# Functions

## Reading Assignment

- Chapter 5 Sections 1, 2, 3, 4, 5 and 8.

# Chapter Learning Outcomes

### At the end of this chapter, you will be able to

- implement functions
- become familiar with the concept of parameter passing
- develop strategies for decomposing complex tasks into simpler ones
- determine the scope of a variable

# Functions as Black Boxes

- A **function** is a sequence of instructions with a name.

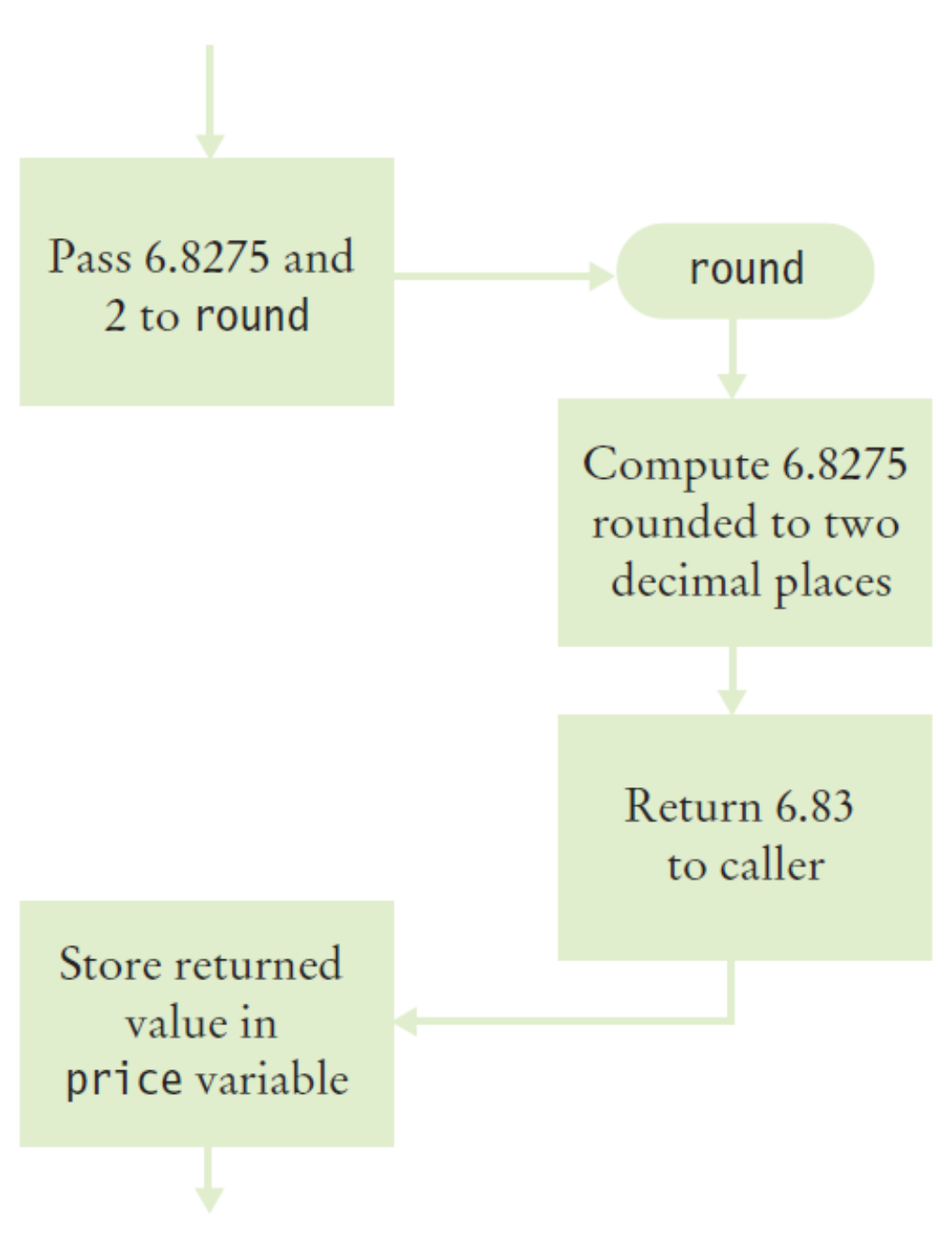
- For example, the **round** function, contains instructions to round a floating point value to a specified number of decimal places.
- You **call** a function in order to execute its instruction.

