Beginner Programming Fundamentals With Python

Module 5

Functional Programming

Ali Samanipour

Jan 2022

Module Roadmap

Functions

A ***function*** is a **group of statements** that exist within a program for the **purpose of performing a specific task.**

Most programs perform tasks that are large enough to be broken down into several subtasks. For this reason, programmers usually break down their programs into small manageable pieces known as *functions*.

Functions …

suppose you have been asked to write a pseudocode to calculate and display the gross pay for an hourly paid employee

- Getting the employee’s hourly pay rate
- Getting the number of hours worked
- Calculating the employee’s gross pay
- Calculating overtime pay
- Calculating withholdings for taxes and benefits
- Calculating the net pay
- Printing the paycheck

- A function that gets the employee’s hourly pay rate
- A function that gets the number of hours worked
- A function that calculates the employee’s gross pay
- A function that calculates the overtime pay
- A function that calculates the withholdings for taxes and benefits
- A function that calculates the net pay
- A function that prints the paycheck

Functions and Modularization

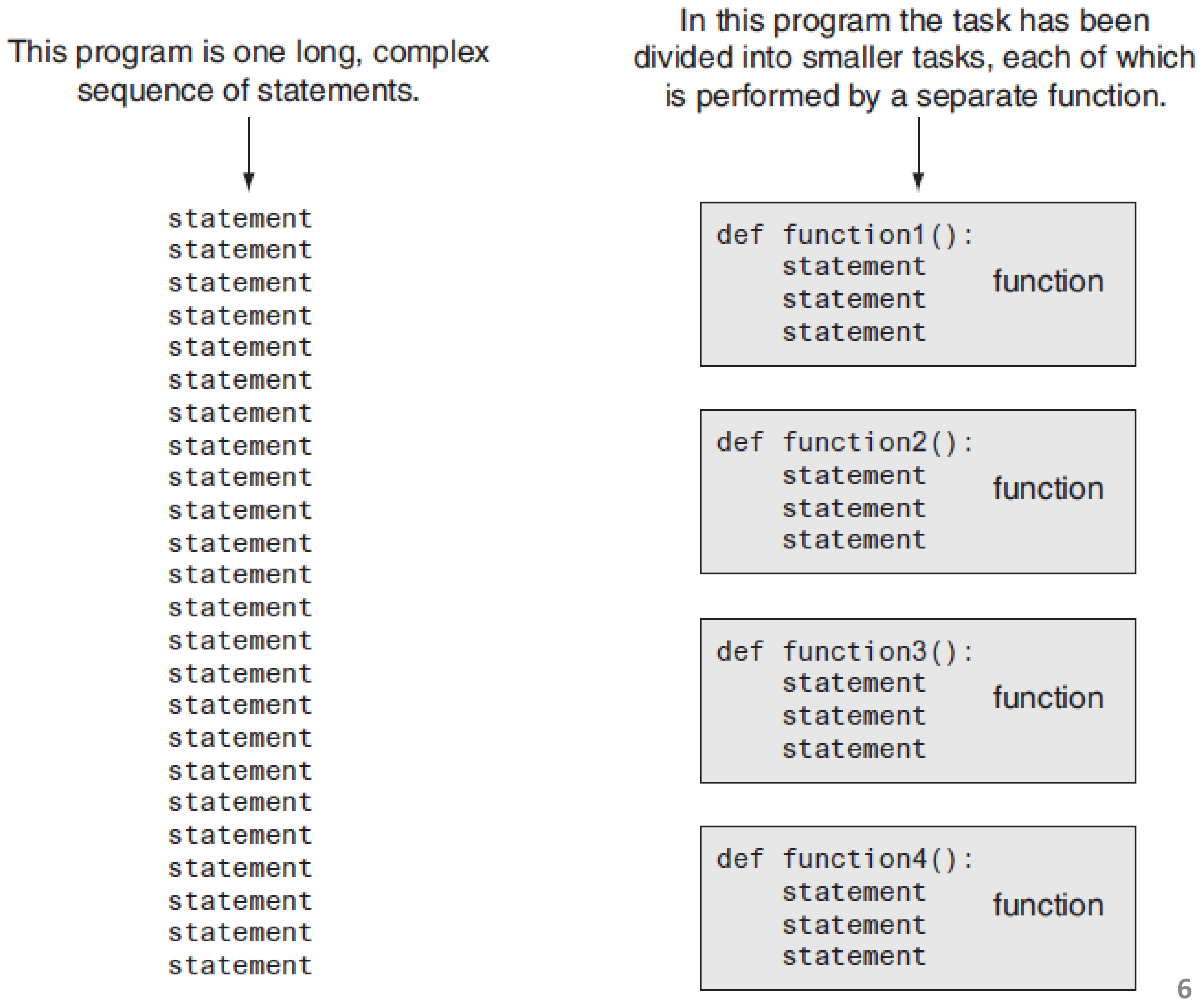
Instead of writing a large program as one long sequence of statements, it can be written as **several small functions**, each one performing a specific part of the task.

These small **functions can then be executed in the desired order to perform the overall task**.

This approach is sometimes called ***divide and conquer*** because a large task is divided into several smaller tasks that are easily performed.

A program that has been written with each task in its own function is called a ***modularized*** program.

Using functions to divide and conquer a large task



This program is one long, complex sequence of statements.

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

In this program the task has been divided into smaller tasks, each of which is performed by a separate function.

```
def function1():
    statement
    statement
    statement
```

```
def function2():
    statement
    statement
    statement
```

```
def function3():
    statement
    statement
    statement
```

```
def function4():
    statement
    statement
    statement
```

Benefits of Modularizing a Program with Functions

Simpler Code: Several small functions are much easier to read than one long sequence of statements.

Code Reuse: writing the code to perform a task once, then reusing it each time you need to perform the task.

Better Testing: When each task within a program is contained in its own function, testing and debugging becomes simpler

Faster Development: functions can be written for the commonly needed tasks, and those functions can be incorporated into each program that needs them.

Easier Facilitation of Teamwork: different programmers can be assigned the job of writing different functions.

Void Functions and Value-Returning Functions

When you call a ***void function***, it simply executes the statements it contains and then terminates.

When you call a ***value-returning function***, it executes the statements that it contains, then returns a value back to the statement that called it

The ***input*** function is an example of a value-returning function. When you call the input function, <u>**it gets the data that the user types on the keyboard**</u> and <u>**returns that data as a string**</u>

Module Roadmap

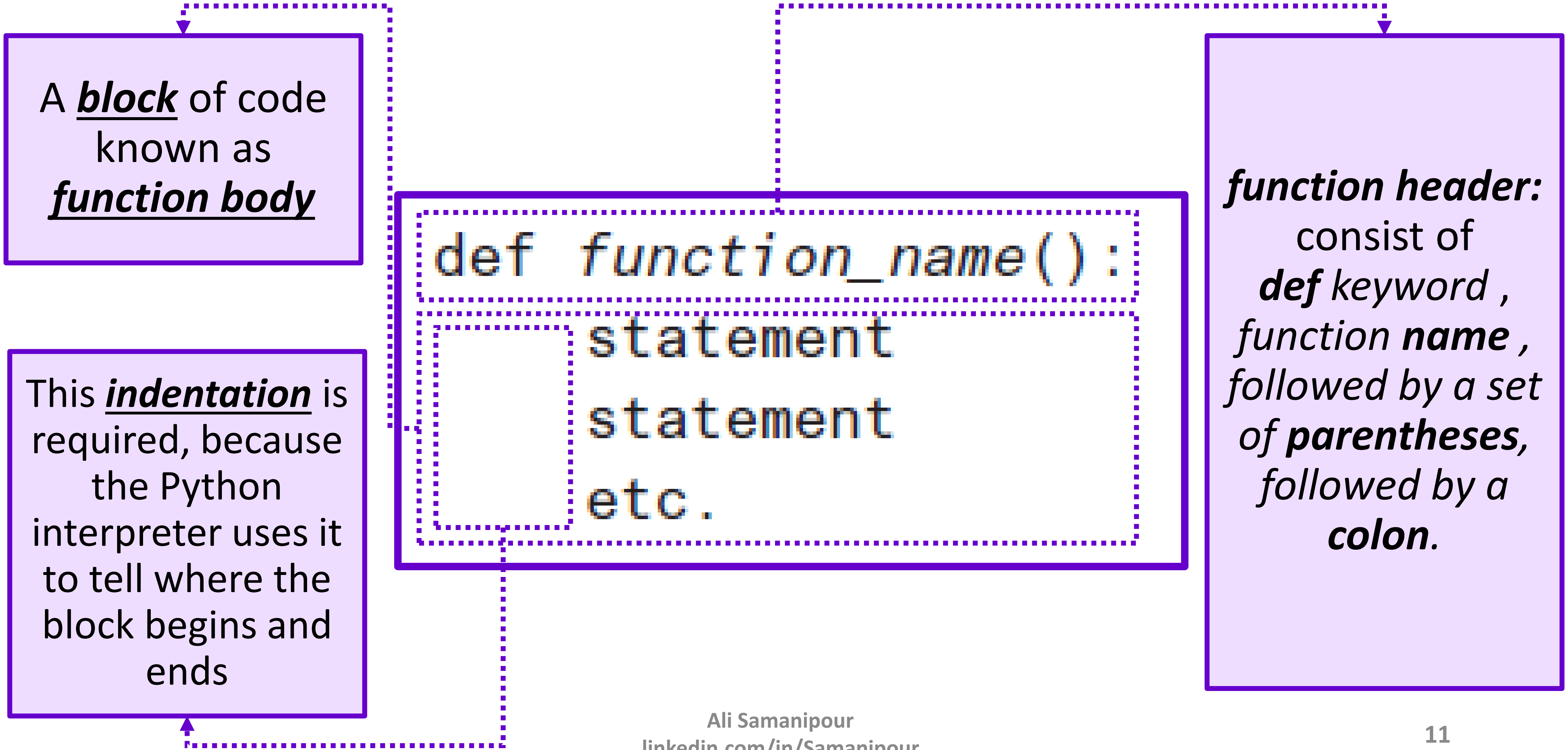
Defining and Calling a Void Function

The code for a function is known as a **function definition**. To execute the function, you write a statement that **calls** it.

Python requires that you follow the same rules that you follow when naming variables

A **function's name should be descriptive** enough so anyone reading your code can reasonably guess what the function does

Defining and Calling a Void Function …



A ***block*** of code known as ***function body***

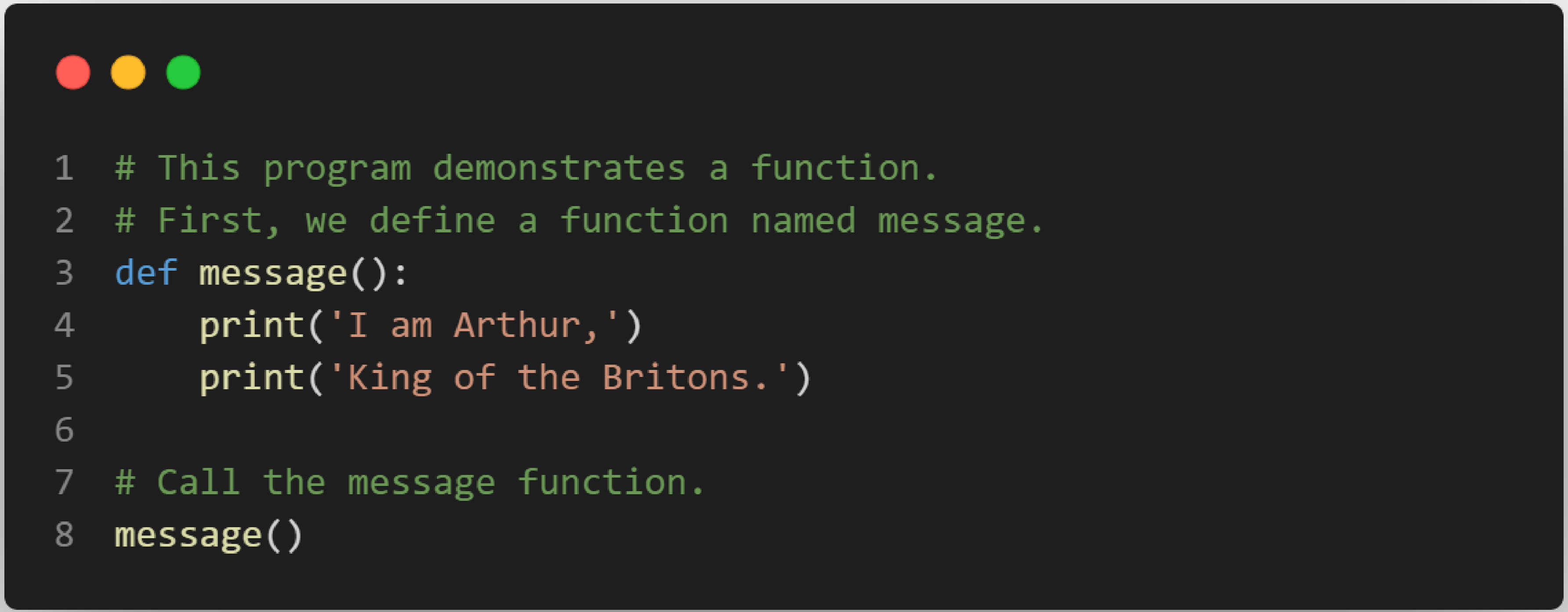
```
def function_name():
        statement
        statement
        etc.
```

This ***indentation*** is required, because the Python interpreter uses it to tell where the block begins and ends

function header: consist of ***def*** *keyword* , *function* ***name*** *,* *followed by a set of* ***parentheses,*** *followed by a* ***colon.***

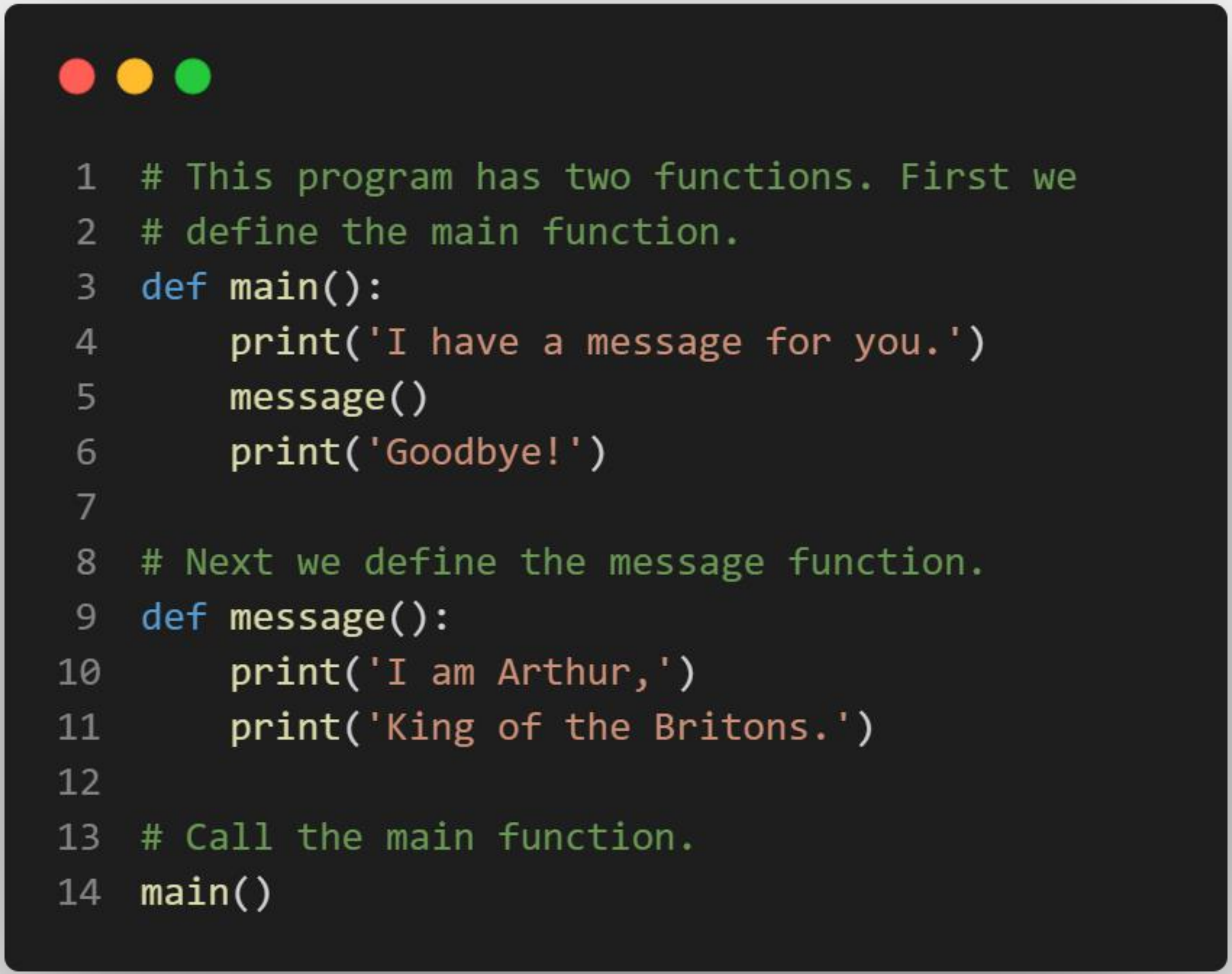
Calling a Function

A function definition specifies what a function does, but it does not cause the function to execute. **To execute a function, you must *call* it**.



