Chapter 5

Functions, Modules and Packages

1. Learning objectives

By the end of this chapter, students will be able to:

- Explain why functions are useful in Python programs.
- Define and call functions using `def`.
- Distinguish between *parameters* and *arguments*.
- Use `return` to send results back to the calling code.
- Return more than one value from a function.
- Define functions with default parameters.
- Understand function signatures and variable scope.
- Distinguish between *local* and *global* variables.
- Import and use Python modules.
- Use `import`, `from ... import ...`, and `import aliases`.
- Understand the purpose of packages and create a simple Python package.
- Organize a small program into reusable functions and modules.

2. Introduction

As programs become longer, writing all instructions in a single sequence can make them difficult to understand, test, and maintain. Consider a program that calculates the area of several rectangular surfaces. If the same calculation is required many times, we could repeat the multiplication instruction whenever it is needed. However, repeating code creates problems. If the calculation must be changed, we may need to modify several parts of the program. Repeated instructions also make the program longer and harder to maintain. Python provides functions to solve this problem.

A function groups instructions under a meaningful name. Once defined, it can be called whenever its operation is required. The program separates the calculation from its use. The function can be called with different dimensions without rewriting the formula.

For example:

```
def calculate_area(length, width):
    area = length * width
    return area


result = calculate_area(5, 4)
print("Area:", result)
```

Output:

```
Area: 20
```

Python also provides modules and packages to organize code across multiple files. A module is a Python file that can contain functions, variables, and other definitions. A package organizes related modules into a directory structure. These concepts help programmers move from writing simple scripts to developing organized and reusable applications.

3. Concepts

3.1. Why functions?

A function is a named, reusable block of code designed to perform a particular task.

Functions provide several advantages:

- **Code reuse:** write an operation once and use it many times.

- **Readability:** divide a program into meaningful parts.
- **Maintainability:** modify one function instead of changing repeated instructions.
- **Testing:** test individual operations separately.
- **Problem decomposition:** divide a complex problem into smaller, manageable tasks.

3.2. Defining and calling functions

A function is defined using the **`def`** keyword, followed by *its name*, *parentheses*, and a *colon* followed by indented instructions, which form the function body. Defining a function does not normally execute its body and runs when it is called.

For example:

```
def greet():
    print("Welcome to Python")


greet()
```

Output:

```
Welcome to Python
```

The instruction `def greet():` defines the function, while `greet()` calls it. A function may be called several times without repeating its definition.

3.3.Parameters and arguments

Functions often need information to perform their tasks. Parameters allow a function to receive that information.

Consider the following function that calculates speed:

```
def calculate_speed(distance, time):
    speed = distance / time
    print(speed)


calculate_speed(150, 2)
```

Output:

```
75.0
```

The names `distance` and `time` are *parameters* which introduced in the function definition. The values `150` and `2` are *arguments* and they are supplied when the function is called and they are assigned to the corresponding parameters.

In this example:

- `distance` receives `150`.
- `time` receives `2`.
- The function calculates the speed and prints `75.0`.

Important: The formula is meaningful only when the time is nonzero. A more robust function should check this condition before performing the division.

3.4. Return values

A function may perform an operation and return a result to the part of the program that called it. The `return` statement ends the current function call and sends a value back to the caller.

For example, considering the following function that calculates the square of a number.

```
def calculate_speed(distance, time):
    speed = distance / time
    return speed

result = calculate_speed(150, 2)

print(result)
```

Output:

```
75.0
```

The variable `speed` is evaluated, and its result is returned and stored in `result` variable.

A returned value can also be used directly in another expression:

```
mean_spead = (calculate_speed(150, 2) + calculate_speed(100, 4))/2

print(mean_spead)
```

Output:

```
50.0
```

`return` versus `print()`

These two instructions have different purposes.

- `print()` displays information on the screen.
- `return` sends a value back to the calling code.

For example:

```
def calc_speed_1(distance, time):
    speed = distance / time
    print(speed)


def calc_speed_2(distance, time):
    speed = distance / time
    return speed



calc_speed1(150, 40) * 2
```

```
3.75
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    calc_speed1(150, 40) * 2
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

```
calc_speed_2(150, 40) * 2
```

```
7.5
```

The first function displays the speed value but does not return it. The second function returns the speed value, allowing the caller to multiply it by two. A function without an explicit `return` statement returns `None` when it finishes.

3.5.Returning multiple values

A function can return more than one value in a single `return` statement. For example, a function may calculate both the area and perimeter of a rectangle.