In [ ]:
```
price = round(6.8275,2) # Sets results to 6.83
print("Price:",price)
```

# Functions as Black Boxes

- By using the expression **round(6.8275,2)**, your program **calls** the **round** function, asking it to round 6.8275 to two decimal digits.
- The instructions of the round function execute and compute the result.
- The round function returns its result back to where the function was called and your program resumes execution

# Functions as Black Boxes


Pass 6.8275 and 2 to round
round
Compute 6.8275 rounded to two decimal places
Return 6.83 to caller
Store returned value in price variable

# Functions as Black Boxes

- When another function calls the round function, it provides **“inputs”**, such as the values 6.8275 and 2 in the call round(6.8275, 2).
- These values are called the **arguments** of the function call.
  - Note that they are not necessarily inputs provided by a human user.
  - They are simply the values for which we want the function to compute a result.

- The **“output”** that the round function computes is called the **return value**.

# Functions as Black Boxes

- Functions can receive multiple arguments, but they return only one value.
- It is also possible to have functions with no arguments.
- An example is the **random** function that requires no argument to produce a random number

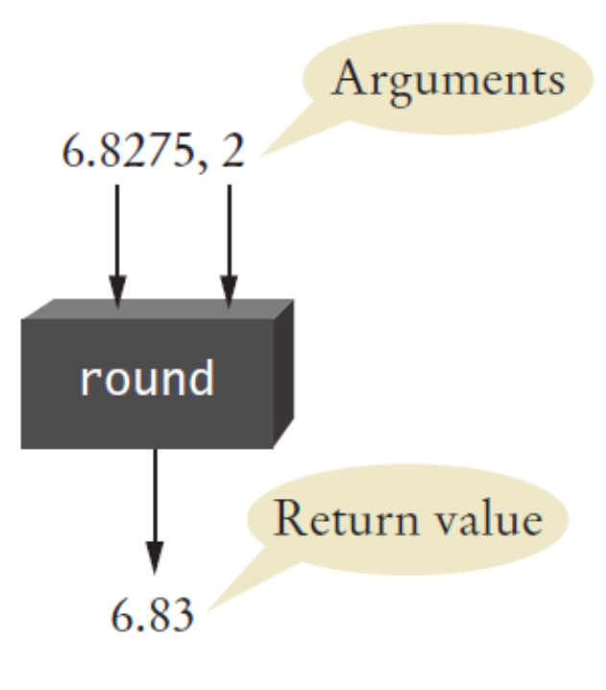
- At this point, you may wonder how the round function performs its job.
- How does round compute that 6.8275 rounded to two decimal digits is 6.83?

In [ ]:
```
price = round(6.8275,2)
print("Price:",price)
```

# Functions as Black Boxes

- Fortunately, as a user of the function, you do not need to know how the function is implemented.

- You just need to know the specification of the function:
  - If you provide arguments x and n, the function returns x rounded to n decimal digits.



- We can think of round as a black box.

# Functions as Black Boxes

- When you design your own functions, you will want to make them appear as black boxes to other programmers.
- Those programmers want to use your functions without knowing what goes on inside.
- Even if you are the only person working on a program, making each function into a black box pays off: there are fewer details that you need to keep in mind.

# Implementing and Testing Functions

## Implementing a Function

- When defining a **function**, you provide a **name** for the function and a **variable** for each **argument**.

- Let us start with a very simple example:
  - a function to compute the volume of a cube with a given side length.

© studioaraminta/iStockphoto.

# Implementing and Testing Functions

- When writing this function, you need to
  - Pick a **name** for the function (cubeVolume)
  - Define a variable for each **argument** (sideLength). These variables are called the **parameter variables**.
- Put all this information together along with the **def** reserved word to form the first line of the function's definition:
  - **def cubeVolume(sideLength):**

