Chapter Five: Functions

Chapter Goals

- To be able to implement functions
- To become familiar with the concept of parameter passing
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To learn how to think recursively

In this chapter, you will learn how to design and implement your own functions

Using the process of stepwise refinement, you will be able to break up complex tasks into sets of cooperating functions

Contents

- Functions as Black Boxes
- Implementing and Testing Functions
- Parameter Passing
- Return Values
- Functions without Return Values
- Reusable Functions
- Stepwise Refinement
- Variable Scope
- Graphics: Building an Image Processing Toolkit
- Recursive Functions

Functions as Black Boxes

SECTION 5.1

Functions as Black Boxes

- A function is a sequence of instructions with a name
- For example, the round function, which was introduced in Chapter 2, contains instructions to round a floating-point value to a specified number of decimal places

Calling Functions

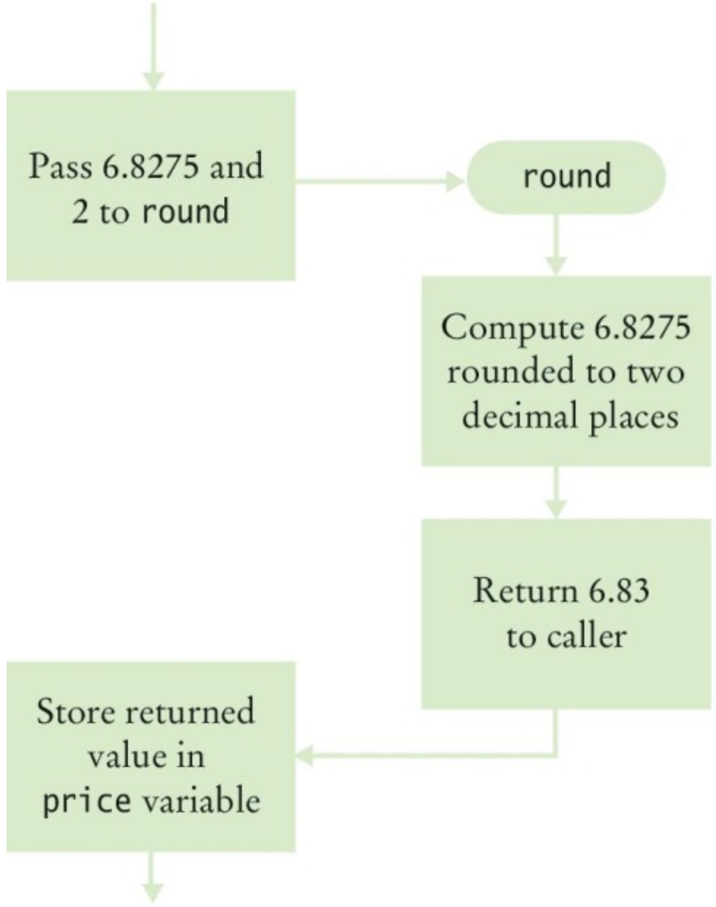
- You *call* a function in order to execute its instructions

```
price = round(6.8275, 2) # Sets result to 6.83
```

- By using the expression round(6.8275, 2), your program *calls* the round function, asking it to round 6.8275 to two decimal digits

Calling Functions (2)

- The round function *returns* its result back to where the function was called and your program resumes execution


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

Function Arguments

- 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 the values for which we want the function to compute a result

Function Arguments

- Functions can receive multiple arguments or it is also possible to have functions with no arguments

Function Return Values

- The “output” that the round function computes is called the **return value**
- Functions return only one value
- The return value of a function is returned to the point in your program where the function was called

```
price = round(6.8275, 2)
```

- When the round function returns its result, the return value is stored in the variable ‘`price`’ statement)

Function Return Values (2)