```
def rectangle(length, width):
    area = length * width
    perimeter = 2 * (length + width)

    return area, perimeter


a, p = rectangle(5, 3)

print("Area:", a)
print("Perimeter:", p)
```

Output:

```
Area: 15
Perimeter: 16
```

Python returns the two values as a tuple, and the caller assigns them to two variables. The same approach can be used to return several related results, such as the minimum and maximum of two measurements.

3.6. Default parameters

A parameter can have a default value. If the caller does not supply an argument for that parameter, Python uses its default.

For example:

```
def calculate_force(mass, acceleration=9.81):
    return mass * acceleration

force1 = calculate_force(10)
force2 = calculate_force(10, 9.0)

print(force1)
print(force2)
```

Output:

```
98.10000000000001
90.0
```

The first call uses the default acceleration of `9.81`, where the second call supplies `9.0`, replacing the default for that call. A parameter with a default value is optional when calling the function. In a function definition, parameters without defaults must appear before parameters with defaults.

3.7.Function signatures

A function signature describes the function's name and the parameters it accepts, including their arrangement and any default values.

Consider:

```
def calculate_power(voltage, current=1):
    return voltage * current
```

Its signature is: `calculate_power(voltage, current=1)`

This signature indicates that:

- `voltage` is a required parameter.
- `current` has a default value of `1`.
- The function can be called with one or two arguments.

Python also allows arguments to be passed by parameter name. These are called *keyword arguments*.

```
pow = calculate_power(voltage=12, current=2)
print(pow)
```

Output:

```
24
```

Keyword arguments can improve readability when a function has several parameters. For example, the following call is also valid because the argument names identify the corresponding parameters.

```
result = calculate_power(current=2, voltage=12)
```

3.8. Variable scope: Local and global variables

Variable scope determines where a variable can be accessed in a program.

Local variables:

A local variable is created inside a function and is normally accessible only within that function.

```
def calculate_area(length, width):
    area = length * width
    return area

print(calculate_area(4, 6))
```

The variable `area` is local to `calculate_area()`. It cannot normally be accessed directly outside the function.

Global variables

A global variable is defined outside functions and can generally be read from within them.

```
acceleration=9.81
def calculate_force(mass):
    return mass * acceleration

calculate_force(10)
```

Output:

```
98.10000000000001
```

The function can read the global variable `acceleration`. However, modifying global variables from inside functions can make programs harder to understand and maintain. Passing information through parameters and returning results is usually preferable.

3.9.Modules

A module is a Python file that contains definitions and instructions. Modules allow related code to be stored separately and reused in other programs.

Python provides built-in modules and modules from its standard library. In addition, programmers can also create their own modules.

For example, the `math` module provides mathematical functions and constants.

```
import math

radius = 3
area = math.pi * radius ** 2

print(f"Area: {area:.2f}")
```

The instruction `import math` makes the module available. The expression `math.pi` accesses its value of `pi`.

Other useful functions in the module include:

- `math.sqrt(x)`: calculates the square root of a nonnegative number.
- `math.ceil(x)`: rounds a number upward to an integer.
- `math.floor(x)`: rounds a number downward to an integer.

Modules help programmers reuse existing functionality instead of implementing every operation from scratch.

3.10. Importing Modules

Python supports several ways to import functionality from modules.

Method 1: Import the module

```
import math

print(math.sqrt(49))
```

The function is accessed using the module name followed by a dot. This approach makes the origin of the function clear.

Method 2: Import a specific function

```
from math import sqrt

print(sqrt(49))
```

The function can now be called directly without writing `math.sqrt()`. Several names can be imported in one statement:

```
from math import sqrt, pi

print(sqrt(64))
print(pi)
```

This approach can be convenient when only a few names are required.

Method 3: Use an import alias

An alias gives a module or imported name a different name in the current program.

```
import math as m

print(m.sqrt(10))
print(m.pi)
```

In this case, the alias `m` is used instead of `math`.

3.11. Creating a user-defined module

A programmer can create a module by saving Python code in a `.py` file. For example, if we want to reuse several basic geometric calculations, we create a file named `geometry.py` containing:

```
def rectangle_area(length, width):
    return length * width

def rectangle_perimeter(length, width):
    return 2 * (length + width)


def circle_area(radius):
    return 3.14 * radius ** 2
```

This file defines three functions and it does not need to contain a separate program that calls them. After that, we can import this file as a module. For example, we can create another file named `main.py` in the same directory. The main program imports `geometry` and uses its functions.