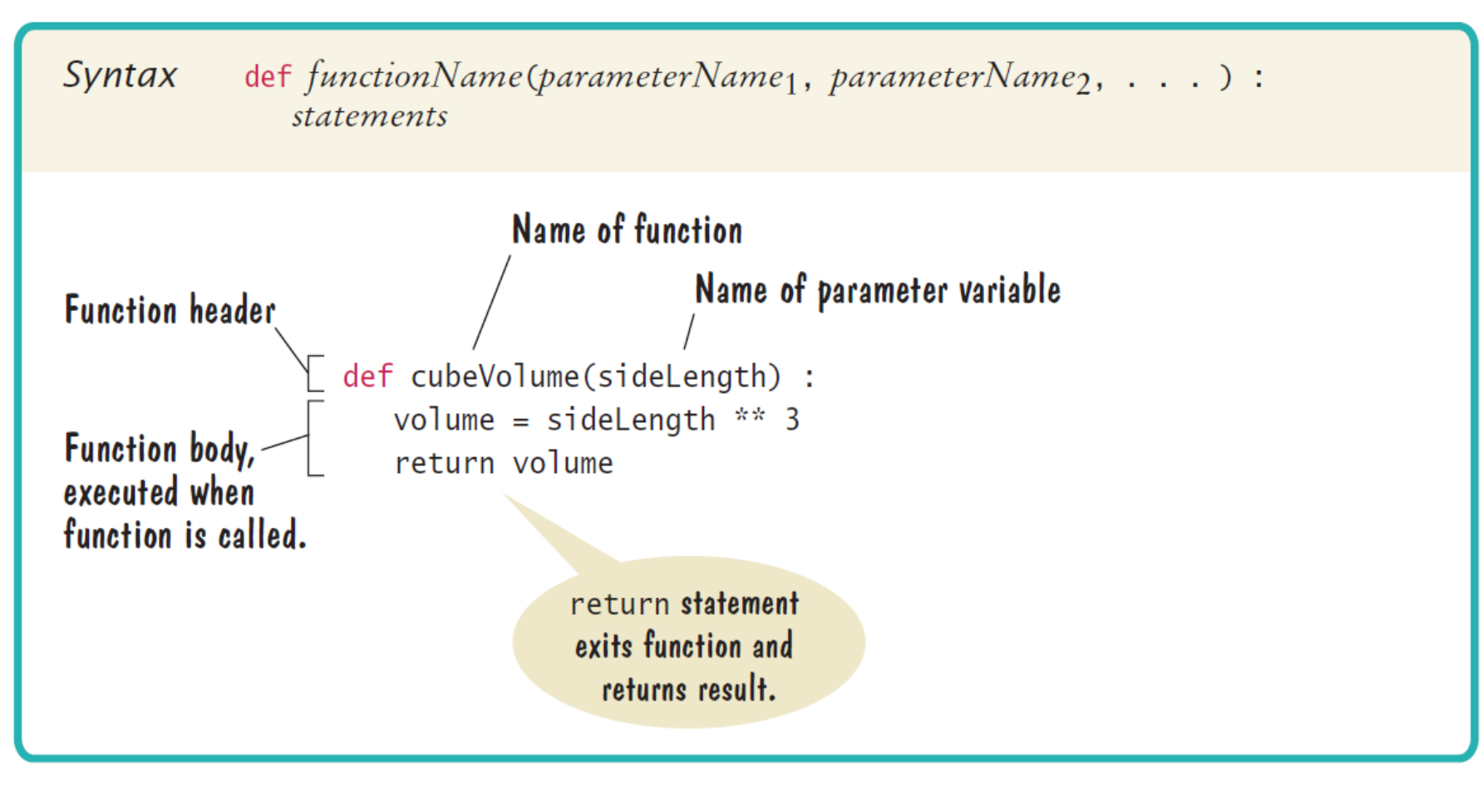
In [ ]:
```
def cubeVolume(sideLength):
    volume = sideLength ** 3
    return volume
```

- This line is called the **header** of the function.
- Next, specify the **body** of the function.
  - The body contains the statements that are executed when the function is called.

- In order to return the result of the function, use the **return** statement:
  - **return volume**

# Implementing and Testing Functions

Function Definition


```
Syntax    def functionName(parameterName1, parameterName2, . . . ) :
              statements
```

```
def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume
```
Name of function
Name of parameter variable
Function header
Function body, executed when function is called.
return statement exits function and returns result.

# Implementing and Testing Functions

## Testing a Function

- In order to test the function, your program should contain:
  - The definition of the function.
  - Statements that call the function and print the result.

