FACULTY OF COMPUTING
& INFORMATION TECHNOLOGY
KING ABDULAZIZ UNIVERSITY

FCIT
KAU

كـلـيـة الـحـاسـبـات
وتـقـنـيـة الـمـعـلـومـات
جامعة الملك عبدالعزيز

Chapter 6
Functions

CPIT 110 (Problem-Solving and Programming)

Introduction to Programming Using Python, By: Y. Daniel Liang

Version 2.0

Sections

- 6.1. Motivations ▶
- 6.2. Defining a Function ▶
- 6.3. Calling a Function ▶
- 6.4. Functions with/without Return Values ▶
- 6.5. Positional and Keyword Arguments ▶
- 6.6. Passing Arguments by Reference Values ▶
- 6.7. Modularizing Code ▶
- 6.9. The Scope of Variables ▶
- 6.10. Default Arguments ▶
- 6.11. Returning Multiple Values ▶

Python Online IDE

Programs

- Program 1: Sum Many Numbers
- Program 2: Testing max Function
- Program 3: Testing Void Function
- Program 4: Testing getGrade Function
- Program 5: Finding the GCD (Modularizing Code)
- Program 6: Prime Number (Modularizing Code)

Check Points

- Section 6.4
  - #1
  - #2
  - #3
  - #4
  - #5
  - #6
  - #7
  - #8
  - #9
  - #10
- Section 6.5
  - #11
- Section 6.6
  - #12
  - #13
- Section 6.9
  - #14
  - #15
  - #16
- Section 6.10
  - #17
  - #18
  - #19
- Section 6.11
  - #20

Objectives

- To define functions with formal parameters (6.2).
- To invoke functions with actual parameters (i.e., arguments) (6.3).
- To distinguish between functions that return and do not return a value (6.4).
- To invoke a function using positional arguments or keyword arguments (6.5).
- To pass arguments by passing their reference values (6.6).
- To develop reusable code that is modular and is easy to read, debug, and maintain (6.7).
- To determine the scope of variables (6.9).
- To define functions with default arguments (6.10).
- To define a function that returns multiple values (6.11).

6.1. Motivations

- Program 1: Sum Many Numbers
- Functions

Sum Many Numbers
Program 1

Write a program that will sum three sets of numbers and then display the sum of each:

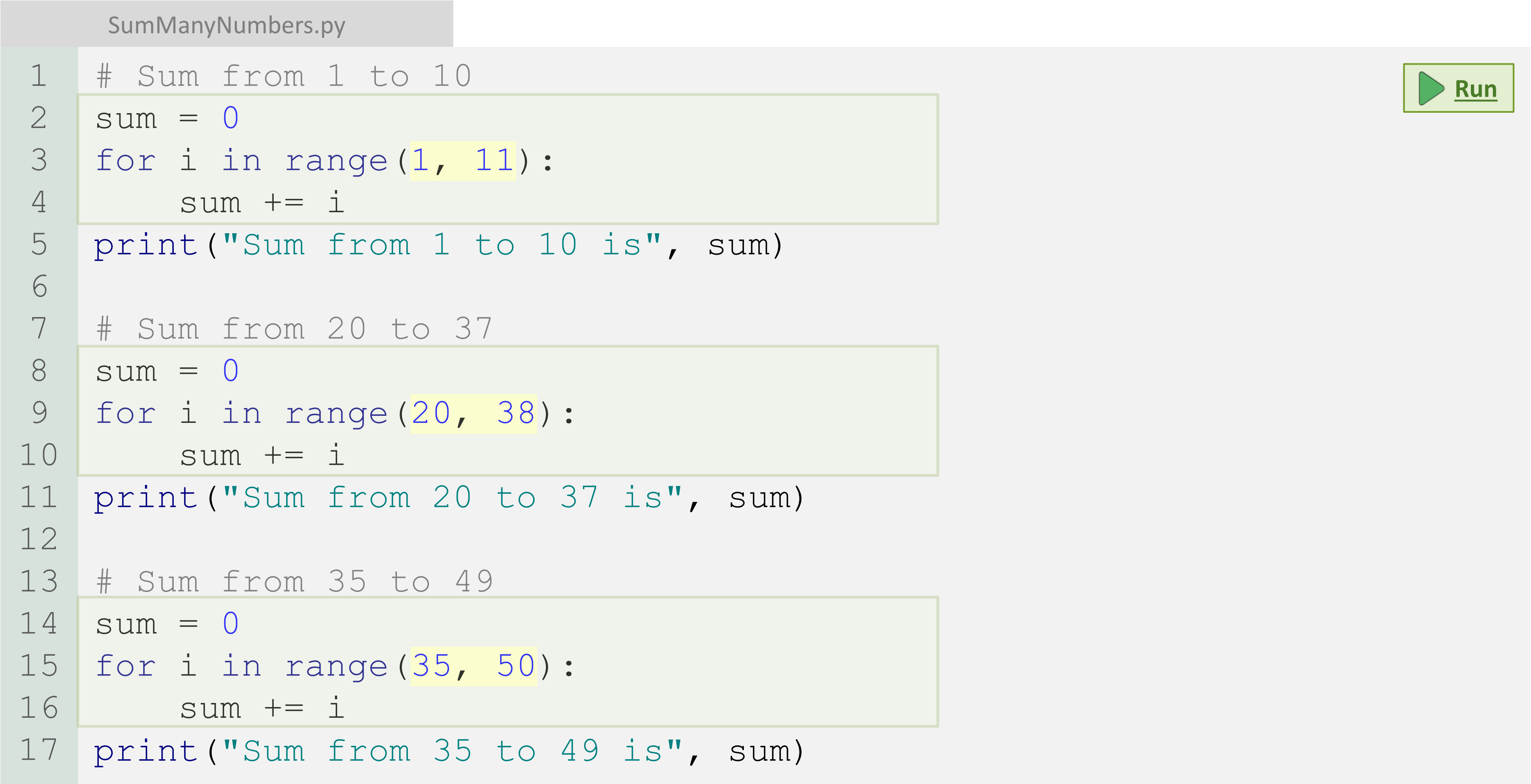
- Sum of integers from 1 to 10.
- Sum of integers from 20 to 37.
- Sum of integers from 35 to 49.

```
Sum from 1 to 10 is 55
Sum from 20 to 37 is 513
Sum from 35 to 49 is 630
```

Sum Many Numbers
Phase 1: Problem-solving

- This program is really easy.
- Algorithm:
  - For each set of numbers:
    - Make a variable sum.
    - Make a for loop and sum from the first number to the second number.
    - Print the final sum.
- So this is very easy to do.
- Unfortunately, we have to do it three times because we have three sets of numbers.

Sum Many Numbers
Phase 2: Implementation



```python
# Sum from 1 to 10
sum = 0
for i in range(1, 11):
    sum += i
print("Sum from 1 to 10 is", sum)

# Sum from 20 to 37
sum = 0
for i in range(20, 38):
    sum += i
print("Sum from 20 to 37 is", sum)

# Sum from 35 to 49
sum = 0
for i in range(35, 50):
    sum += i
print("Sum from 35 to 49 is", sum)
```

Sum Many Numbers
Observation

- Each sum is doing something very similar.
- In fact, each sum is essentially doing the same thing.
- The only difference is the range of numbers.
  - The starting and ending numbers of the sum.
- So why do we *repeat* our code three times?
- Wouldn't it be nice if we could write "common" code and then reuse it when needed?
  - That would be PERFECT!
- This is the idea of functions!

Sum Many Numbers
Phase 2: Implementation (Improved)

The first implementation can be simplified by using functions, as follows:

SumManyNumbersUsingFucntions.py

Run

```python
def sum(i1, i2):
    result = 0
    for i in range(i1, i2 + 1):
        result += i
    return result

def main():
    print("Sum from 1 to 10 is", sum(1, 10))
    print("Sum from 20 to 37 is", sum(20, 37))
    print("Sum from 35 to 49 is", sum(35, 49))

main() # Call the main function
```

```
Sum from 1 to 10 is 55
Sum from 20 to 37 is 513
Sum from 35 to 49 is 630
```

Sum Many Numbers
Discussion

- Lines 1–6 define the function named sum with the two parameters i1 and i2.
- Lines 8–11 define the main function that invokes:
  - sum(1, 10) to compute the sum from 1 to 10.
  - sum(20, 37) to compute the sum from 20 to 37.
  - sum(35, 49) to compute the sum from 35 to 49.
- Lines 12 calls the main function to execute the program.

Functions

- What is a function?
  - A function is a collection of statements grouped together to perform an operation.
- Guess what?
  - You have already used many predefined functions!
  - Examples:
    - print("message")
    - eval("numericString")
    - random.randint(a, b)
- These functions are defined in the Python library.
- In this chapter, you will learn how to create your own functions!

6.2. Defining a Function

- Anatomy of a Function
- Remember: Naming Conventions

Defining a Function

- A function definition consists of:
  - Function name
  - Parameters
  - Body
- Syntax:

```
def functionName(list of parameters)
    # Function body
```

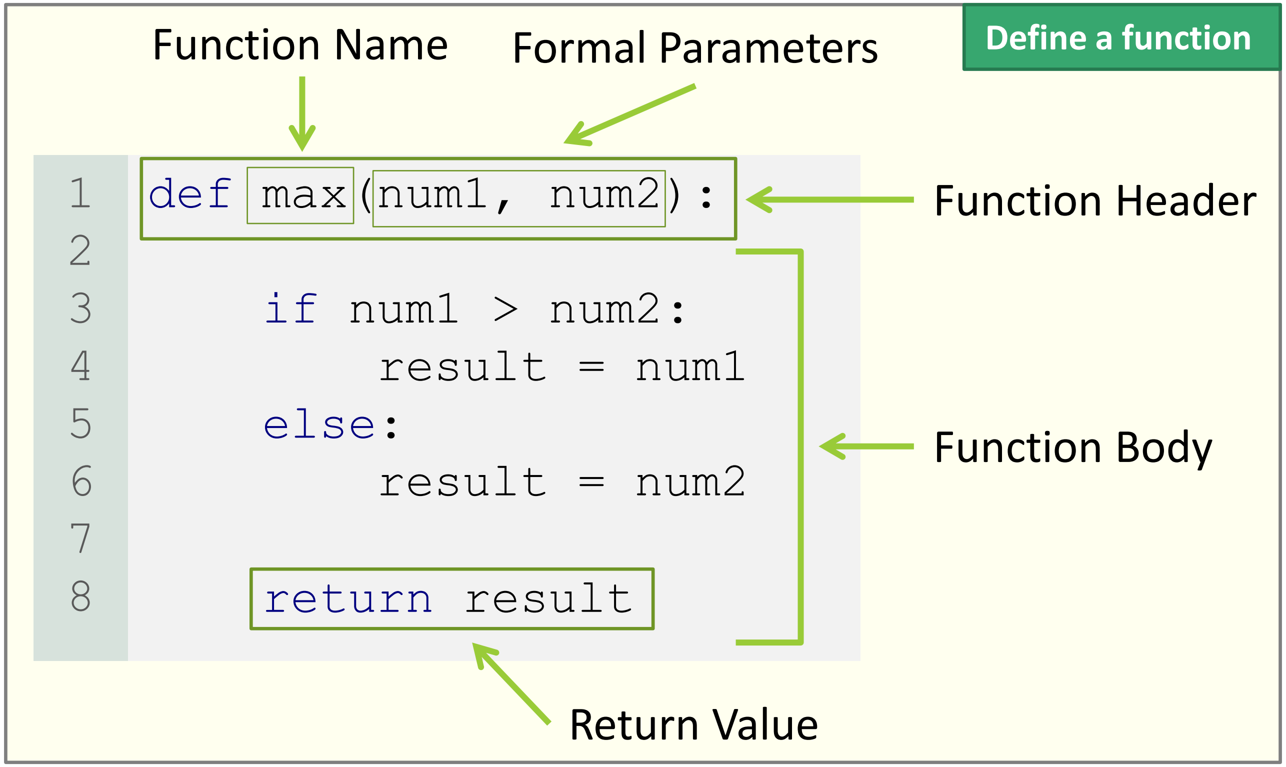
- Function’s definition defines the function, but it does **not** cause the function to execute.
  - A function is being executed when it is called or invoked.

Anatomy of a Function

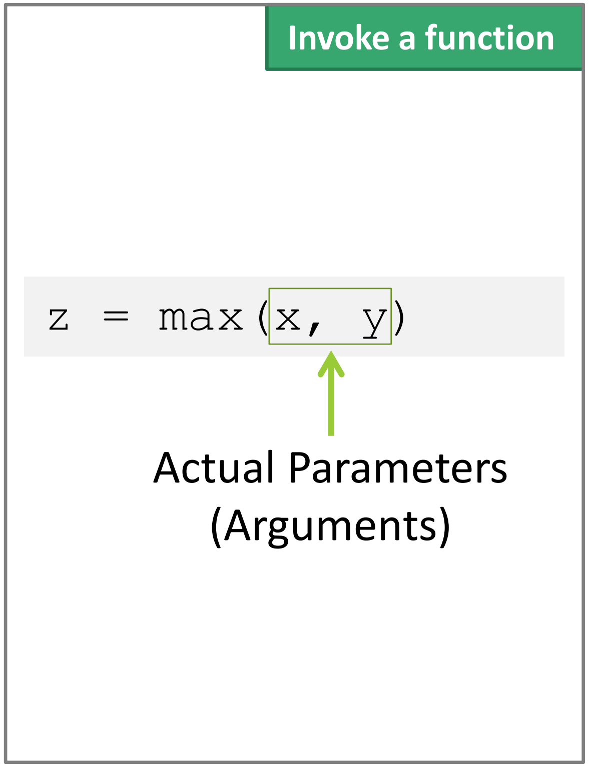
- We will now look at a sample function.
- This function is very easy.
  - Given two integers, find the larger value.
- Although the logic is easy, we will study this sample function in detail.
- We need to understand the anatomy of a function.
  - Anatomy: a study of the structure or internal workings of something.
  - In summary: we need to fully understand all components of the function and how it works!

Anatomy of a Function
Defining a Function

- This function, named max, has two parameters, num1 and num2. It returns the largest number from these parameters.



```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

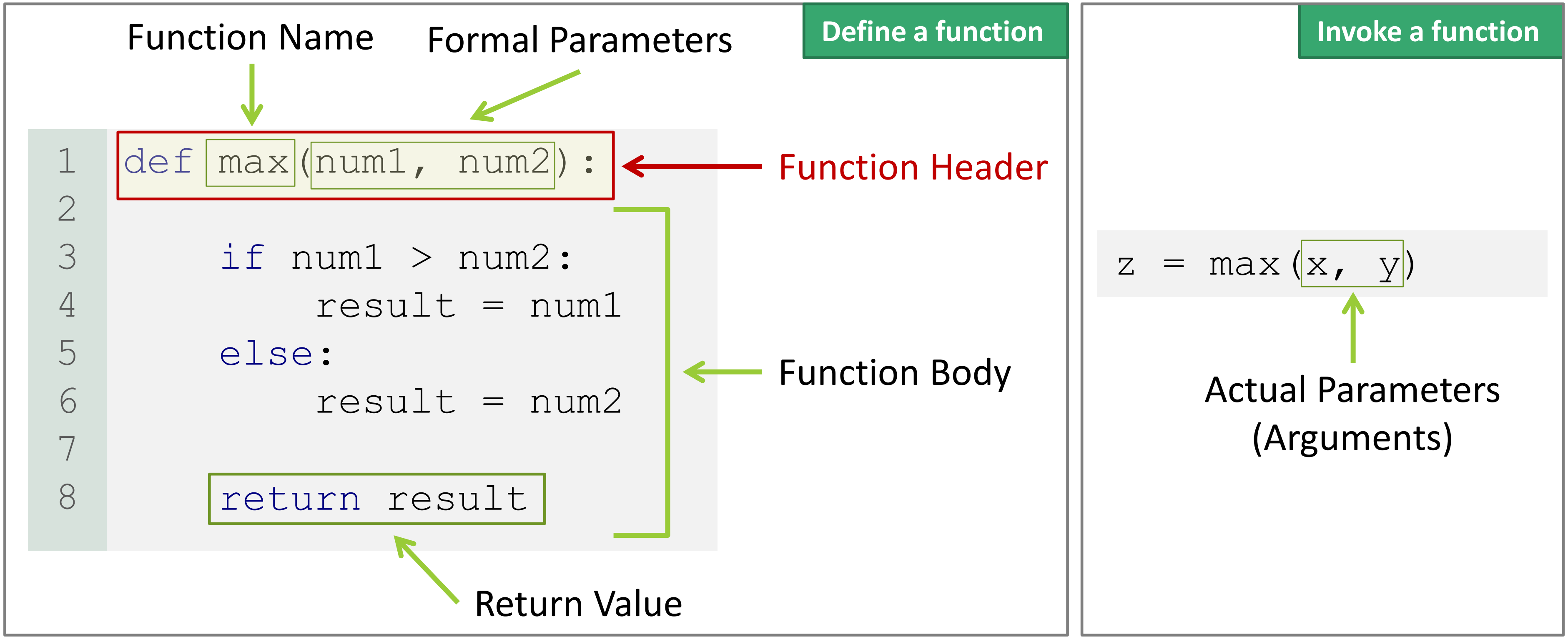


```
z = max(x, y)
```

Anatomy of a Function

Function Header

- The header begins with the def keyword, followed by function's name and parameters, and ends with a colon (:).

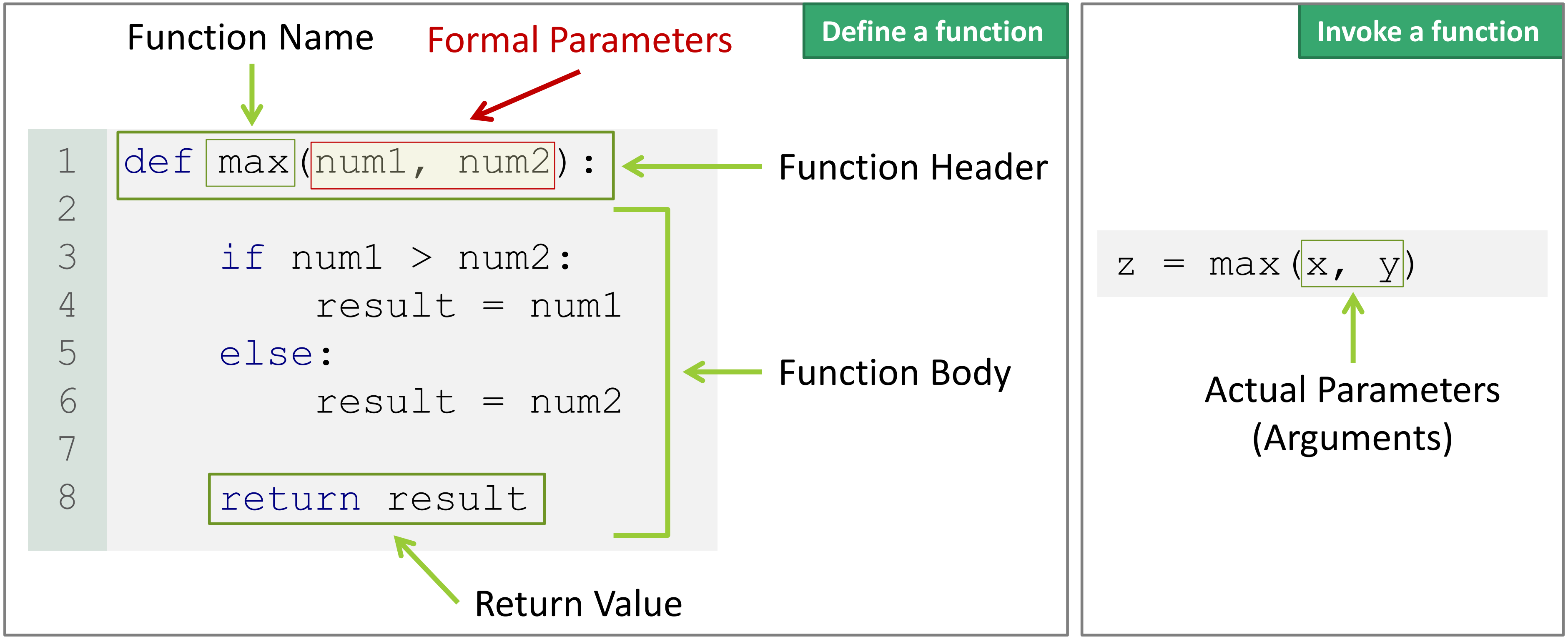


```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

```
z = max(x, y)
```

Anatomy of a Function
Formal Parameters

- The variables in the function header are known as formal parameters or simply parameters.