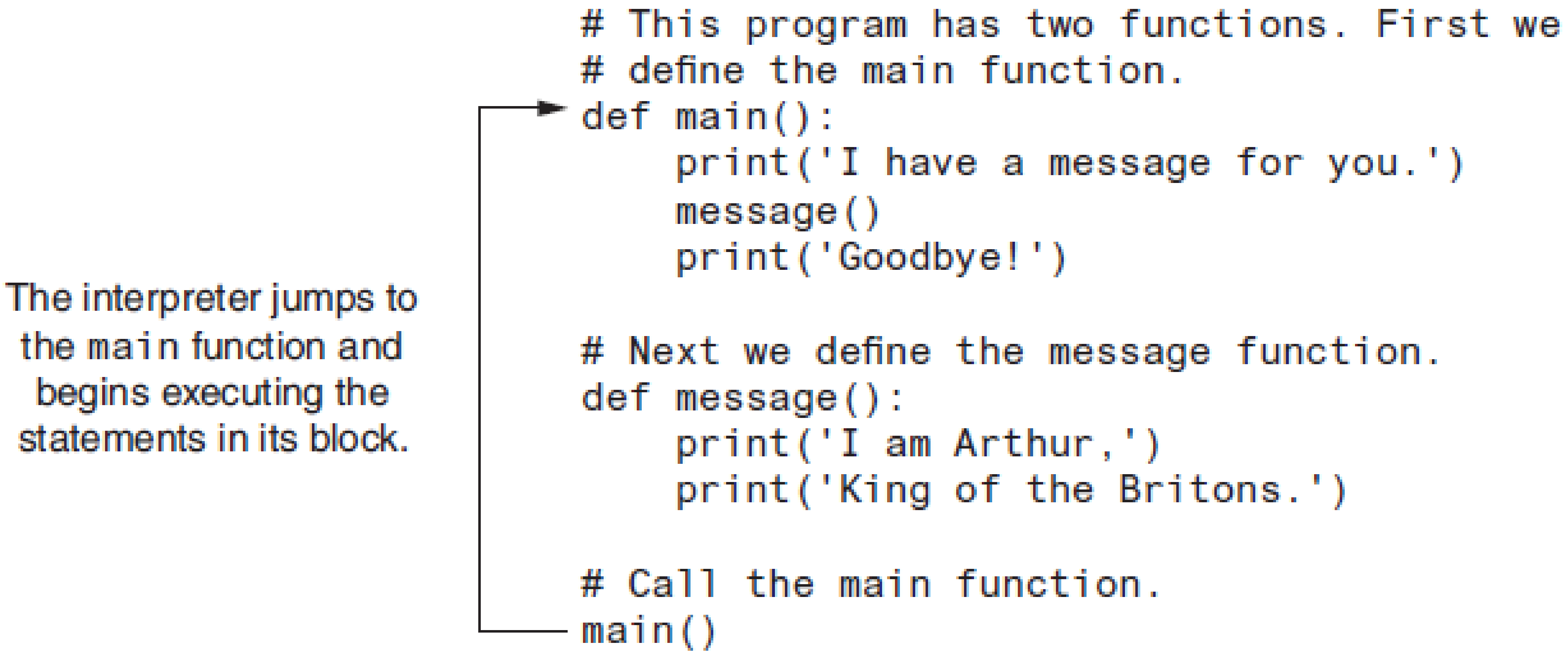
```python
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the message function.
message()
```

Tracing function calls



```python
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

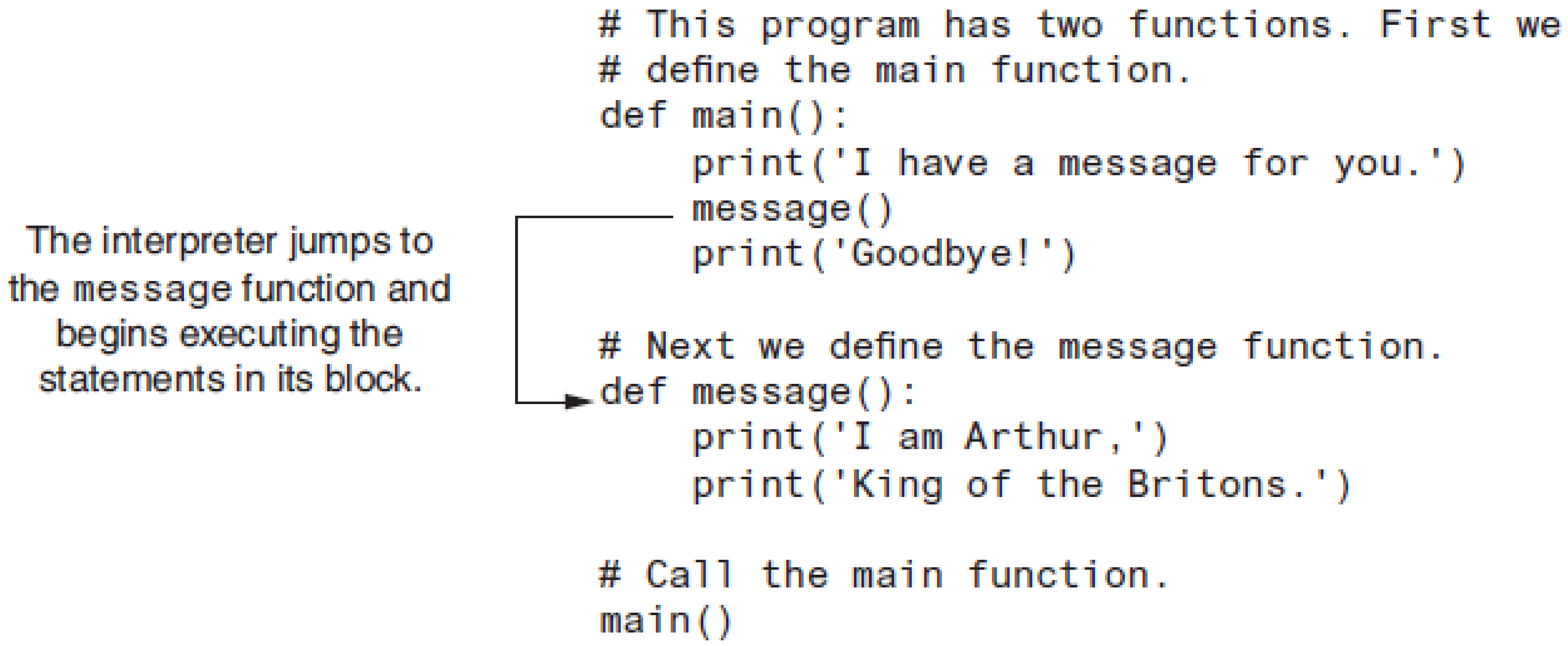
Tracing function calls



The interpreter jumps to the main function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

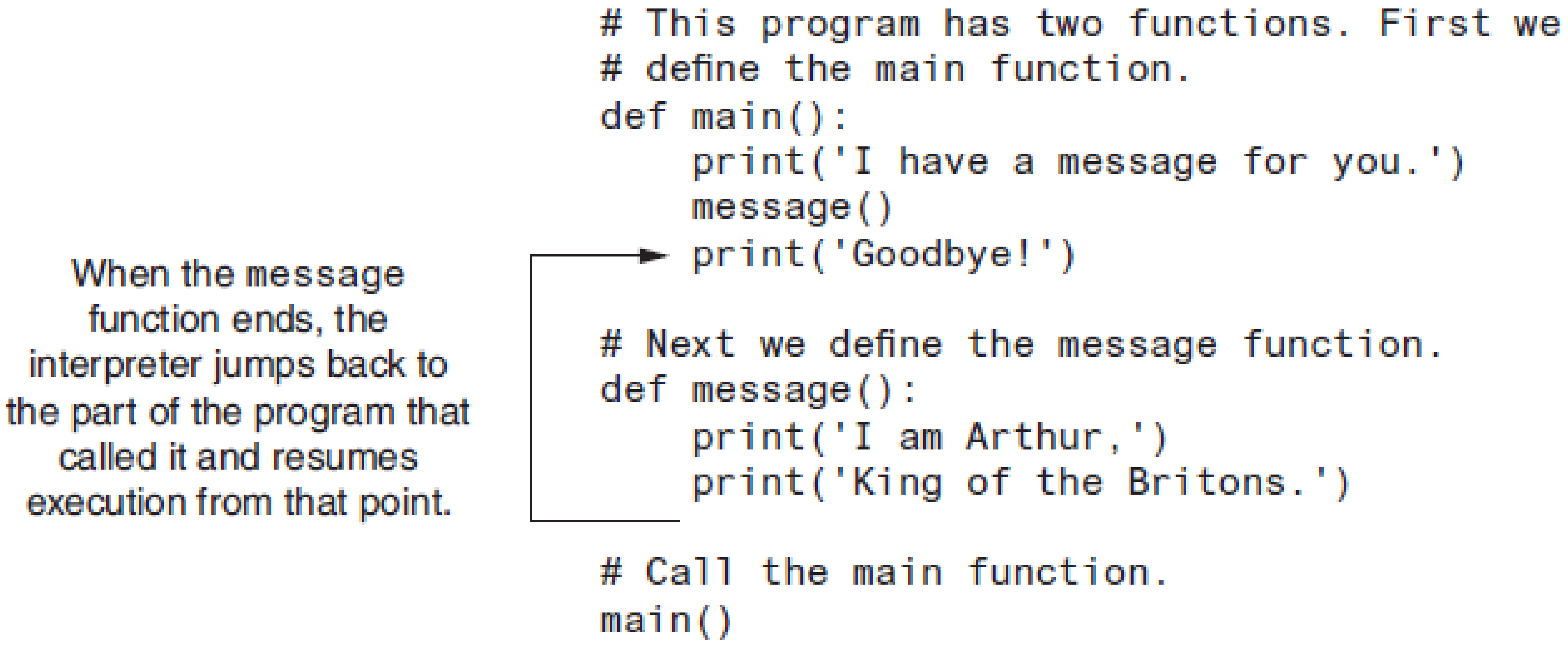
Tracing function calls



The interpreter jumps to the message function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

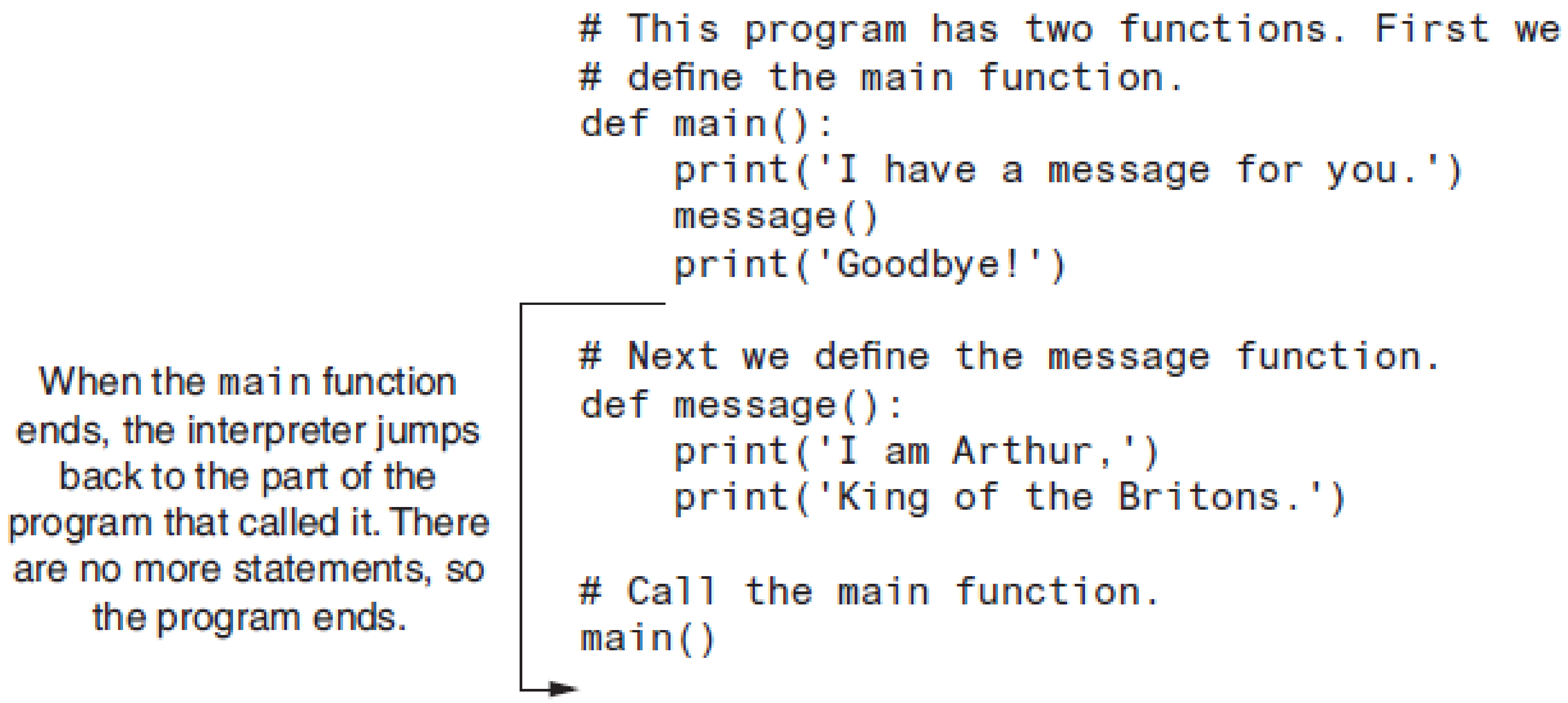
Tracing function calls



When the message function ends, the interpreter jumps back to the part of the program that called it and resumes execution from that point.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

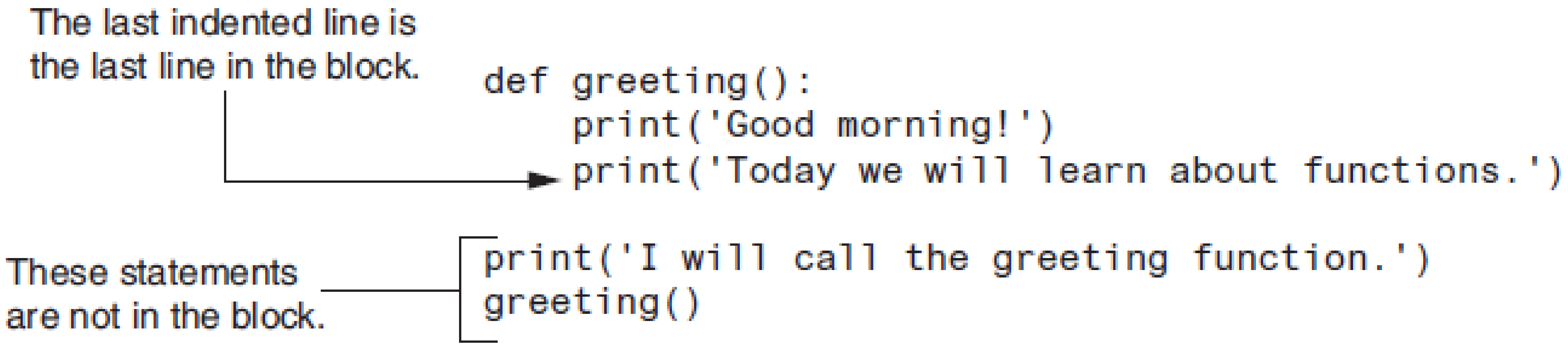
Tracing function calls



When the main function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

Indentation in Python


The last indented line is the last line in the block.
These statements are not in the block.

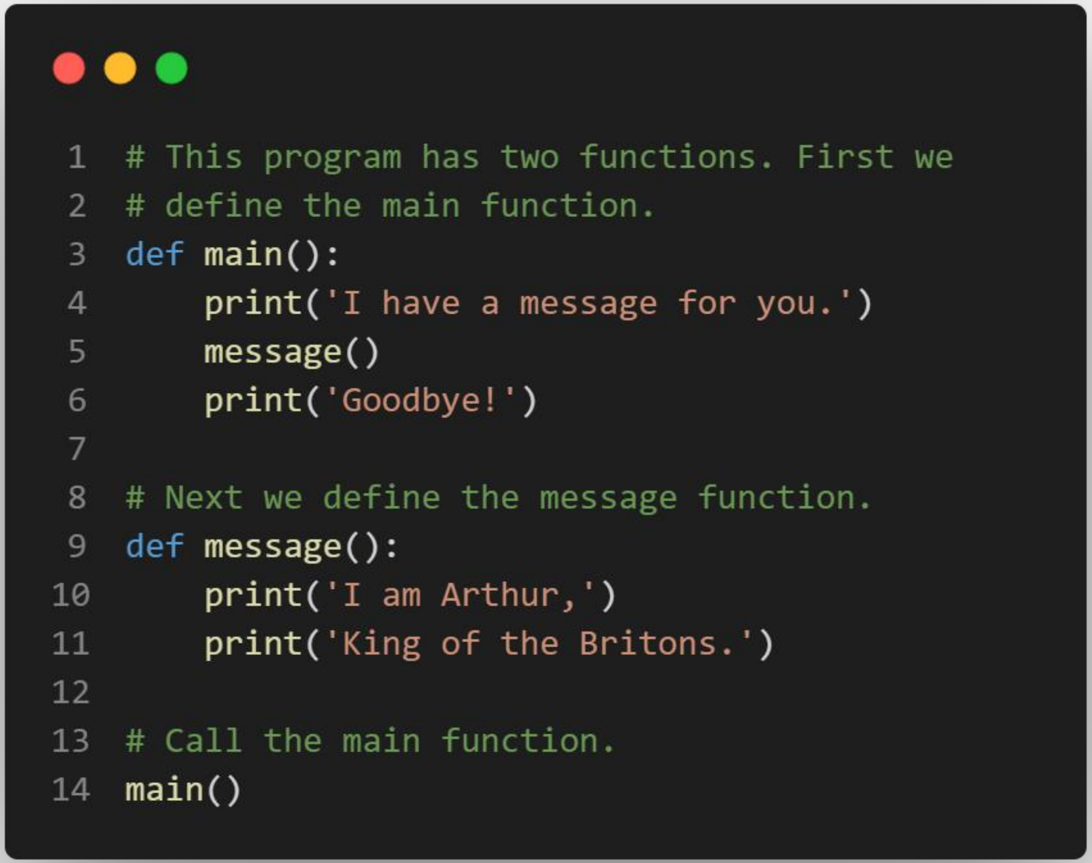
```python
def greeting():
    print('Good morning!')
    print('Today we will learn about functions.')