In [ ]:
```
def cubeVolume(sideLength):
    volume = sideLength ** 3
    return volume
result1 = cubeVolume(2)
result2 = cubeVolume(10)
print("A cube with side length 2 has volume", result1)
print("A cube with side length 10 has volume", result2)
```

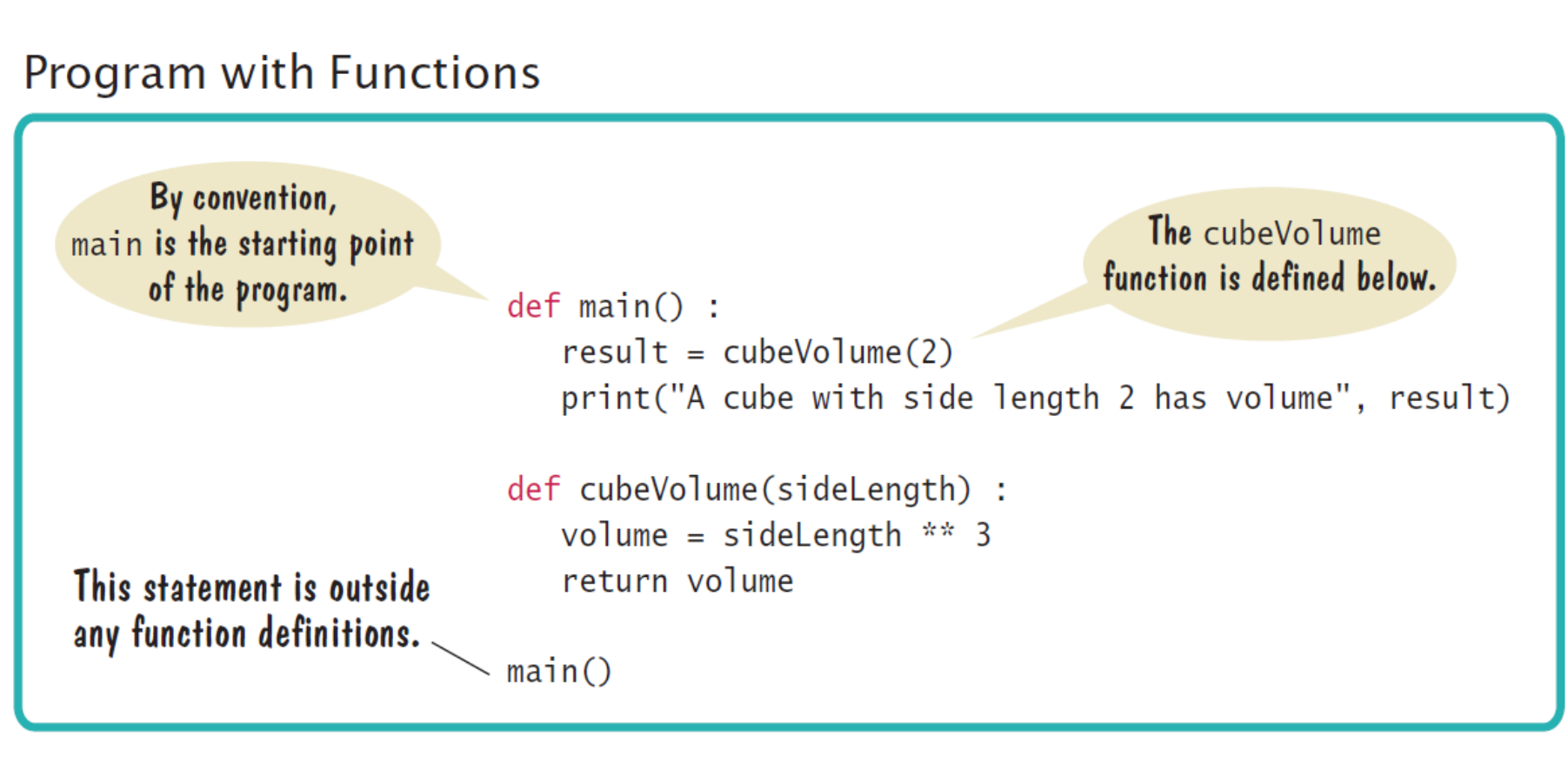
# Implementing and Testing Functions

## Programs that Contain Functions

- When you write a program that contains one or more functions, you need to pay attention to the order of the function definitions and statements in the program.
- As the Python interpreter reads the source code, it reads each function definition and each statement.
  - The statements in a function definition are not executed until the function is called.
  - Any statement not in a function definition, on the other hand, is executed as it is encountered.
- Therefore, it is important that you define each function before you call it.

# Implementing and Testing Functions

Program with Functions


```
def main() :
   result = cubeVolume(2)
   print("A cube with side length 2 has volume", result)

def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

main()
```
By convention, main is the starting point of the program.
The cubeVolume function is defined below.
This statement is outside any function definitions.

