something, modify a global variable, or write to a file, but they do not use the return keyword.

**Example:**

Let's write a simple void function that prints a greeting:

def greet(name):

    print(f"Hello, {name}!")

# Calling the function

greet("Alice")

(Explanation**)**

* The function greet takes one parameter, name.
* Inside the function, it prints a greeting message.
* When you call greet("Alice"), it prints "Hello, Alice!".

This function does something (prints a message) but does not return any value.

**Value-Returning Functions**

**Value-returning functions** are functions that perform an action and then return a value using the return keyword. These functions can be used to compute a result and give it back to the caller.

**Example:**

Let's write a simple value-returning function that adds two numbers and returns the result:

def add(a, b):

    return a + b

# Calling the function and storing the result

result = add(3, 5)

print(result)

(Explanation**)**

* The function ***add*** takes two parameters, ***a*** and ***b***.
* Inside the function, it returns the sum of ***a*** and ***b***.
* When you call ***add(3, 5)***, it returns the value ***8***, which is then stored in the variable ***result***.
* Finally, we print the value of ***result***, which is ***8***.

This function does something (adds two numbers) and returns the result to the caller.

1. **Function Arguments and Parameters**

In Python, functions are a core component for building reusable code. Understanding the concepts of arguments and parameters is essential for effective function use.

**Parameters** are the variables listed inside the parentheses in the function definition. They act as placeholders for the values that will be passed to the function when it is called.

(Example)

|  |
| --- |
| def greet(name, age):  print(f"Hello, {name}. You are {age} years old.")  greet("John", 36) # Output: "Hello, John. You are 36 years old." |

In this example, ***name*** and ***age*** are parameters of the ***greet*** function.

**Arguments** are the actual values passed to the function when it is called. These values are assigned to the corresponding parameters in the function definition.

(Example)

|  |
| --- |
| greet("Alice", 30) # Output: "Hello, Alice. You are 30 years old." |

Here, ***"Alice"*** and ***30*** are arguments passed to the ***greet*** function.

**Differences:**

* **Definition vs. Usage**: Parameters are defined in the function signature (definition), while arguments are the actual values provided during the function call.
* **Placeholder vs. Actual Value**: Parameters act as placeholders within the function, whereas arguments are the real data values that replace the placeholders.

**Similarities:**

* **Purpose**: Both parameters and arguments serve the purpose of passing data to functions.
* **Binding**: During a function call, arguments are bound to the corresponding parameters, allowing the function to use these values.

**Types of Parameters and Arguments**

* **Positional Parameters and Arguments**: The most straightforward way to pass arguments, where the order matters.

|  |
| --- |
| def multiply(x, y):  return x \* y  result = multiply(5, 3) # Positional arguments  print(result) # Output: 15 |

* **Keyword Parameters and Arguments**: Allows you to pass arguments by explicitly naming them.

|  |
| --- |
| result = multiply(x=5, y=3) # Keyword arguments  print(result) # Output: 15 |

* **Default Parameters**: You can provide default values for parameters, making them optional when calling the function.

|  |
| --- |
| def greet(name, age=25):  print(f"Hello, {name}. You are {age} years old.")  greet("Bob") # Uses default age  greet("Alice", 30) # Overrides default age |

* **Variable-Length Parameters**: Functions can accept an arbitrary number of arguments using ***\*args*** for positional arguments and ***\*\*kwargs*** for keyword arguments.

|  |
| --- |
| def summarize(\*args):  return sum(args)  print(summarize(1, 2, 3)) # Output: 6  def print\_info(\*\*kwargs):  for key, value in kwargs.items():  print(f"{key}: {value}")  print\_info(name="Alice", age=30, city="New York") |

**Example 1: Positional and Keyword Arguments**

|  |
| --- |
| def describe\_pet(pet\_name, animal\_type='dog'):  print(f"\nI have a {animal\_type}.")  print(f"My {animal\_type}'s name is {pet\_name}.")  describe\_pet('Willie') # Positional argument  describe\_pet(pet\_name='Harry', animal\_type='hamster') # Keyword arguments |

(Explanation)

* **Function Definition**: The function ***describe\_pet*** is defined with two parameters: ***pet\_name*** and ***animal\_type***. The parameter ***animal\_type*** has a default value of ***'dog'***.
* **Positional Argument Call**: ***describe\_pet('Willie')***
  + Here, 'Willie' is passed as a positional argument to pet\_name.
  + Since no value is provided for animal\_type, it uses the default value 'dog'.
  + Output:

I have a dog.