Define a function

Function Name

Formal Parameters

```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

Function Header

Function Body

Return Value

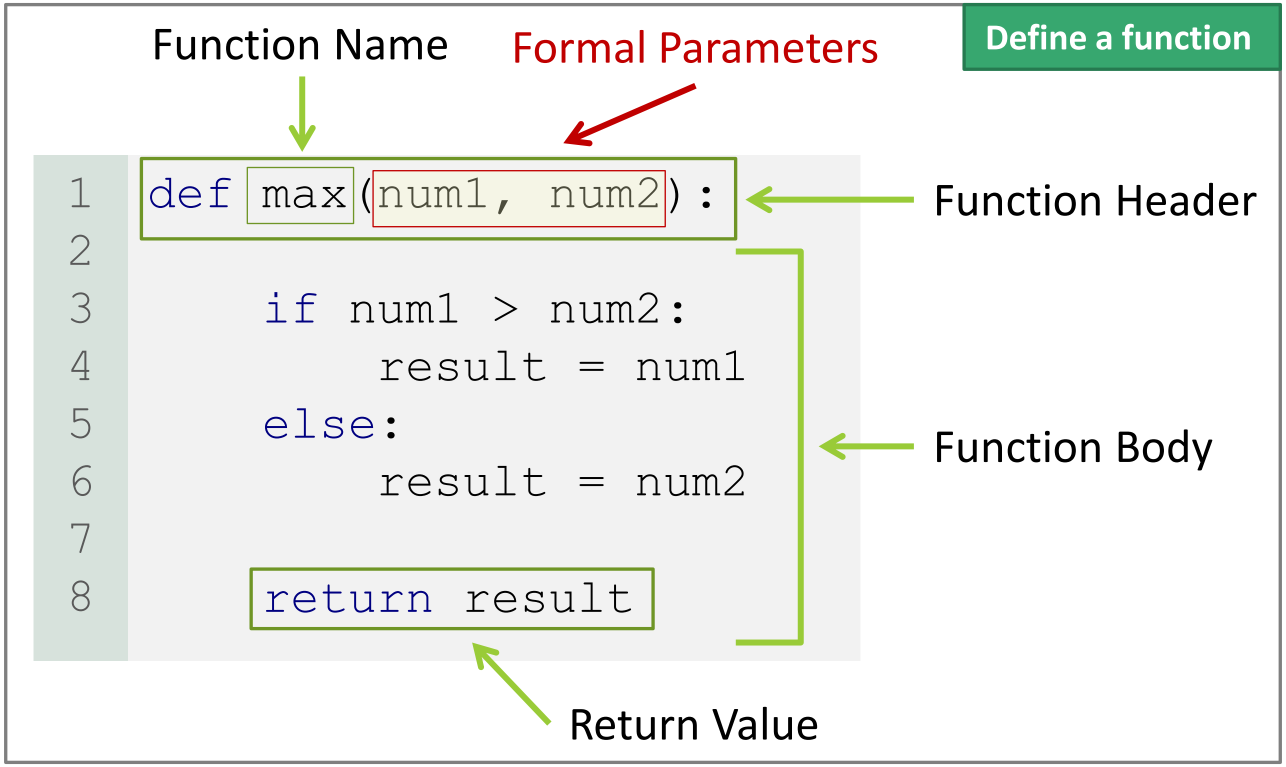
Invoke a function

```
z = max(x, y)
```

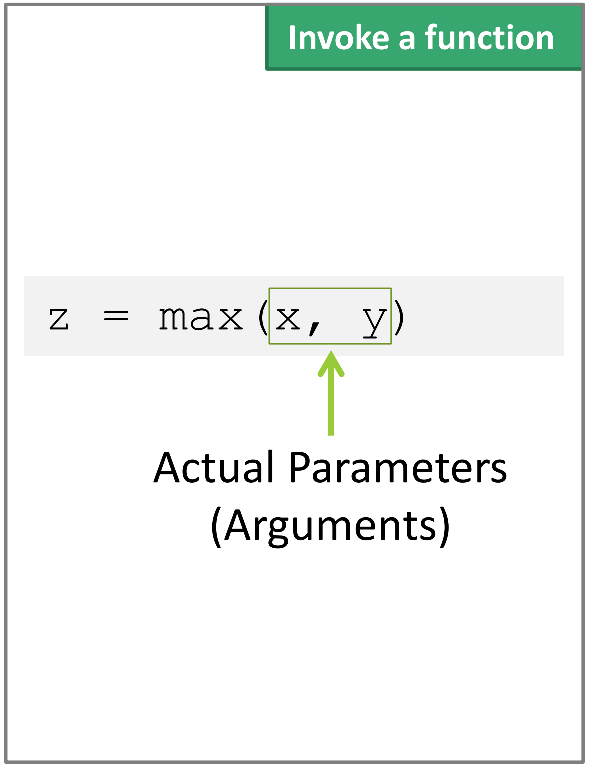
Actual Parameters
(Arguments)

Anatomy of a Function
Formal Parameters

- Parameters are **optional**; that is, a function may **not** have any parameters.
- Example: the random.random() function has no parameters.



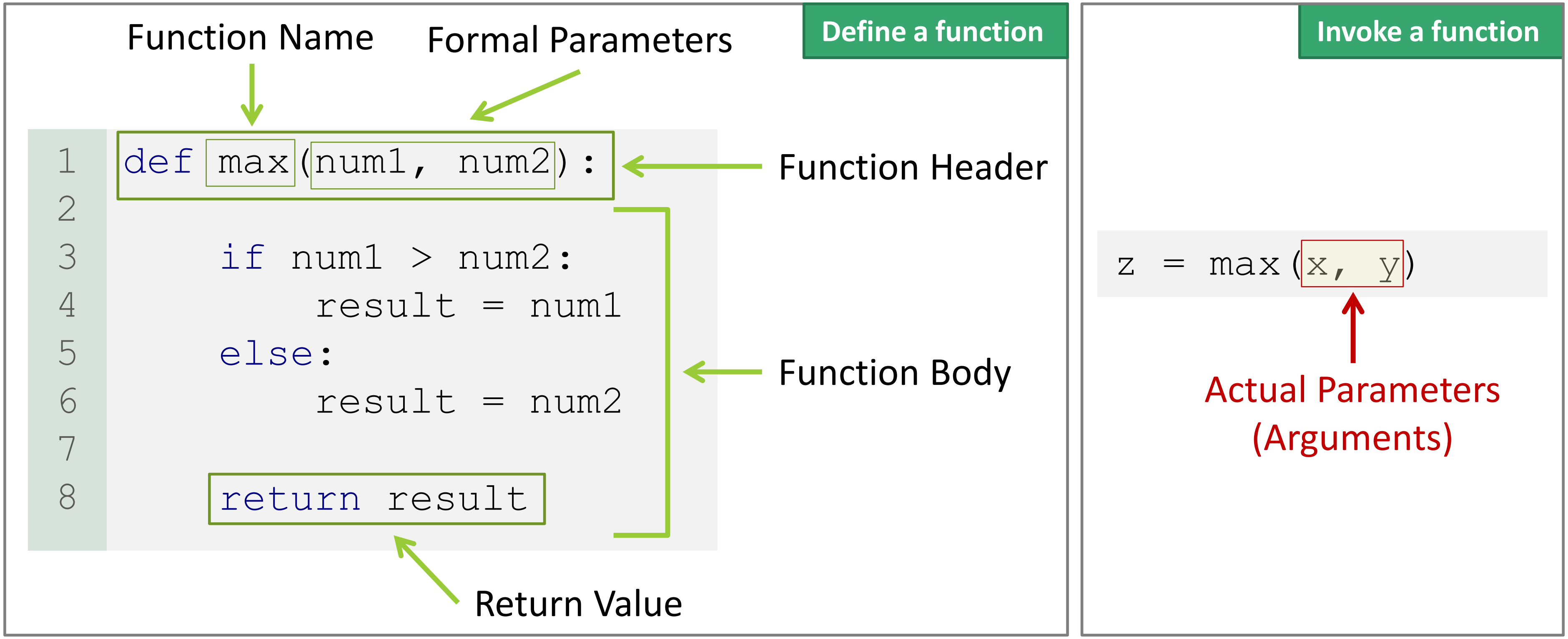
```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```



```
z = max(x, y)
```

Anatomy of a Function
Actual Parameters

- A parameter is like a placeholder: When a function is invoked, you pass a value to the parameter.
- This value is referred to as an actual parameter or argument.



Define a function

Function Name

Formal Parameters

```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

Function Header

Function Body

Return Value

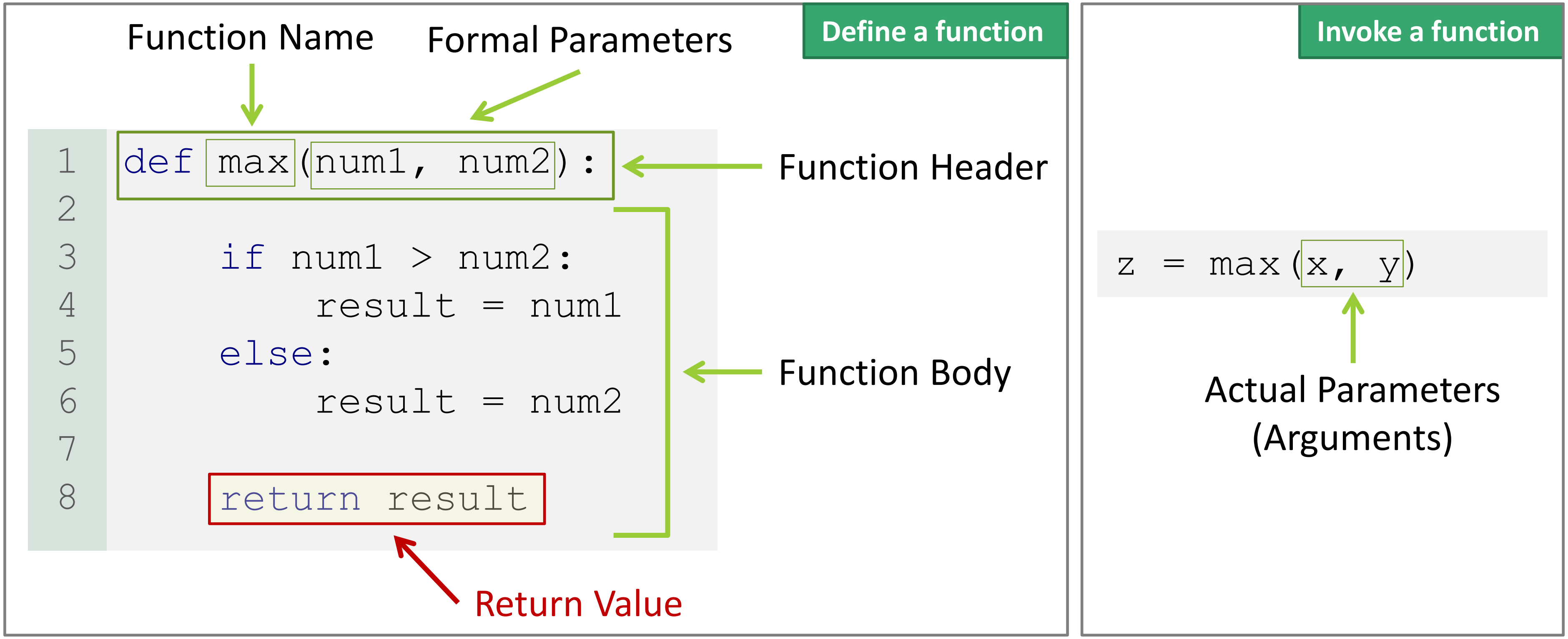
Invoke a function

```
z = max(x, y)
```

Actual Parameters (Arguments)

Anatomy of a Function
Return Value

- A function **may** return a value using the return keyword.
- Some functions return a value, while other functions perform desired operations **without** returning a value.



Define a function

Function Name

Formal Parameters

```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

Function Header

Function Body

Return Value

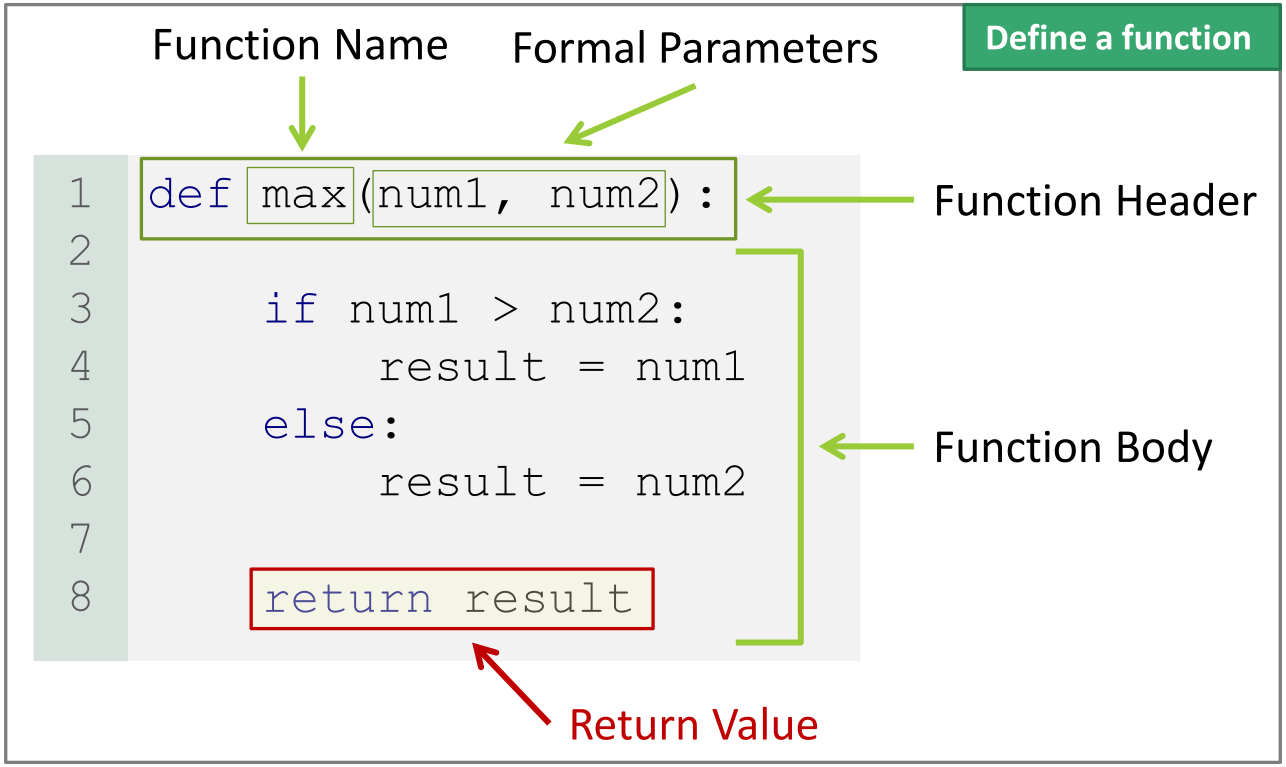
Invoke a function

```
z = max(x, y)
```

Actual Parameters
(Arguments)

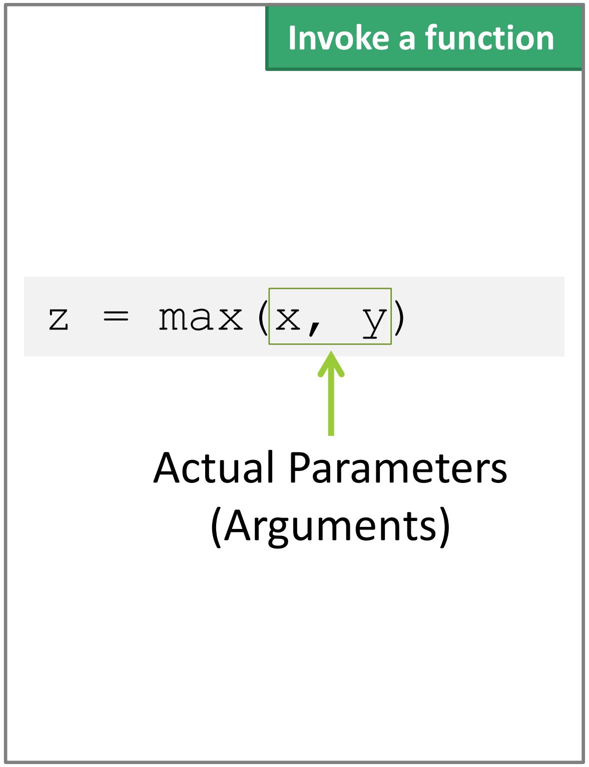
Anatomy of a Function
Return Value

- If a function returns a value, it is called a value-returning function.
- A return statement using the keyword return is required for a value-returning function to return a result.



Define a function

```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

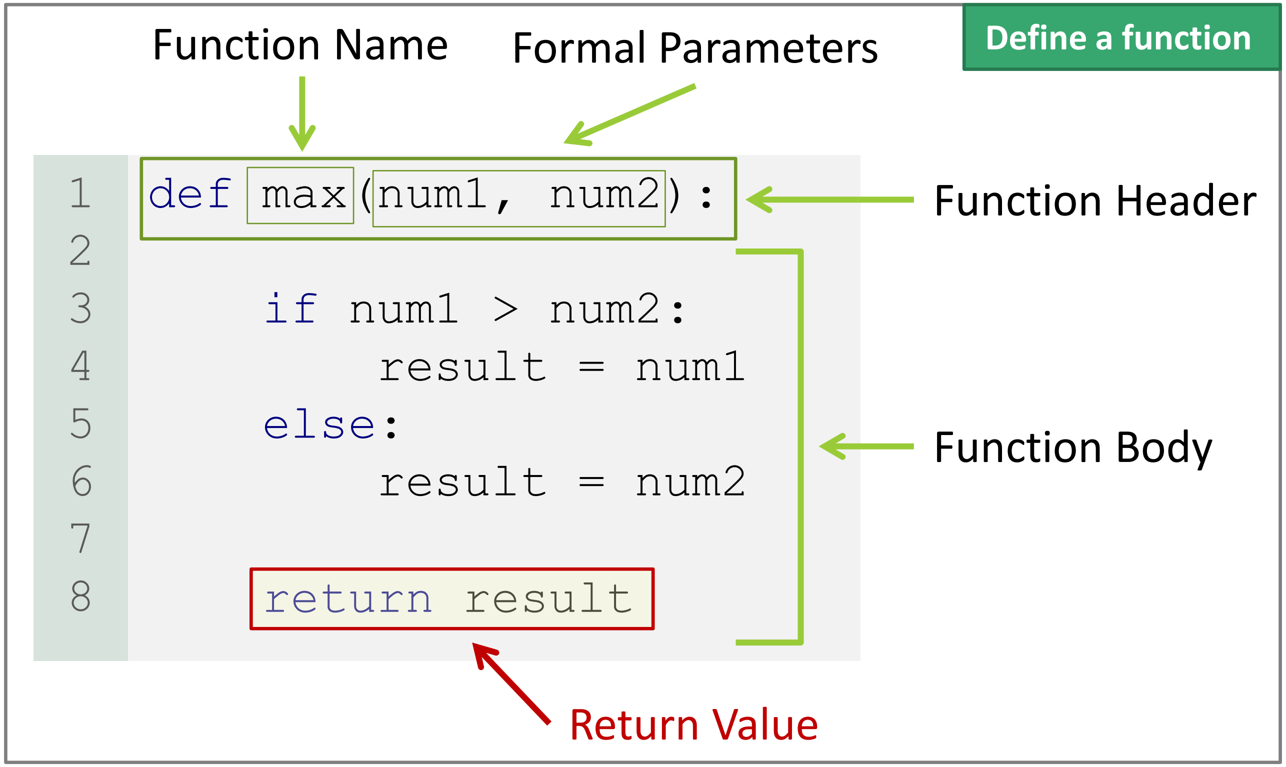


Invoke a function

```
z = max(x, y)
```

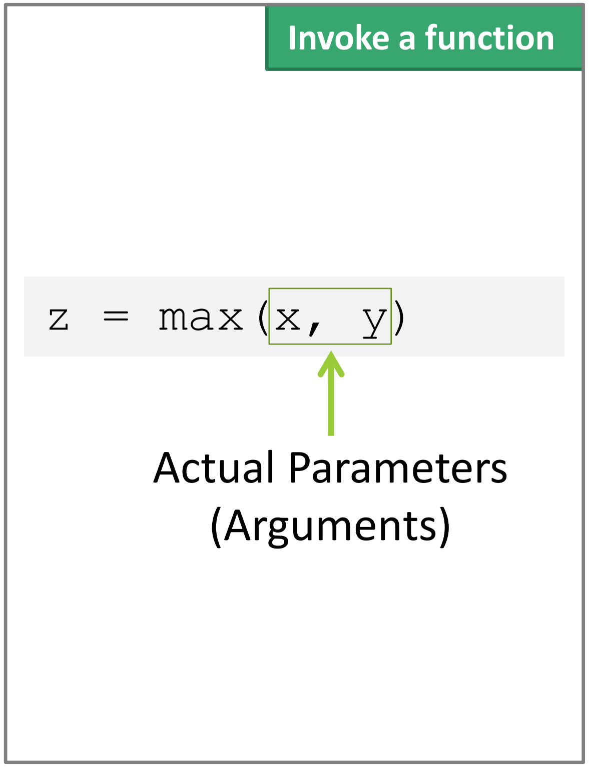
Anatomy of a Function
Return Value

- The function **terminates** when a return statement is executed.



Define a function

```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

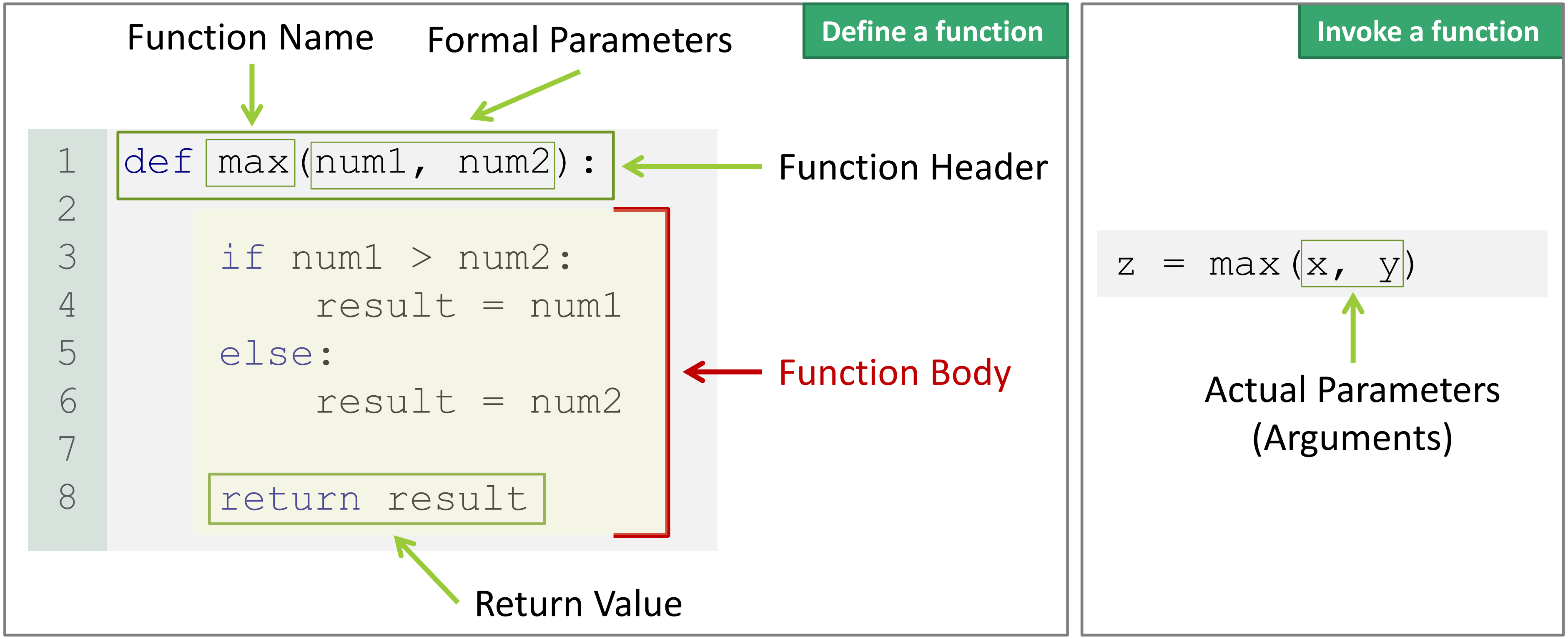


Invoke a function

```
z = max(x, y)
```

Anatomy of a Function
Function Body

- The function body contains a collection of statements that define what the function does.