# Implementing and Testing Functions

In [ ]:
```
#  This program computes the volumes of two cubes.

def main() :
   result1 = cubeVolume(2)
   result2 = cubeVolume(10)
   print("A cube with side length 2 has volume", result1)
   print("A cube with side length 10 has volume", result2)

## Computes the volume of a cube.
#  @param sideLength the length of a side of the cube
#  @return the volume of the cube
#
def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

# Start the program.
main()
```

# Implementing and Testing Functions

## Student Activity

- Define a function squareArea that computes the area of a square of a given side length.

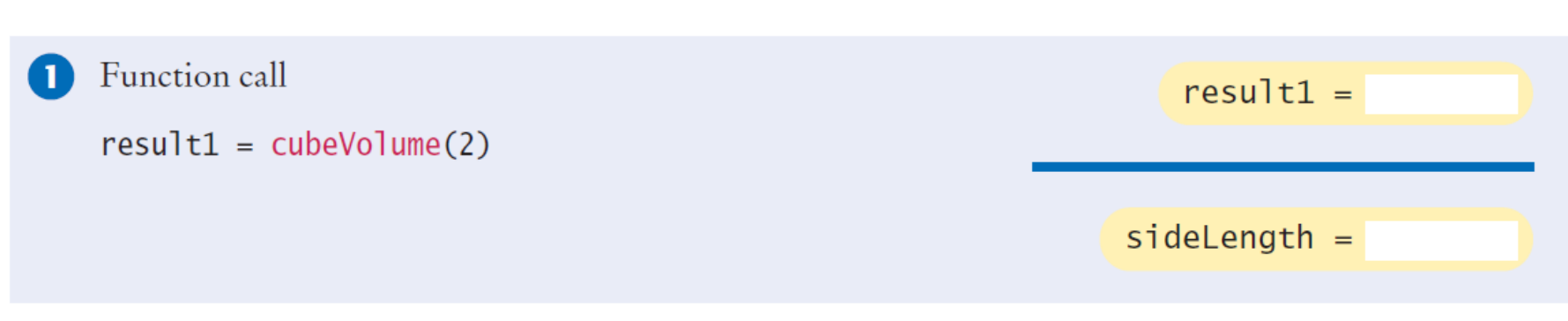
In [ ]:

# Parameter Passing

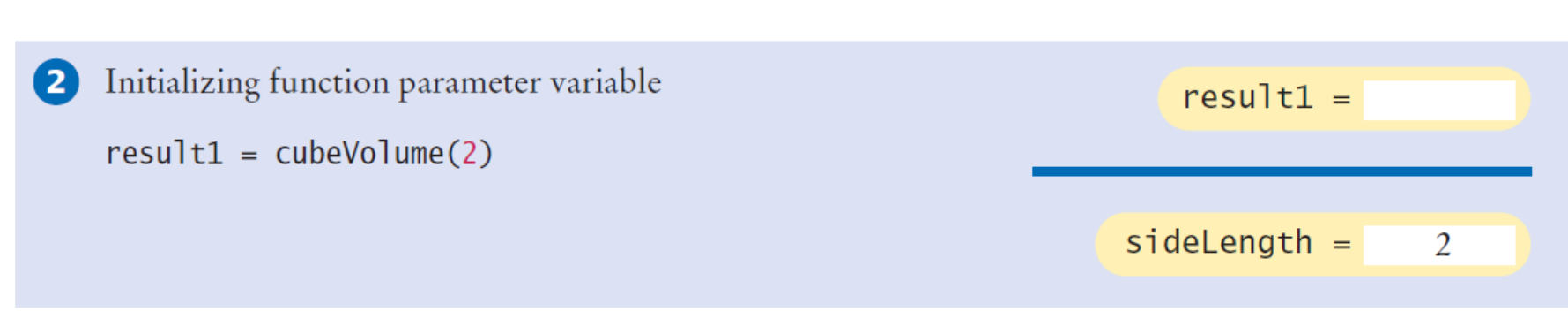
- When a function is called, variables are created for receiving the function’s arguments.
- These variables are called **parameter variables**.
  - (Another commonly used term is **formal parameters**.)
- The values that are supplied to the function when it is called are the **arguments** of the call.
  - (These values are also commonly called the **actual parameters**.)