My dog's name is Willie.

* **Keyword Arguments Call**: ***describe\_pet(pet\_name='Harry', animal\_type='hamster')***
  + The arguments are passed using their parameter names.
  + pet\_name is assigned the value 'Harry' and animal\_type is assigned the value 'hamster'.
  + Output:

I have a hamster.

My hamster's name is Harry.

**Example 2: Default Parameters**

|  |
| --- |
| def make\_shirt(size='Large', message='I love Python'):  print(f"Making a {size} shirt with the message: '{message}'")  make\_shirt()  make\_shirt(size='Medium')  make\_shirt(message='Hello World', size='Small') |

**Example 3: Variable-Length Arguments**

|  |
| --- |
| def make\_pizza(\*toppings):  print("\nMaking a pizza with the following toppings:")  for topping in toppings:  print(f"- {topping}")  make\_pizza('pepperoni')  make\_pizza('mushrooms', 'green peppers', 'extra cheese')  def build\_profile(first, last, \*\*user\_info):  profile = {'first\_name': first, 'last\_name': last}  profile.update(user\_info)  return profile  user\_profile = build\_profile('albert', 'einstein', location='princeton', field='physics')  print(user\_profile) |

By understanding and using parameters and arguments effectively, you can create flexible and reusable functions that handle various scenarios in your programs.

1. **The *if \_\_name\_\_ == "\_\_main\_\_":*  Statement**

The ***if \_\_name\_\_ == "\_\_main\_\_":*** statement is a common construct in Python programming. It is used to ensure that certain code is only executed when the script is run directly, and not when it is imported as a module in another script.

**What is \_\_name\_\_?**

In Python, ***\_\_name\_\_*** is a special built-in variable that represents the name of the module. Depending on how the module is being used, the value of ***\_\_name\_\_*** changes:

* If the module is being run directly (i.e., you run the script from the command line or an IDE), ***\_\_name\_\_*** is set to ***"\_\_main\_\_"***.
* If the module is being imported into another module, ***\_\_name\_\_*** is set to the module's name.

**Why Use *if \_\_name\_\_ == "\_\_main\_\_":*?**

This construct allows you to control the execution of code based on whether the module is run as a standalone script or imported as a module. It's particularly useful for:

* **Running Tests or Example Code:**
  + You can include tests or example usage of your functions in the same file, and they will only be executed if the file is run directly.
* **Avoiding Side Effects:**
  + When a module is imported, you usually don't want certain parts of the code (like script-specific setup, user prompts, or other side effects) to run automatically.

**How Does It Work?**

Here's a basic example to illustrate the use of ***if \_\_name\_\_ == "\_\_main\_\_":***:

|  |
| --- |
| # my\_module.py  def my\_function():  print("Hello from my\_function!")  if \_\_name\_\_ == "\_\_main\_\_":  print("This is executed when the script is run directly")  my\_function() |

* **Scenario 1: Running the script directly**

If you run the my\_module.py script in your Command Prompt or Terminal:

|  |
| --- |
| > python my\_module.py |

The output will be:

|  |
| --- |
| This is executed when the script is run directly  Hello from my\_function! |

In this scenario, since the script is run directly, the condition ***if \_\_name\_\_ == "\_\_main\_\_":*** evaluates to ***True***, and the code inside the block is executed.

**Scenario 2: Importing the module**

If you import the my\_module.py module from another script and call the function using the module name as follows:

|  |
| --- |
| # another\_script.py  import my\_module  my\_module.my\_function() # Call my\_function() using the module name |

The output will be:

|  |
| --- |
| Hello from my\_function! |

In this scenario, when ***my\_module*** is imported, the condition ***if \_\_name\_\_ == "\_\_main\_\_":*** evaluates to ***False***, so the code inside the block is not executed. Only ***my\_function()*** is called explicitly in ***another\_script.py***.

1. **The yield Statement**

The ***yield*** statement in Python is used within a function to create a generator. A generator is a special type of iterator that allows you to iterate through a sequence of values, producing them one at a time, only when requested. This is different from returning all values at once.

**Basics of yield**

When a function includes the yield statement, it becomes a generator function. Calling this function does not execute its code immediately. Instead, it returns a generator object. You can then use the ***next()*** function on this generator to get the next value produced by the generator.