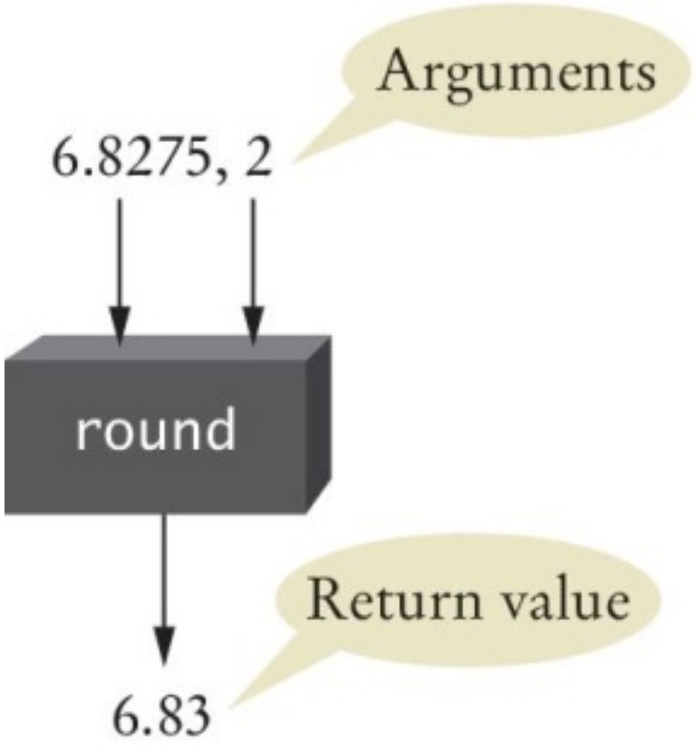
- Do not confuse returning a value with producing program output which is produced when using a `print()` statement

Black Box Analogy

- A thermostat is a ‘black box’
  - Set a desired temperature
  - Turns on heater/AC as required
  - You don’t have to know how it really works!
    - How does it know the current temp?
    - What signals/commands does it send to the heater or A/C?
- Use functions like ‘black boxes’
  - Pass the function what it needs to do its job
  - Receive the answer

The round Function as a Black Box

- You pass the round function its necessary arguments (6.8275 & 2) and it produces its result (6.83)



The round Function as a Black Box

- You may wonder how the round function performs its job
- As a user of the function, you don't 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

Designing Your Own Functions

- When you design your own functions, you will want to make them appear as black boxes to other programmers
    - 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

SECTION 5.2

Implementing and Testing Functions

- A function to calculate the volume of a cube
  - What does it need to do its job?
  - What does it answer with?
- When writing (‘defining’) this function
  - Pick a name for the function (cubeVolume)
  - Declare a variable for each incoming argument
    (sideLength) (called parameter variables)
  - Put all this information together along with the def keyword to form the first line of the function’s definition:

```
def cubeVolume(sideLength):
```

This line is called the **header** of the function

Testing a Function

- If you run a program containing just the function definition, then nothing happens
  - After all, nobody is calling the 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