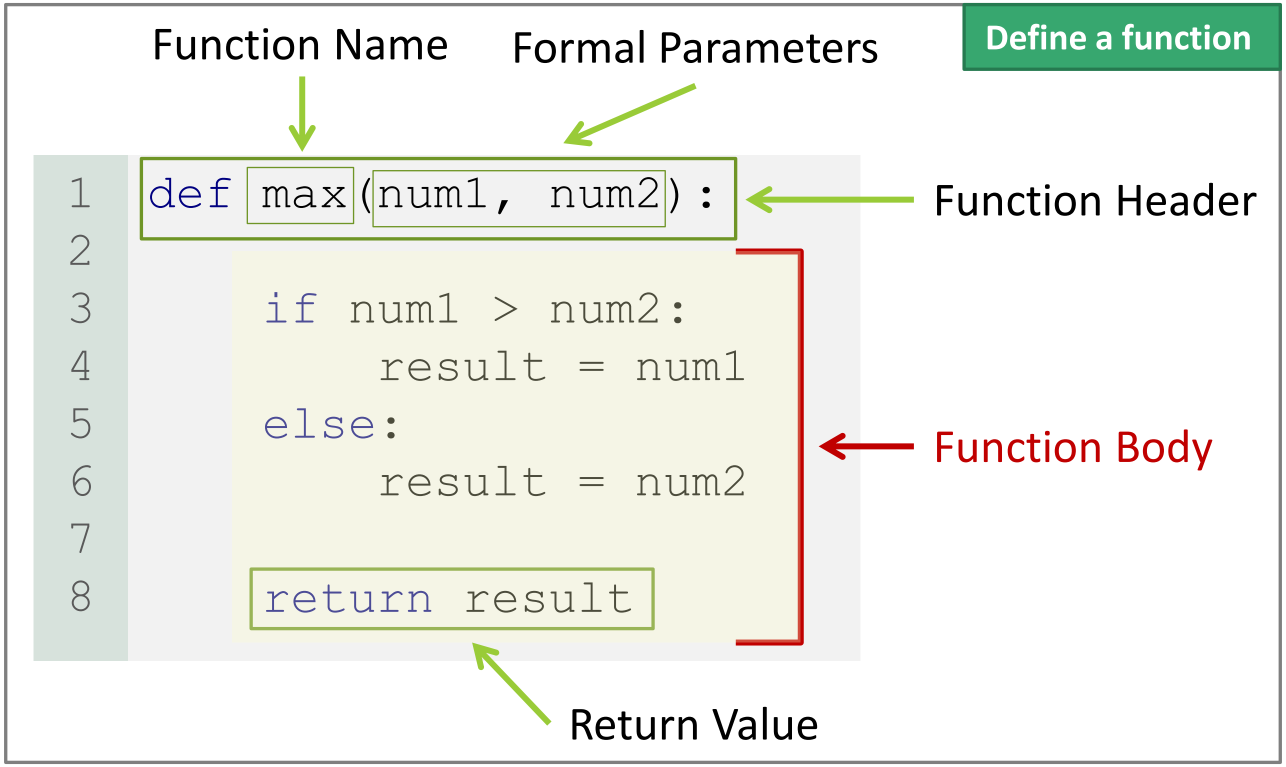


```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

```
z = max(x, y)
```

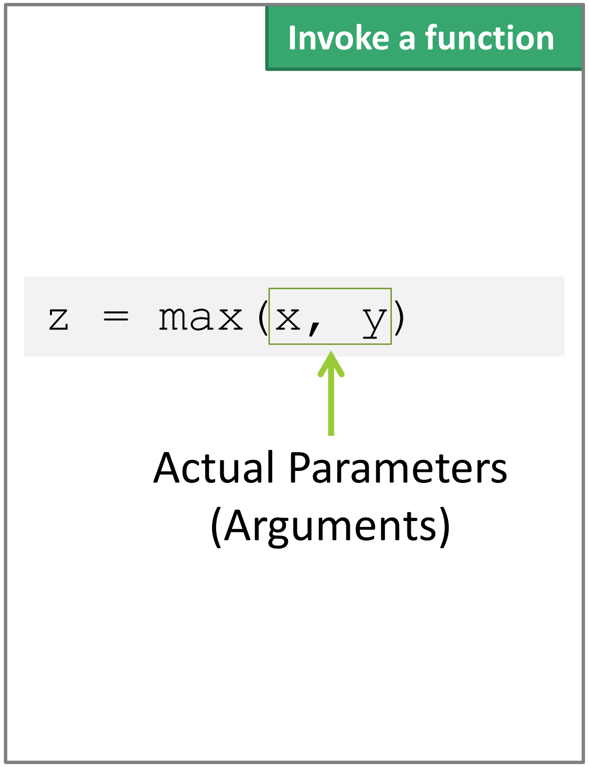
Anatomy of a Function
Function Body

- For example, the function body of the max function uses an if statement to determine which number is larger and return the value of that number.



Define a function

```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

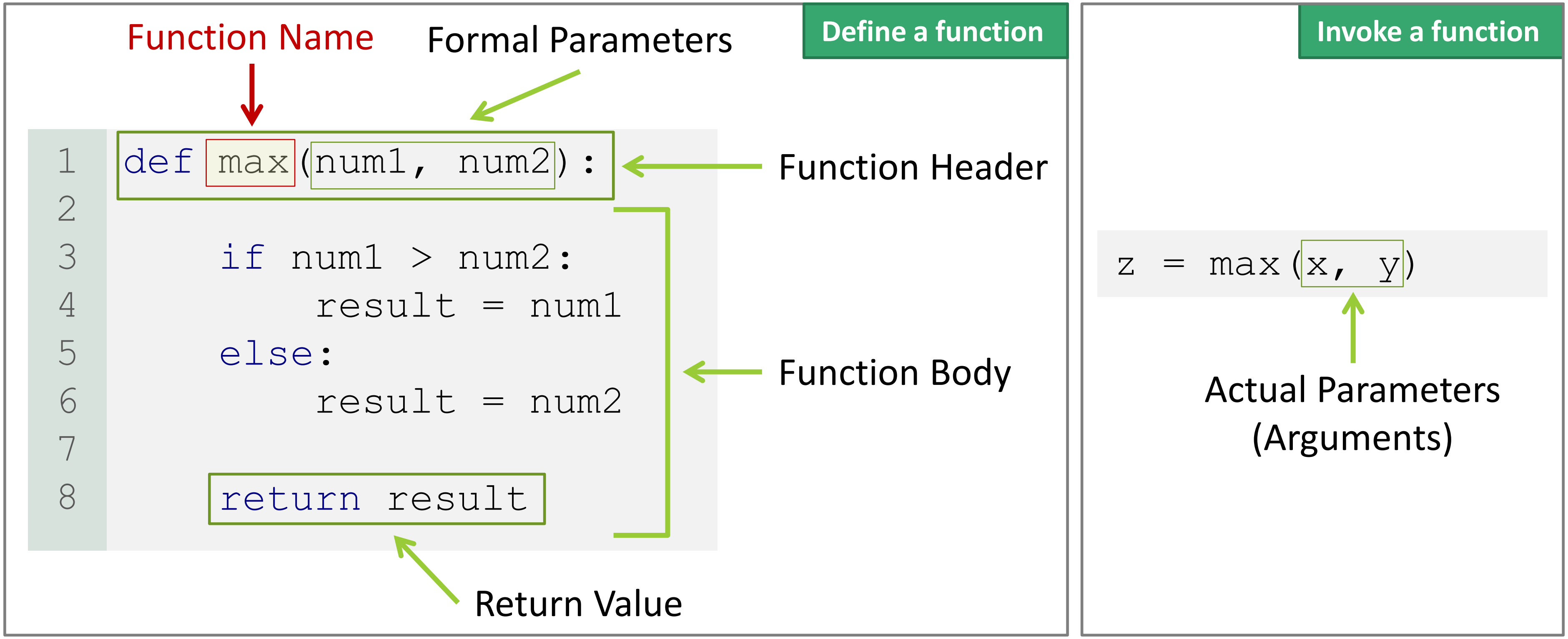


Invoke a function

```
z = max(x, y)
```

Anatomy of a Function
Function Name

- The function name is used to invoke (call) the function.
- The function is being executed when it is called or invoked.



```
def max(num1, num2):

    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

```
z = max(x, y)
```

Remember
Naming Conventions

- In Chapter 2 slides, Section 2.7, we have learned naming conventions of variables and functions.
  - Choose meaningful and descriptive names.
  - Use lowercase.
  - If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name (camelCase).
    - For example: computeArea, interestRate, yourFirstName.
  - **Or** use lowercase for all words and concatenate them using underscore (_).
    - For example: compute_area, interest_rate, your_first_name.
- Do you have to follow these rules?
  - No. But it makes your program much easier to read!

6.3. Calling a Function

- Program Control
- Program 2: Testing max Function
- Trace Function Invocation
- Activation Record
- Trace Call Stack
- Activation Record and Call Stacks

Calling a Function

- Remember:
  - A function is a collection of statements grouped together to perform an action.
  - So inside the function, you define the actions.
    - ➢ You "do" everything that you want the function to "do“.
- How do we "start" the function? How do we run it?

➢ Answer: We call or invoke the function.

- Calling a function executes the code in the function.
- The program that calls the function is called a caller.

Calling a Function
That Returns a Value

There are two ways to call a function, depending on whether or not it returns a value:

1. If the function returns a value, a call to that function is usually treated as a value.
   - Example #1:

     ```
     larger = max(3, 4)
     ```

     - Here, we "call" the function, max(3, 4).
     - The maximum number, which is **4**, will get returned.
     - We save that value (**4**) into the variable larger.
   - Example #2:

     ```
     print(max(3, 4))
     ```

     - Here, we directly print the result, which is **4**.

Calling a Function
That Does Not Return a Value

There are two ways to call a function, depending on whether or not it returns a value:

2. If a function does not return a value, the call to the function must be a statement.

   - Example:

     ```
     print("This is a parameter!")
     ```

     - Here, we "call" the print function.
     - We send over the string, "This is a parameter!".
     - That function receives the string and prints to output.

Note

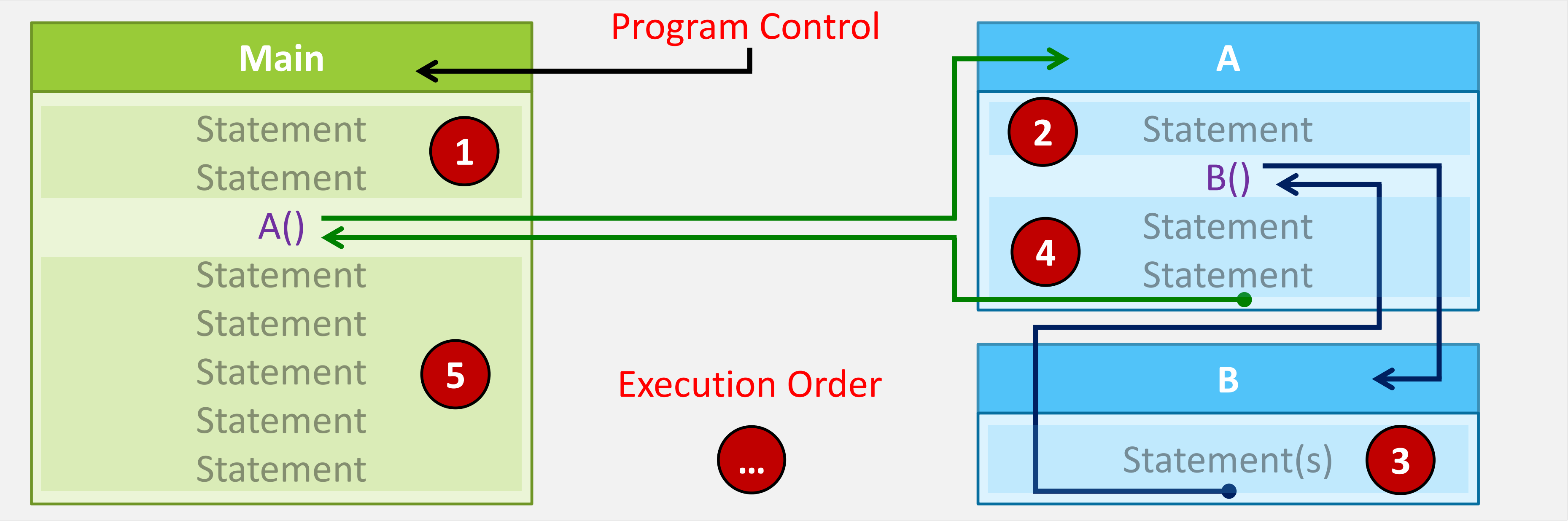
- A value-returning function also can be invoked as a statement.
- Example:

```
max(3, 4)
```

- In this case, the return value is ignored.
- This is rare but is permissible if the caller is not interested in the return value.

Program Control

- When a program calls a function, program control is transferred to the called function.
- A called function returns control to the caller when:
  - Its return statement is executed.
  - **Or** the function is finished.



Testing max Function
Program 2

Write a program that will call a function, max, to determine the maximum of two numbers. Function max should return the maximum value.

Suppose the two numbers are **2** and **5**.

```
The larger number of 5 and 2 is 5
```

Testing max Function
Phase 1: Problem-solving

- Define a main function (It is a good practice).
- In main function, we just make two integers and give a value.
  - Of course, we could ask the user for two numbers.
  - Or we could generate two random numbers.
  - These are easy things and are not the purpose of this example.
- Next, we call the max function inside the main function.
- This means we need to write a max function!
  - max function should be easy.
  - Just check which number is larger.
  - Save the larger number into a variable.
  - Finally, return that variable (the larger number).
- At the end, outside of the functions, call the main function to be the first function that will be executed by Python interpreter when it runs the program.

Testing max Function
Phase 2: Implementation

LISTING 6.1 TestMax.py

Run

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

main() # Call the main function
```

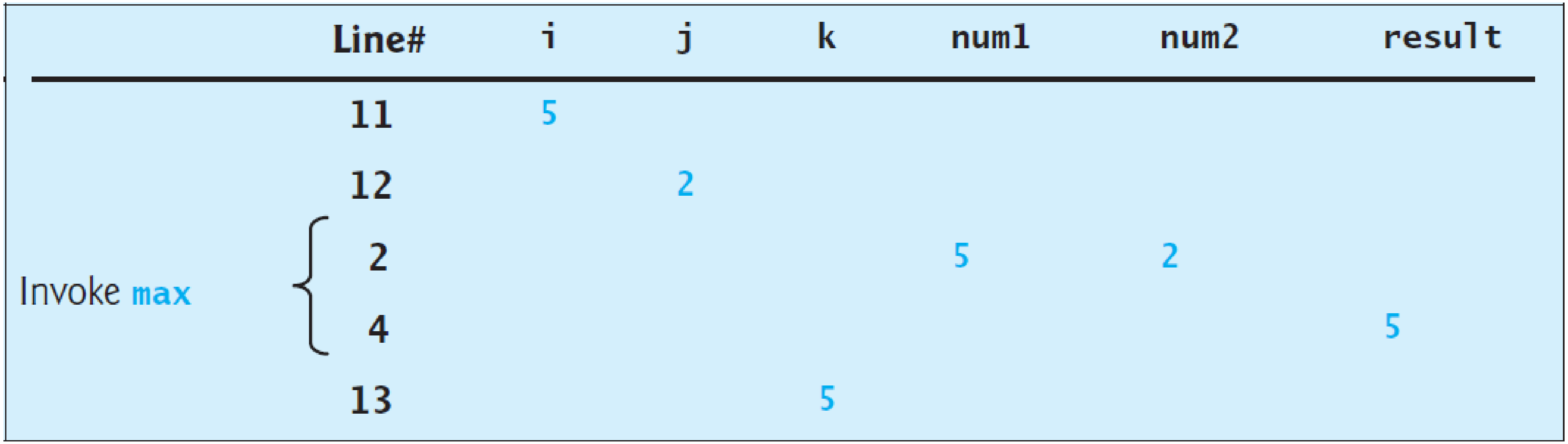
```
The larger number of 5 and 2 is 5
```

Testing max Function
Details

- This program contains the max and main functions.
- The program script invokes the main function in line 16.
- By convention, programs often define a function named main that contains the main functionality for a program.

Testing max Function
Trace The Program Execution

```
The larger number of 5 and 2 is 5
```

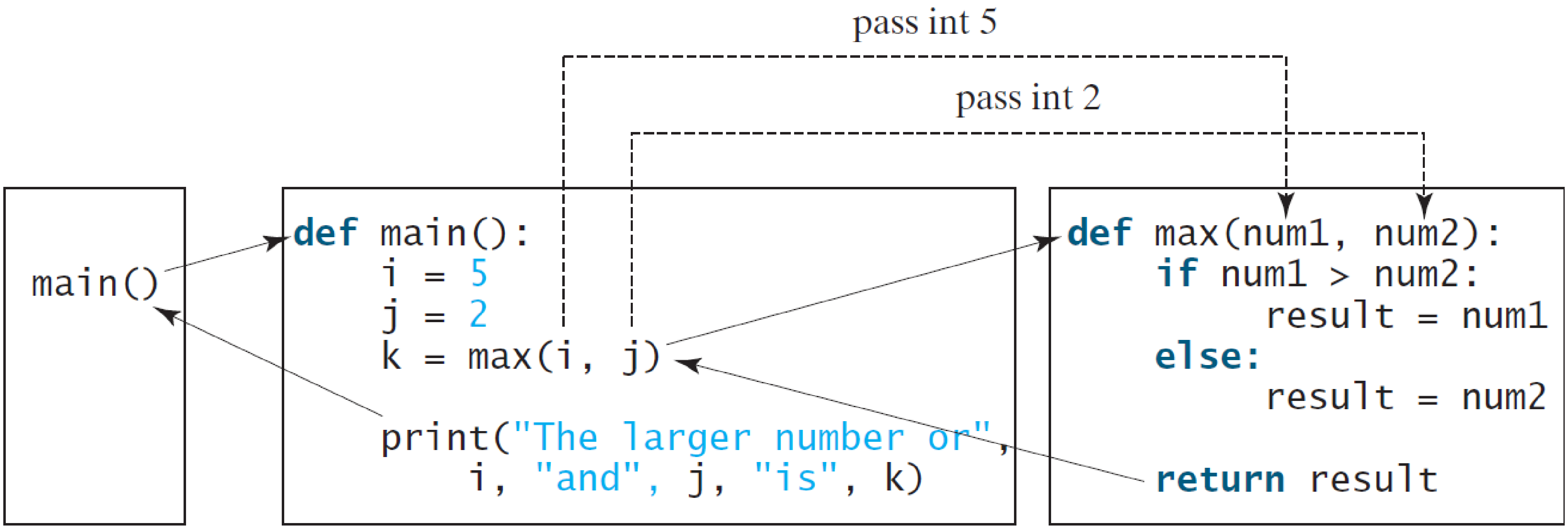


	Line#	i	j	k	num1	num2	result
	11	5					
	12		2				
Invoke max	2				5	2	
	4						5
	13			5			

Testing max Function
Discussion

- How is this program executed? The interpreter reads the script in the file line by line starting from line 1.
- Since line 1 is a comment, it is ignored.
- When it reads the function header in line 2, it stores the function with its body (lines 2–8) in the memory.
- Remember that a function's definition defines the function, but it **does not** cause the function to execute.
- The interpreter then reads the definition of the main function (lines 10–14) to the memory.
- Finally, the interpreter reads the statement in line 16, which invokes the main function and causes the main function to be executed.
- The control is now transferred to the main function.

Testing max Function
Discussion



pass int 5

pass int 2

```
main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The larger number or",
          i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

- When a function is invoked, the control is transferred to the function.
- When the function is finished, the control is returned to where the function was called.

Testing max Function
Discussion

- The execution of the main function begins in line 11.
- It assigns **5** to i and **2** to j (lines 11–12) and then invokes max(i, j) (line 13).
- When the max function is invoked (line 13), variable i's value is passed to num1 and variable j's value is passed to num2.
- The control is transferred to the max function, and the max function is executed.
- When the return statement in the max function is executed, the max function returns the control to its caller (in this case the caller is the main function).

Testing max Function
Discussion

- After the max function is finished, the returned value from the max function is assigned to k (line 13).
- The main function prints the result (line 14).
- The main function is now finished, and it returns the control to its caller (line 16).
- The program is now finished.

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Print Start ...