print('I will call the greeting function.')
greeting()
```

Module Roadmap

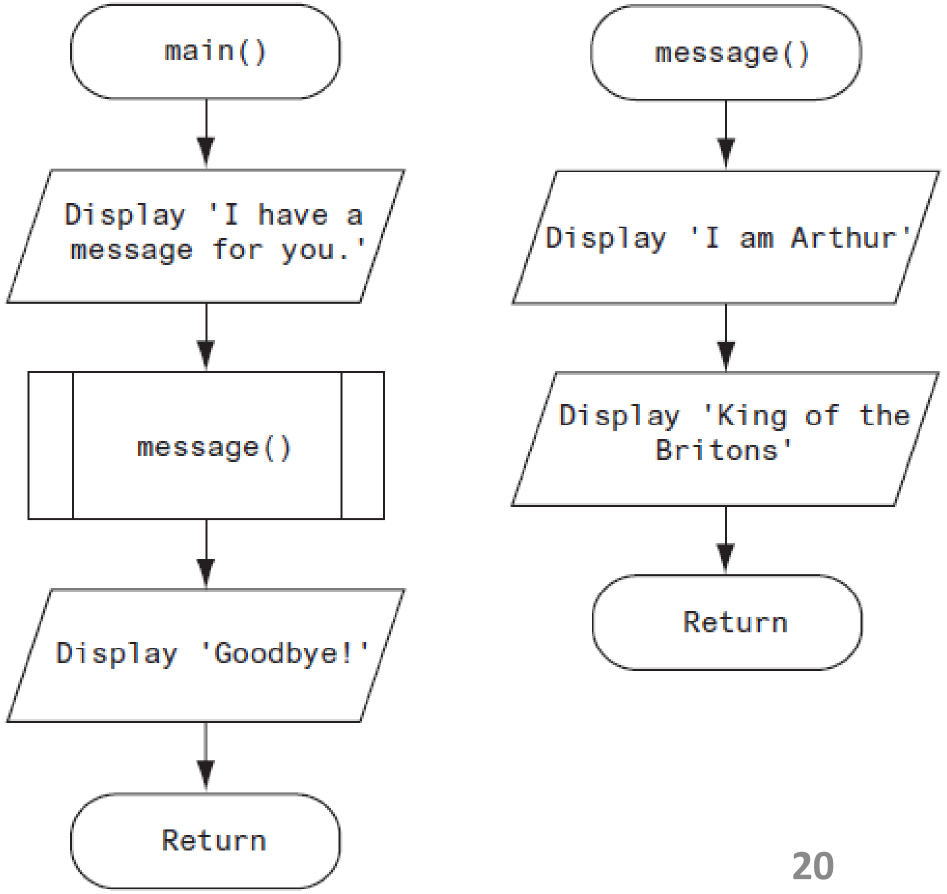
1. Introduction to Functions
2. Void Functions
3. Designing Programs Using Functions
4. Local and Global Variables
5. Value-Returning Functions
6. Storing Functions in Modules

Designing a Program to Use Functions

Programmers commonly use a technique known as ***top-down design*** to break down an algorithm into functions.



```python
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```



Top-Down Design

Programmers commonly use a technique known as ***top-down design*** to break down an algorithm into functions.

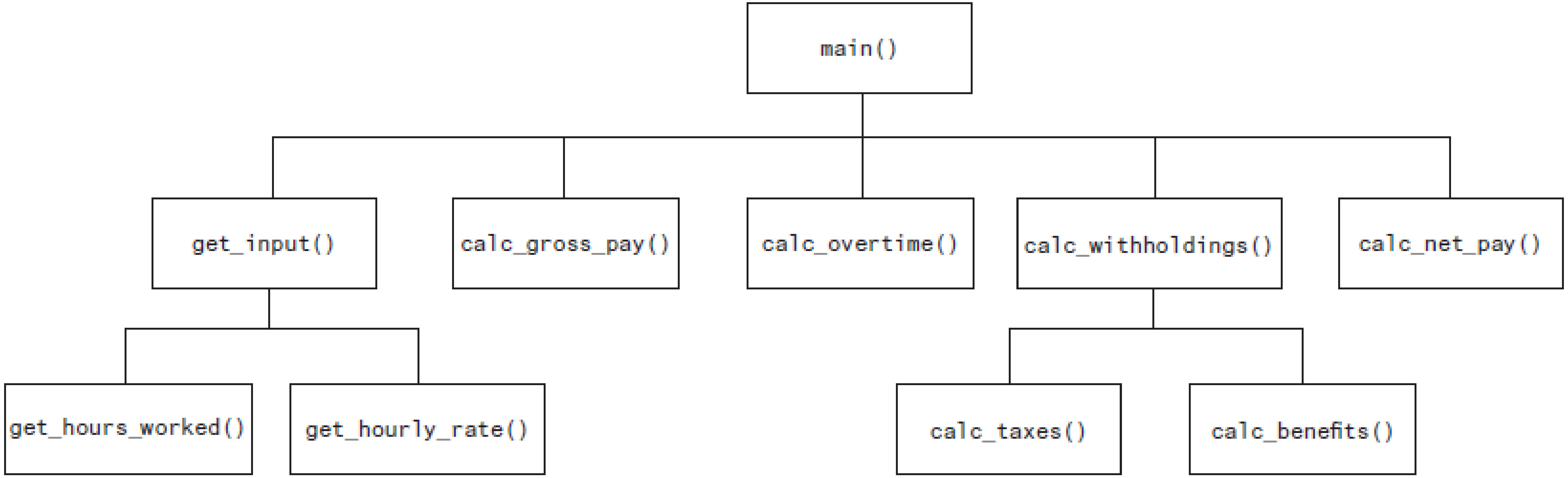
Step 1 : The overall task that the program is to perform is broken down into a series of subtasks.

Step 2 : Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.

Step 3 : Once all of the subtasks have been identified, they are written in code.

Hierarchy Charts

A hierarchy chart, which is also known as a structure chart, shows boxes that represent each function in a program



Module Roadmap

1. Introduction to Functions
2. Void Functions
3. Designing Programs Using Functions
4. Local and Global Variables
5. Value-Returning Functions
6. Storing Functions in Modules

Local Variables

A ***local variable*** is created inside a function and <u>cannot be accessed by statements that are outside the function.</u>

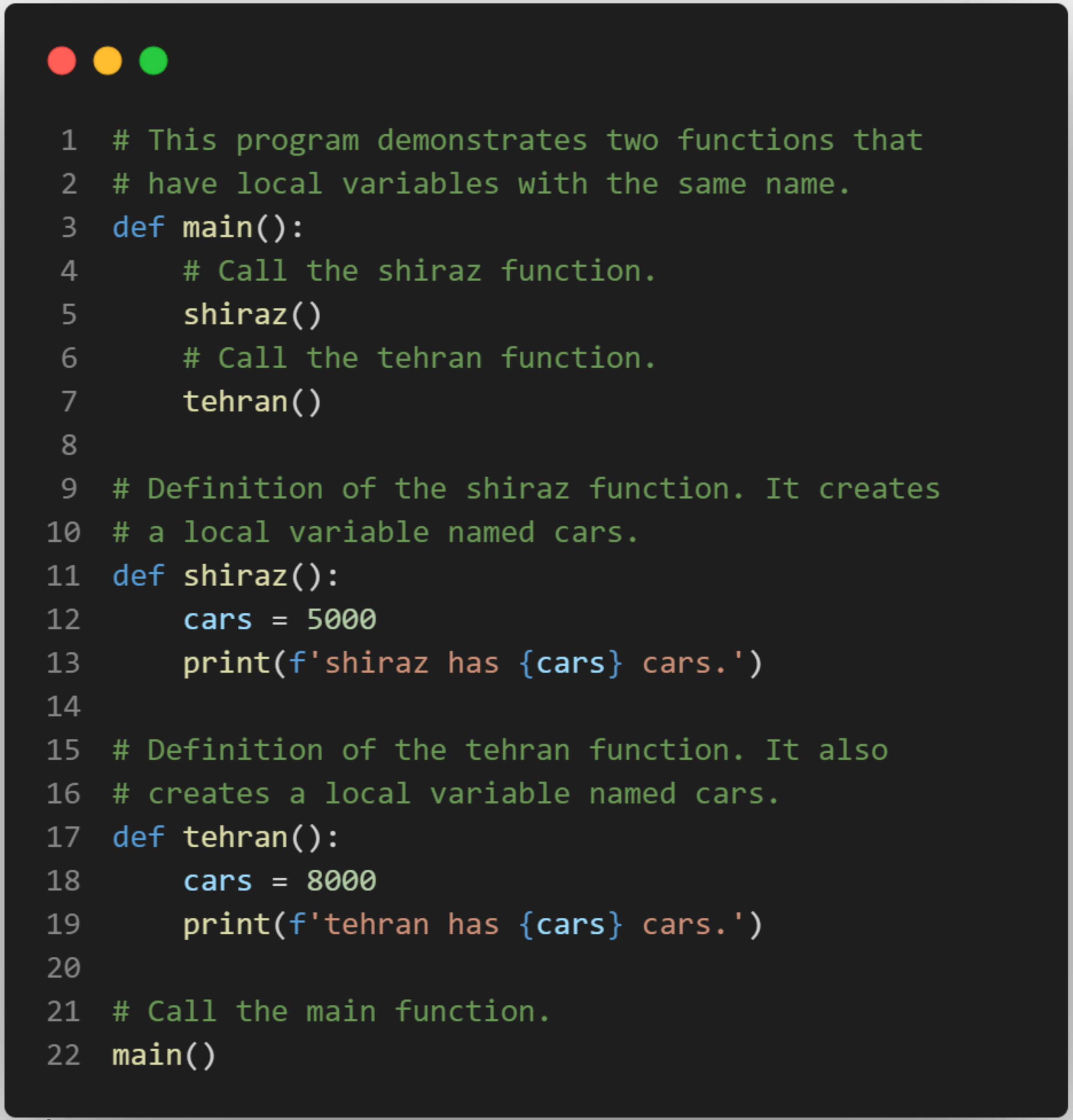
Different functions can have local variables with the same names because the functions cannot see each other's local variables.

Scope and Local Variables

A variable's ***scope*** is the part of a program in which the variable may be accessed. ***A variable is visible only to statements in the variable's scope***

A local variable's scope is the function in which the variable is created.

Scope and Local Variables



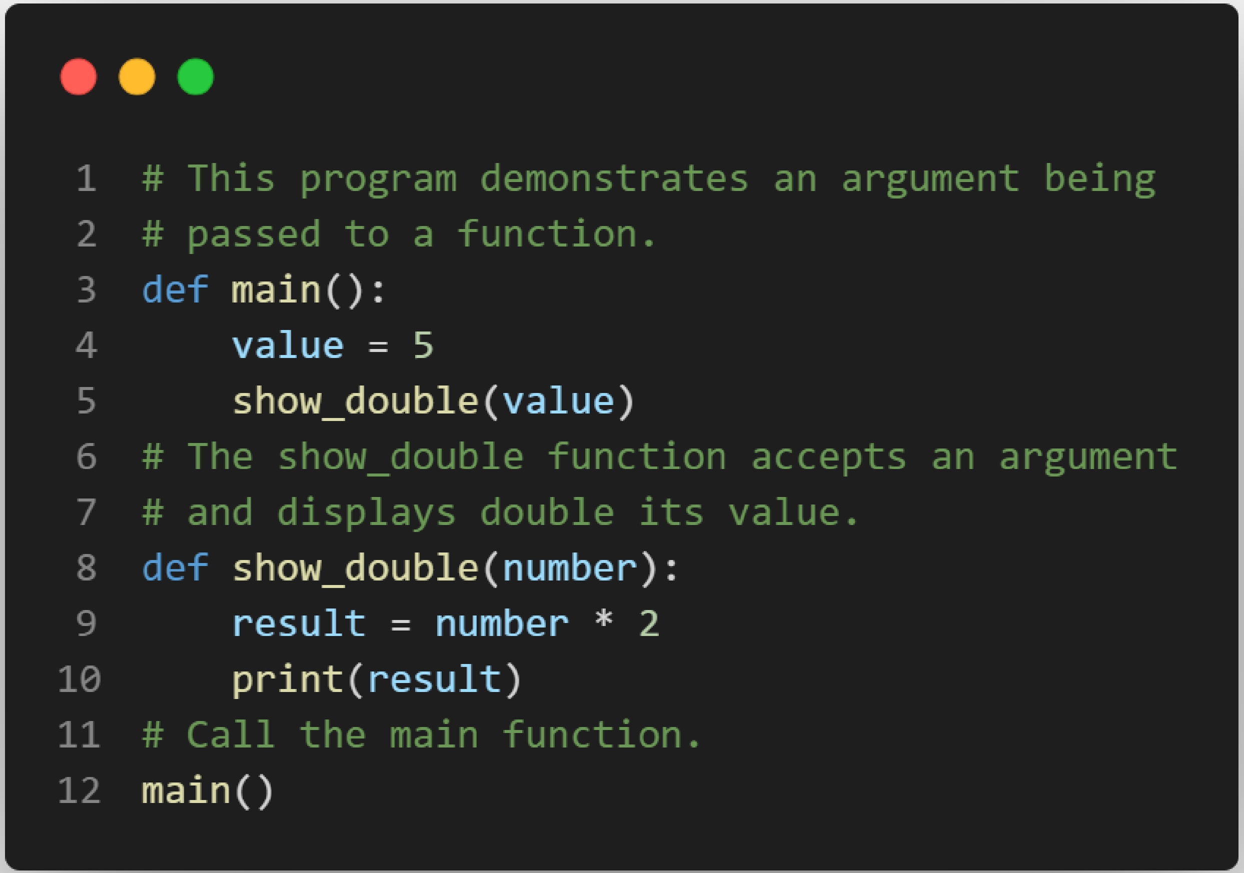
```python
# This program demonstrates two functions that
# have local variables with the same name.
def main():
    # Call the shiraz function.
    shiraz()
    # Call the tehran function.
    tehran()

# Definition of the shiraz function. It creates
# a local variable named cars.
def shiraz():
    cars = 5000
    print(f'shiraz has {cars} cars.')

# Definition of the tehran function. It also
# creates a local variable named cars.
def tehran():
    cars = 8000
    print(f'tehran has {cars} cars.')

# Call the main function.
main()
```

Passing Arguments to Functions

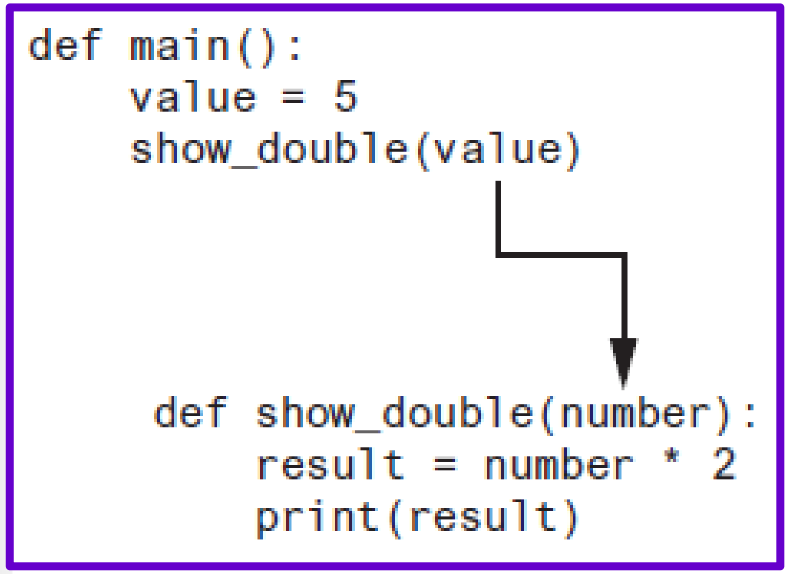
An ***argument*** is any piece of data that is passed into a function when the function is called.