```
import geometry

area = geometry.rectangle_area(5, 3)
perimeter = geometry.rectangle_perimeter(5, 3)
circle = geometry.circle_area(2)

print("Rectangle area:", area)
print("Rectangle perimeter:", perimeter)
print(f"Circle area: {circle:.2f}")
```

The two files have different responsibilities:

- `geometry.py` contains reusable calculations.
- `main.py` uses those calculations to perform a task and display results.

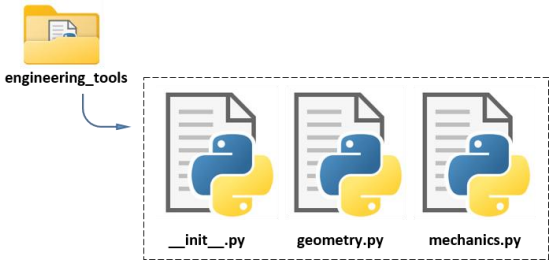
Both files should be saved in the same directory for this example to work as expected. This separation makes the code easier to test, modify, and reuse in other programs.

3.12. Packages

A package is a way of organizing related Python modules into a directory structure. A module is typically a single Python file, whereas a package groups modules that belong to a larger subject or application.

For example, a program for basic engineering calculations could be organized as follows:

```
engineering_tools/
    __init__.py
    geometry.py
    mechanics.py
```



Here:

- `engineering_tools` is the package directory.
- `__init__.py` marks the directory as a regular package and can contain initialization code or selected imports.
- `geometry.py` contains geometric calculations.
- `mechanics.py` contains calculations related to mechanics.

An `__init__.py` file can be empty. Although modern Python also supports certain namespace packages without this file, using it is appropriate for this introductory example.

Importing from a package

Suppose `geometry.py` contains:

```
def rectangle_area(length, width):
    return length * width
```

and `mechanics.py` contains:

```
def calculate_force(mass,
acceleration):
    return mass * acceleration
```



The `main` program, saved outside the package directory at the same project level, can import these modules as follows:

```
from engineering_tools import geometry, mechanics

area = geometry.rectangle_area(8, 4)
force = mechanics.calculate_force(5, 9.81)

print("Area:", area)
print(f"Force: {force:.2f} N")
```

The package keeps related functions organized while allowing the main program to access them.

For this example, the project directory should contain the `engineering_tools` directory and the main program. Run the program from the project directory or configure the Python import path appropriately. Packages are especially useful when a project contains many modules or when several programmers work on different parts of the same application.

4. Syntax Reference

The following table summarizes the principal syntax introduced in this chapter.

Purpose	Syntax
Define a function	`def function_name():`
Define parameters	`def function_name(a, b):`
Call a function	`function_name()`
Return a result	`return expression`
Return multiple values	`return value1, value2`

Define a default parameter	`def function_name(a, b=10):`
Use a keyword argument	`function_name(a=5, b=10)`
Import a module	`import module_name`
Import a specific name	`from module_name import name`
Import with an alias	`import module_name as alias`
Import modules from a package	`from package_name import module_name`

Remember that indentation is required inside a function definition, where the statements belonging to the function must be indented consistently.

5. Worked Examples

Example 1- Converting temperature

Problem

Write a function that converts a temperature from Celsius to Fahrenheit.

Solution

```
def celsius_to_fahrenheit(celsius):
    fahrenheit = (9 / 5) * celsius + 32
    return fahrenheit

# use of the function
temperature = float(input("Enter temperature in Celsius: "))

result = celsius_to_fahrenheit(temperature)

print(f"Temperature: { result:.2f} °F")
```

Example output:

```
Enter temperature in Celsius: 30

Temperature: 86.00 °F
```

Explanation

The function receives the Celsius temperature as an argument. It applies the conversion formula and returns the Fahrenheit temperature. The main program receives the input, calls the function, and displays the result.

Example 2 - Calculating the average of three measurements

Problem

A technician records three measurements. Write a function that calculates their average and returns both the average and the largest measurement.

Solution