# Parameter Passing

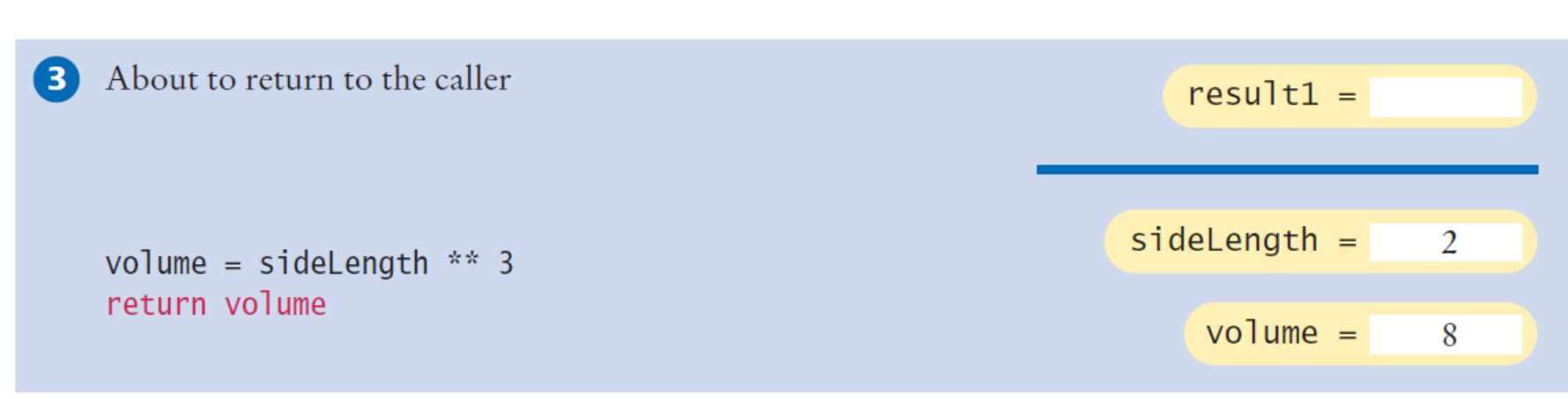
In [ ]:
```
#  This program computes the volumes of two cubes.

def main() :
   result1 = cubeVolume(2)
   result2 = cubeVolume(10)
   print("A cube with side length 2 has volume", result1)
   print("A cube with side length 10 has volume", result2)

## Computes the volume of a cube.
#  @param sideLength the length of a side of the cube
#  @return the volume of the cube
#
def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

# Start the program.
main()
```


① Function call
```
result1 = cubeVolume(2)
```

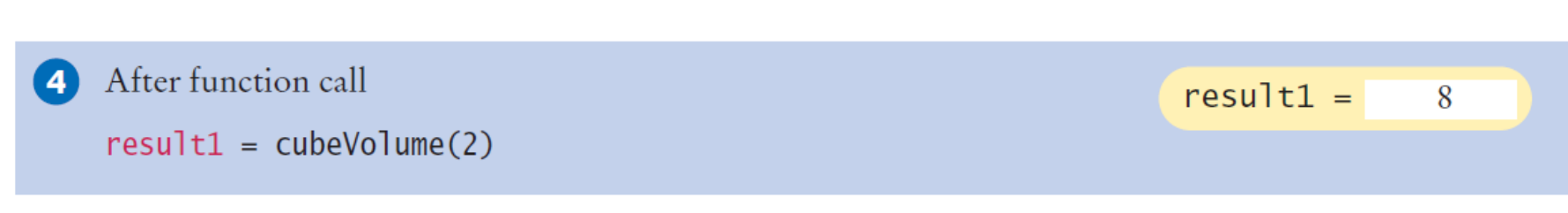
- The parameter variable **sideLength** of the **cubeVolume** function is created when the function is called.


② Initializing function parameter variable
```
result1 = cubeVolume(2)
```

- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, **sideLength** is set to 2.


③ About to return to the caller
```
volume = sideLength ** 3
return volume
```

- The function computes the expression **sideLength ** 3**, which has the value 8.
- That value is stored in the variable **volume**.


④ After function call
```
result1 = cubeVolume(2)
```

- The function returns. All of its variables are removed.
- The return value is transferred to the **caller**, that is, the function calling the **cubeVolume** function.
- The caller puts the return value in the **result1** variable.

# Parameter Passing

## Student Activity

- What does this program print? Use a diagram to find the answer.