A ***parameter*** is a variable that receives an argument that is passed into a function.

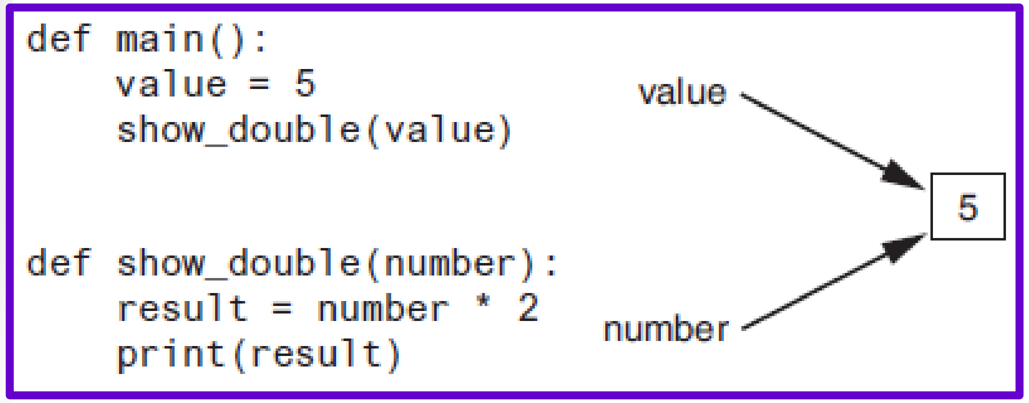
Passing Arguments to Functions



```python
# This program demonstrates an argument being
# passed to a function.
def main():
    value = 5
    show_double(value)
# The show_double function accepts an argument
# and displays double its value.
def show_double(number):
    result = number * 2
    print(result)
# Call the main function.
main()
```

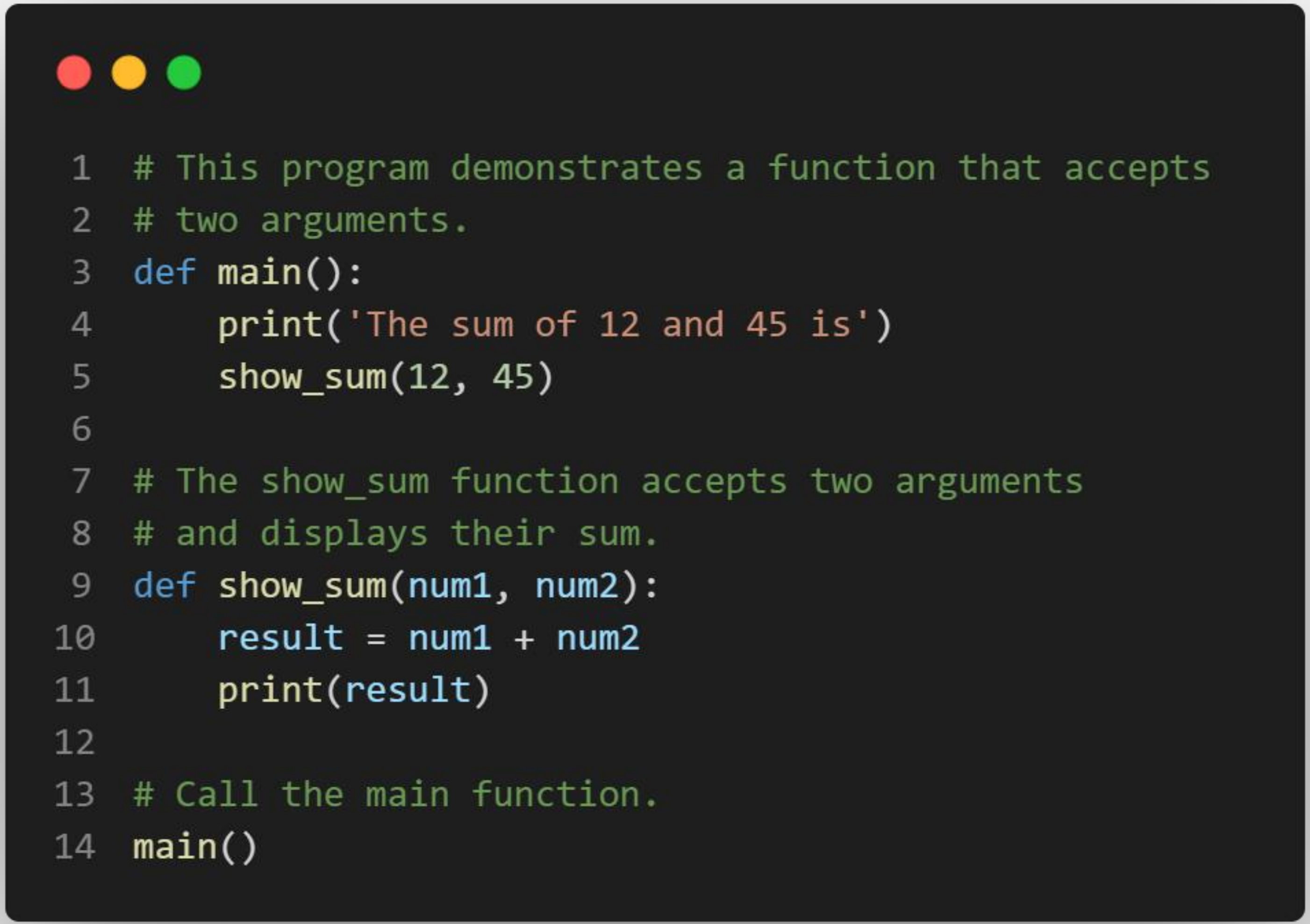


```python
def main():
    value = 5
    show_double(value)

    def show_double(number):
        result = number * 2
        print(result)
```

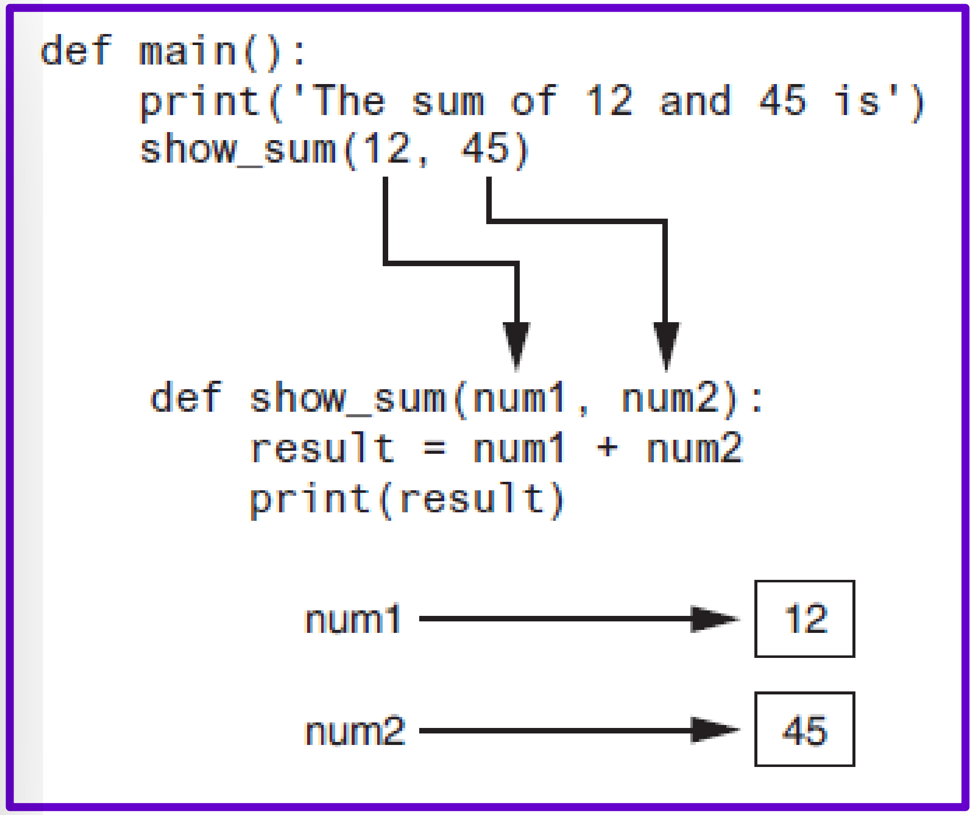


```python
def main():
    value = 5
    show_double(value)

def show_double(number):
    result = number * 2
    print(result)
```

Passing Multiple Arguments



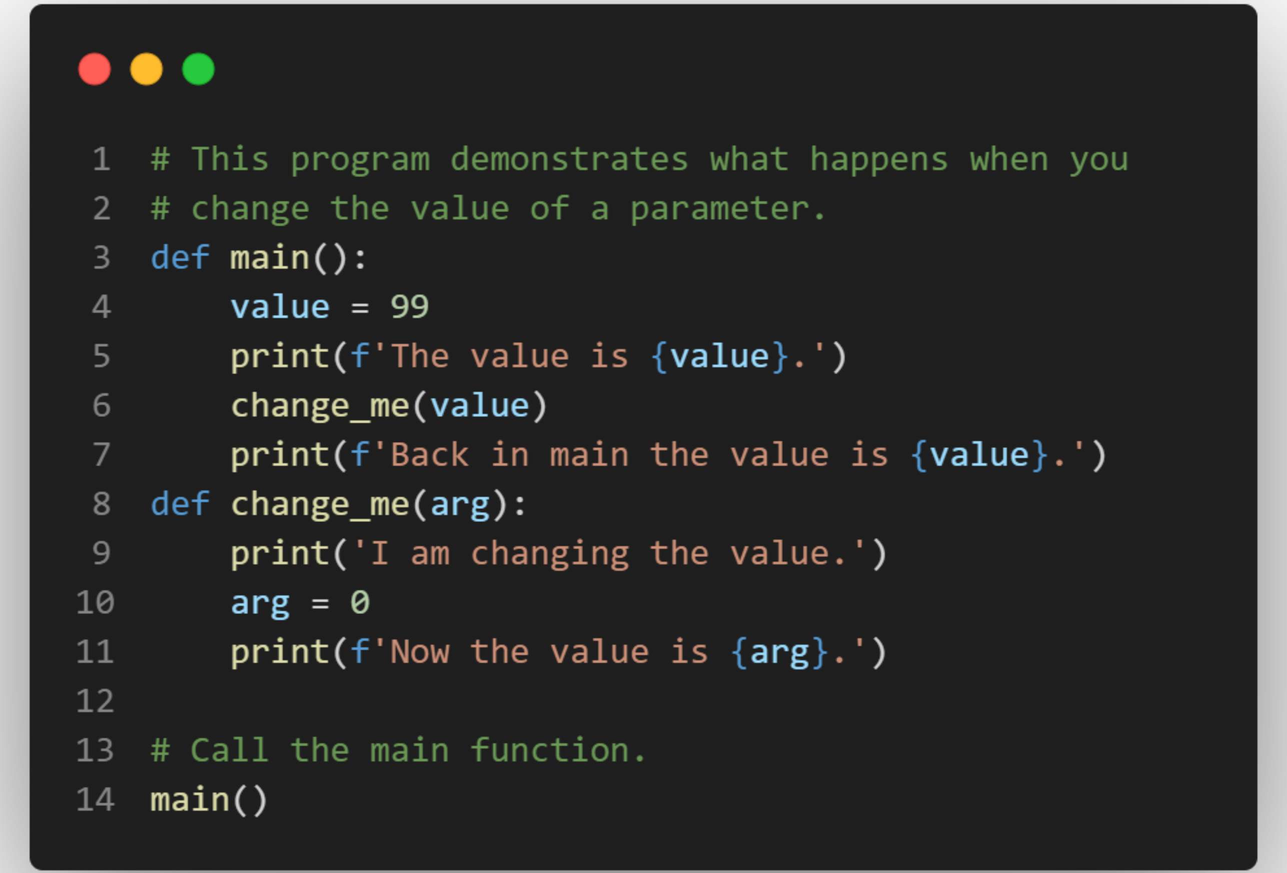
```python
# This program demonstrates a function that accepts
# two arguments.
def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)

# The show_sum function accepts two arguments
# and displays their sum.
def show_sum(num1, num2):
    result = num1 + num2
    print(result)

# Call the main function.
main()
```



```python
def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)

def show_sum(num1, num2):
    result = num1 + num2
    print(result)
```

Making Changes to Parameters

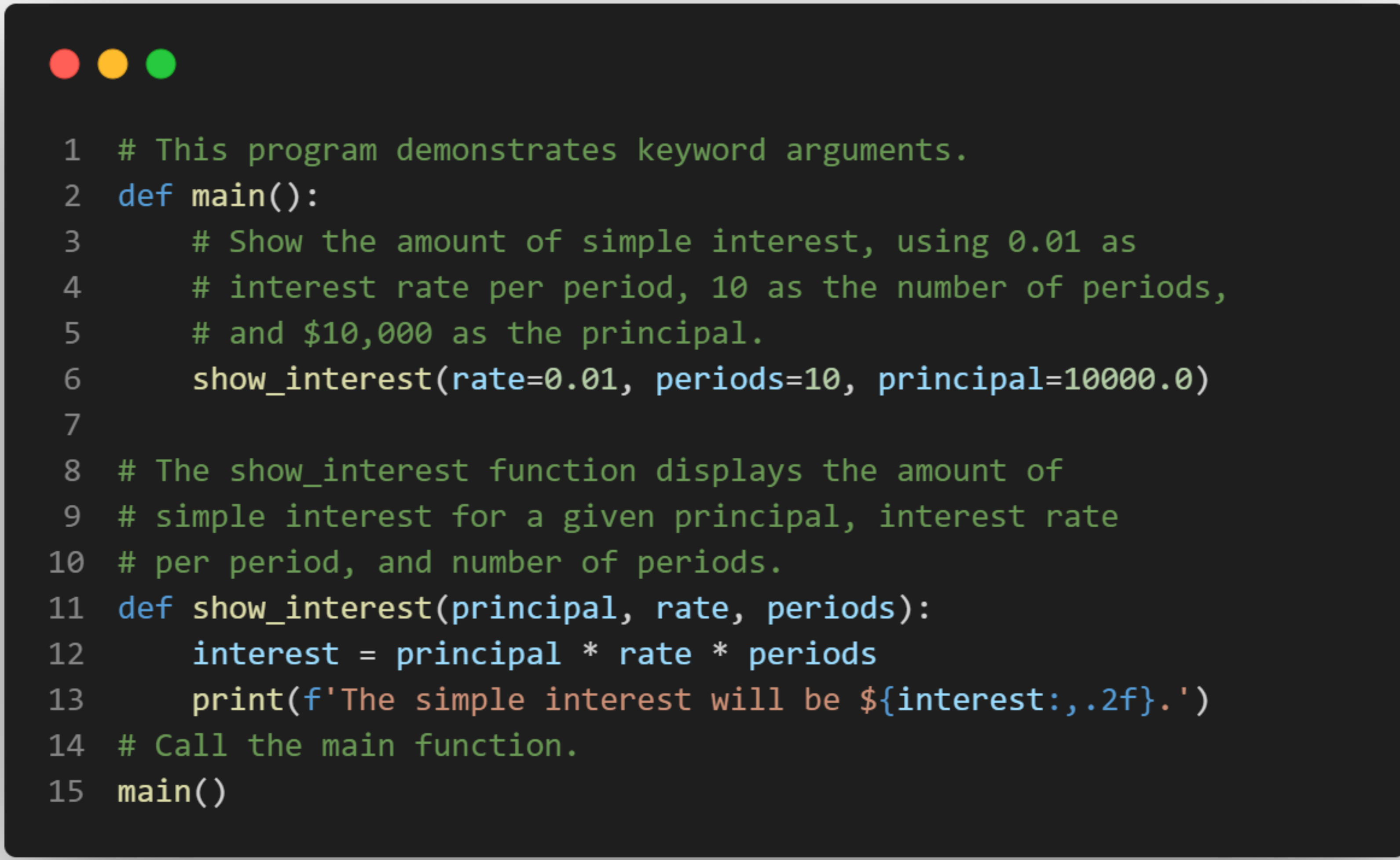
any changes that are made to the parameter variable <u>will not affect</u> the argument



```python
# This program demonstrates what happens when you
# change the value of a parameter.
def main():
    value = 99
    print(f'The value is {value}.')
    change_me(value)
    print(f'Back in main the value is {value}.')
def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print(f'Now the value is {arg}.')

# Call the main function.
main()
```