```
def analyze_measurements(a, b, c):
    average = (a + b + c) / 3
    maximum = max(a, b, c)

    return average, maximum

# use of the function
average, maximum = analyze_measurements(12.5, 14.0, 13.5)

print(f"Average: { average:.2f}")
print(f"Maximum: { maximum:.2f}")
```

Output:

```
Average: 15.33
Maximum: 17.00
```

Explanation

The function performs two related calculations and returns both results. The caller assigns the returned values to `average` and `maximum`. The built-in function `max()` is used to identify the largest value.

Example 3 - Reusing a function with a default parameter

Problem

Write a function that calculates the final price of an item after applying a discount. The discount should be 5% by default, but the user should be able to specify another discount rate.

Solution

```
def final_price(price, discount=5):
    reduction = price * discount / 100
    return price - reduction


# use of the function
price1 = final_price(700)
price2 = final_price(700, 10)


print(f"Price with default discount: { price1:.2f}")
print(f"Price with 10% discount: { price2:.2f}")
```

Output:

```
Price with default discount: 665.00
Price with 10% discount: 630.00
```

Explanation

In the first call, the function uses the default discount of 5%. In the second call, the supplied argument replaces the default value with 10%. This example illustrates how default parameters make functions more flexible. The function assumes that the supplied price and discount are valid.

Example 4 - Organizing a small program into modules

Problem

A laboratory needs a program to calculate the volume of a rectangular container. Organize the calculation into a separate module.

Step 1: Create `container_tools.py`

```
def calculate_volume(length, width, height):
    return length * width * height
```

Step 2: Create `main.py`

```
import container_tools

length = float(input("Enter length (cm): "))
width = float(input("Enter width (cm): "))
height = float(input("Enter height (cm): "))

volume = container_tools.calculate_volume(length, width, height)

print(f"Volume: {volume:.2f} cm3")
```

Example output:

```
Enter length (cm): 20
Enter width (cm): 10
Enter height (cm): 5
Volume: 1000.00 cm^3
```

Explanation

The calculation is stored in `container_tools.py`, while `main.py` handles input and output. Both files belong in the same directory for this example. The module can later be reused by another program that needs the same calculation and the function does not need to be rewritten.

6. Common mistakes

1. Confusing `print()` with `return`

Incorrect:

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


result = multiply(3, 4)
print(result)
```

Output:

```
12
None
```

The function displays the product but does not return it.

Correct:

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


result = multiply(3, 4)
print(result)
```

Output:

```
12
```

2. Using a local variable outside its function

Incorrect:

```
def calculate(a):
    result = 10*a + 5


calculate(2)
print(result)
```

The program raises a `NameError` because result is local to `calculate(a)`.

Correct:

```
def calculate(a):
    result = 10*a + 5
    return result


result = calculate(2)
print(result)
```

3. Placing a required parameter after a default parameter

Incorrect:

```
def calculate(a=1, b):
    return a + b
```

Python raises a `SyntaxError` because a parameter without a default cannot follow a parameter with a default in this function definition.

Correct:

```
def calculate(b, a=10):
    return a + b
```

4. Forgetting the module name when using `import`

Incorrect:

```
import math

print(sqrt(25))
```

The name `sqrt` has not been imported directly.

Correct:

```
import math

print(math.sqrt(25))
```

Alternatively:

```
from math import sqrt

print(sqrt(25))
```

5. Naming a file after a standard library module

A file named `math.py`, for example, may interfere with importing Python's standard `math` module when the local file is found first. Choose descriptive names that do not conflict with standard library modules or other important dependencies. Also, keep the main program and the user-defined module in the expected directory, and run the program from an appropriate working directory.

6. Repeating code instead of reusing a function

If the same calculation appears several times, consider defining a function and calling it with different arguments. However, not every instruction needs its own function. Functions should represent meaningful operations and make the program easier to understand.

7. Exercises

Exercise 1

Write a function named `is_even(number)` that returns `True` if the integer is even and `False` otherwise.

Exercise 2

Write a function named `circle_properties(radius)` that returns the area and perimeter of a circle.

Exercise 3

Write two functions:

- `km_to_miles(km)`, which converts kilometers to miles.
- `miles_to_km(miles)`, which converts miles to kilometers.

Use the approximate conversion factor: 1 mile = 1.60934 km

Exercise 4

Write a function that calculates the area and the volume of a cylinder.

Exercise 5

Write a function to find the roots of a quadratic equation $ax^2 + bx + c = 0$.

Exercise 6

1. Create a file named `basic_math.py` containing the following functions:
   - `add(a, b)`: returns the sum.
   - `subtract(a, b)`: returns the difference.
   - `multiply(a, b)`: returns the product.
   - `divide(a, b)`: returns the quotient when the divisor is not zero.
2. Create a second file named `main.py` that imports the module and calls its functions.
3. Test the division function with a zero divisor and handle that case appropriately.

Exercise 7

1. Create a module named `geometry.py` containing functions to calculate:
   - The area of a rectangle.

- The area of a triangle.
- The area of a circle.

2. Write a separate `main` program that imports the module and uses all three functions.

Exercise 8

Develop a small engineering calculator organized into separate modules.

The program should:

1. Provide functions for calculating the area of a rectangle and a circle.
2. Provide a function for calculating force using mass and acceleration.
3. Store these functions in appropriate modules.
4. Use a main program to obtain the required inputs and display the results.
5. Use meaningful function names and avoid repeating calculation formulas.
6. Handle invalid inputs or mathematically invalid cases, such as a negative radius or a negative mass, according to rules you define and explain.

8. Summary

In this chapter, we learned how to organize Python programs using functions, modules, and packages.

The main concepts are:

- A function is a reusable block of code defined with `def`.
- A function call executes the function.
- *Parameters* receive information, while *arguments* are the values supplied during a call.
- The `return` statement sends results back to the caller.
- A function can returns multiple values.
- Default parameters allow functions to be called without supplying every argument.
- A *function signature* describes its name and parameters.

- *Local variables* belong to their function's scope, while *global variables* are defined outside functions.
- *Modules* store related Python definitions in separate `.py` files.
- The `import` statement makes modules available to a program.
- *Packages* organize related modules into directories.
- Dividing a program into functions and modules improves readability, reuse, testing, and maintenance.

A well-organized program does more than produce the correct result. It also makes the solution easier to understand, verify, and extend.

9. Review Questions

1. What is a function, and why is it useful?
2. What is the difference between defining a function and calling it?
3. What is the difference between a *parameter* and an *argument*?
4. What is the purpose of the `return` statement?
5. How does `return` differ from `print()`?
6. What happens when a function finishes without an explicit `return` statement?
7. What is a default parameter, and when is it useful?
8. What information does a function signature provide?
9. What is the difference between a *local variable* and a *global variable*?
10. What is a Python module?
11. Explain the difference between `import math` and `from math import sqrt`.
12. What is an import alias, and why might it be useful?
13. What is the difference between a module and a package?
14. What is the purpose of `__init__.py` in a regular Python package?

10. Key Terms

Term	Meaning
Function	A reusable block of code designed to perform a specific task.
Function definition	The process of creating a function using the `def` keyword.
Function call	An instruction that executes a function.
Parameter	A variable listed in a function definition that receives a value.
Argument	A value supplied to a function when it is called.
Return value	A value sent back to the part of the program that called a function.
`return` statement	An instruction that ends a function call and optionally returns a value.
Default parameter	A parameter that has a predefined value used when no argument is supplied for it.
Function signature	The function's name and parameter specification.
Keyword argument	An argument passed by explicitly specifying the parameter name.
Variable scope	The region of a program where a variable can be accessed.
Local variable	A variable defined inside a function and normally accessible only within that function.
Global variable	A variable defined outside functions and accessible from the global scope.
Module	A Python file containing code, such as functions, variables, and other definitions, that can be imported into another program.
`Import statement`	An instruction used to make a module or its contents available to a program.
Import Alias	An alternative name assigned to an imported module or object.
Standard library	A collection of modules provided with Python.
Package	A way of organizing related Python modules into a directory structure.
`__init__.py`	A file commonly used to identify a regular Python package and optionally initialize it.
Modular Programming	An approach to software development that divides a program into separate, organized components.

11. References

1. **Python Software Foundation.** *Python 3 Documentation — The Python Tutorial and Python User Guide.*
2. **Severance, C. R.** *Python for Everybody: Exploring Data Using Python 3.*
3. **Downey, A. B.** *Think Python. How to Think Like a Computer Scientist. 2nd Edition,*
4. **Sweigart, A.** *Automate the Boring Stuff with Python.*