**Example 1: Fixed Result Generator**

Here’s a simple generator function that yields the same string every time:

|  |
| --- |
| def my\_generator():  """  Generator function that yields the same string when 'next' is called  """  yield "More research is needed…"  def main():  # Use the generator and obtain its next value  print("I saved a bunch on the latest A.I.")  print("Unfortunately, all it replies with is", next(my\_generator()))  print(next(my\_generator()))  print(next(my\_generator()))  print()  if \_\_name\_\_=="\_\_main\_\_":  main() |

Output:

I saved a bunch on the latest A.I.

Unfortunately, all it replies with is More research is needed...

More research is needed...

More research is needed...

In this example, each call to ***next(myGenerator())*** creates a new generator and yields ***"More research is needed…"***.

**Example 2: Collection Generator**

A more practical example of a generator function yields successive elements from a collection:

|  |
| --- |
| def my\_generator2(my\_aliaslist):  """  Generator function that yields the same string when 'next' is called  """  yield from my\_aliaslist  def main():  alias\_list = ["monty", "python", "lamer"]  it = my\_generator2(alias\_list)  print("So, what do you call yourself?")  print(next(it), "?")  print(next(it), "?")  print(next(it), "?")    if \_\_name\_\_=="\_\_main\_\_":  main() |

Output:

So, what do you call yourself?

monty ?

python ?

lamer ?

In this example:

* The ***my\_generator2*** function takes a list (***my\_aliaslist***) and yields each element one at a time.
* yield from ***my\_aliaslist*** is a shorthand for yielding each element in ***my\_aliaslist***.

**Key Points of yield**

1. **State Retention**: Generators retain their state between calls. This means they remember where they left off in the iteration process.
2. **Lazy Evaluation**: Generators produce items only when requested, which is useful for handling large datasets efficiently.
3. **Simplified Code**: Using yield can make code more readable and maintainable compared to manually managing an iterator.
4. **Nesting Function Calls**

Nesting function calls is a common technique in Python and many other programming languages. It allows you to combine multiple function calls into a more concise and organized structure. This involves placing a function call within the argument list of another function call, so that the result of the inner call is used as an argument for the outer call.

**Example: Finding the Maximum Score**

|  |
| --- |
| def main():  monty\_scores = [77, 88, 75]  bob\_scores = [88, 99, 95]  print("The highest score this term is: ")  print(max([max(monty\_scores), max(monty\_scores)]))  if \_\_name\_\_=="\_\_main\_\_":  main() |

Output:

The highest score this term is:

99

(Explanation)

* **Individual Max Calculations**:
  + ***max(monty\_scores)*** calculates the maximum value in ***monty\_scores***, which is ***[77, 88, 75]***. The result is ***88***.
  + ***max(bob\_scores)*** calculates the maximum value in ***bob\_scores***, which is ***[88, 99, 95]***. The result is ***99***.
* **Creating a New List**:
  + A new list is created with the results of the previous max calculations: ***[88, 99]***.
* **Outer Max Calculation**:
  + ***max([88, 99])*** calculates the maximum value in the new list, which is 99.
* **Print Statement**:
  + The result of ***max([88, 99])***, which is ***99***, is passed to the print function and printed.

**Advantages of Nesting**

* **Conciseness**: Nesting reduces the number of intermediate variables and makes the code more concise.
* **Readability**: For experienced programmers, nested calls can be more readable as they reduce clutter.
* **Efficiency**: By avoiding temporary variables, nesting can sometimes improve the efficiency of the code.

1. **Calling from the Call**

The pattern of 'immediate call' in Python allows you to chain multiple operations in a concise manner. This technique involves using the result of one call immediately as the input for another operation or function call. It effectively condenses a sequence of processing steps into a single, compact statement.

**Example: Processing a List of Usernames**

|  |
| --- |
| def main():  monty\_usernames = [" Super Monty112 ", " Rico Suave155 ", " Monty man178 "]  print("It's incredible that you can call yourself ")  print(monty\_usernames[0].strip()[:-3].upper())  print(" with a straight face")  if \_\_name\_\_=="\_\_main\_\_":  main() |

Output:

It's incredible that you can call yourself

SUPER MONT

with a straight face

(Explanation)

* **List Indexing**: ***monty\_usernames[0]*** retrieves the first username from the ***monty\_usernames*** list, which is ***" Super Monty112 "***.
* **String Stripping**: The ***strip()*** method is called on the result of the first step, removing any leading or trailing whitespace. The resulting string is ***"Super Monty112"***.
* **String Slicing**: The sliced ***string [:-3]*** removes the last three characters from the string ***"Super Monty112"***, resulting in ***"Super Mont"***.
* **String Uppercase Conversion**: The ***upper()*** method is called on the sliced string to convert it to uppercase, resulting in ***"SUPER MONT"***.