Keyword Arguments

In addition to this conventional form of argument passing, the Python language allows you to write an argument in the following format, to specify which parameter variable the argument should be passed to:
parameter_name=value



```python
# This program demonstrates keyword arguments.
def main():
    # Show the amount of simple interest, using 0.01 as
    # interest rate per period, 10 as the number of periods,
    # and $10,000 as the principal.
    show_interest(rate=0.01, periods=10, principal=10000.0)

# The show_interest function displays the amount of
# simple interest for a given principal, interest rate
# per period, and number of periods.
def show_interest(principal, rate, periods):
    interest = principal * rate * periods
    print(f'The simple interest will be ${interest:,.2f}.')
# Call the main function.
main()
```

Global Variables

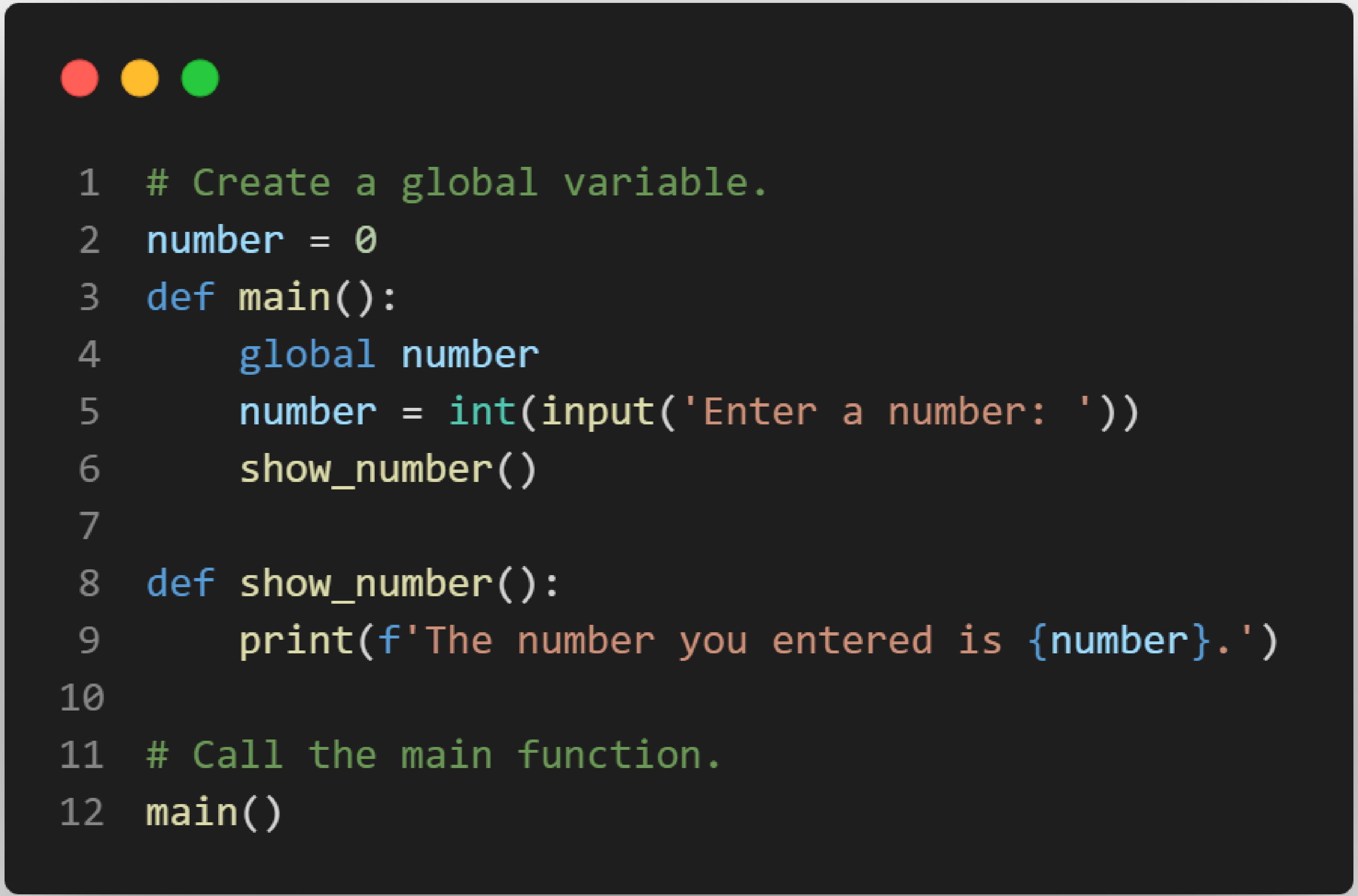
A global variable is accessible to all the functions in a program file.

When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is global

Most programmers agree that you should restrict the use of global variables, or not use them at all. The reasons are as follows:

- **Global variables make debugging difficult.**
- **Functions that use global variables are usually dependent on those variables.**
- **Global variables make a program hard to understand.**

Global Variables



```python
# Create a global variable.
number = 0
def main():
    global number
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    print(f'The number you entered is {number}.')

# Call the main function.
main()
```

Module Roadmap

Introduction to Value-Returning Functions: Generating Random Numbers

A ***value-returning function*** is a function that returns a value back to the part of the program that called it.

Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks.

Standard Library Functions and the import Statemen

Python, as well as most programming languages, comes with a ***standard library*** of functions that have already been written for you

Some of Python’s library functions are ***built into*** the Python interpreter. (like *print*, *input*, *range*)

Some of Python’s library are stored in files that are known as ***modules***.

Standard Library Functions and the import Statemen

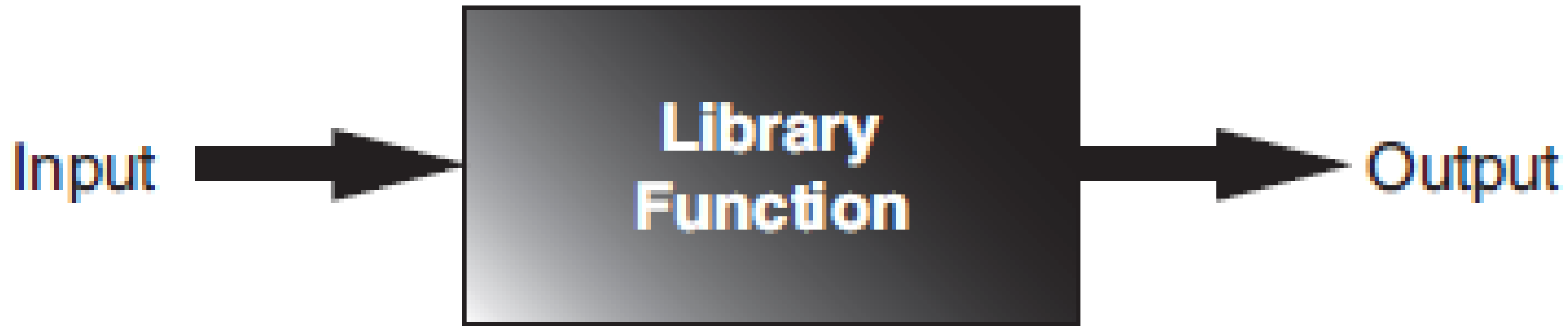
In order to call a function that is stored in a module, you have to write an ***import*** statement at the top of your program.

If you want to use any of the math module's functions in a program, you should write the following import statement at the top of the program:

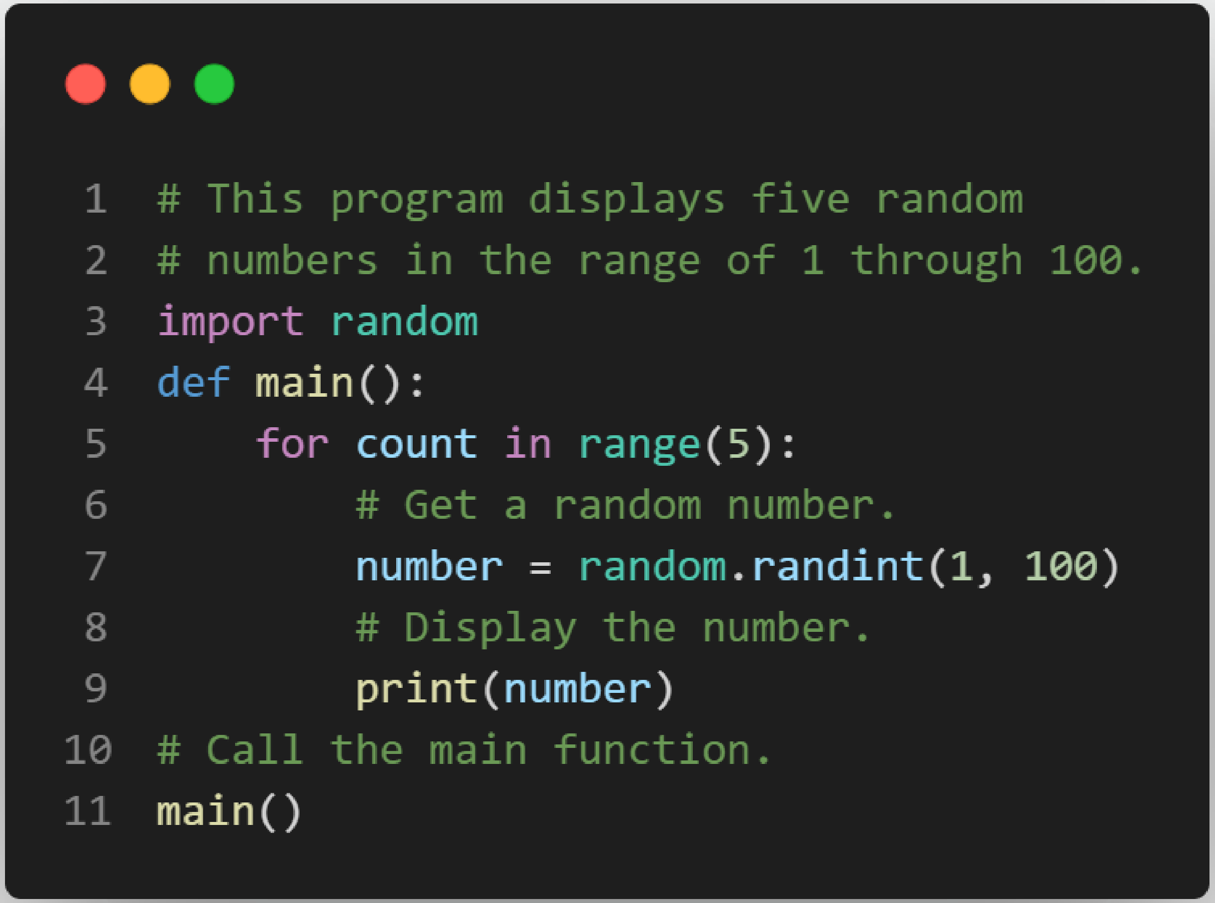
import math

A library function viewed as a black box

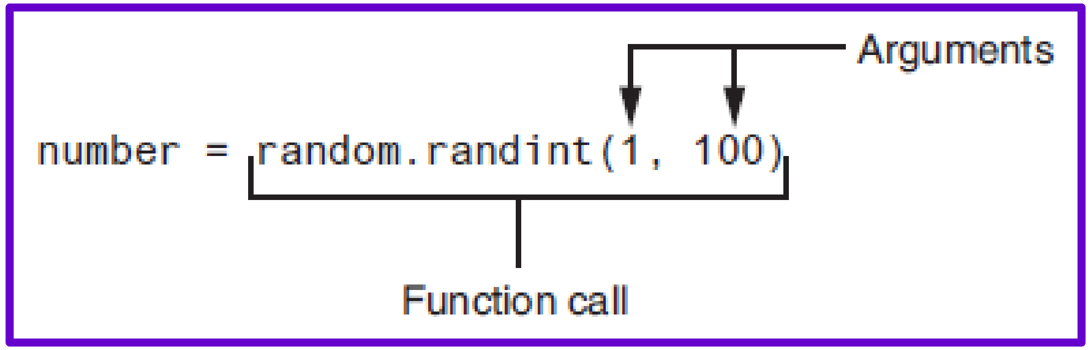
Because **you do not see the internal workings** of library functions, many programmers think of them as *black boxes*.

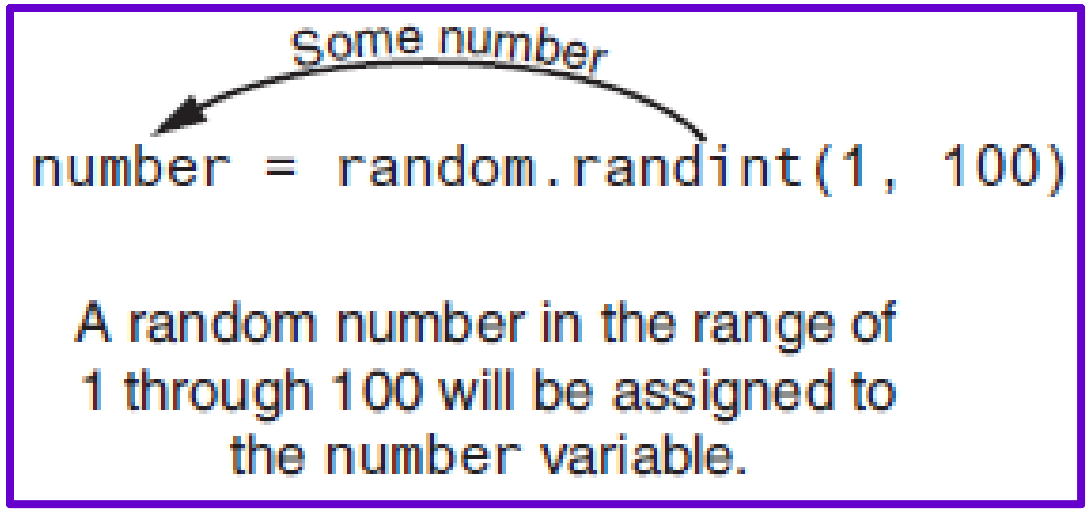


Generating Random Numbers



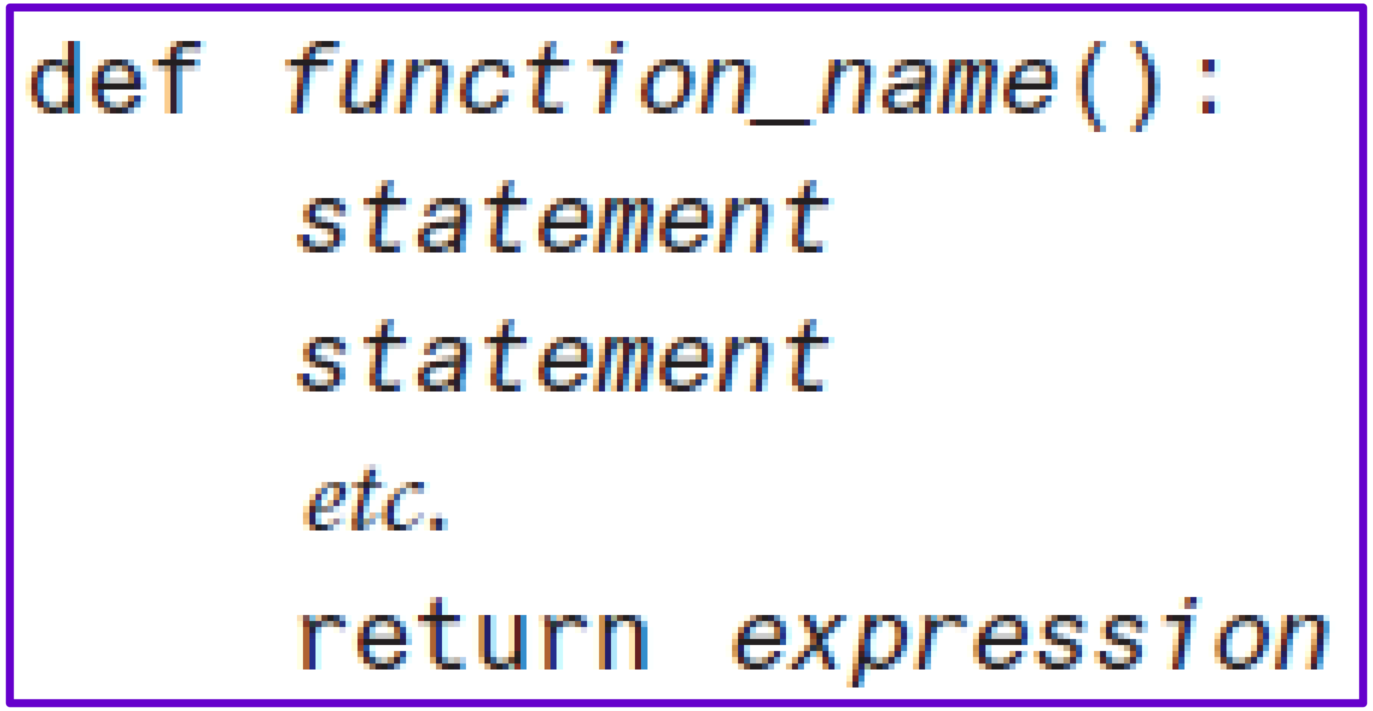
```python
# This program displays five random
# numbers in the range of 1 through 100.
import random
def main():
    for count in range(5):
        # Get a random number.
        number = random.randint(1, 100)
        # Display the number.
        print(number)
# Call the main function.
main()
```




Some number
number = random.randint(1, 100)
A random number in the range of 1 through 100 will be assigned to the number variable.