Calling/Testing the Cube Function

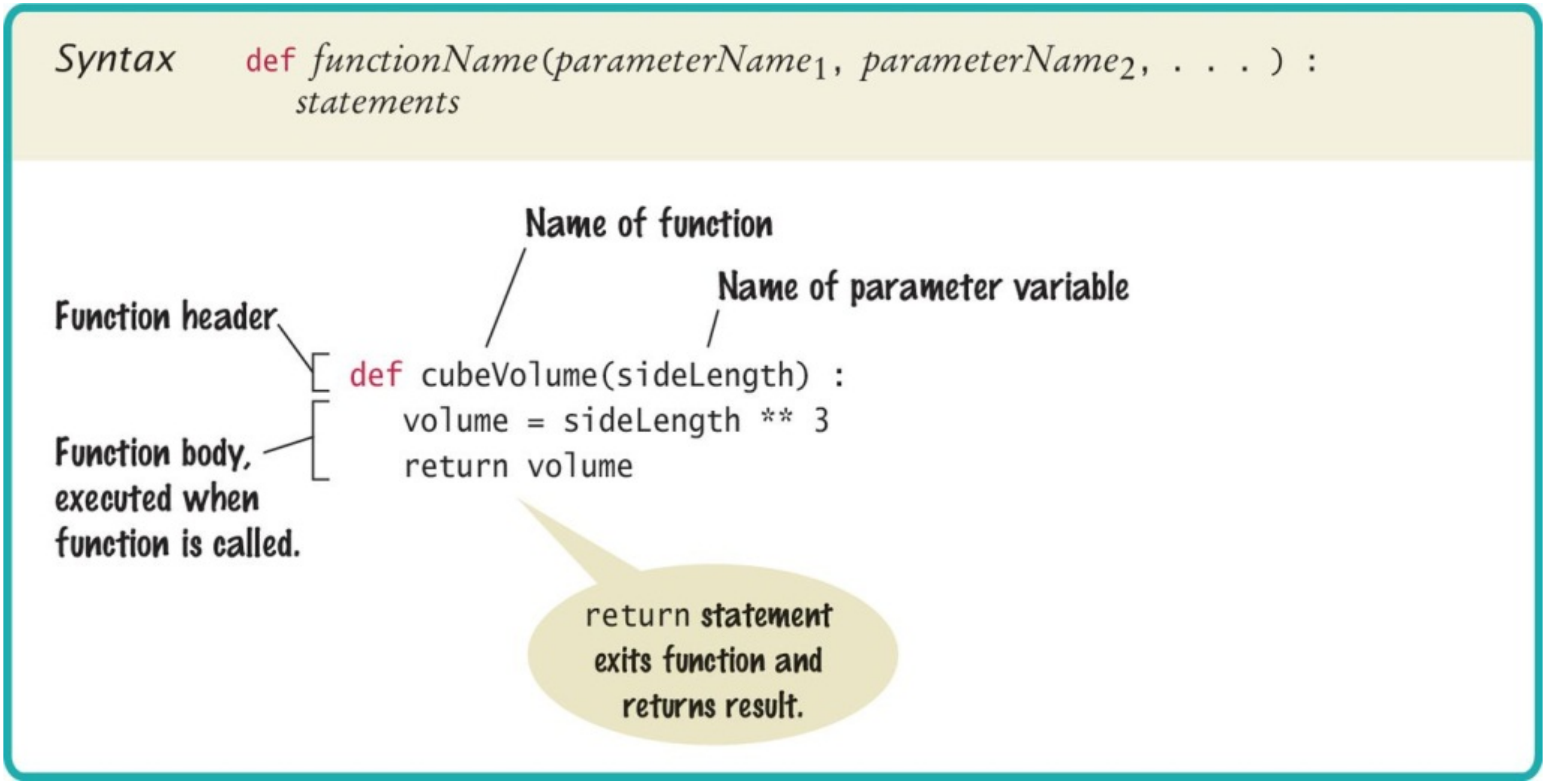
Implementing the function (function definition)

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

Calling/testing the function

```
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)
```

Syntax: 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.