In [ ]:
```
def main():
    a = 5
    b = 7
    print(mystery(a,b))
def mystery(x,y):
    z = x + y
    z = z / 2.0
    return z
main()
```

# Return Values

- The **return** statement terminates a function call and yields the function result.
- In the preceding examples, each **return** statement returned a variable.

- However, the **return** statement can return the value of any expression.
- Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return the value of a more complex expression:

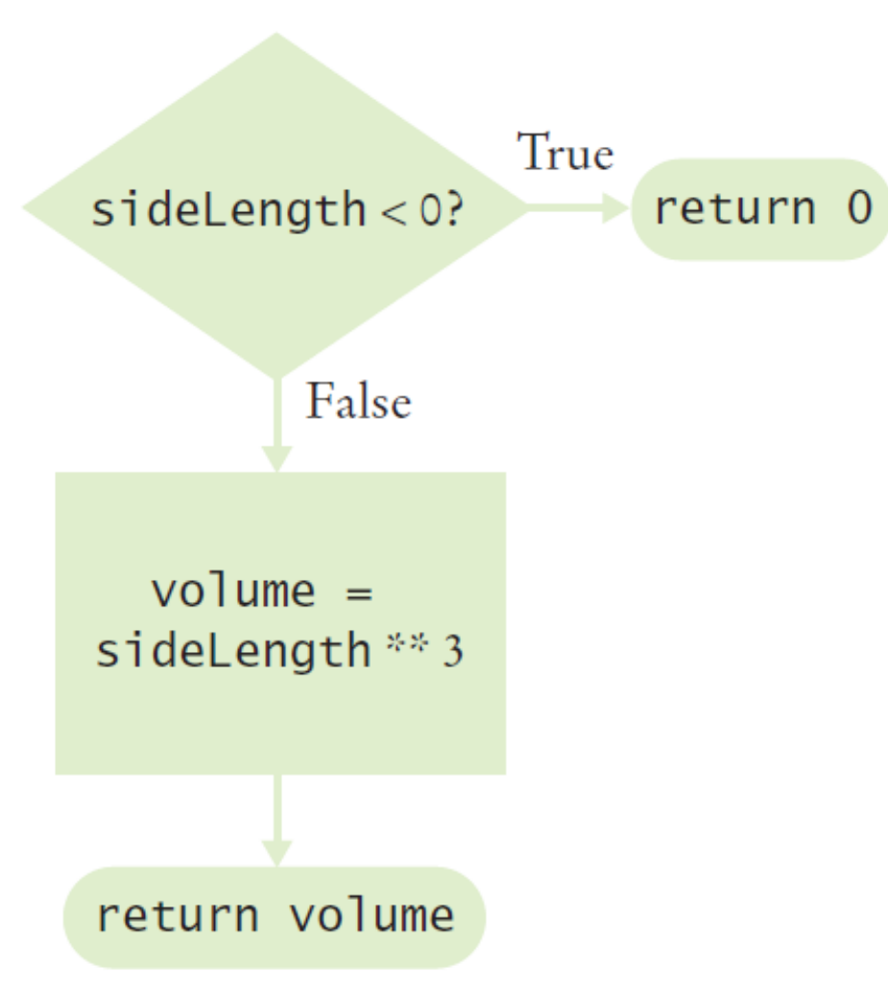
In [ ]:
```
def cubeVolume(sideLength):
    return sideLength ** 3
```

- When the **return** statement is processed, the function exits **immediately**.

# Return Values

- Some programmers find this behavior convenient for handling exceptional cases at the beginning of the function:

In [ ]:
```
def cubeVolume(sideLength):
    if sidelength < 0:
        return 0
    # Handle the regular case.
```



# Return Values

- Some programmers dislike the use of multiple **return** statements in a function.
- You can avoid multiple returns by storing the function result in a variable that you return in the last statement of the function.
- For example:

In [ ]:
```
def cubeVolume(sideLength):
    if sideLength >= 0:
        volume = sideLength ** 3
    else:
        volume = 0
    return volume
```

# Return Values

## Student Activity

- What does this function do?