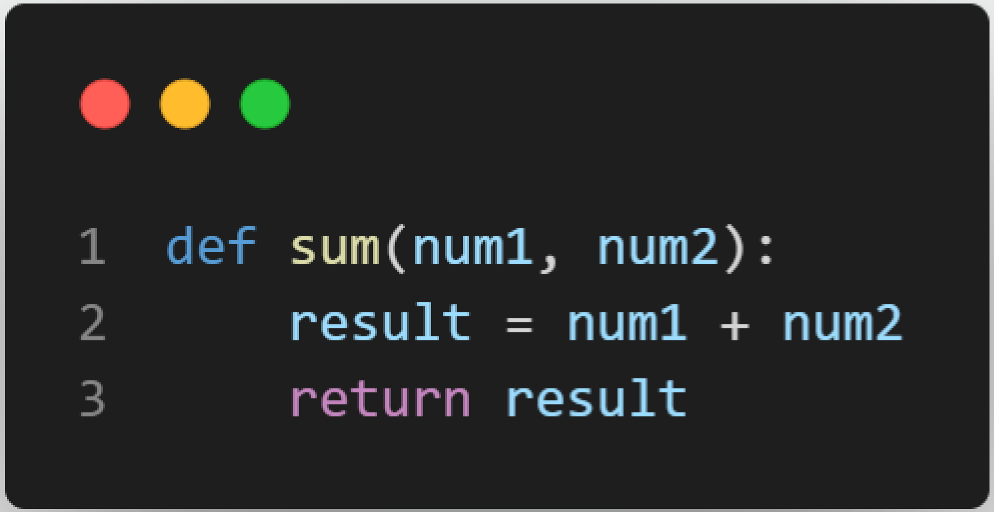
Writing Your Own Value-Returning Functions

A value-returning function has a return statement that returns a value back to the part of the program that called it.

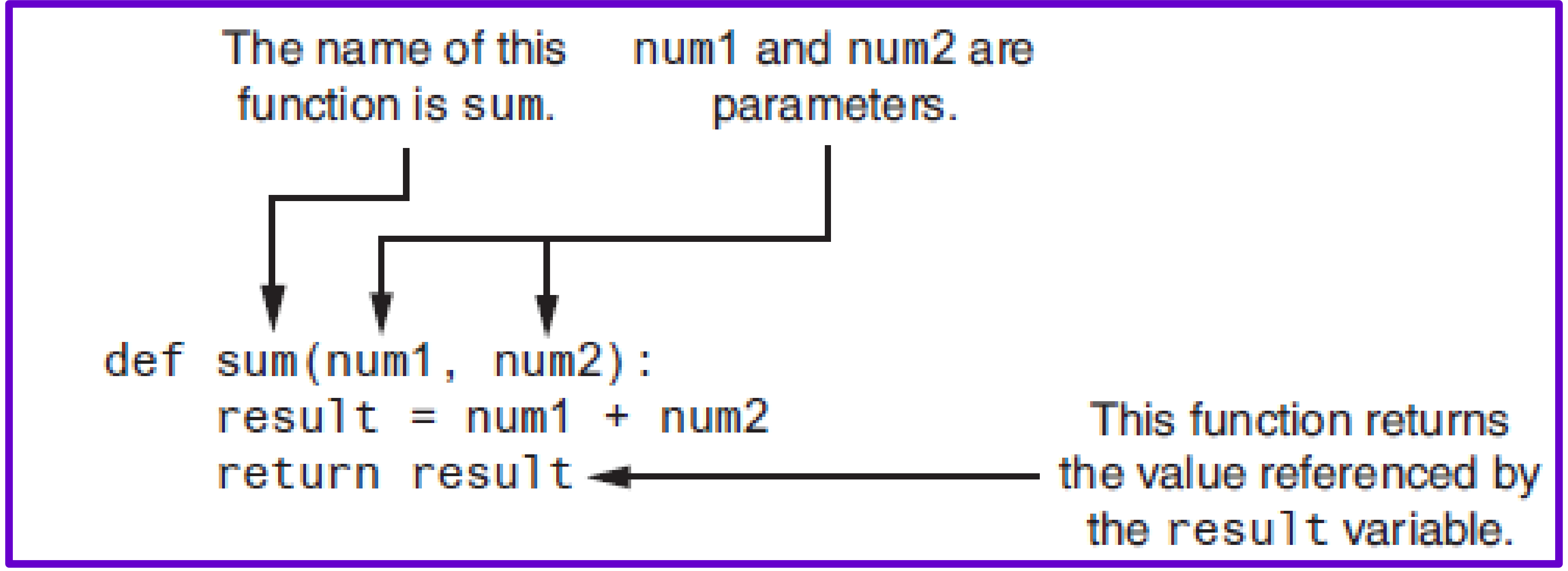


```
def function_name():
    statement
    statement
    etc.
    return expression
```

Writing Your Own Value-Returning Functions (Example)

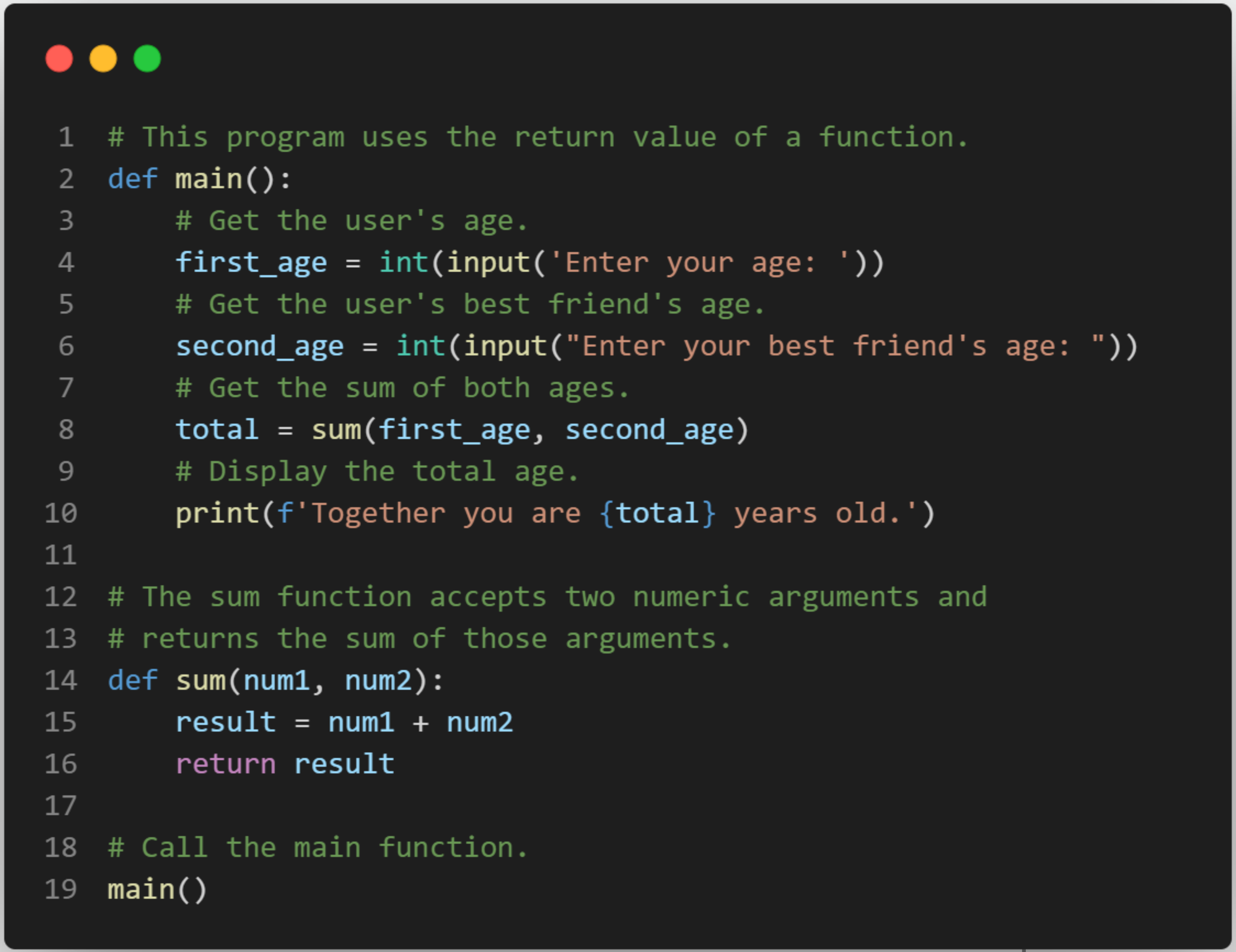


```python
def sum(num1, num2):
    result = num1 + num2
    return result
```

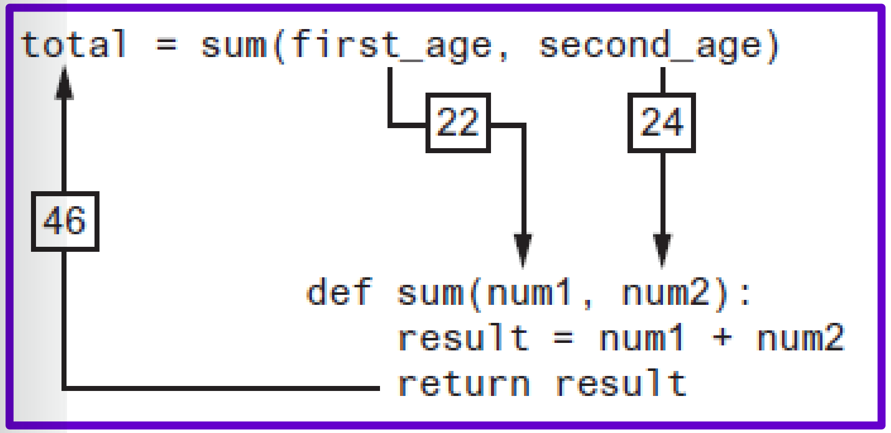

The name of this function is sum.
num1 and num2 are parameters.
This function returns the value referenced by the result variable.

```python
def sum(num1, num2):
    result = num1 + num2
    return result
```

Writing Your Own Value-Returning Functions (Example)

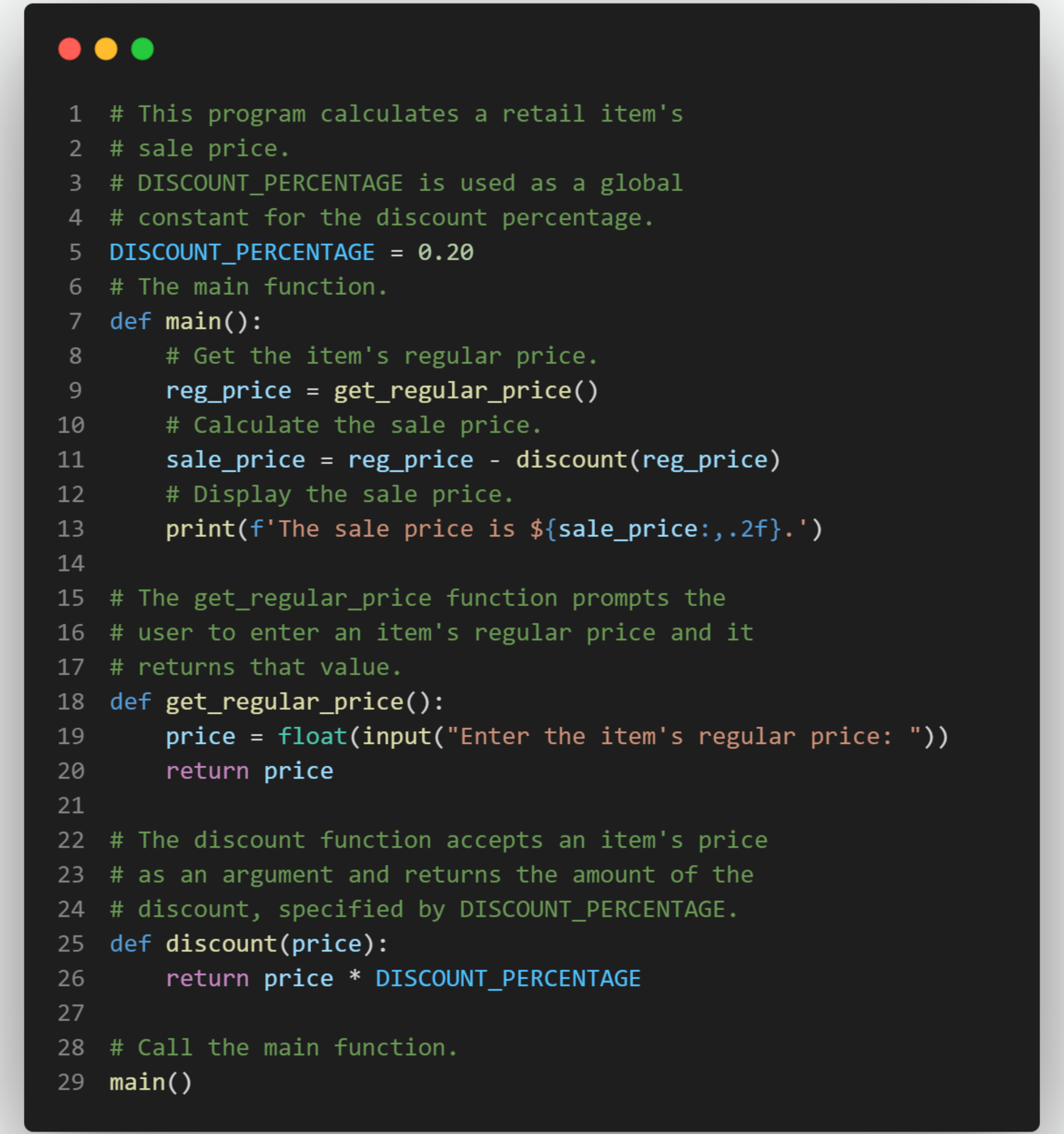


```python
# This program uses the return value of a function.
def main():
    # Get the user's age.
    first_age = int(input('Enter your age: '))
    # Get the user's best friend's age.
    second_age = int(input("Enter your best friend's age: "))
    # Get the sum of both ages.
    total = sum(first_age, second_age)
    # Display the total age.
    print(f'Together you are {total} years old.')

# The sum function accepts two numeric arguments and
# returns the sum of those arguments.
def sum(num1, num2):
    result = num1 + num2
    return result

# Call the main function.
main()
```



```
total = sum(first_age, second_age)

def sum(num1, num2):
    result = num1 + num2
    return result
```

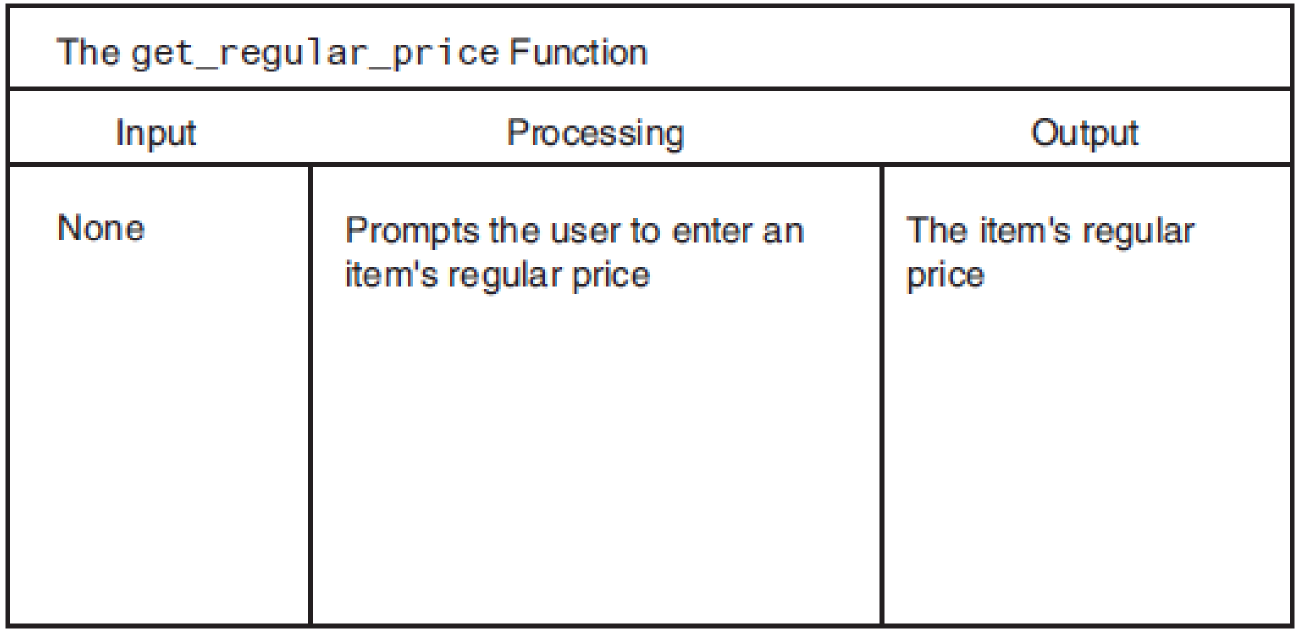
Writing Your Own Value-Returning Functions (Example)



```python
# This program calculates a retail item's
# sale price.
# DISCOUNT_PERCENTAGE is used as a global
# constant for the discount percentage.
DISCOUNT_PERCENTAGE = 0.20
# The main function.
def main():
    # Get the item's regular price.
    reg_price = get_regular_price()
    # Calculate the sale price.
    sale_price = reg_price - discount(reg_price)
    # Display the sale price.
    print(f'The sale price is ${sale_price:,.2f}.')

# The get_regular_price function prompts the
# user to enter an item's regular price and it
# returns that value.
def get_regular_price():
    price = float(input("Enter the item's regular price: "))
    return price

# The discount function accepts an item's price
# as an argument and returns the amount of the
# discount, specified by DISCOUNT_PERCENTAGE.
def discount(price):
    return price * DISCOUNT_PERCENTAGE

# Call the main function.
main()
```