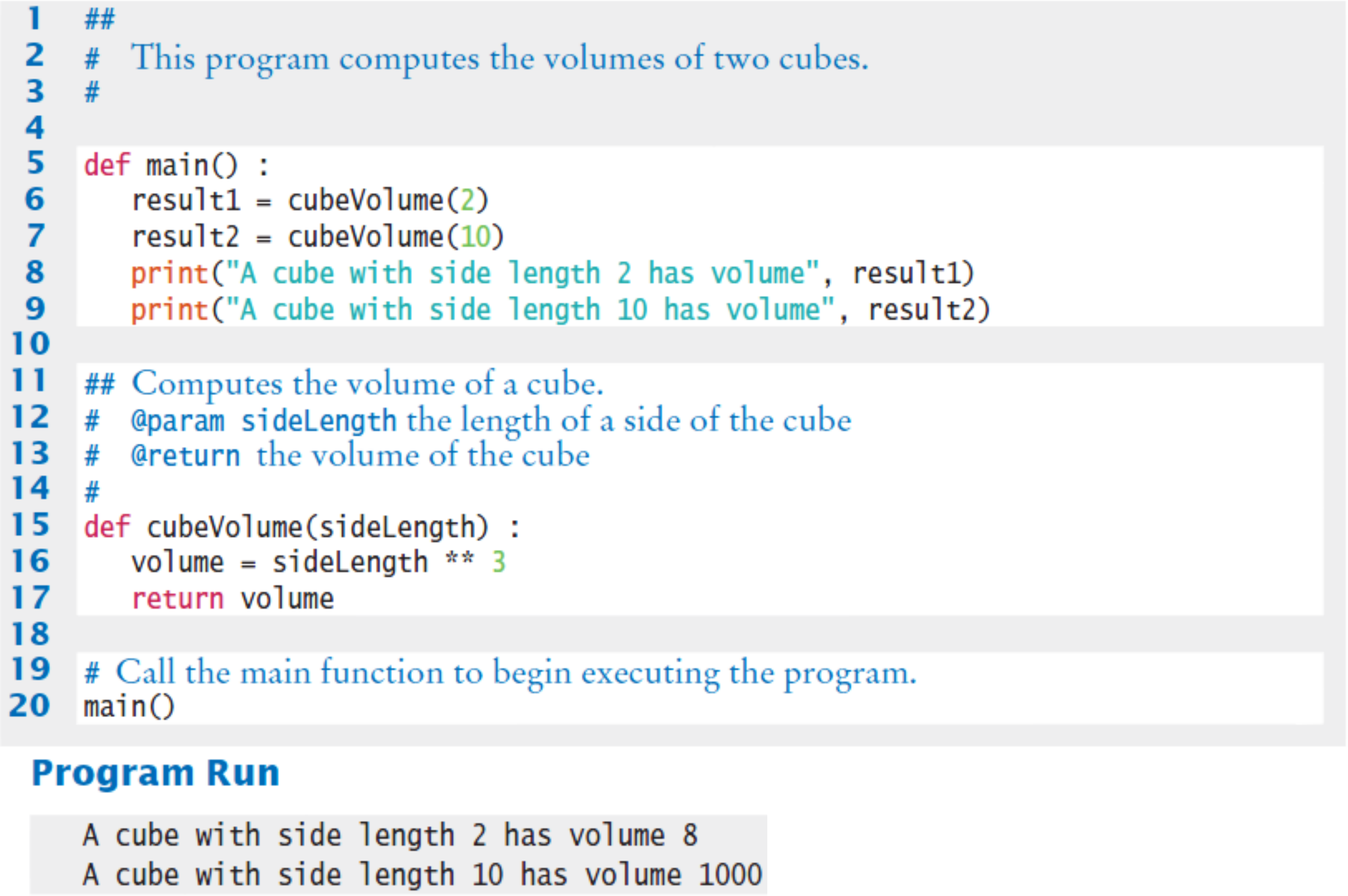
Programming Tip: Function Comments

- Whenever you write a function, you should *comment* its behavior
- Remember, comments are for human readers, not compilers (sort of)

```
## 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
```

Function comments explain the purpose of the function, the meaning of the parameter variables and the return value, as well as any special requirements

Cubes.py with Documentation



```python
##
#  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

# Call the main function to begin executing the program.
main()
```

Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

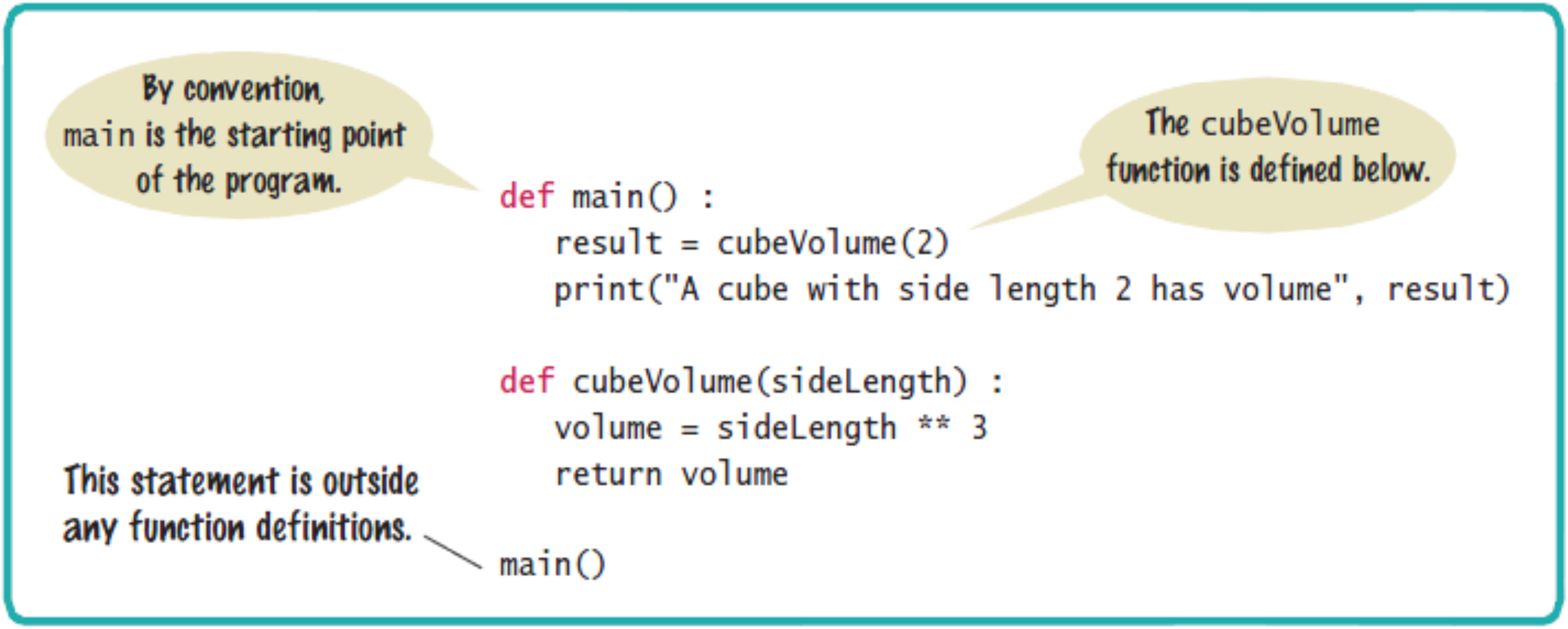
Cubes.py

- Open the file Cubes.py in PyCharm
- The files contains to functions:
  - main
  - cubeVolume
- Line 20 contains the call to the function “main”

The `main` Function

- When defining and using functions in Python, it is good programming practice to place all statements into functions, and to specify one function as the starting point
- Any legal name can be used for the starting point, but we chose ‘main’ since it is the required function name used by other common languages
- Of course, we must have one statement in the program that calls the main function

Syntax: The main Function



By convention, main is the starting point of the program.

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

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

main()
```

The cubeVolume function is defined below.

This statement is outside any function definitions.

Using Functions: Order (1)

- It is important that you define any function before you call it
- For example, the following will produce a compile-time error:

```
print(cubeVolume(10))
def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume
```

- The compiler does not know that the cubeVolume function will be defined later in the program

Using Functions: Order (2)

- However, a function can be called from within another function before the former has been defined
- The following is perfectly legal:

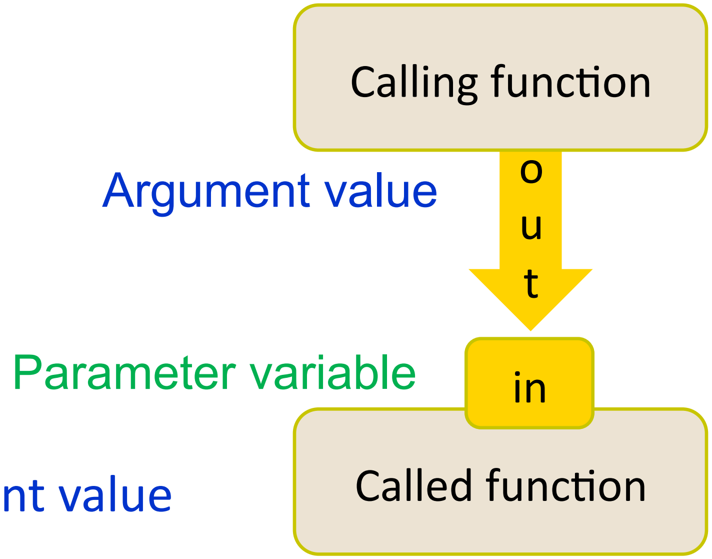
```
def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume",
      result)