```
Start ...
```

Trace Function Invocation

After this line executes, program control go to function main.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Invoke the main function

```
Start ...
```

Trace Function Invocation

Program control is now at function main.

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Execute main()

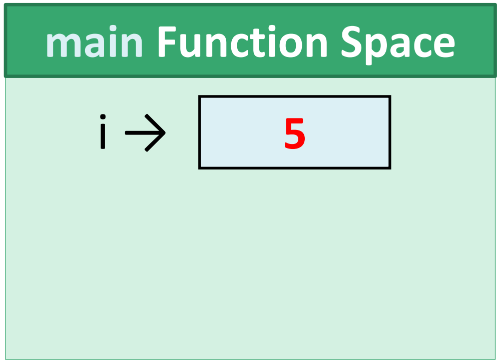
main Function Space

```
Start ...
```

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

i is now **5**


main Function Space

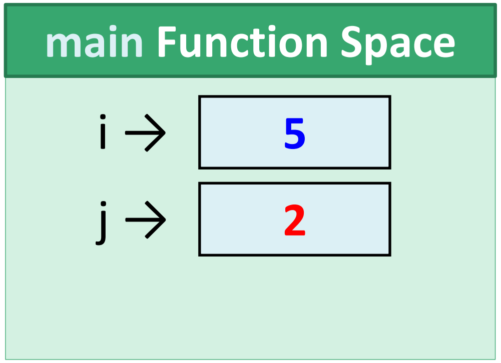
i → **5**

```
Start ...
```

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

j is now 2



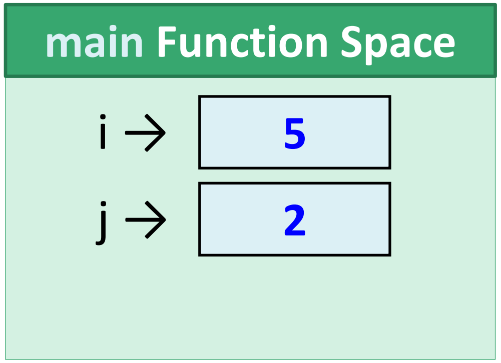
```
Start ...
```

Trace Function Invocation

After this line executes, program control go to function max.

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

invoke max(i, j)


main Function Space

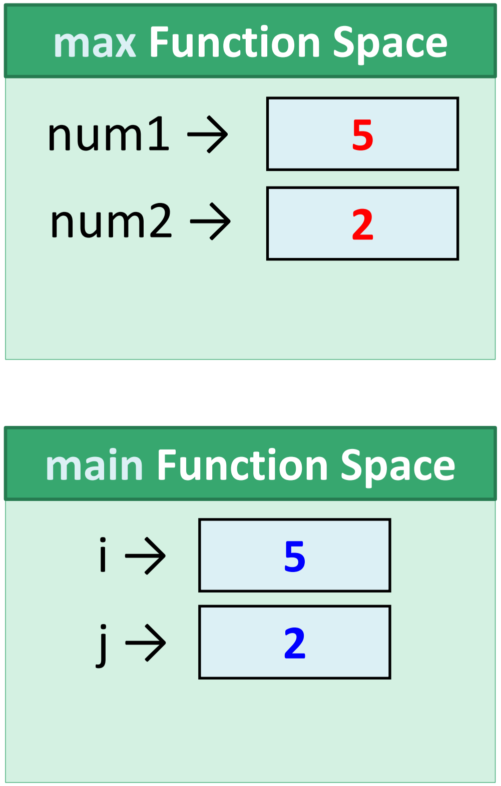
```
Start ...
```

Trace Function Invocation

Program control is now at function max.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Execute max(i, j)
Pass the value of i to num1
Pass the value of j to num2



max Function Space

num1 → 5

num2 → 2

main Function Space

i → 5

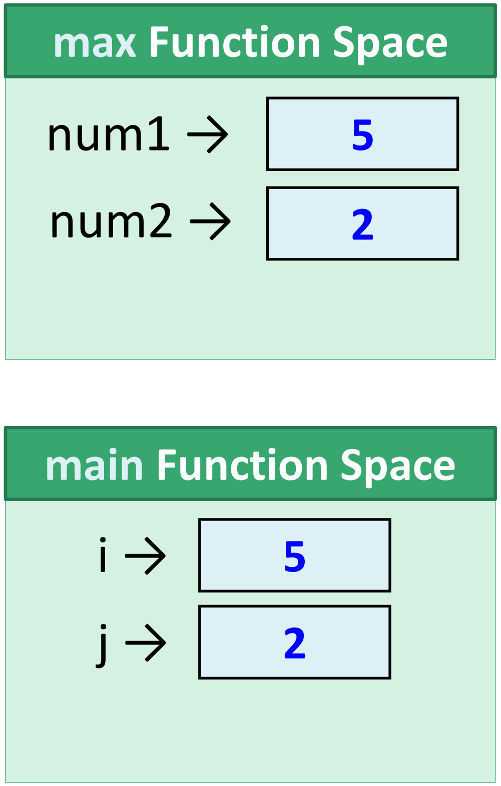
j → 2

```
Start ...
```

Trace Function Invocation

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

(num1 > num2) is True
since num1 is 5 and num2 is 2

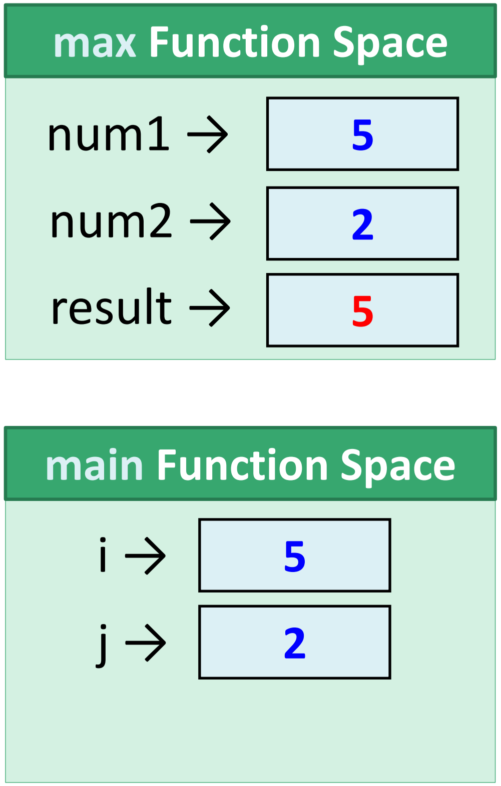


```
Start ...
```

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

result is now **5**



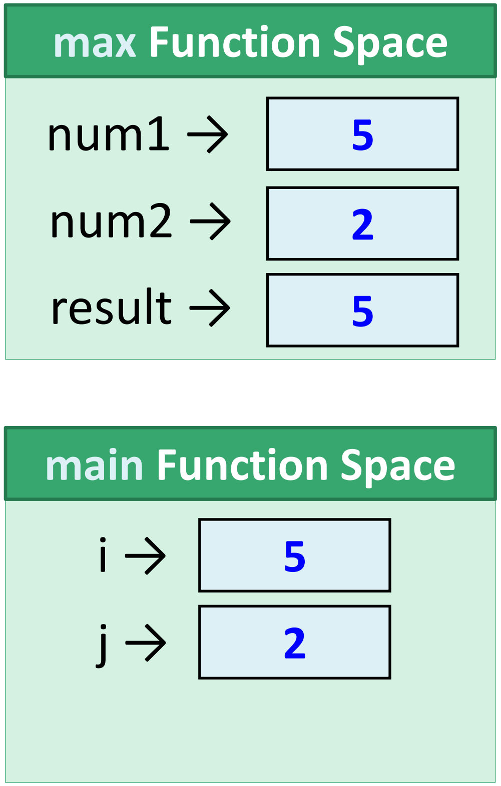
```
Start ...
```

Trace Function Invocation

Now, the maximum value is returned. Program control returns to main.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Return result, which is **5**



max Function Space

num1 → 5

num2 → 2

result → 5

main Function Space

i → 5

j → 2

```
Start ...
```

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Return max(i, j) and assign the return value (5) to k

max Function Space

num1 → 5

num2 → 2

result → 5

main Function Space

i → 5

j → 2

k → 5

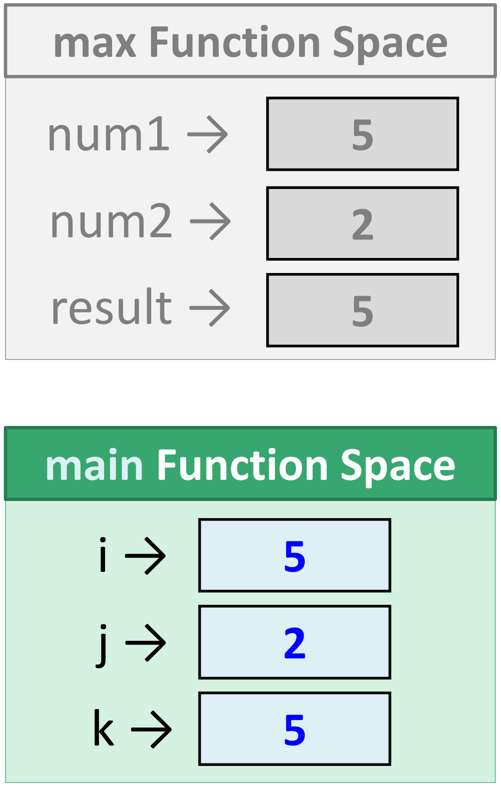
```
Start ...
```

Trace Function Invocation

After this line executes, program control returns to the script.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

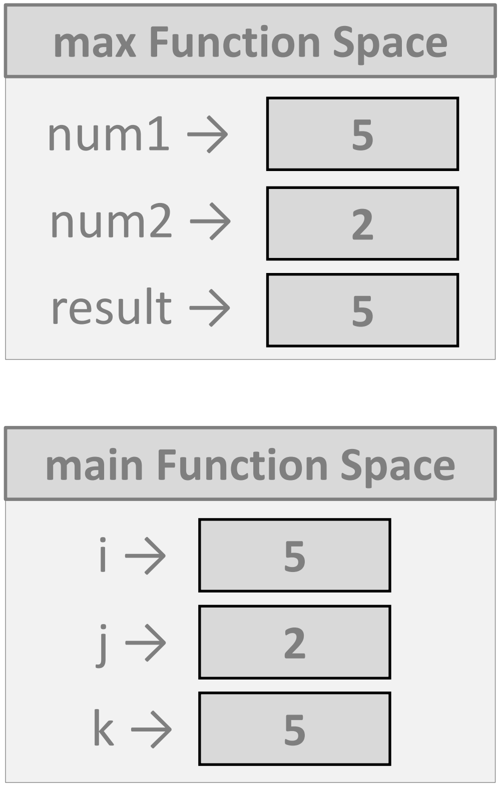
Execute the print statement.



```
Start ...
The maximum between 5 and 2 is 5
```

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```



max Function Space

num1 → 5

num2 → 2

result → 5

main Function Space

i → 5

j → 2

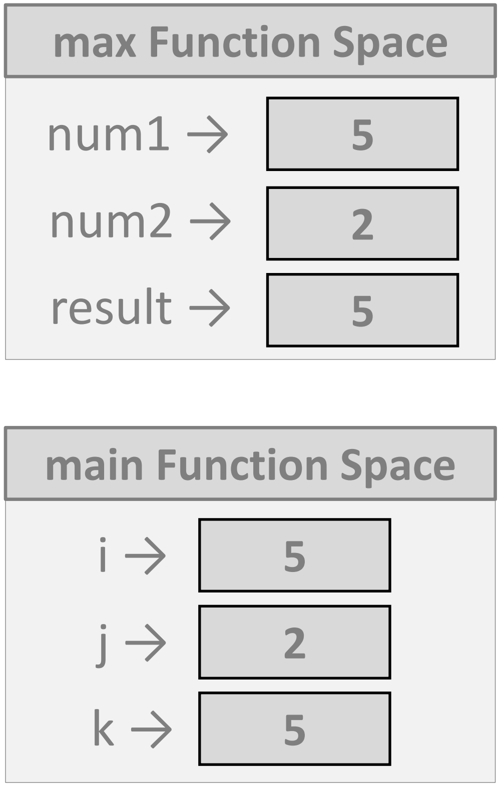
k → 5

main() returns nothing (None)

```
Start ...
The maximum between 5 and 2 is 5
```

Trace Function Invocation

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```



max Function Space

num1 → 5

num2 → 2

result → 5

main Function Space

i → 5

j → 2

k → 5

```
Start ...
The maximum between 5 and 2 is 5
... End
```

Execute the print statement

Note

- In the preceding example, main is defined after max.
- In Python, functions can be defined in any order in a script file as long as the function is in the memory when it is called.
- You can also define main before max.

Activation Record

- **Each time** a function is called, the system creates an activation record.
- The activation record stores all parameters and variables for the function.
- The activation record is then placed in a specific area of memory known as a call stack.
  - Also known as "execution stack", "machine stack" or just "the stack".
- A call stack stores the activation records in a last-in, first-out fashion.

Activation Record

- When functionA calls functionB, for example, the activation record for functionA is kept intact.
- A new activation record for functionB is created for this new function that was just called.
- When functionB finishes its work and returns to the caller, which was functionA, the activation record of functionB is then removed from the stack of records.
- Why?
  - Because functionB is finished!
  - Confused? Let us see an example…

Trace Call Stack

Program control is now at the script.

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Stack is now empty

Call Stack

Execute the print statement

Trace Call Stack

Program control is now at the script.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Stack is now empty

Call Stack

Invoke the main function

Trace Call Stack

The main function is invoked

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Stack is
now empty

Call Stack

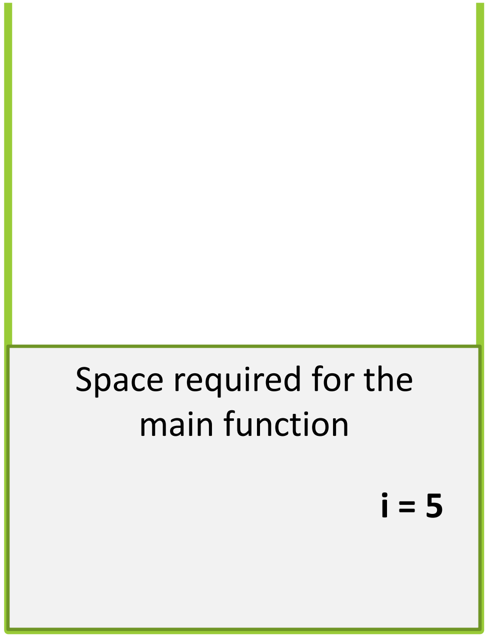
Execute main()

Trace Call Stack

The main function is invoked

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

i is now **5**



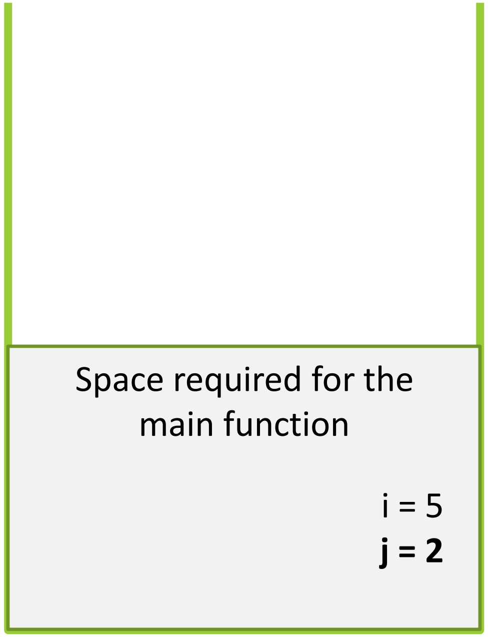
Call Stack

Trace Call Stack

The main function is invoked

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

j is now **2**



Space required for the main function

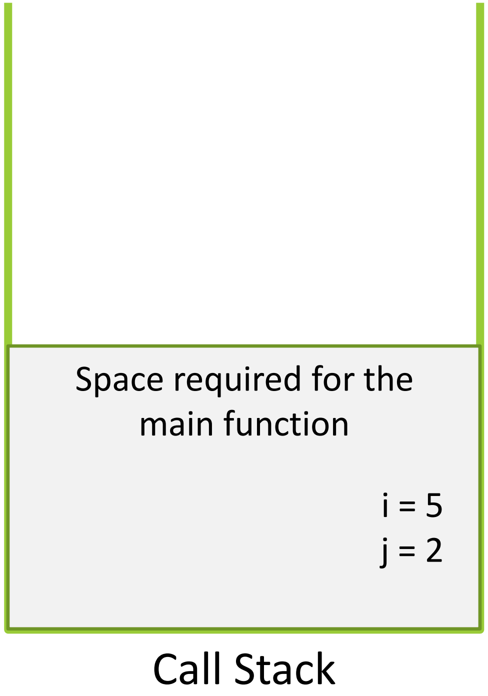
i = 5
j = 2

Call Stack

Trace Call Stack

The main function is invoked

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

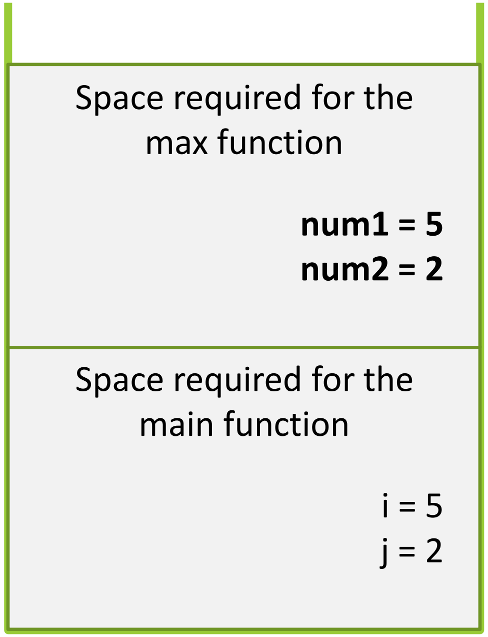


invoke max(i, j)

Trace Call Stack

The max function is invoked

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```


Space required for the max function
num1 = 5
num2 = 2
Space required for the main function
i = 5
j = 2

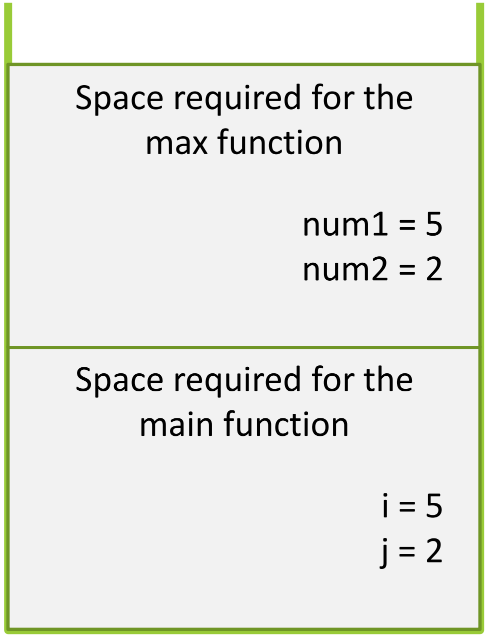
Call Stack

Execute max(i, j)
Pass the value of i to num1, and pass the value of j to num2

Trace Call Stack

The max function is invoked

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```



Space required for the max function

num1 = 5
num2 = 2

Space required for the main function

i = 5
j = 2

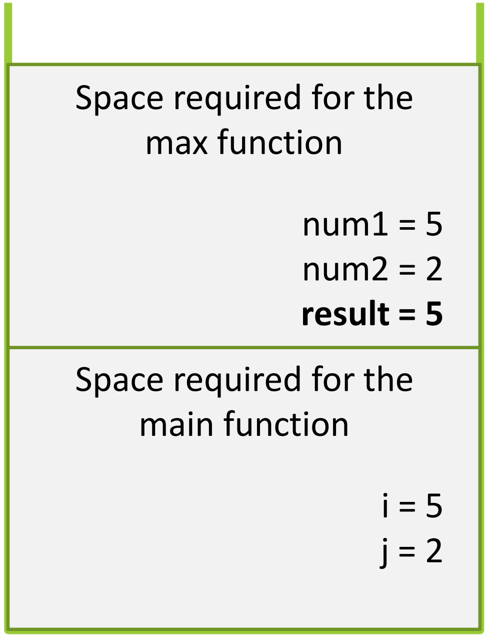
Call Stack

(num1 > num2) is True
since num1 is **5** and num2 is **2**

Trace Call Stack

The max function is invoked

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```



Space required for the max function

num1 = 5
num2 = 2
result = 5

Space required for the main function

i = 5
j = 2

Call Stack

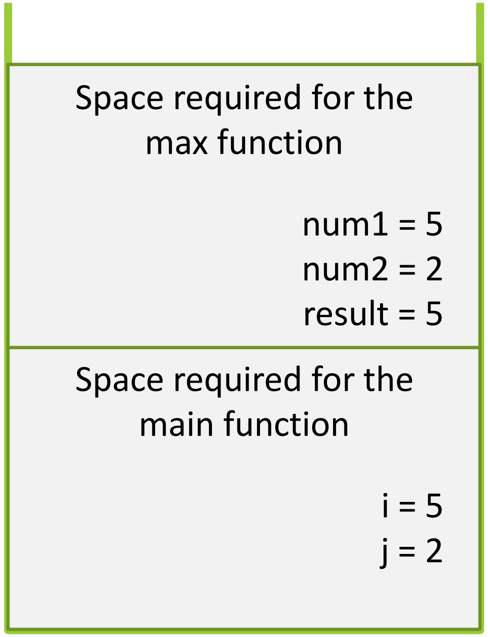
result is now **5**

Trace Call Stack

The max function is invoked