By chaining operations, you avoid intermediate variables, making the code more concise.

Each operation is performed sequentially on the result of the previous operation. While chaining can improve conciseness, it may sometimes reduce readability if overused or if the chain is too long. Use it judiciously to maintain a balance.

**Summary**

1. Functions provide modularity in software development, allowing independent components to work together.
2. Breaking down complex problems into smaller parts using functions follows the "divide and conquer" approach.
3. Without functions, code becomes repetitive and difficult to maintain, similar to writing an essay without paragraphs.
4. Functions enhance reusability by defining logic once and reusing it multiple times, reducing redundancy.
5. Functions improve code quality by minimizing repetition and the potential for errors.
6. Defining a function in Python involves using the ***def*** keyword, followed by the function name, parameters, and a colon.
7. The function body contains indented code that performs the function’s tasks, and the pass statement can be used for empty functions.
8. Void functions perform actions without returning values, like printing output or modifying global variables.
9. Value-returning functions compute results and return values using the return keyword, useful for further computations.
10. Parameters are placeholders in function definitions, while arguments are actual values passed during function calls.
11. Functions can have positional, keyword, default, and variable-length parameters to handle various argument types.
12. The ***yield*** statement in Python creates a generator, producing values one at a time upon request, retaining state between calls.
13. Nesting function calls allows combining multiple operations into a concise structure, improving code efficiency and readability.
14. Immediate call patterns chain multiple operations in a single statement, avoiding intermediate variables for conciseness.
15. Using advanced function techniques like yield and nesting can enhance coding efficiency and maintainability.

**Programming Exercises**

1. **Temperature Converter**

Write a program that asks the user to enter a temperature in Celsius, then converts that temperature to Fahrenheit using a function. The conversion formula is as follows:

Create and use a function for the conversion.

1. **Compound Interest Calculator**

Write a program that calculates the compound interest earned on an investment. The program should ask for the principal amount (P), the annual interest rate (r), the number of years (t), and the number of times interest is applied per time period (n) during the investment period.

Use the following formula for compound interest:

* A – final Amount
* P – principal Amount
* r – annual interest rate
* n – number of times interest applied per time period
* t – number of time periods (years) elapsed

Create and use a function to perform the compound interest calculation.

1. **Body Mass Index (BMI) Calculator**

Write a program that asks the user to enter their weight in kilograms and height in meters. The program should then calculate and display the user's BMI using a function.

The formula for BMI is:

1. **Grade Average Calculator**

Write a program that asks the user to enter five test scores. The program should calculate the average of the test scores using a function and display it. Implement the calculation and display logic in separate functions.

1. **Tip Calculator**

Write a program that asks the user to enter the cost of a meal at a restaurant. The program should then calculate and display the tip amounts for 15%, 18%, and 20% of the meal cost using functions. Create and use a function to calculate the tip for a given percentage.

1. **Leap Year Checker**

Write a program that asks the user to enter a year. The program should then determine whether the year is a leap year using a function. A year is a leap year if it is divisible by 4, but not divisible by 100, unless it is also divisible by 400.

1. **Retail Price Calculator**

Write a program that asks the user to enter the wholesale cost of an item and its markup percentage. The program should calculate and display the retail price of the item using a function.

The formula is:

1. **Fibonacci Sequence Generator**

Write a program that asks the user for a number ***n*** and then generates and displays the first ***n*** numbers in the Fibonacci sequence using a function. Implement the sequence generation in a function and use it to produce the output.

1. **Grade Calculator**

Write a program that asks the user to enter a score out of 100. The program should then display the corresponding letter grade using a function based on the following scale:

* A: 90-100
* B: 80-89
* C: 70-79
* D: 60-69
* F: 0-59 Create and use a function to determine the letter grade.

1. **Prime Number Checker**

Write a program that asks the user to enter a number. The program should then determine whether the number is a prime number using a function.

Implement the prime-checking logic in a function and use it to check the input number.