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

main()
```

Parameter Passing

SECTION 5.3

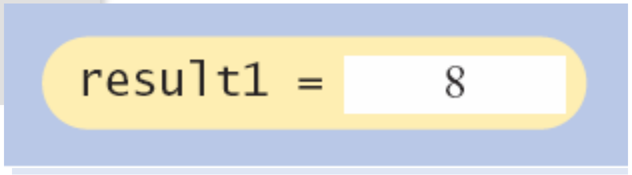
Parameter Passing

- Parameter variables receive the argument values supplied in the function call
- The argument value may be:
  - The contents of a variable
  - A 'literal' value (2)
    - Aka, 'actual parameter' or argument
- The parameter variable is:
  - Declared in the called function
  - Initialized with the value of the argument value
  - Used as a variable inside the called function
    - Aka, 'formal parameter'

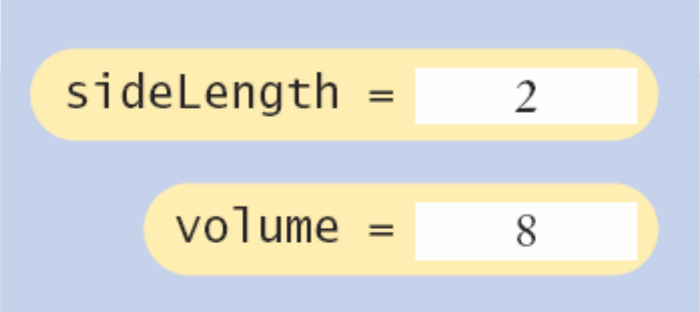


Parameter Passing Steps

```
result1 = cubeVolume(2)
```

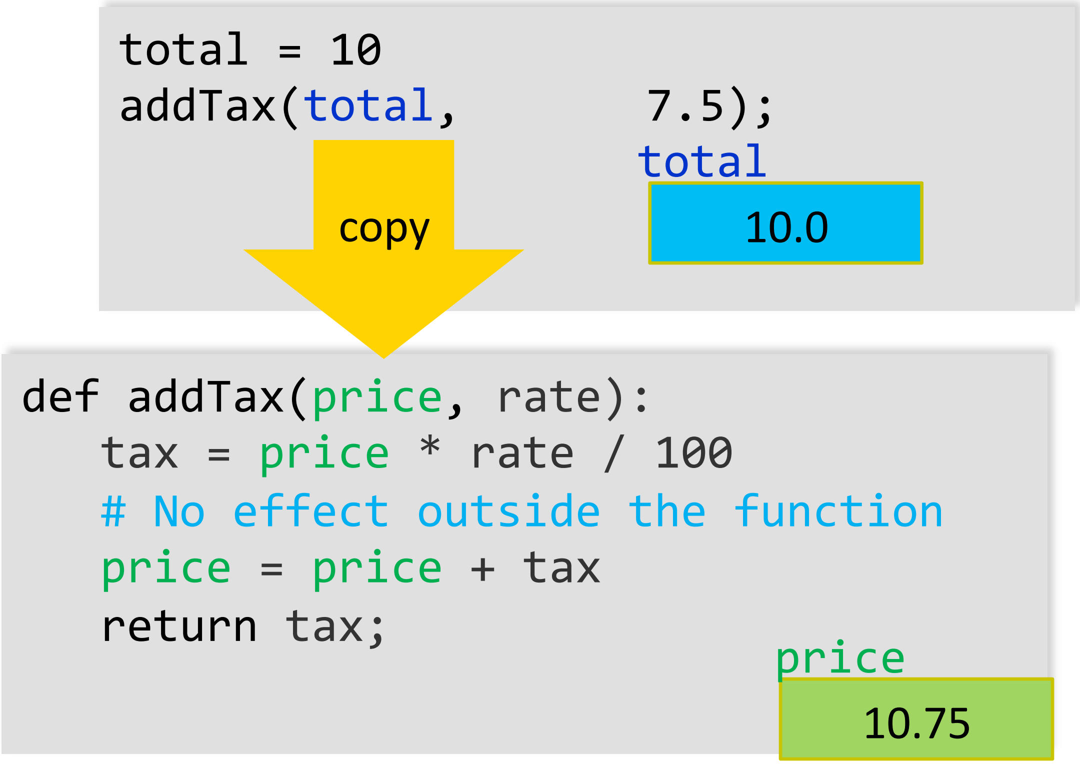


```
def cubeVolume(sideLength):
    volume = sideLength * 3
    return volume
```



Common Error 5.1

- Trying to modify parameter variables
- A copy of the argument values is passed (the ***Value*** is passed)
  - Called function (addTax) can modify local copy (price)