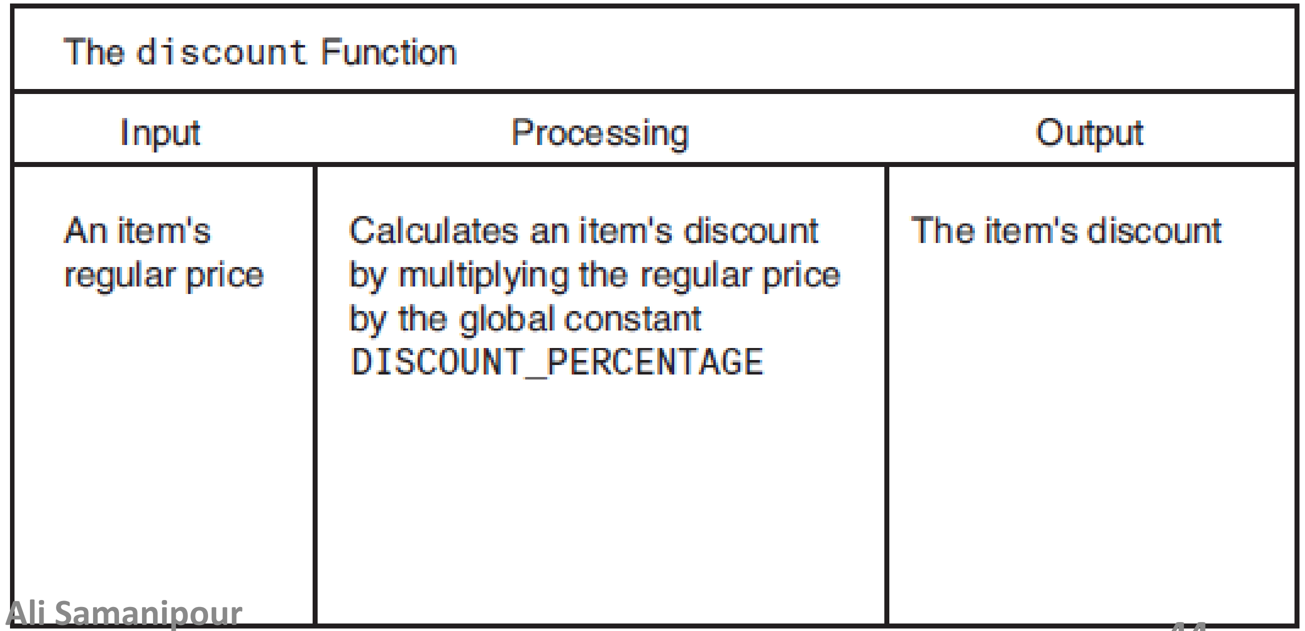
Using IPO Charts

An ***IPO*** chart is a simple but effective tool that programmers sometimes use for designing and documenting functions.



The `get_regular_price` Function

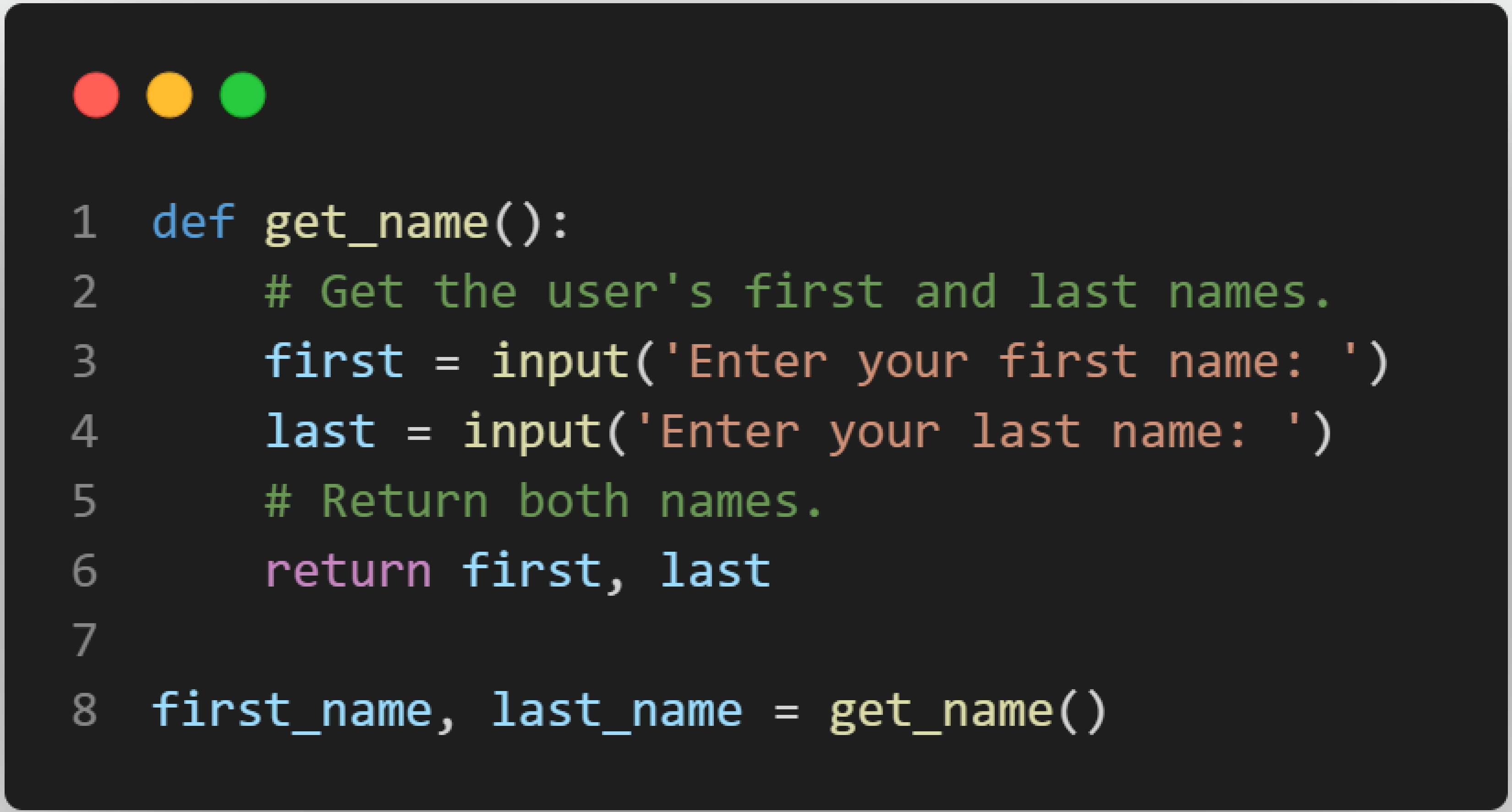
Input	Processing	Output
None	Prompts the user to enter an item's regular price	The item's regular price



The `discount` Function

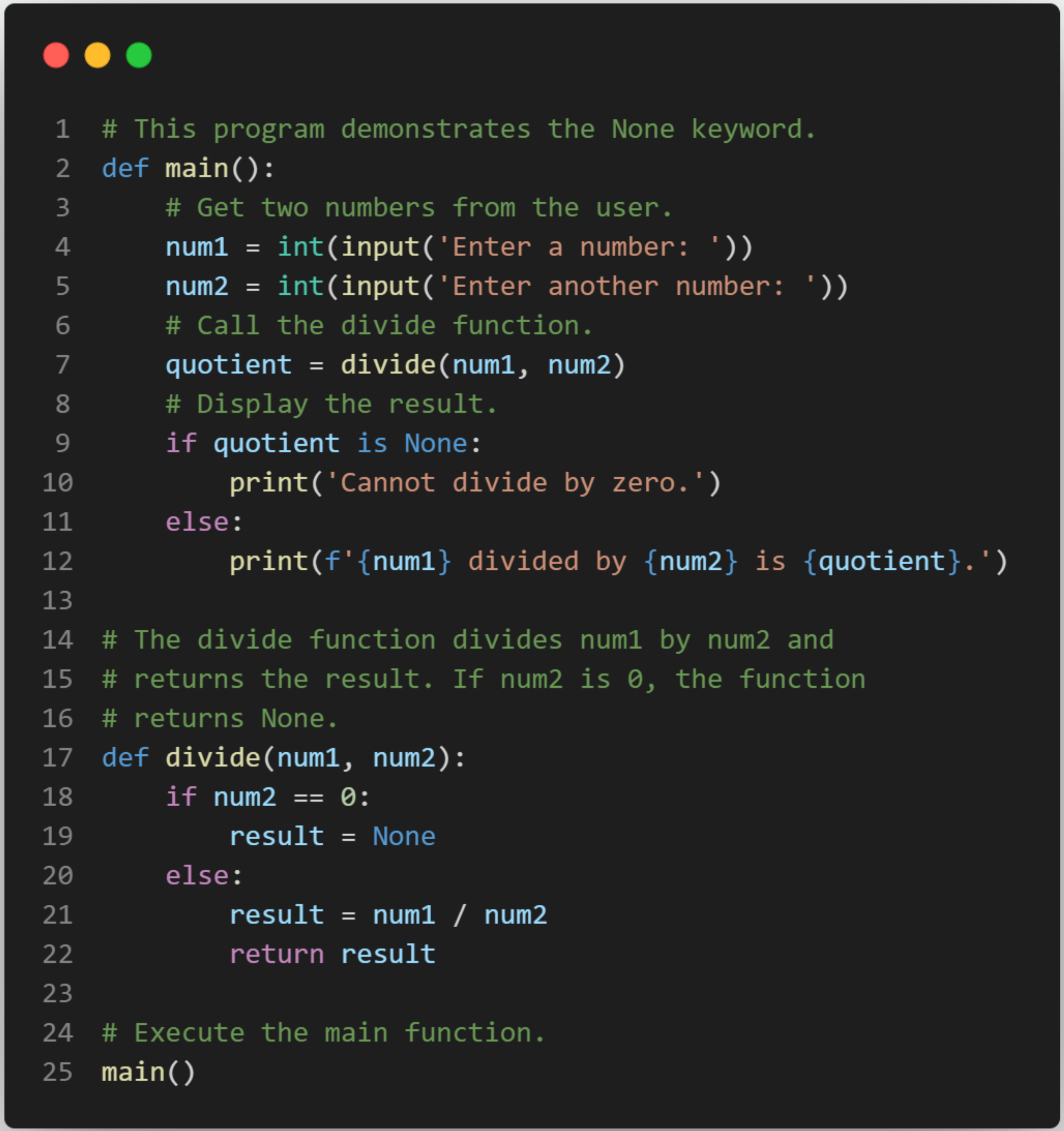
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant `DISCOUNT_PERCENTAGE`	The item's discount

Returning Multiple Values



```python
def get_name():
    # Get the user's first and last names.
    first = input('Enter your first name: ')
    last = input('Enter your last name: ')
    # Return both names.
    return first, last

first_name, last_name = get_name()
```

Returning None from a Function



```python
# This program demonstrates the None keyword.
def main():
    # Get two numbers from the user.
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    # Call the divide function.
    quotient = divide(num1, num2)
    # Display the result.
    if quotient is None:
        print('Cannot divide by zero.')
    else:
        print(f'{num1} divided by {num2} is {quotient}.')

# The divide function divides num1 by num2 and
# returns the result. If num2 is 0, the function
# returns None.
def divide(num1, num2):
    if num2 == 0:
        result = None
    else:
        result = num1 / num2
        return result

# Execute the main function.
main()
```

Module Roadmap

1. Introduction to Functions
2. Void Functions
3. Designing Programs Using Functions
4. Local and Global Variables
5. Value-Returning Functions
6. **Storing Functions in Modules**

Storing Functions in Modules

A ***module*** is a file that contains Python code. Large programs are easier to debug and ***maintain*** when they are divided into modules.

When you break a program into modules, **each module should contain functions that perform related tasks**

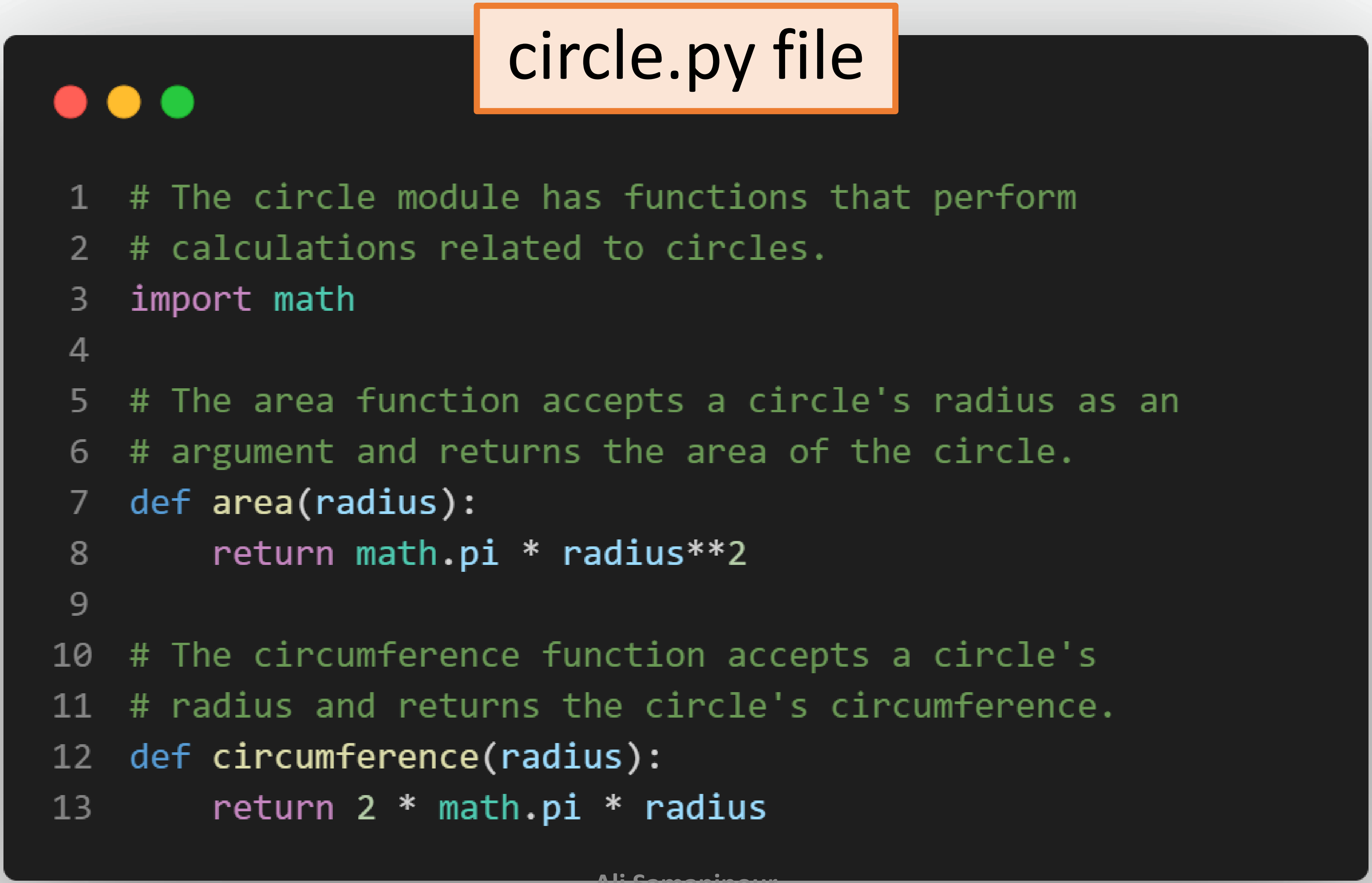
This approach, which is called modularization, makes the program easier to understand, test, and maintain

Storing Functions in Modules (Example)

Suppose you've been asked you to write a program that calculates the following:

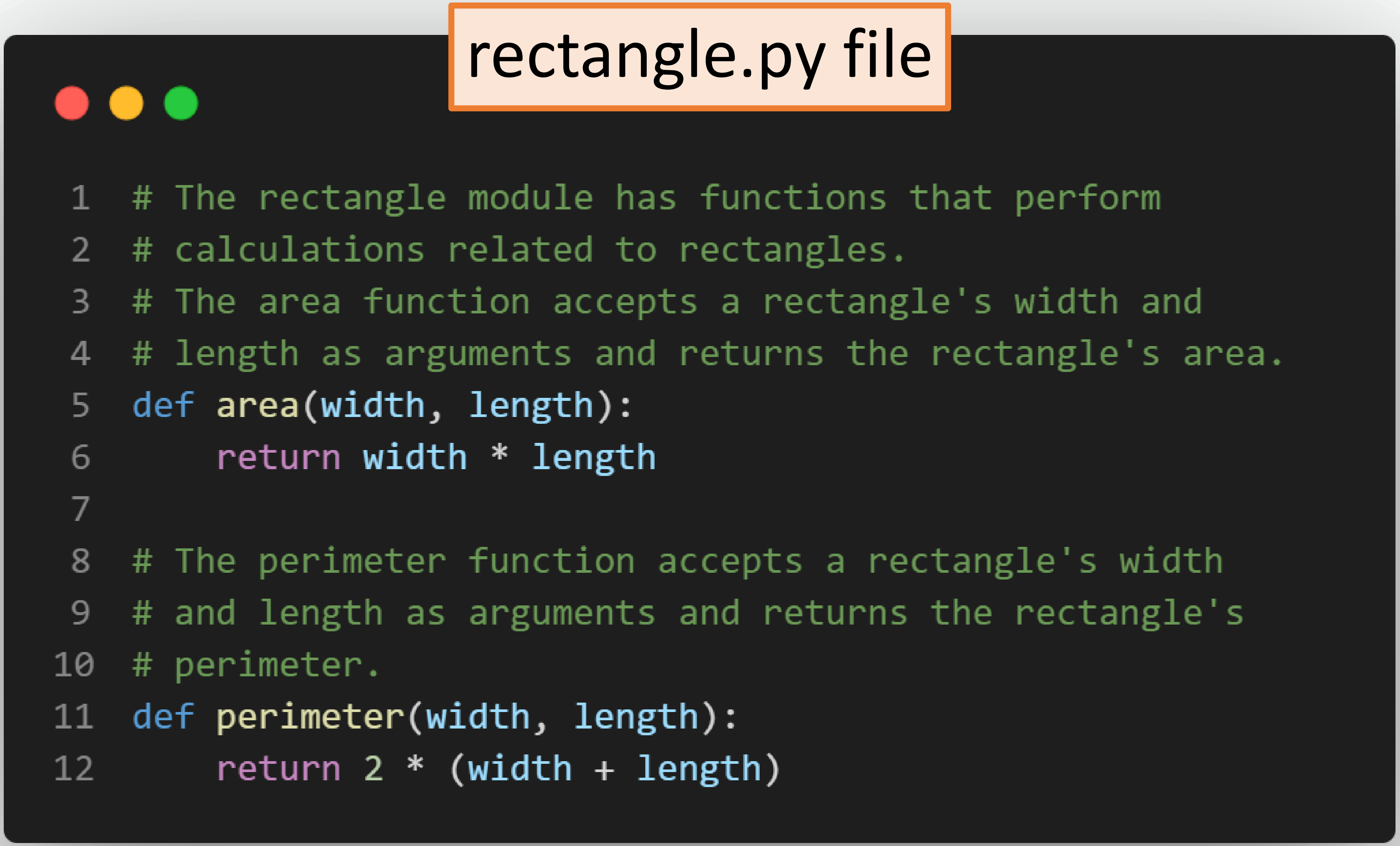
- The area of a circle
- The circumference of a circle
- The area of a rectangle
- The perimeter of a rectangle

Storing Functions in Modules (Example ...)


circle.py file

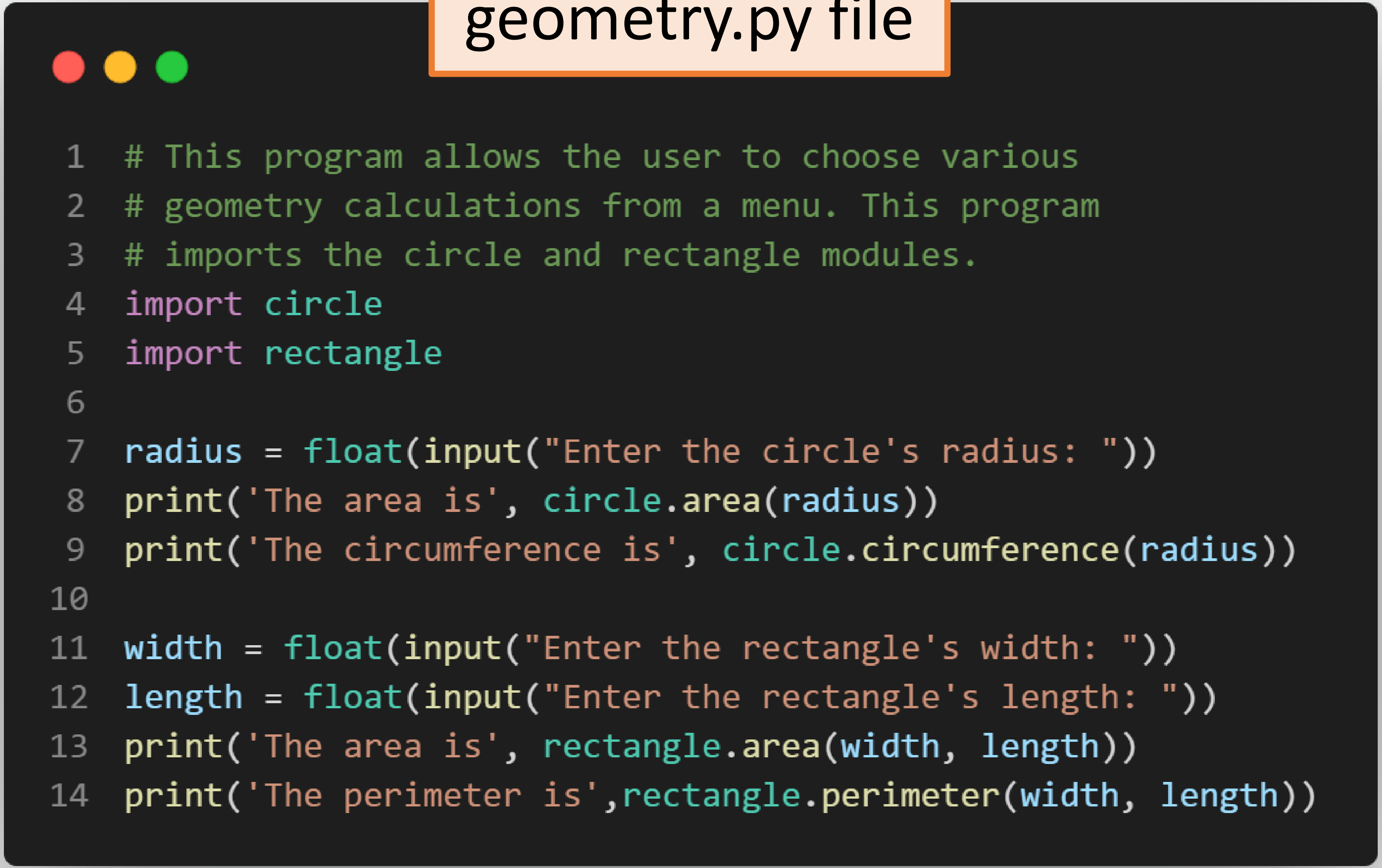
```python
# The circle module has functions that perform
# calculations related to circles.
import math

# The area function accepts a circle's radius as an
# argument and returns the area of the circle.
def area(radius):
    return math.pi * radius**2

# The circumference function accepts a circle's
# radius and returns the circle's circumference.
def circumference(radius):
    return 2 * math.pi * radius
```

Storing Functions in Modules (Example ...)


rectangle.py file

```python
# The rectangle module has functions that perform
# calculations related to rectangles.
# The area function accepts a rectangle's width and
# length as arguments and returns the rectangle's area.
def area(width, length):
    return width * length

# The perimeter function accepts a rectangle's width
# and length as arguments and returns the rectangle's
# perimeter.
def perimeter(width, length):
    return 2 * (width + length)
```

Storing Functions in Modules (Example …)



geometry.py file

```python
# This program allows the user to choose various
# geometry calculations from a menu. This program
# imports the circle and rectangle modules.
import circle
import rectangle

radius = float(input("Enter the circle's radius: "))
print('The area is', circle.area(radius))
print('The circumference is', circle.circumference(radius))

width = float(input("Enter the rectangle's width: "))
length = float(input("Enter the rectangle's length: "))
print('The area is', rectangle.area(width, length))
print('The perimeter is',rectangle.perimeter(width, length))
```

Conditionally Executing the main Function in a Module

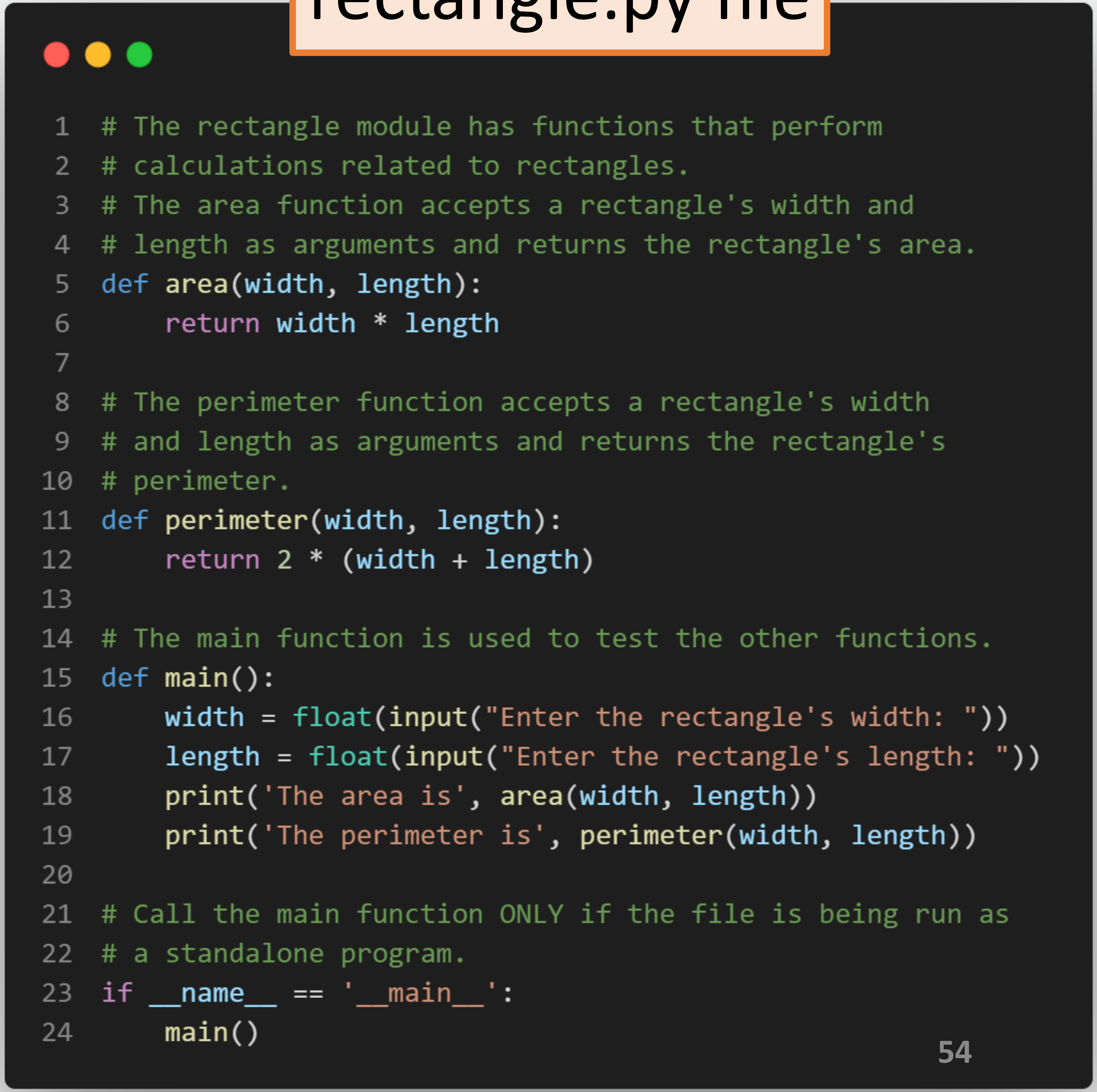
When a module is imported, the Python interpreter executes the statements in the module just as if the module were a standalone program.

When the Python interpreter is processing a source code file, it creates a special variable named ***__name__.***

If the file is being imported as a module, the __name__ variable will be set to the name of the module. Otherwise, if the file is being executed as a standalone program, the __name__ variable will be set to the string ***'__main__'***.

Conditionally Executing the main Function in a Module (Example ...)

rectangle.py file



```python
# The rectangle module has functions that perform
# calculations related to rectangles.
# The area function accepts a rectangle's width and
# length as arguments and returns the rectangle's area.
def area(width, length):
    return width * length

# The perimeter function accepts a rectangle's width
# and length as arguments and returns the rectangle's
# perimeter.
def perimeter(width, length):
    return 2 * (width + length)

# The main function is used to test the other functions.
def main():
    width = float(input("Enter the rectangle's width: "))
    length = float(input("Enter the rectangle's length: "))
    print('The area is', area(width, length))
    print('The perimeter is', perimeter(width, length))

# Call the main function ONLY if the file is being run as
# a standalone program.
if __name__ == '__main__':
    main()
```

Ali Samanipour
linkedin.com/in/Samanipour

Appendix : Pausing Execution Until the User Presses Enter

Sometimes you want a program to pause so the user can read information that has been displayed on the screen. **When the user is ready for the program to continue execution, he or she presses the Enter key and the program resumes**.

In Python, you can use the ***input*** function to cause a program to pause until the user presses the Enter key

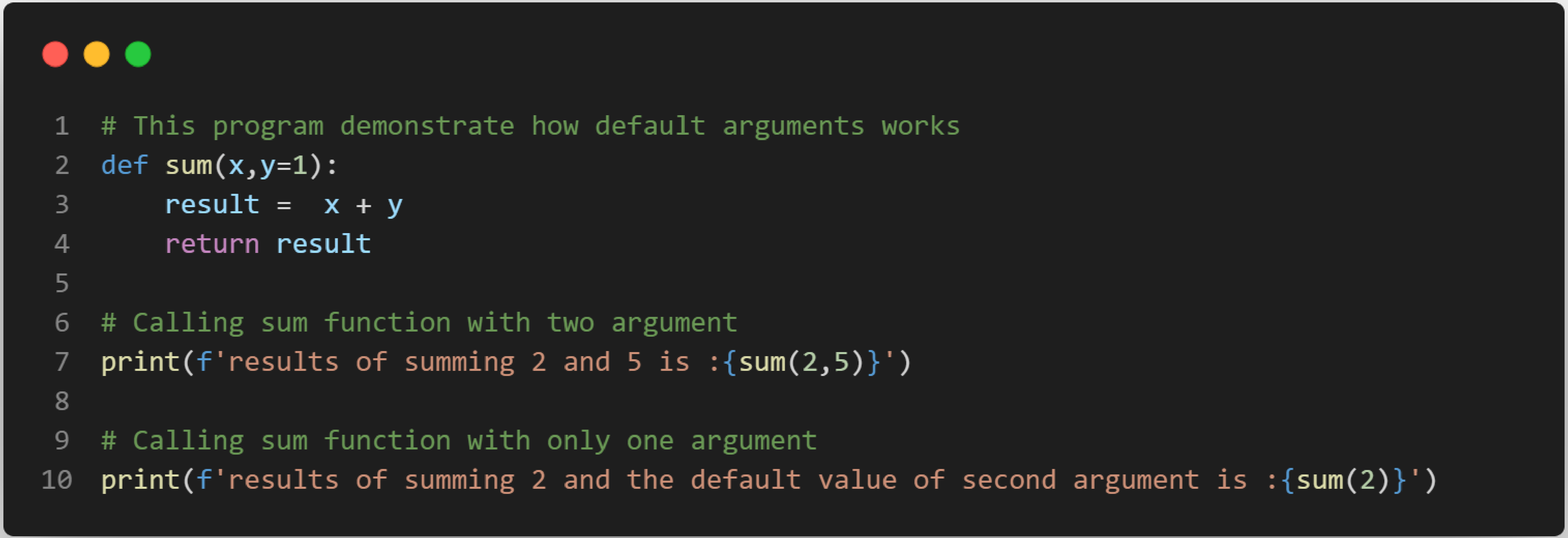
Appendix : Using the pass Keyword

Sometimes when you are initially writing a program’s code, **you know the names of the functions you plan to use, but you might not know all the details of the code that will be in those functions**

When this is the case, you can use the ***pass*** keyword to create empty functions.

The ***pass*** keyword can be used as a placeholder anywhere in your Python code.

Appendix : Default Function Arguments



```python
# This program demonstrate how default arguments works
def sum(x,y=1):
    result =  x + y
    return result

# Calling sum function with two argument
print(f'results of summing 2 and 5 is :{sum(2,5)}')

# Calling sum function with only one argument
print(f'results of summing 2 and the default value of second argument is :{sum(2)}')
```