```
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Return result, which is **5**

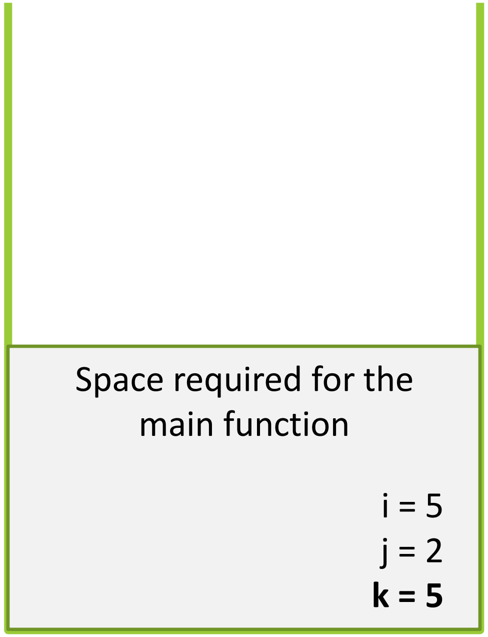

Space required for the max function
num1 = 5
num2 = 2
result = 5
Space required for the main function
i = 5
j = 2

Call Stack

Trace Call Stack

The main function is invoked

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```



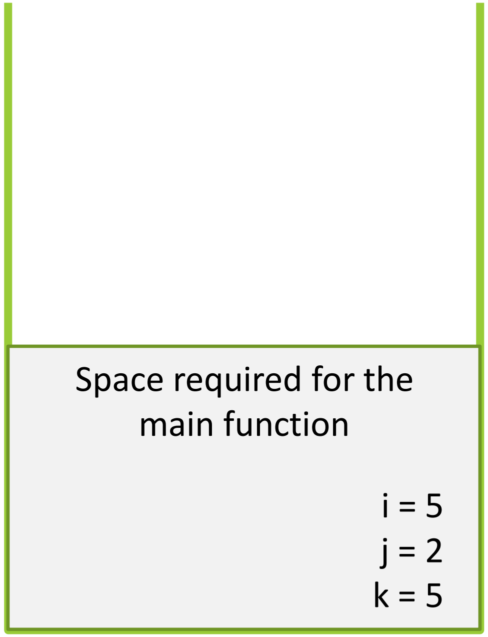
Call Stack

Return result, which is **5**

Trace Call Stack

The main function is invoked

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```


Space required for the main function

i = 5
j = 2
k = 5

Call Stack

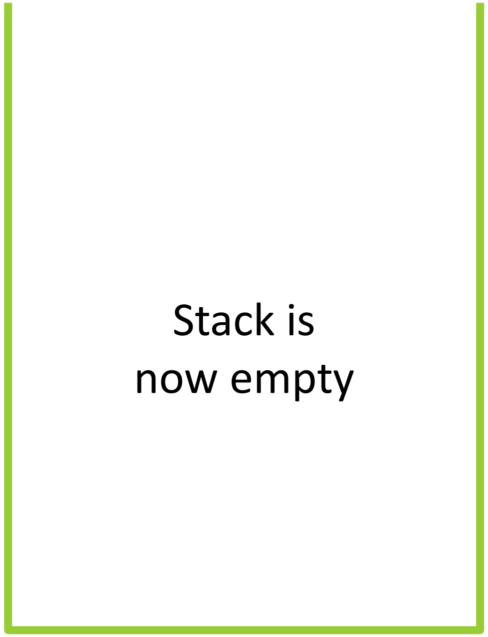
Execute print statement

Trace Call Stack

Program control is now at the script.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

main() returns nothing (None)



Stack is now empty

Call Stack

Trace Call Stack

Program control is now at the script.

```python
# Return the max between two numbers
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result

def main():
    i = 5
    j = 2
    k = max(i, j) # Call the max function
    print("The maximum between", i, "and", j, "is", k)

print("Start ...")
main() # Call the main function
print("... End")
```

Stack is now empty

Call Stack

Execute the print statement

Activation Record and Call Stacks

Summary

- When a function is invoked, an activation record is created to store variables in the function.
- The activation record is released after the function is finished.

6.4. Functions with/without Return Values

- Functions without Return Values
- Program 3: Testing Void Function
- Functions with Return Values
- Program 4: Testing getGrade Function
- None Value
- Terminating Void Functions
- Check Point #1 - #10

Functions without Return Values

- The previous example (max function) was a value-returning function.
  - Meaning, it returned a value (the max) to the caller.
- Some functions do not return anything at all.
  - A function **does not have** to return a value.
- This kind of function is commonly known as a void function in programming terminology.
- The following program (Program 3) defines a function named printGrade and invokes (calls) it to print the grade based on a given score.

Testing Void Function
Program 3

LISTING 6.2 PrintGradeFunction.py

```python
# Print grade for the score
def printGrade(score):
    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')

def main():
    score = eval(input("Enter a score: "))
    print("The grade is ", end = "")
    printGrade(score)

main() # Call the main function
```

Run

Testing Void Function
Discussion

- Example runs of the program:

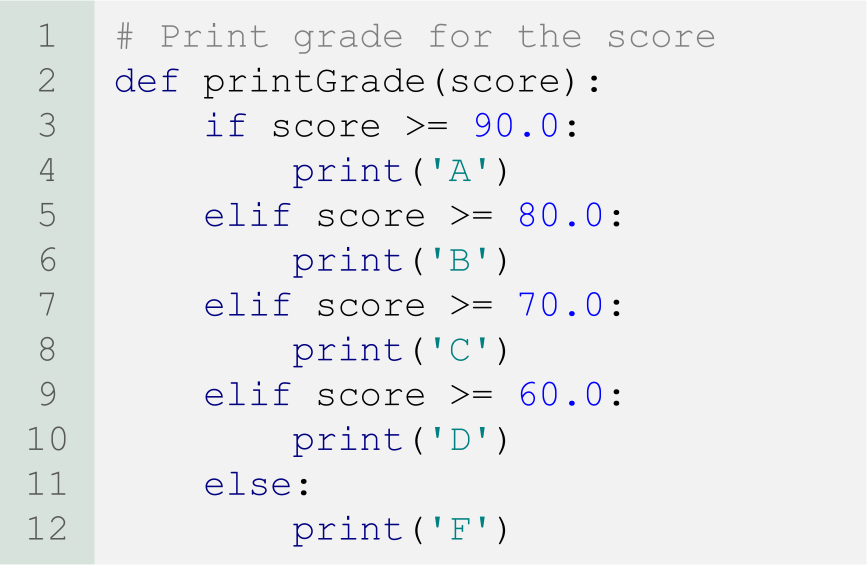
```
Enter a score: 91  <Enter>
The grade is A
```

```
Enter a score: 85  <Enter>
The grade is B
```

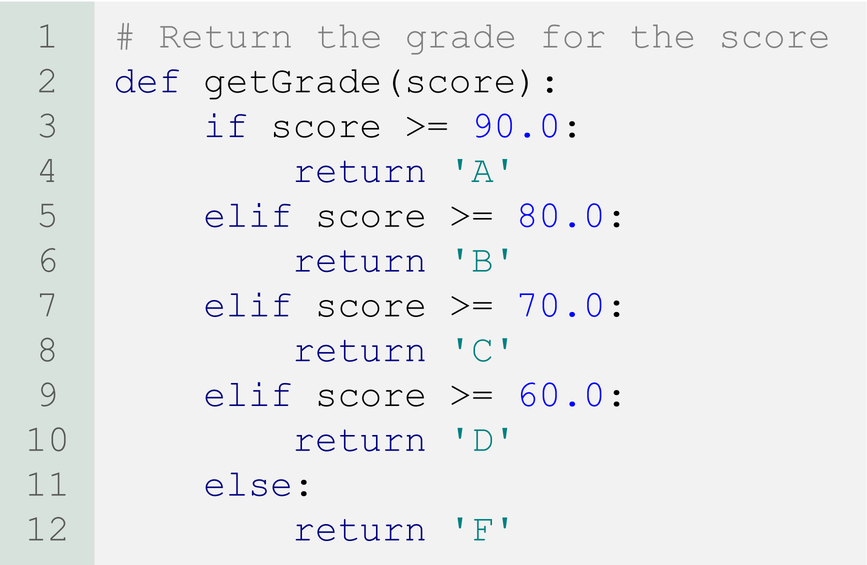
- The printGrade function does not return any value.
- So, it is invoked as a statement in line 17 in the main function.

Functions with Return Values

- To see the differences between a function that does not return a value and a function that returns a value, let's redesign the printGrade function (in Program 3) to return a value.



```
# Print grade for the score
def printGrade(score):
    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')
```

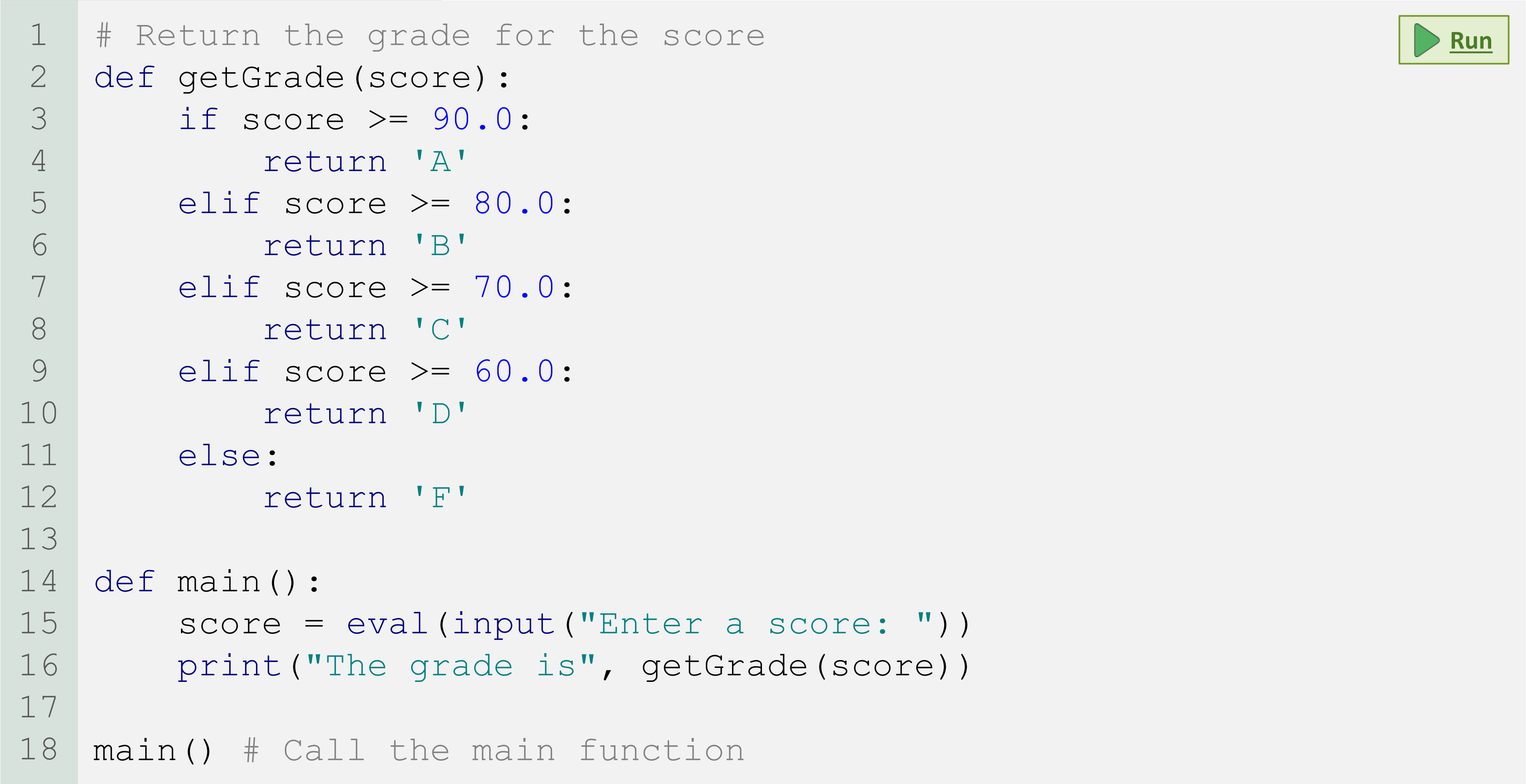


```
# Return the grade for the score
def getGrade(score):
    if score >= 90.0:
        return 'A'
    elif score >= 80.0:
        return 'B'
    elif score >= 70.0:
        return 'C'
    elif score >= 60.0:
        return 'D'
    else:
        return 'F'
```

- We call the new function that returns the grade, getGrade, as shown in following program (Program 4).

Testing getGrade Function
Program 4

LISTING 6.3 ReturnGradeFunction.py



```python
# Return the grade for the score
def getGrade(score):
    if score >= 90.0:
        return 'A'
    elif score >= 80.0:
        return 'B'
    elif score >= 70.0:
        return 'C'
    elif score >= 60.0:
        return 'D'
    else:
        return 'F'

def main():
    score = eval(input("Enter a score: "))
    print("The grade is", getGrade(score))

main() # Call the main function
```

Testing getGrade Function
Discussion

- Example runs of the program:

```
Enter a score: 66  <Enter>
The grade is D
```

```
Enter a score: 55  <Enter>
The grade is F
```

- The getGrade function defined in lines 2–12 returns a character grade based on the numeric score value.
  - It is invoked in line 16.
- The getGrade function returns a character, and it can be invoked and used just like a character.
- The printGrade function does not return a value, and it must be invoked as a statement.

None Value

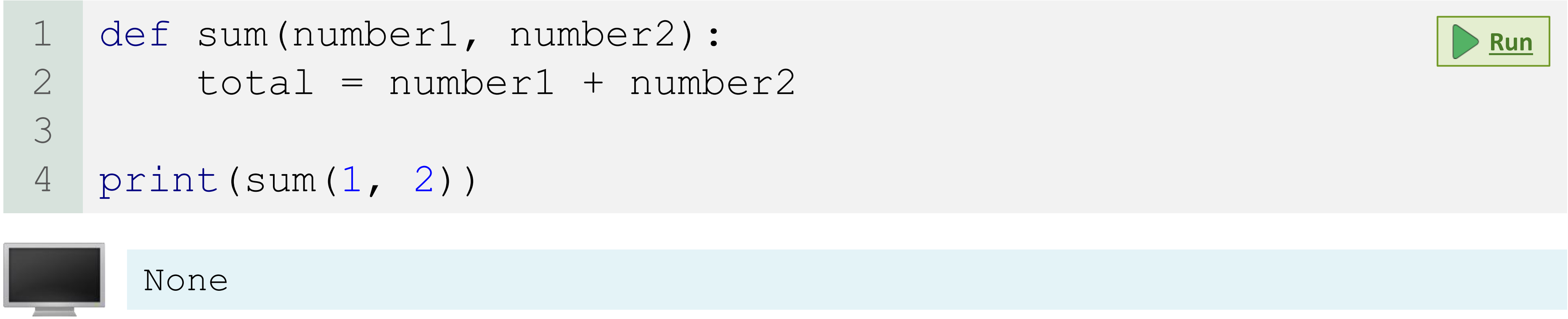
- Technically, every function in Python returns a value whether you use return or not.
- If a function does not return a value, by default, it returns a special value None.
  - For this reason, a function that does not return a value is also called a None function.
- The None value can be assigned to a variable to indicate that the variable does not reference any object (data).

Python

```
>>> x = None
>>> print(x)
None
>>> x == None
True
>>> x != None
False
```

None Value
Example

- For example, if you run the following program:



```python
def sum(number1, number2):
    total = number1 + number2

print(sum(1, 2))
```

```
None
```

- You will see the output is None, because the sum function does not have a **return statement**.
- By default, it returns None.

Terminating Void Functions

- A return statement is not needed for a None function (void).
- But it can be used for terminating the function and returning control to the function’s caller.
- The syntax is simply:

```
return
```

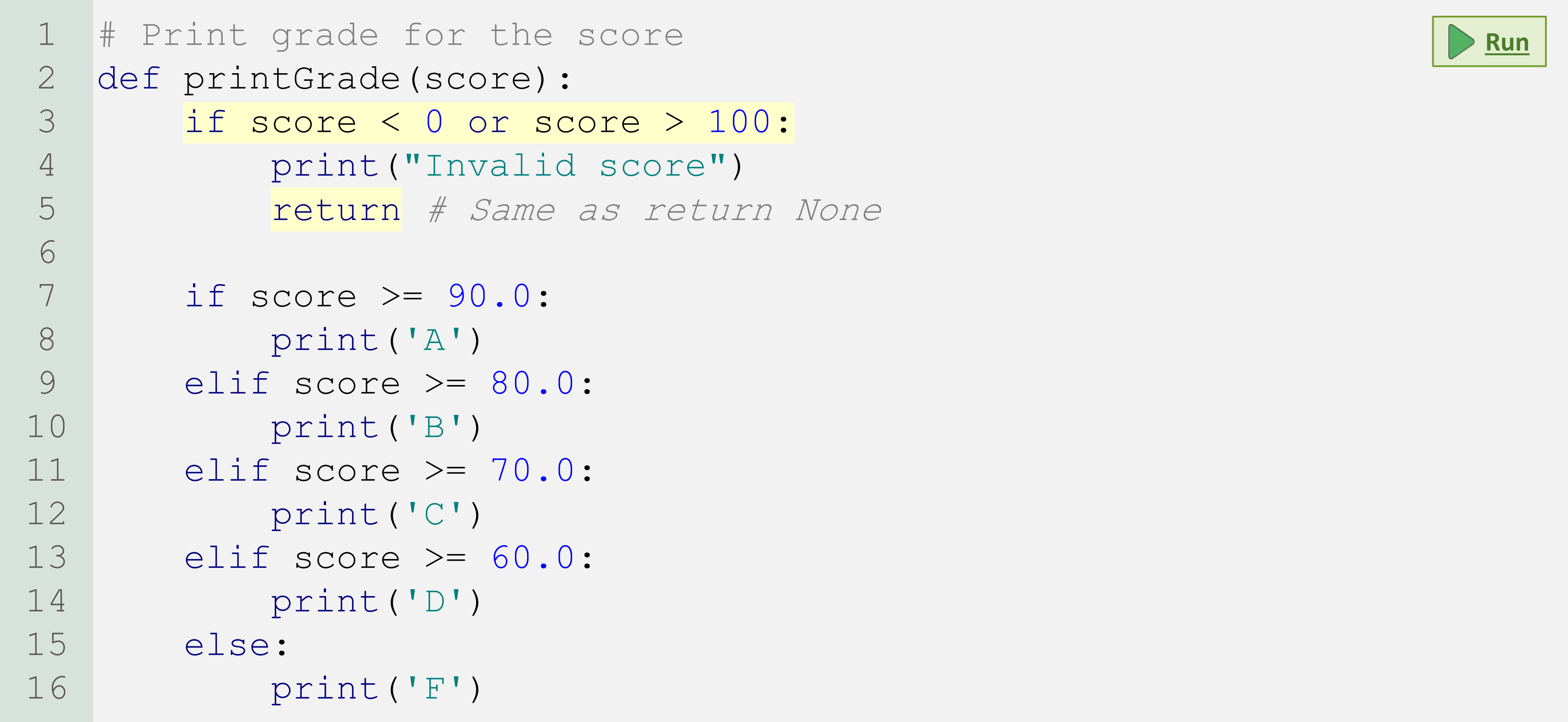
- Or

```
return None
```

- This is rarely used, but it is sometimes useful for circumventing (avoiding) the normal flow of control in a function that does not return any value.

Terminating Void Functions
Example

- For example, the following code has a return statement to terminate the function **when** the score is invalid.



```python
# Print grade for the score
def printGrade(score):
    if score < 0 or score > 100:
        print("Invalid score")
        return # Same as return None

    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')
```

Check Point #1

What are the benefits of using a function?

- Answer: At least three benefits:

1) Reuse code.
2) Reduce complexity.
3) Easy to maintain.

Check Point
#2

Can you simplify the max function by using a conditional expression?

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

- Solution:

```
def max(num1, num2):
    return num1 if num1 > num2 else num2
```

Check Point
#3

Can you have a return statement in a None function? Does the return statement in the following function cause syntax errors?

```
def xFunction(x, y):
    print(x + y)
    return
```

- Answer:
  - Yes, we can have a return statement in a None function.
  - No, the return statement in the previous function **does not** cause syntax errors.

Check Point
#4

Can a call to a value-returning function be a statement by itself?

- Answer:
  - Yes, it can.
  - But the returned value it will be ignored.

Check Point
#5

Write function headers for the following functions (and indicate whether the function returns a value):

- Computing a sales commission, given the sales amount and the commission rate.
  - ➢ getCommission(salesAmount, commissionRate)
  - ➢ The function returns a value.
- Printing the calendar for a month, given the month and year.
  - ➢ printCalendar(month, year)
  - ➢ The function does not return a value.
- Computing a square root.
  - ➢ sqrt(value)
  - ➢ The function returns a value.