```
total = 10
addTax(total,       7.5);
```

```
def addTax(price, rate):
   tax = price * rate / 100
   # No effect outside the function
   price = price + tax
   return tax;
```

Programming Tip 5.2

- Do not modify parameter variables

Many programmers find this practice confusing

```
def totalCents(dollars, cents) :
    cents = dollars * 100 + cents # Modifies parameter variable.
    return cents
```

To avoid the confusion, simply introduce a separate variable:

```
def totalCents(dollars, cents) :
    result = dollars * 100 + cents
    return result
```

Return Values

SECTION 5.4

Return Values

- Functions can (optionally) return one value
  - Add a return statement that returns a value
    - A return statement does two things:
      1) Immediately terminates the function
      2) Passes the return value back to the calling function

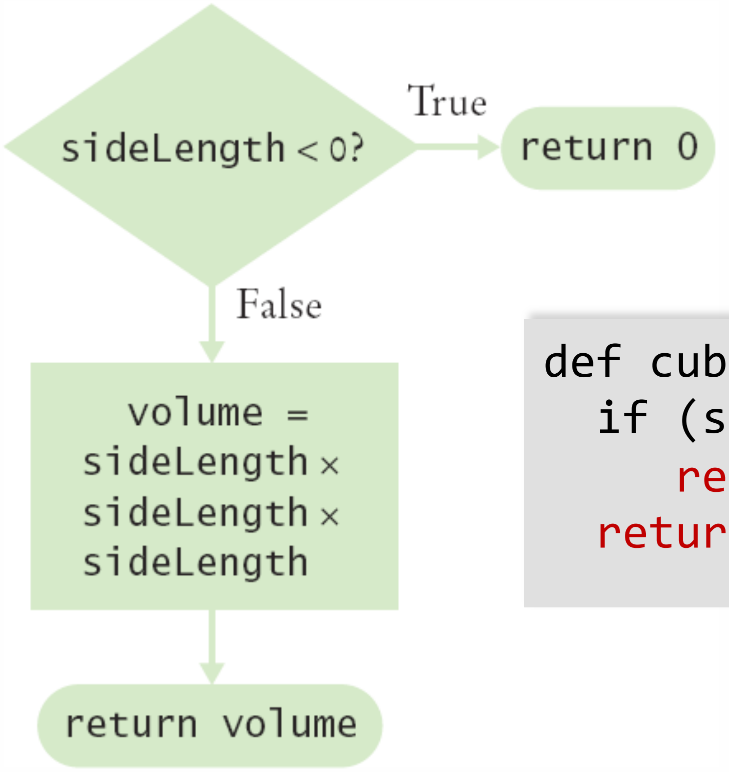
```
def cubeVolume (sideLength):
   volume = sideLength * 3
   return volume
```

return statement

The return value may be a value, a variable or a calculation

Multiple return Statements

- A function can use multiple return statements
  - But every branch must have a return statement



```
def cubeVolume(sideLength):
  if (sideLength < 0):
     return 0
  return sideLength * 3
```

Multiple `return` Statements (2)

- Alternative to multiple returns (e.g., one for each branch):
  - 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:

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

Make Sure A Return Catches All Cases

- Missing return statement
  - Make sure all conditions are handled
  - In this case, `sideLength` could be equal to 0
    - No return statement for this condition
    - The compiler will *not* complain if any branch has no return statement
    - It may result in a run-time error because Python returns the special value **None** when you forget to return a value

```
def cubeVolume(sideLength) :
    if sideLength >= 0 :
        return sideLength ** 3
    # Error—no return value if sideLength < 0
```

Make Sure A Return Catches All Cases (2)

- A correct implementation:

```
def cubeVolume(sideLength) :
    if sideLength >= 0
        return sideLength ** 3
    else :
        return 0
```