In [ ]:
```
def mystery(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

# Functions Without Return Values

- Some functions may not return a value, but they can produce output.

- Sometimes, you need to carry out a sequence of instructions that does not yield a value.
- If that instruction sequence occurs multiple times, you will want to package it into a function.

# Functions Without Return Values

- Here is a typical example: Your task is to print a string in a box, like this:

```
-------
!Hello!
-------
```

In [ ]:
```
## Prints a string in a box.
# @param contents the string to enclose in a box
#
def boxString(contents):
    n = len(contents)
    if n == 0:
        return #Return immediately
    print("-"*(n+2))
    print("!"+contents+"!")
    print("-"*(n+2))

def main():
    boxString("Hello")
main()
```

## Student Activity

- What is wrong with the following statement?

In [ ]:
```
print(boxString("Hello"))
```

# Variable Scope

- As your programs get larger and contain more variables, you may encounter problems where you cannot access a variable that is defined in a different part of your program, or where two variable definitions conflict with each other.
- In order to resolve these problems, you need to be familiar with the concept of variable scope.

- The **scope** of a variable is the part of the program in which you can access it.

# Variable Scope

- In the following code segment, the **scope** of the parameter variable **sideLength** is the entire **cubeVolume** function but **not** the **main** function.

```
def main() :
   print(cubeVolume(10))

def cubeVolume(sideLength) :
   return sideLength ** 3
```

# Variable Scope

- A variable that is defined within a function is called a **local variable**.
- When a local variable is defined in a block, it becomes available from that point until the end of the function in which it is defined.

- For example, in the code segment below, the scope of the square variable is highlighted.

```
def main() :
   sum = 0
   for i in range(11) :
      square = i * i
      sum = sum + square

   print(square, sum)
```

In [ ]:
```
def main() :
    sum = 0
    for i in range(11) :
        square = i * i
        sum = sum + square
    print(square,sum)

main()
```

# Variable Scope

```
def main() :
   sideLength = 10
   result = cubeVolume()
   print(result)

def cubeVolume() :
   return sideLength ** 3   # Error

main()
```

In [ ]:
```
def main() :
    sideLength = 10
    result = cubeVolume()
    print(result)

def cubeVolume() :
    return sideLength **3

main()
```

- Note the scope of the variable **sideLength**.
- The **cubeVolume** function attempts to read the variable, but it cannot;
  - The scope of **sideLength** does not extend outside the **main** function.

# Variable Scope

- It is possible to use the variable name more than once in a program.
- For example,

```
def main() :
   result = square(3) + square(4)
   print(result)

def square(n) :
   result = n * n
   return result

main()
```

In [ ]:
```
def main() :
    result = square(3) + square(4)
    print(result)

def square(n) :
    result = n * n
    return result

main()
```

- Each **result** variable is defined in a separate function, and their scope do not overlap.

# Variable Scope

- Any variable that is defined outside a function is a **global variable**.
- A **global variable** is **visible** to all functions defined after it.
  - i.e., you can get the value of the variable.
- However, any function that wishes to update a **global variable** must include a **global** declaration:

In [ ]:
```
balance = 1000 # A global varaible

def withdraw(amount)  :
    global balance # This function intends to update the global balance variable
    if balance >= amount:
        balance = balance - amount

withdraw(200)
print(balance)
```

In [ ]:
```
balance = 1000 # A global varaible

def withdraw(amount)  :
#    global balance # This function intends to update the global balance variable
    if balance >= amount:
        newBalance = balance - amount
    print("New Balance =", newBalance)

withdraw(200)
print("Current value of balance =", balance)
```

- If you omit the **global** declaration, then the balance variable inside the **withdraw** function is considered a local variable.

- Generally, global variables are not a good idea.
- When multiple functions update global variables, the result can be difficult to predict.
- Particularly in larger programs developed by multiple programmers, it is important that the effect of each function be clear and easy to understand.
- You should avoid global variables in your programs.

# Summary

- A function is a named sequence of instructions.
- Arguments are supplied when a function is called.
- The return value is the result that the function computes.
- When declaring a function, you provide a name for the function and a variable for each argument.
- Function comments explain the purpose of the function, the meaning of the parameters and return values, as well as any special requirements.
- Parameter variables hold the arguments supplied in the function call.

# Summary

- The return statement terminates a function call and yields the function result.
  - Complete computations that can be reused into functions.
- Use the process of stepwise refinement to decompose complex tasks into simpler ones.
  - When you discover that you need a function, write a description of the parameter variables and return values.
  - A function may require simpler functions to carry out its work.

## Summary

- The scope of a variable is the part of the program in which the variable is visible.
  - Two local or parameter variables can have the same name, provided that their scope do not overlap.
  - You can use the same variable name within different functions since their scope does not overlap.
  - Local variable declared inside a function are not visible to code inside other functions.