Check Point
#5

Write function headers for the following functions (and indicate whether the function returns a value):

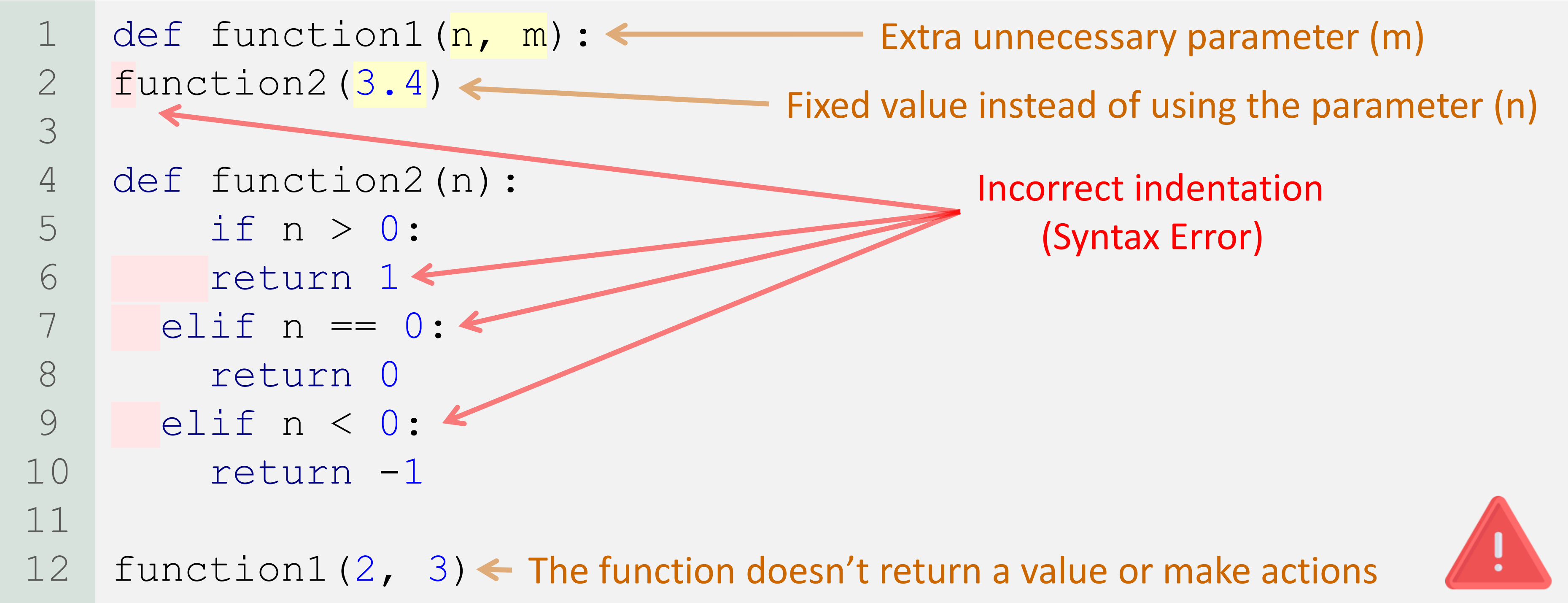
- Testing whether a number is even and returning true if it is.
  - ➢ isEven(value)
  - ➢ The function returns a value.

- Printing a message a specified number of times.
  - ➢ printMessage(message, times)
  - ➢ The function does not return a value.

- Computing the monthly payment, given the loan amount, number of years, and annual interest rate.
  - ➢ monthlyPayment(loan, numberOfYears, annualInterestRate)
  - ➢ The function returns a value.

Check Point
#6

Identify and correct the errors in the following program:



```
def function1(n, m):
function2(3.4)

def function2(n):
    if n > 0:
    return 1
  elif n == 0:
    return 0
  elif n < 0:
    return -1

function1(2, 3)
```

Extra unnecessary parameter (m)

Fixed value instead of using the parameter (n)

Incorrect indentation
(Syntax Error)

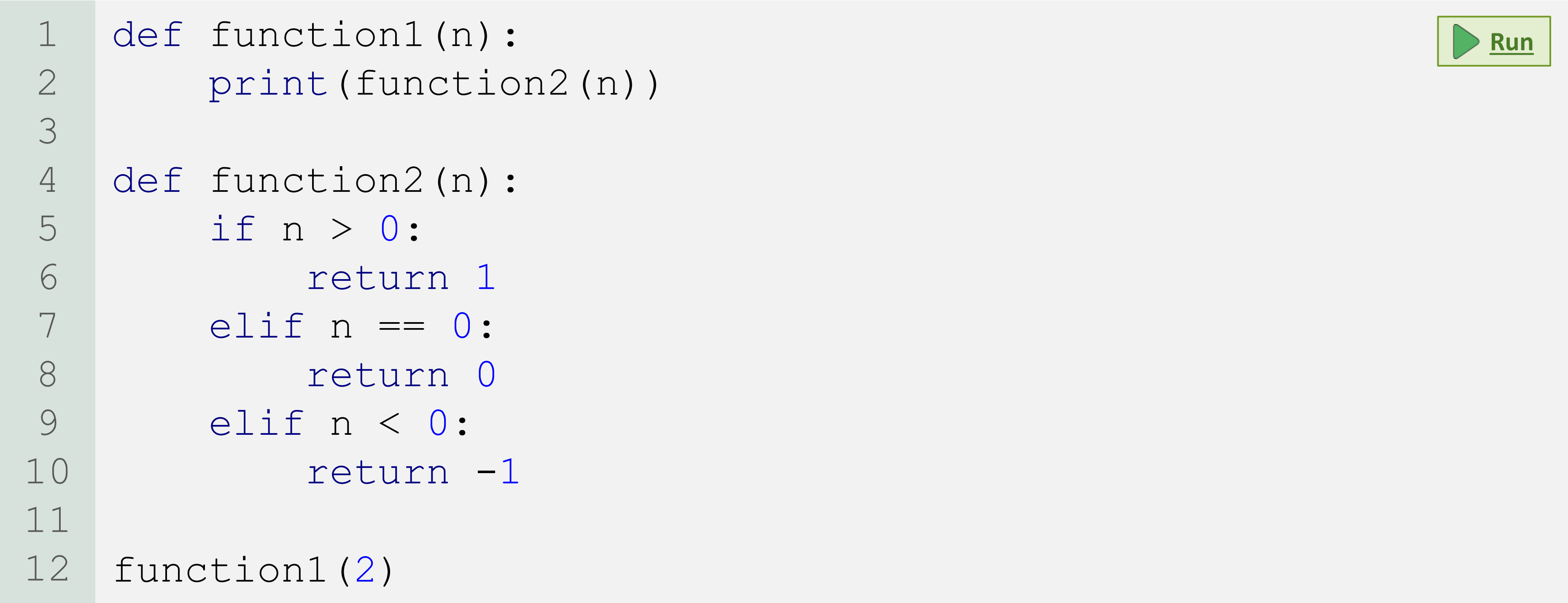
The function doesn't return a value or make actions

- Solution: the following slide has the corrected code.

Check Point
#6

Identify and correct the errors in the following program:

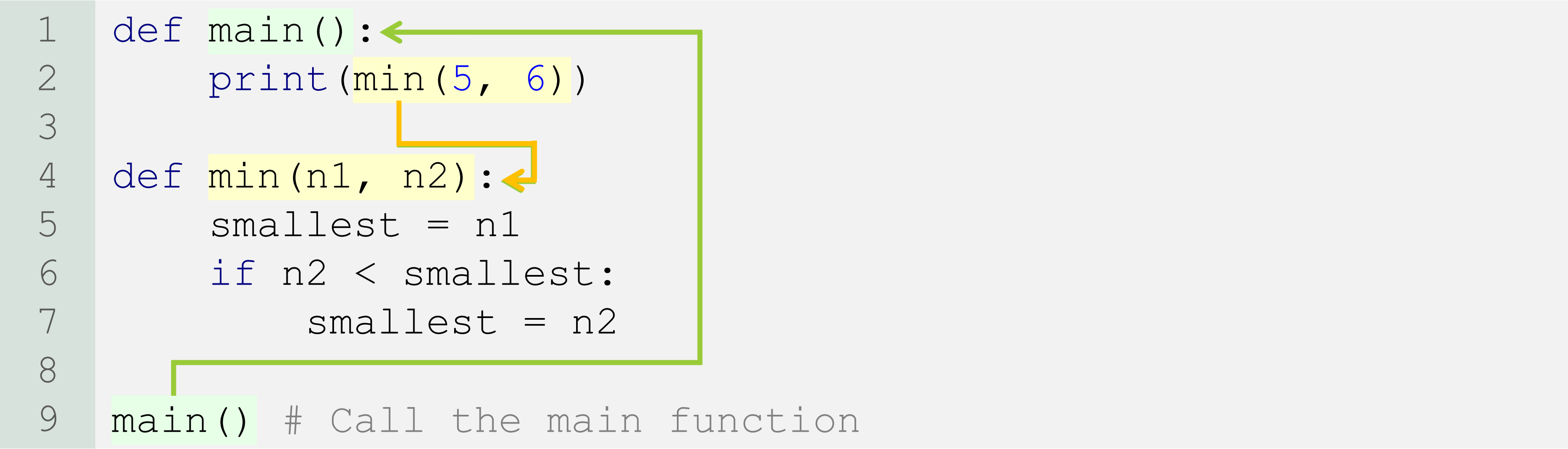
- Solution: the following code is the corrected code:



```python
def function1(n):
    print(function2(n))

def function2(n):
    if n > 0:
        return 1
    elif n == 0:
        return 0
    elif n < 0:
        return -1

function1(2)
```

Check Point
#7

Show the output of the following code:



```
def main():
    print(min(5, 6))

def min(n1, n2):
    smallest = n1
    if n2 < smallest:
        smallest = n2

main() # Call the main function
```

```
None
```

- Solution: the following slide has the corrected code.

Check Point
#7

Show the output of the following code:

- Solution: the following code is the corrected code.

```
def main():
    print(min(5, 6))

def min(n1, n2):
    smallest = n1
    if n2 < smallest:
        smallest = n2

    return smallest

main() # Call the main function
```

```
5
```

Check Point
#8

Show the output of the following code:

```
def main():
    print( min( min(5,  6) , min(51,  3) ) )

def min(n1, n2):
    smallest = n1
    if n2 < smallest:
        smallest = n2

    return smallest

main() # Call the main function
```

```
3
```

Check Point
#9

Show the output of the following code:

```
def printHi(name):
    message = "Hi " + name

def printHello(name):
    message = "Hello " + name
    print(message)

def getHello(name):
    return "Hello " + name

printHi("Omar")
getHello("Ali")
printHello("Ahmad")
print("#", getHello("Jamal"), "#")
```

```
Hello Ahmad
# Hello Jamal #
```

Check Point
#10

Show the output of the following code:

```python
def A():
    return 1
    print("A")
    return 2
def B():
    print("B")
    if not True:
        return 10
    else:
        return 3
    return 5
r = A()
r += B()
print(r)
```

```
B
4
```

6.5. Positional and Keyword Arguments

- Positional Arguments
- Keyword Arguments
- Mixing Keyword and Positional Arguments
- Check Point #11

Positional and Keyword Arguments

- The power of a function is its ability to work with parameters.
- When calling a function, you need to pass arguments to parameters.
- There are two kinds of arguments:
  - Positional arguments.
  - Keyword arguments.
- This means that a function’s arguments can be passed as positional arguments or keyword arguments.

Positional Arguments

- Using positional arguments requires that the arguments be passed in the same order as their respective parameters in the function header.
- Example, the following function prints a message n times:

```
def nPrintln(message, n):
    for i in range(n):
        print(message)
```

- You can use nPrintln('Ahmad', 3) to print Ahmad three times.
- The nPrintln('Ahmad', 3) statement:
  - Passes Ahmad to message.
  - Passes **3** to n.
  - Prints Ahmad three times.

Positional Arguments

- Example, the following function prints a message n times:

```python
def nPrintln(message, n):
    for i in range(n):
        print(message)
```

- However, the statement nPrintln(3, 'Ahmad') has a different meaning.
  - It passes **3** to message and Ahmad to n.
  - So, this will cause an error.
- When we call a function like this, it is said to use positional arguments.
  - The arguments must match the parameters in order, number, and compatible type, as defined in the function header.

Keyword Arguments

- Example, the following function prints a message n times:

```
def nPrintln(message, n):
    for i in range(n):
        print(message)
```

- You can also call a function using keyword arguments, passing each argument in the form name = value.
- For example, nPrintln(n = 5, message = "good")
  - Passes **5** to n.
  - Passes "good" to message.
- The arguments can appear in any order using keyword arguments.

Mixing Keyword and Positional Arguments

- It is possible to mix positional arguments with keyword arguments, **but** the positional arguments **cannot appear after** any keyword arguments.

- Suppose a function header is:

```
def f(p1, p2, p3):
```

- You can invoke it by using:

```
f(30, p2 = 4, p3 = 10)
```

- However, it would be wrong to invoke it by using:

```
f(30, p2 = 4, 10)
```

➢ Because the positional argument **10** **appears after** the keyword argument **p2 = 4**.

Check Point
#11

Suppose a function header is as follows:

```
def f(p1, p2, p3, p4):
```

Which of the following calls are correct?

- `f(1, p2 = 3, p3 = 4, p4 = 4)` **Correct** ✓
- `f(1, p2 = 3, 4, p4 = 4)` **Wrong** ✗
- `f(p1 = 1, p2 = 3, 4, p4 = 4)` **Wrong** ✗
- `f(p1 = 1, p2 = 3, p3 = 4, p4 = 4)` **Correct** ✓
- `f(p4 = 1, p2 = 3, p3 = 4, p1 = 4)` **Correct** ✓

6.6. Passing Arguments by Reference Values

- Passing Arguments By Values
- Check Point #12 - #13

Passing Arguments By Values

- For your information:
  - All data are objects in Python, a variable for an object is actually a reference to the object.
  - When you invoke a function with an argument, the reference value of the argument is passed/sent to the formal parameter inside the function.
  - This is referred to as pass-by-value.
- For simplicity, we say that if the argument is a variable, the **value** of the variable is passed to a parameter when invoking a function.
- If the variable is a number or a string, the variable **is not affected**, regardless of the changes made to the parameter inside the function.

Passing Arguments By Values
Example

▶ Run

```
 1  def main():
 2      x = 1
 3      print("Before the call, x is", x)
 4      increment(x)
 5      print("After the call, x is", x)
 6
 7  def increment(n):
 8      n += 1
 9      print("\tn inside the function is", n)
10
11  main() # Call the main function
```

```
Before the call, x is 1
       n inside the function is 2
After the call, x is 1
```

- As shown in the output, the value of x (**1**) is passed to the parameter n to invoke the increment function (line 4).
- The parameter n is incremented by **1** in the function (line 8), but x is not changed no matter what the function does.

Check Point
#12

Can the argument have the same name as its parameter?

- Answer: Yes, the actual parameter (argument) can have the same name as its formal parameter (parameter).

```python
def main():
    x = 1
    print("Before the call, x is", x)
    increment(x)
    print("After the call, x is", x)

def increment(x):
    x += 1
    print("\tx inside the function is", x)

main() # Call the main function
```

```
Before the call, x is 1
        x inside the function is 2
After the call, x is 1
```

Check Point
#13

Show the result of the following programs:

```python
def main():
    max = 0
    getMax(1, 2, max)
    print(max)

def getMax(value1, value2, max):
    if value1 > value2:
        max = value1
    else:
        max = value2

main()
```

(a)

```
0
```

Check Point
#13

Show the result of the following programs:

```
def main():
    i = 1
    while i <= 6:
        print(function1(i, 2))
        i += 1

def function1(i, num):
    line = ""
    for j in range(1, i):
        line += str(num) + " "
        num *= 2
    return line

main()
```

(b)

```

2
2 4
2 4 8
2 4 8 16
2 4 8 16 32
```

Check Point
#13

Show the result of the following programs:

```
def main():
    # Initialize times
    times = 3
    print("Before the call, variable",
    "times is", times)
    # Invoke nPrintln and display times
    nPrint("Welcome to CS!", times)
    print("After the call, variable",
    "times is", times)

# Print the message n times
def nPrint(message, n):
    while n > 0:
        print("n = ", n)
        print(message)
        n -= 1

main()
```

(c)

```
Before the call, variable times is 3
n =  3
Welcome to CS!
n =  2
Welcome to CS!
n =  1
Welcome to CS!
After the call, variable times is 3
```

Check Point
#13

Show the result of the following programs:

```
def main():
    i = 0
    while i <= 4:
        function1(i)
        i += 1
        print("i is", i)

def function1(i):
    line = " "
    while i >= 1:
        if i % 3 != 0:
            line += str(i) + " "
            i -= 1
    print(line)

main()
```

(d)

```
i is 1
 1
i is 2
 2 1
i is 3
(... infinite Loop)
```

6.7. Modularizing Code

- Program 5: Finding the GCD (Modularizing Code)
- Program 6: Prime Number (Modularizing Code)

Modularizing Code

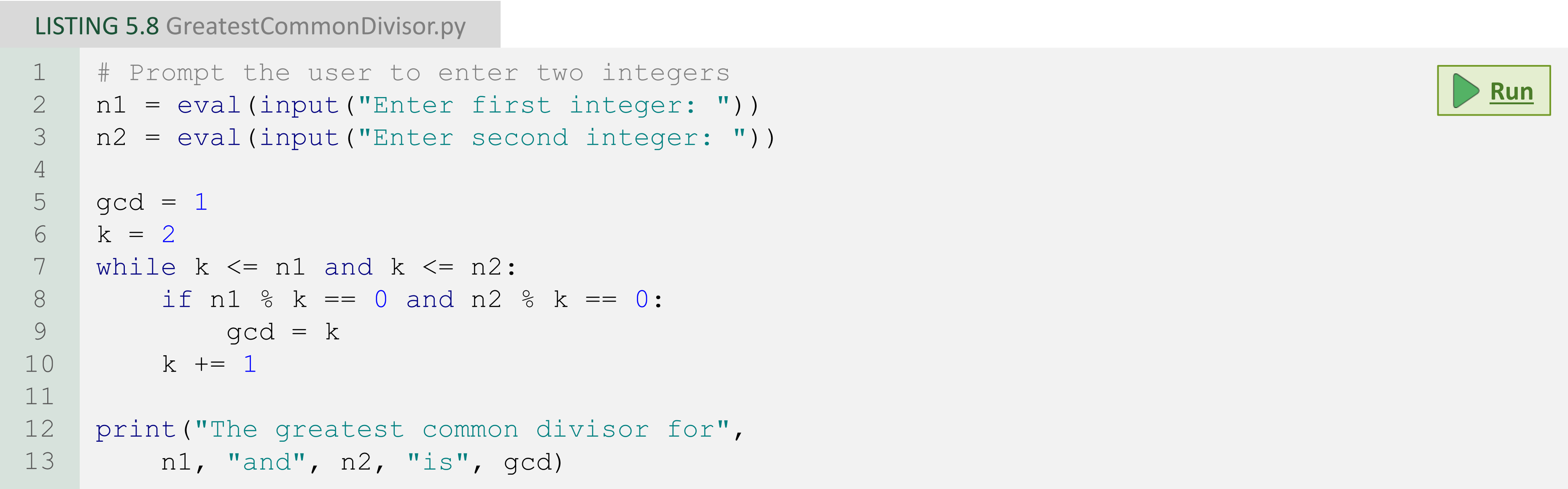
- What is the idea of modularizing code?
  - To answer this, let us ask another question: What is a module?
  - Answer: a sub-group of a larger entity.
  - For example, you have a Chapter in your book, and then inside the chapter, maybe you have 8 modules.
  - These are small, independent sections of the Chapter.
- Imagine if the chapter did not have modules, and you were told to "modularize the chapter".
  - This means, divide the chapter into modules!
- This same idea applies to code.

Modularizing Code

- New programmers often write long un-modularized code, which is very difficult to read.
- So we tell them: modularize the code!
- This makes the code easier:
  - To maintain
  - To read
  - To debug
  - and a best of all, it makes the code reusable!
- Use of functions:
  - We already learned that functions can be used to reduce redundant code and they facilitate reuse of code.
  - Functions are also used to modularize code and to help improve the overall quality of the program.

Finding the GCD (Modularizing Code)
Program 5

In Chapter 5, Program 7, we wrote a program to find the GCD of two integers.



LISTING 5.8 GreatestCommonDivisor.py

```
# Prompt the user to enter two integers
n1 = eval(input("Enter first integer: "))
n2 = eval(input("Enter second integer: "))

gcd = 1
k = 2
while k <= n1 and k <= n2:
    if n1 % k == 0 and n2 % k == 0:
        gcd = k
    k += 1

print("The greatest common divisor for",
    n1, "and", n2, "is", gcd)
```

Re-write the program in a modularized fashion by using a function to compute the GCD.

Finding the GCD (Modularizing Code)
Phase 1: Problem-solving

- First, let’s write a function that find and return the GCD of two numbers.
- The header of the new function can be as the following:

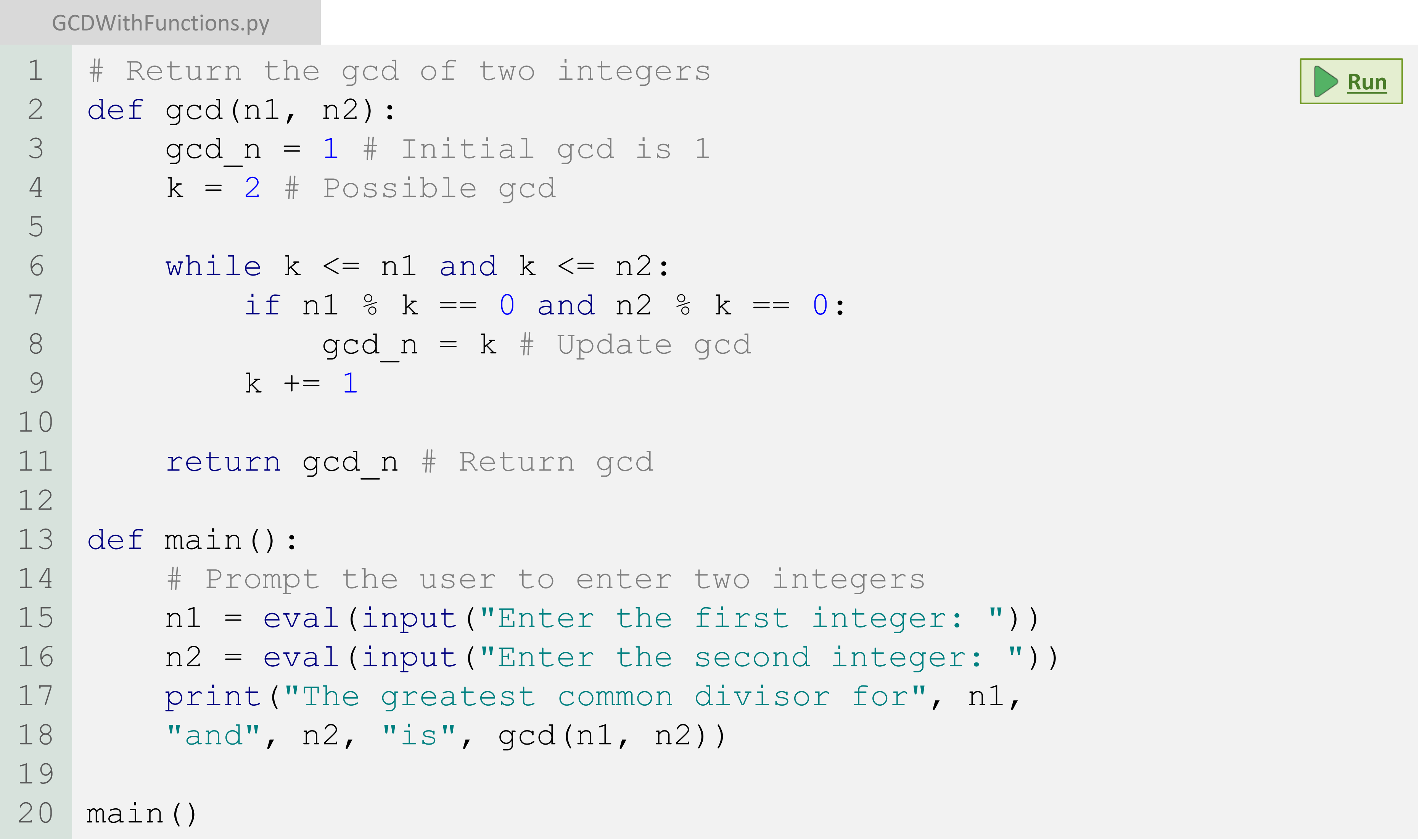
```
def gcd(n1, n2):
```

- Now, let’s implement the function:

```
# Return the gcd of two integers
def gcd(n1, n2):
    gcd = 1 # Initial gcd is 1
    k = 2 # Possible gcd

    while k <= n1 and k <= n2:
        if n1 % k == 0 and n2 % k == 0:
            gcd = k # Update gcd
        k += 1

    return gcd # Return gcd
```

Finding the GCD (Modularizing Code)

Phase 2: Implementation



GCDWithFunctions.py

```python
# Return the gcd of two integers
def gcd(n1, n2):
    gcd_n = 1 # Initial gcd is 1
    k = 2 # Possible gcd

    while k <= n1 and k <= n2:
        if n1 % k == 0 and n2 % k == 0:
            gcd_n = k # Update gcd
        k += 1

    return gcd_n # Return gcd

def main():
    # Prompt the user to enter two integers
    n1 = eval(input("Enter the first integer: "))
    n2 = eval(input("Enter the second integer: "))
    print("The greatest common divisor for", n1,
    "and", n2, "is", gcd(n1, n2))

main()
```

Finding the GCD (Modularizing Code)
Example Runs of The Program

```
Enter the first integer: 20  <Enter>
Enter the second integer: 90  <Enter>
The greatest common divisor for 20 and 90 is 10
```

```
Enter the first integer: 99  <Enter>
Enter the second integer: 13  <Enter>
The greatest common divisor for 99 and 13 is 1
```

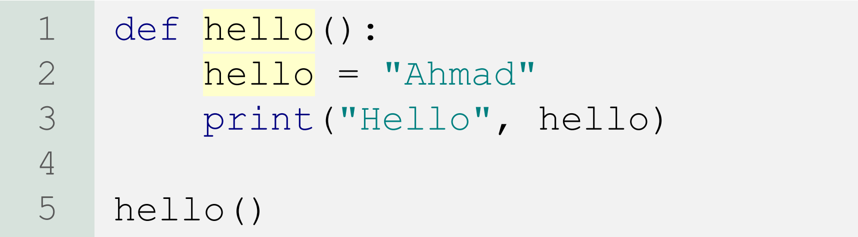
```
Enter the first integer: 12  <Enter>
Enter the second integer: 64  <Enter>
The greatest common divisor for 12 and 64 is 4
```

Note

What happens if you define a variable and a function with the same name?

- **Avoid** naming variables with the same name of functions or vice versa **to prevent conflicts**.

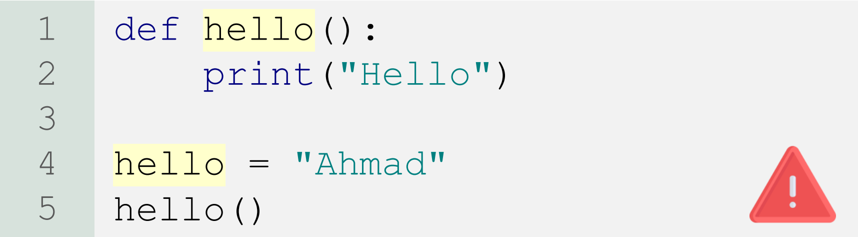
- While the following code is ok:



```python
def hello():
    hello = "Ahmad"
    print("Hello", hello)

hello()
```

```
Hello Ahmad
```

The following code cause a runtime error:



```python
def hello():
    print("Hello")

hello = "Ahmad"
hello()
```

```
 hello()
TypeError: 'str' object
is not callable
```

Remember
Python Is Case-sensitive

- Python is case-sensitive.
- For example, the following identifiers (names) are different in Python (not the same name):
  - hello
  - Hello
  - hEllo
  - helOO
  - hElOo
  - heloO
  - helOo
  - HELO

Prime Number (Modularizing Code)
Program 6

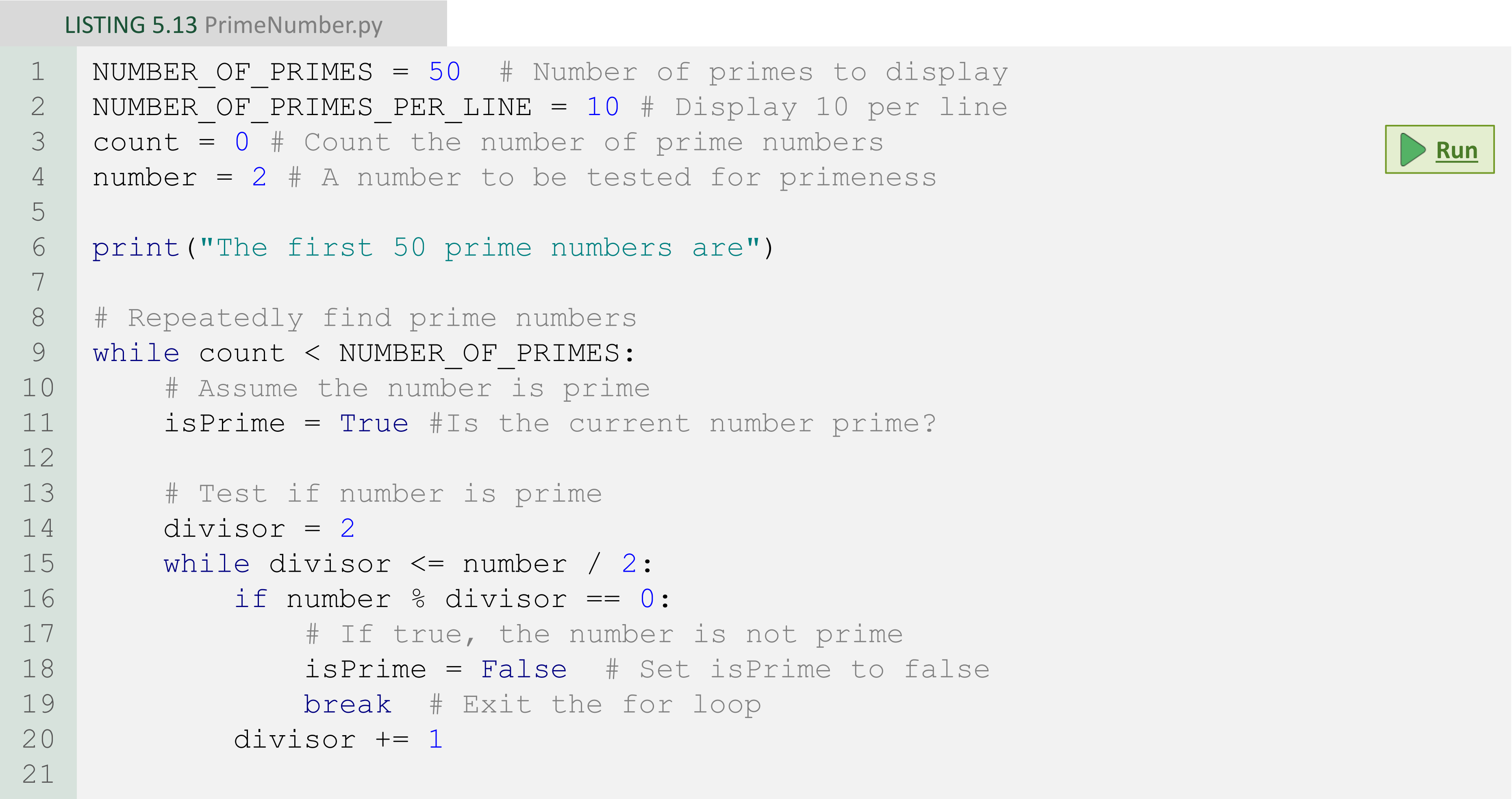
Write a modularized program, which should print the first 50 prime numbers, with ten numbers printed per line.

- Note: In Chapter 5, Program 9, we wrote this program. Re-write the program in a modularized fashion by using a functions.

```
The first 50 prime numbers are
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229
```

Prime Number (Modularizing Code)
Phase 1: Problem-solving

- Recover the Implementation of Program 9 In Chapter 5:



LISTING 5.13 PrimeNumber.py

```python
NUMBER_OF_PRIMES = 50  # Number of primes to display
NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
count = 0 # Count the number of prime numbers
number = 2 # A number to be tested for primeness

print("The first 50 prime numbers are")

# Repeatedly find prime numbers
while count < NUMBER_OF_PRIMES:
    # Assume the number is prime
    isPrime = True #Is the current number prime?

    # Test if number is prime
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            # If true, the number is not prime
            isPrime = False  # Set isPrime to false
            break  # Exit the for loop
        divisor += 1

```

Prime Number (Modularizing Code)
Phase 1: Problem-solving

- Recover the Implementation of Program 9 In Chapter 5:

LISTING 5.13 PrimeNumber.py

```
# If number is prime, display the prime number and increase the count
if isPrime:
    count += 1 # Increase the count

    print(format(number, '5d'), end = '')
    if count % NUMBER_OF_PRIMES_PER_LINE == 0:
        # Display the number and advance to the new line
        print() # Jump to the new line

# Check if the next number is prime
number += 1
```

Run

Prime Number (Modularizing Code)
Phase 1: Problem-solving

- We really have two things to consider for this program:
  1. We need to determine if a number is prime.
  2. We need to print 10 prime numbers per line.
- We can do both of these steps with functions.
- We can make a function, isPrime, to determine prime.
- Also, we can make another function that is used specifically to print the numbers, printPrimeNumbers.

Prime Number (Modularizing Code)
Phase 1: Problem-solving

Step 1: Determine if a number is prime (isPrime)

- In the previous Implementation, we showed how to determine if a number is prime:

```
# Assume the number is prime
    isPrime = True #Is the current number prime?

    # Test if number is prime
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            # If true, the number is not prime
            isPrime = False  # Set isPrime to false
            break  # Exit the for loop
        divisor += 1
```

Prime Number (Modularizing Code)
Phase 1: Problem-solving

Step 1: Determine if a number is prime (isPrime)

- The function isPrime can be implemented as the following:

```
# Check whether number is prime
def isPrime(number):
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            # If true, number is not prime
            return False # number is not a prime
        divisor += 1

    return True # number is prime
```

Prime Number (Modularizing Code)
Phase 1: Problem-solving

Step 2: Print 10 prime numbers per line (printPrimeNumbers)

- We can keep a constant: NUMBER_OF_PRIMES_PER_LINE.
- We keep a counter to count the number of primes found.
- We use the same while loop from Chapter 5.
  - `while (count < numberOfPrimes)`
  - We start with the number **2**, and check if it is prime using isPrime.
  - If so, we increment count.
  - Of course, we increment the tested value (number).
  - And we continue until we find the desired number of primes.

Prime Number (Modularizing Code)
Phase 1: Problem-solving

Step 2: Print 10 prime numbers per line (printPrimeNumbers)

- printPrimeNumbers can be implemented as the following:

```python
def printPrimeNumbers(numberOfPrimes):
    NUMBER_OF_PRIMES = 50 # Number of primes to display
    NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
    count = 0 # Count the number of prime numbers
    number = 2 # A number to be tested for primeness

    # Repeatedly find prime numbers
    while count < numberOfPrimes:
        # Print the prime number and increase the count
        if isPrime(number):
            count += 1 # Increase the count

            print(number, end = " ")
            if count % NUMBER_OF_PRIMES_PER_LINE == 0:
                # Print the number and advance to the new line
                print()

        # Check if the next number is prime
        number += 1
```

Prime Number (Modularizing Code)
Phase 2: Implementation

LISTING 6.7 PrimeNumberFunction.py

▶ Run

```
# Check whether number is prime
def isPrime(number):
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            # If true, number is not prime
            return False # number is not a prime
        divisor += 1

    return True # number is prime

def printPrimeNumbers(numberOfPrimes):
    NUMBER_OF_PRIMES = 50 # Number of primes to display
    NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
    count = 0 # Count the number of prime numbers
    number = 2 # A number to be tested for primeness

    # Repeatedly find prime numbers
```

Prime Number (Modularizing Code)
Phase 2: Implementation

LISTING 6.7 PrimeNumberFunction.py

Run

```
 while count < numberOfPrimes:
        # Print the prime number and increase the count
        if isPrime(number):
            count += 1 # Increase the count

            print(format(number, "4d"), end = " ")
            if count % NUMBER_OF_PRIMES_PER_LINE == 0:
                # Print the number and advance to the new line
                print()

        # Check if the next number is prime
        number += 1

def main():
    print("The first 50 prime numbers are")
    printPrimeNumbers(50)

main() # Call the main function
```

Prime Number (Modularizing Code)

Run of The Program

```
The first 50 prime numbers are
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229
```

Prime Number (Modularizing Code) Discussion

- This program divides a large problem into two subproblems.
- As a result, the new program is easier to read and easier to debug.
- Moreover, the functions printPrimeNumbers and isPrime can be reused by other programs.

Note

What happens if you define two functions with the same name?

- There is no syntax error in this case, but the latter function definition prevails.

- Example:

```python
def hello():
    print("Hello")

def hello():
    print("Hi")

hello()
```

```
Hi
```

6.9. The Scope of Variables

- Local Variables
- Global Variables
- Example #1 - #5
- global Keyword
- Example #6 - #8
- Check Point #14 - #16

Local Variables

- Reminder from Chapter 2, Section 2.5:
  - The scope of a variable is the part of the program where the variable can be referenced.
- A variable created inside a function is referred to as a local variable.
- Local variables can only be accessed inside a function.
- The scope of a local variable starts from its creation and continues to the end of the function that contains the variable.
- A parameter is a local variable.
  - A parameter is "defined" inside the function header.
  - This means the scope of parameters are for the entire function!

Global Variables

- In Python, you can also use global variables.
- Global variables are created outside all functions and are accessible to all functions in their scope.
- A global variable cannot be modified inside a function unless a global statement is used.
  - This is done by using global keyword.

Example 1

```
1  globalVar = 1 # Create a global variable
2  def f1():
3      localVar = 2 # Create a local variable
4      print(globalVar) # Print: 1
5      print(localVar) # Print: 2
6
7  f1()
8  print(globalVar) # Print: 1
9  print(localVar) # Out of scope, so this gives an error
```

The Scope of localVar

The Scope of globalVar

Run

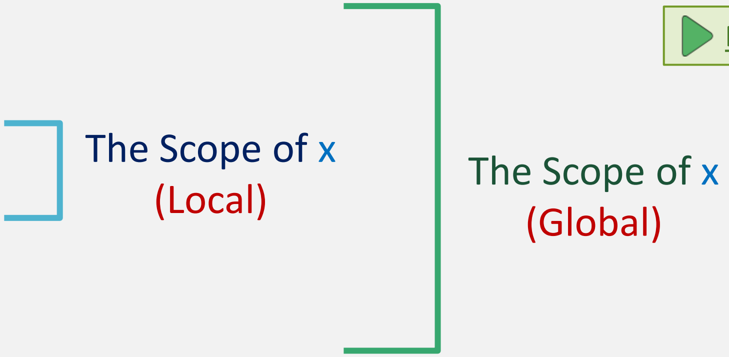
```
1
2
1
 print(localVar) # Out of scope, so this gives an error
NameError: name 'localVar' is not defined
```

- A global variable is created in line 1.
- It is accessed within the function in line 4 and outside the function in line 8.
- A local variable is created in line 3.
- It is accessed within the function in line 5.
- Attempting to access the variable from outside of the function causes an error in line 9.

Example 2

```
1  x = 1 # Create a global variable
2  def f1():
3      x = 2 # create a local variable
4      print(x) # Print: 2
5
6  f1()
7  print(x) # Print: 1
```

Run


The Scope of x (Local)

The Scope of x (Global)

```
2
1
```

- Here a global variable x is created in line 1 and a local variable with the same name (x) is created in line 3.
- From this point on, the global variable x is not accessible in the function.
- Outside the function, the global variable x is still accessible.
- So, it prints 1 in line 7.

Example 3

```
1  x = eval(input("Enter a number: "))
2  if (x > 0):
3      y = 4
4  print(y) # This gives an error if y is not created
```

The Scope of y (If it is not executed)

The Scope of y (If it is executed)

The Scope of x

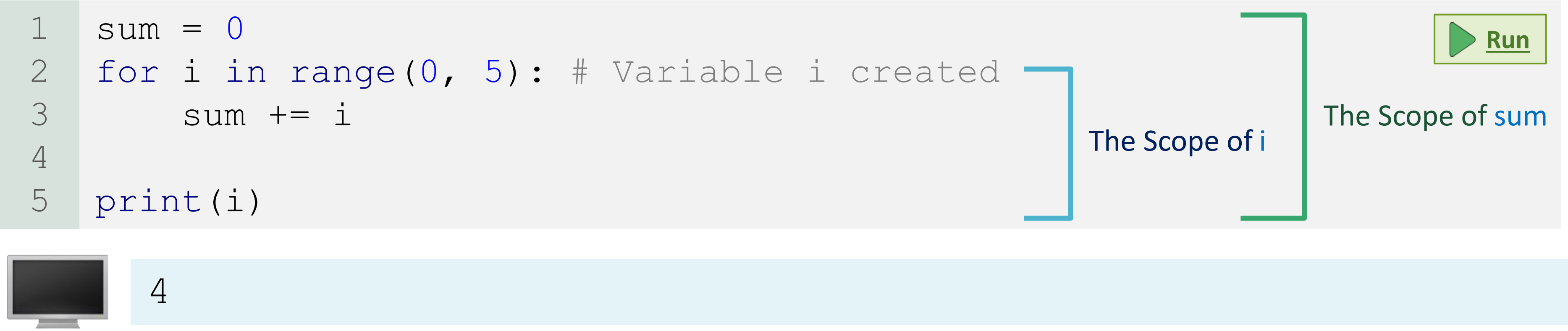
Run

```
Enter a number: 1   <Enter>
4
```

```
Enter a number: 0   <Enter>
 print(y) # This gives an error if y is not created
NameError: name 'y' is not defined
```

- Here the variable y is created if x > 0.
- If you enter a positive value for x (line 1), the program runs fine.
- But if you enter a nonpositive value, line 5 produces an error because y is not created.

Example 4



```
1  sum = 0
2  for i in range(0, 5): # Variable i created
3      sum += i
4
5  print(i)
```

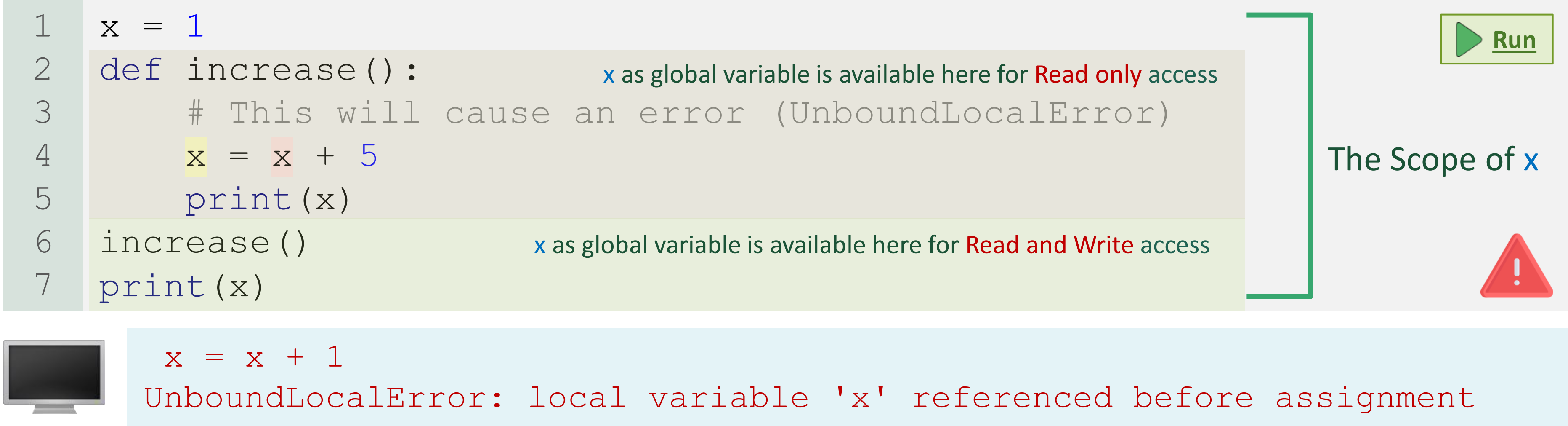
The Scope of i

The Scope of sum

```
4
```

- Here the variable **i** is created in the loop.
- After the loop is finished, **i** is **4**, so line 5 displays **4**.

Example 5



```
1 x = 1
2 def increase():
3     # This will cause an error (UnboundLocalError)
4     x = x + 5
5     print(x)
6 increase()
7 print(x)
```

x as global variable is available here for Read only access

x as global variable is available here for Read and Write access

```
  x = x + 1
UnboundLocalError: local variable 'x' referenced before assignment
```

- In line 1, x is created as global variable (created outside functions).
- Inside the increase function, in line 4, x is modified (incremented by 1).
- However, this will cause an error. **Why?**
- This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope.
- Line 4 implicitly makes x local to the increase function, so trying to execute this line, though, will try to read the value of the local variable x before it is assigned, resulting in an UnboundLocalError.
- Solution: use global keyword. (See next examples)

global Keyword

- In Python, global keyword allows you to:
  - Modify a global variable from a local context (inside a function).
    - In other words, you can bind a local variable in the global scope.
  - Create global variables from a local context (inside a function).
    - In other words, you can create a variable in a function and use it outside the function.
- The basic rules for global keyword in Python are:
  - When we create a variable inside a function, it’s local by default.
  - When we define a variable outside of a function, it’s global by default.
    - ➢ You **don’t** have to use global keyword.
  - We use global keyword to read and write a global variable inside a function.
  - Use of global keyword outside a function has no effect.

Example 6

```
1  x = 1
2  def increase():
3      global x # Now x is available for read and write access
4      x = x + 5
5      print(x) # Print: 6
6
7  increase()
8  print(x) # Print: 6
```

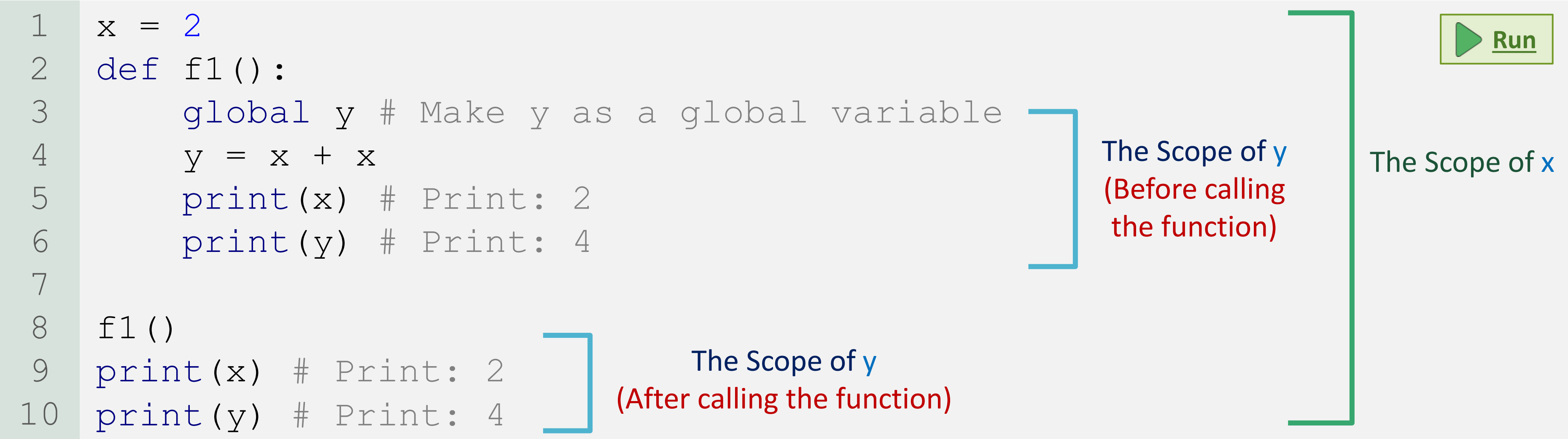
▶ Run

The Scope of x

```
6
6
```

- Here a global variable x is created in line 1 and x is bound in the function in line 3.
- This means that x in the function is the same as x outside of the function, so the program prints **2** in line 5 and in line 8.

Example 7



```
 1  x = 2
 2  def f1():
 3      global y # Make y as a global variable
 4      y = x + x
 5      print(x) # Print: 2
 6      print(y) # Print: 4
 7
 8  f1()
 9  print(x) # Print: 2
10  print(y) # Print: 4
```

Run

The Scope of y (Before calling the function)

The Scope of y (After calling the function)

The Scope of x

```
2
4
2
4
```

- Line 3 creates a global variable y inside the f1 function (local context) by using a global statement.
- y will be available for use as a global variable after executing the f1 function (Line 8).

Example 8

```
1  x = 2
2  def f1():
3      global y # Make y as a global variable
4      y = x + x
5      print(x) # Displays: 2
6      print(y) # Displays: 4
7  
8  print(x) # Displays: 2
9  print(y) # Causes an error (NameError)
10 f1()
```

The Scope of y (Before calling the function)

The Scope of x

The Scope of y (After calling the function)

```
2
 print(y)
NameError: name 'y' is not defined
```

- Line 3 creates a global variable y inside the f1 function (local context) by using a global statement.
- y will be available for use as a global variable after executing the f1 function (Line 10).
- This means that y in Line 9 is **not** existing yet (not defined yet), resulting in a NameError.

Caution

- Although global variables are allowed and you may see global variables used in other programs, it is **not** a good practice to allow them to be modified in a function.
- Because doing so can make programs prone to errors.
- However, it is fine to define global constants so all functions in the module can share them.

Check Point
#14

What is the printout of the following code?

```python
def function(x):
    print(x)
    x = 4.5
    y = 3.4
    print(y)

x = 2
y = 4
function(x)
print(x)
print(y)
```

(a)

```python
def f(x, y = 1, z = 2):
    return x + y + z

print(f(1, 1, 1))
print(f(y = 1, x = 2, z = 3))
print(f(1, z = 3))
```

(b)

```
2
3.4
2
4
```

```
3
6
5
```

Check Point
#15

What is wrong in the following code?

```python
def function():
    x = 4.5
    y = 3.4
    print(x)
    print(y)

function()
print(x)
print(y)
```

The Scope of y

The Scope of x

- Answer:
  - x an y are local variables, and their scopes start from their creation and continue to the end of the function.
  - So x and y are not existing (not defined) outside the function.

Check Point
#16

Can the following code run? If so, what is the printout?

```
x = 10

if x < 0:
    y = -1
else:
    y = 1

print("y is", y)
```

- Answer:
  - Yes, the code is correct. It has not a runtime error because the y variable is going to be defined in all cases after the if statement.

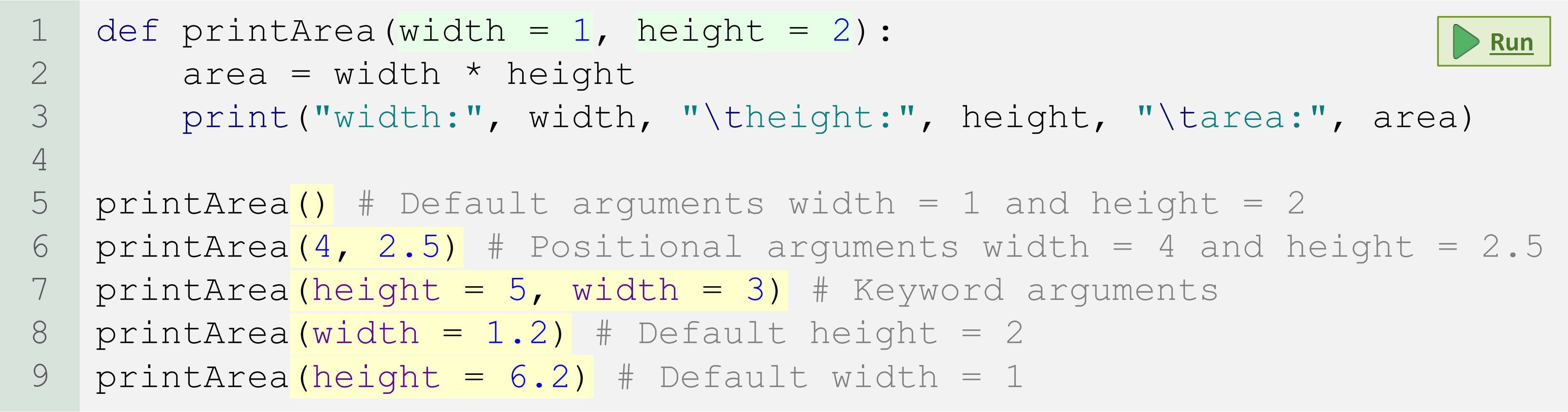
```
y is 1
```

6.10. Default Arguments

- Check Point #17 - #19

Default Arguments

- Python allows you to define functions with default argument values.
- The default values are passed to the parameters when a function is invoked without the arguments.
- The default value is assigned by using assignment (=) operator of the form parameterName = value. For example:



```
def printArea(width = 1, height = 2):
    area = width * height
    print("width:", width, "\theight:", height, "\tarea:", area)

printArea() # Default arguments width = 1 and height = 2
printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
printArea(height = 5, width = 3) # Keyword arguments
printArea(width = 1.2) # Default height = 2
printArea(height = 6.2) # Default width = 1
```

Default Arguments
Example

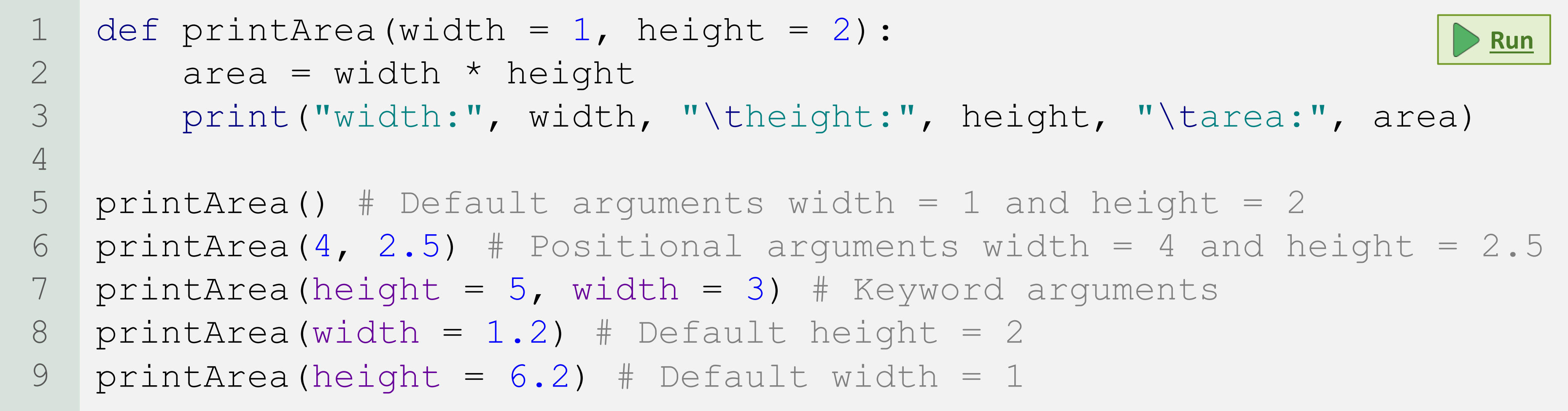
▶ Run

```python
def printArea(width = 1, height = 2):
    area = width * height
    print("width:", width, "\theight:", height, "\tarea:", area)

printArea() # Default arguments width = 1 and height = 2
printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
printArea(height = 5, width = 3) # Keyword arguments
printArea(width = 1.2) # Default height = 2
printArea(height = 6.2) # Default width = 1
```

```
width: 1      height: 2     area: 2
width: 4      height: 2.5   area: 10.0
width: 3      height: 5     area: 15
width: 1.2    height: 2     area: 2.4
width: 1      height: 6.2   area: 6.2
```

Default Arguments

Example



```
1  def printArea(width = 1, height = 2):
2      area = width * height
3      print("width:", width, "\theight:", height, "\tarea:", area)
4
5  printArea() # Default arguments width = 1 and height = 2
6  printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
7  printArea(height = 5, width = 3) # Keyword arguments
8  printArea(width = 1.2) # Default height = 2
9  printArea(height = 6.2) # Default width = 1
```

- Line 1 defines the printArea function with the parameters width and height.
- Width has the default value 1 and height has the default value 2.
- Line 5 invokes the function without passing an argument, so the program uses the default value 1 assigned to width and 2 to height.
- Line 6 invokes the function by passing 4 to width and 2.5 to height.
- Line 7 invokes the function by passing 3 to width and 5 to height.
- Note that you can also pass the argument by specifying the parameter name, as shown in lines 8 and 9.

Note

- A function may mix parameters with default arguments and non-default arguments.
- In this case, the non-default parameters must be defined before default parameters.
- Example:

```
def printInfo(name, age = 25, city = "Jeddah"):
    print("Name:", name, "Age:", age, "City:", city)

printInfo("Ahmad") # Displays: Name: Ahmad Age: 25 City: Jeddah
```

- The following code has a syntax error because the non-default parameters **are not** defined before default parameters:

```
def printInfo(age = 25, name, city = "Jeddah"):
    print("Name:", name, "Age:", age, "City:", city)
```

Note

- Many programming languages support a useful feature that allows you to define two functions with the same name in a module, but it is not supported in Python.
- With default arguments, you can define a function once, and call the function in many different ways.
- This achieves the same effect as defining multiple functions with the same name in other programming languages.

```python
def printInfo(name, age = 0):
    if age > 0:
        print("Name:", name, " # Age:", age)
    else:
        print("Hello", name)

printInfo("Ahmad")
printInfo("Jamal", 23)
```

```
Hello Ahmad
Name: Jamal # Age: 23
```

Check Point
#17

Show the printout of the following code:

```
def f(w = 1, h = 2):
    print(w, h)

f()
f(w = 5)
f(h = 24)
f(4, 5)
```

- Solution:

```
1 2
5 2
1 24
4 5
```

Check Point
#18

Identify and correct the errors in the following program:

```
1  def main():
2      nPrintln(5)
3
4  def nPrintln(message = "Welcome to Python!", n):
5      for i in range(n):
6          print(message)
7
8  main() # Call the main function
```

- Answer: Line 4 has a syntax error because a non-default argument (n) follows a default argument (message). To correct the error:

```
1  def main():
2      nPrintln(5)
3
4  def nPrintln(n, message = "Welcome to Python!"):
5      for i in range(n):
6          print(message)
7
8  main() # Call the main function
```

Check Point
#19

What happens if you define two functions in a module that have the same name?

- Answer: There is no syntax error in this case, but the later definition replaces the previous definitions.

- Example:

```python
def hello():
    print("Hello")

def hello(name = "Ahmad"):
    print("Hi", name)

hello()
```

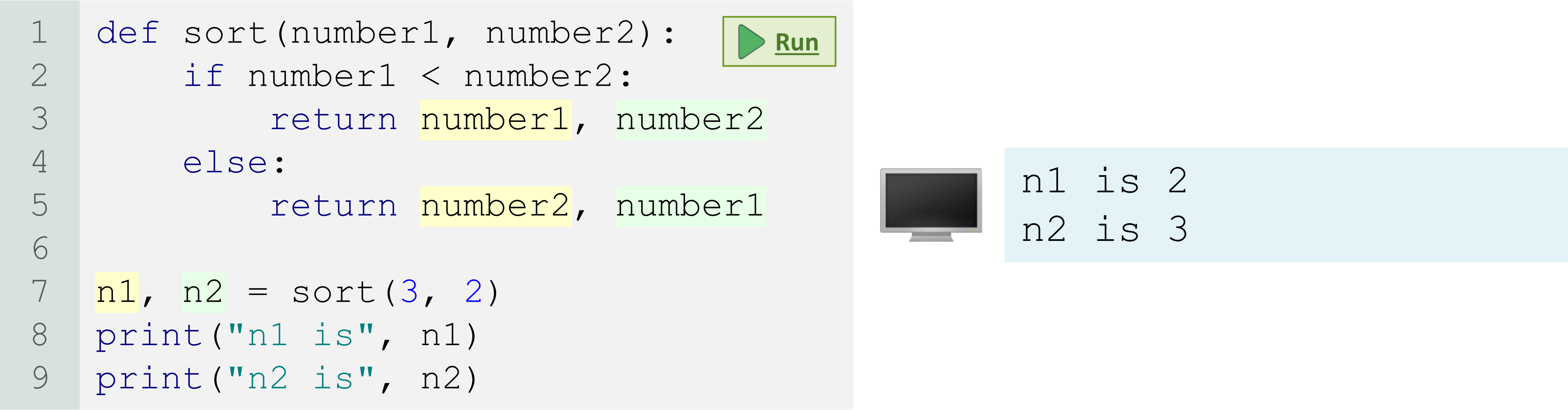
```
Hi Ahmad
```

6.11. Returning Multiple Values

- Check Point #20

Returning Multiple Values

- The Python return statement can return multiple values.
  - ➢ This means that Python allows a function to return multiple values.
- The following example defines a function that takes two numbers and returns them in ascending order:



```
def sort(number1, number2):
    if number1 < number2:
        return number1, number2
    else:
        return number2, number1

n1, n2 = sort(3, 2)
print("n1 is", n1)
print("n2 is", n2)
```

```
n1 is 2
n2 is 3
```

  - ○ The sort function returns two values. When it is invoked, you need to pass the returned values in a simultaneous assignment.

Check Point
#20

Show the printout of the following code:

```
def f(x, y):
    return x + y, x - y, x * y, x / y

t1, t2, t3, t4 = f(9, 5)
print(t1, t2, t3, t4)
```

- Solution:

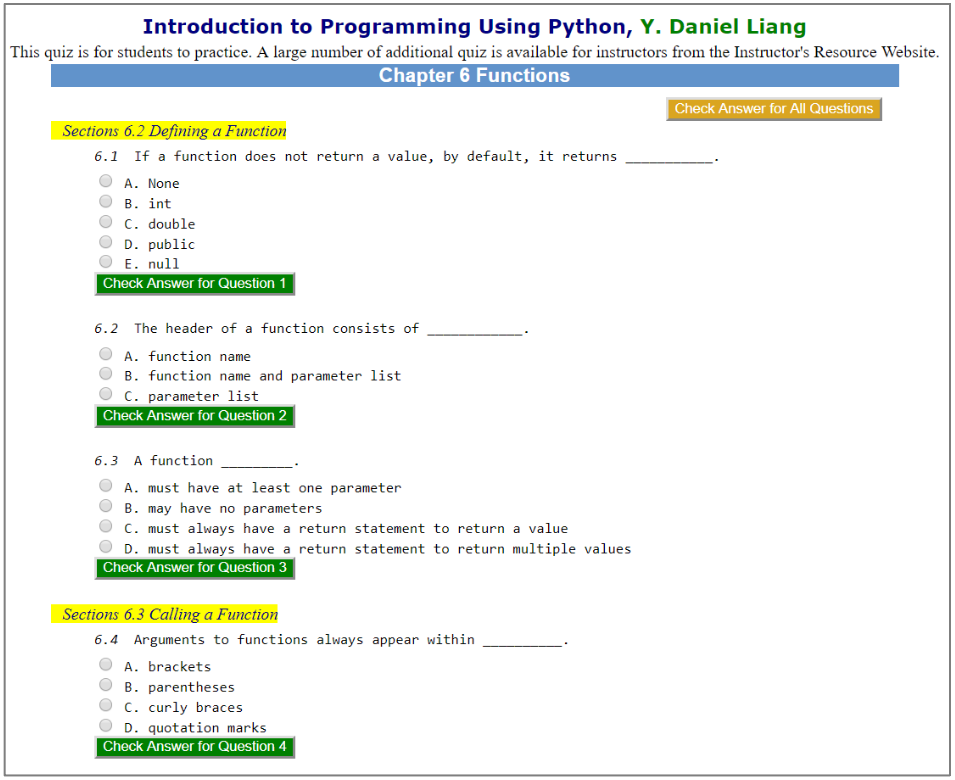
```
14 4 45 1.8
```

End

- Test Questions
- Programming Exercises

Test Questions

- Do the test questions for this chapter online at https://liveexample-ppe.pearsoncmg.com/selftest/selftestpy?chapter=6



Introduction to Programming Using Python, Y. Daniel Liang

This quiz is for students to practice. A large number of additional quiz is available for instructors from the Instructor's Resource Website.

Chapter 6 Functions

Sections 6.2 Defining a Function

6.1 If a function does not return a value, by default, it returns __________.

○ A. None
○ B. int
○ C. double
○ D. public
○ E. null

6.2 The header of a function consists of ____________.

○ A. function name
○ B. function name and parameter list
○ C. parameter list

6.3 A function _________.

○ A. must have at least one parameter
○ B. may have no parameters
○ C. must always have a return statement to return a value
○ D. must always have a return statement to return multiple values

Sections 6.3 Calling a Function

6.4 Arguments to functions always appear within __________.

○ A. brackets
○ B. parentheses
○ C. curly braces
○ D. quotation marks

Programming Exercises

- Page 203 – 212:
  - 6.1 – 6.11
  - 6.13 – 6.34
- Lab #9