PGR107
Python Programming

Lecture 5 – Functions

Kristiania University College

Chapter 5 - 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

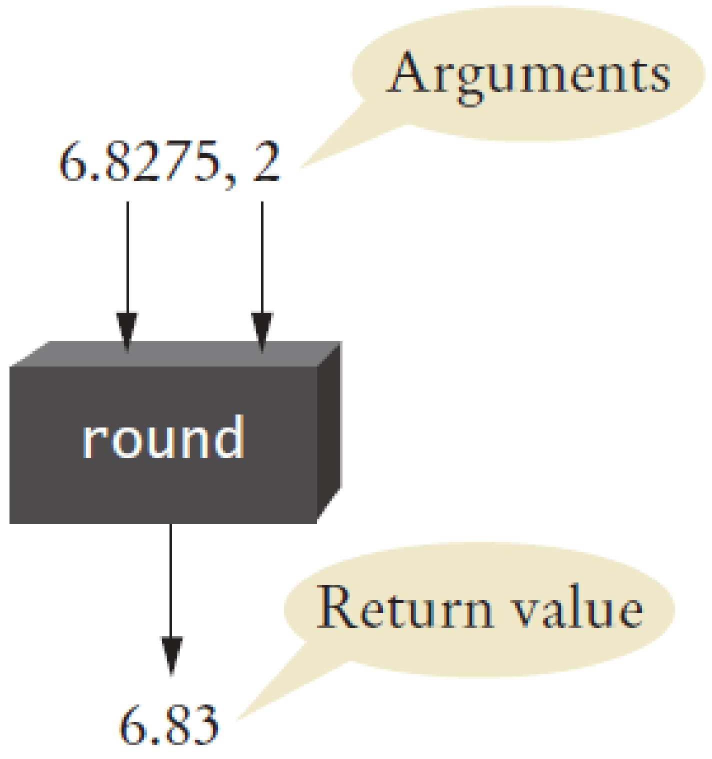
Functions

- A **function** packages a computation consisting of multiple steps into a form that can be easily understood and reused.

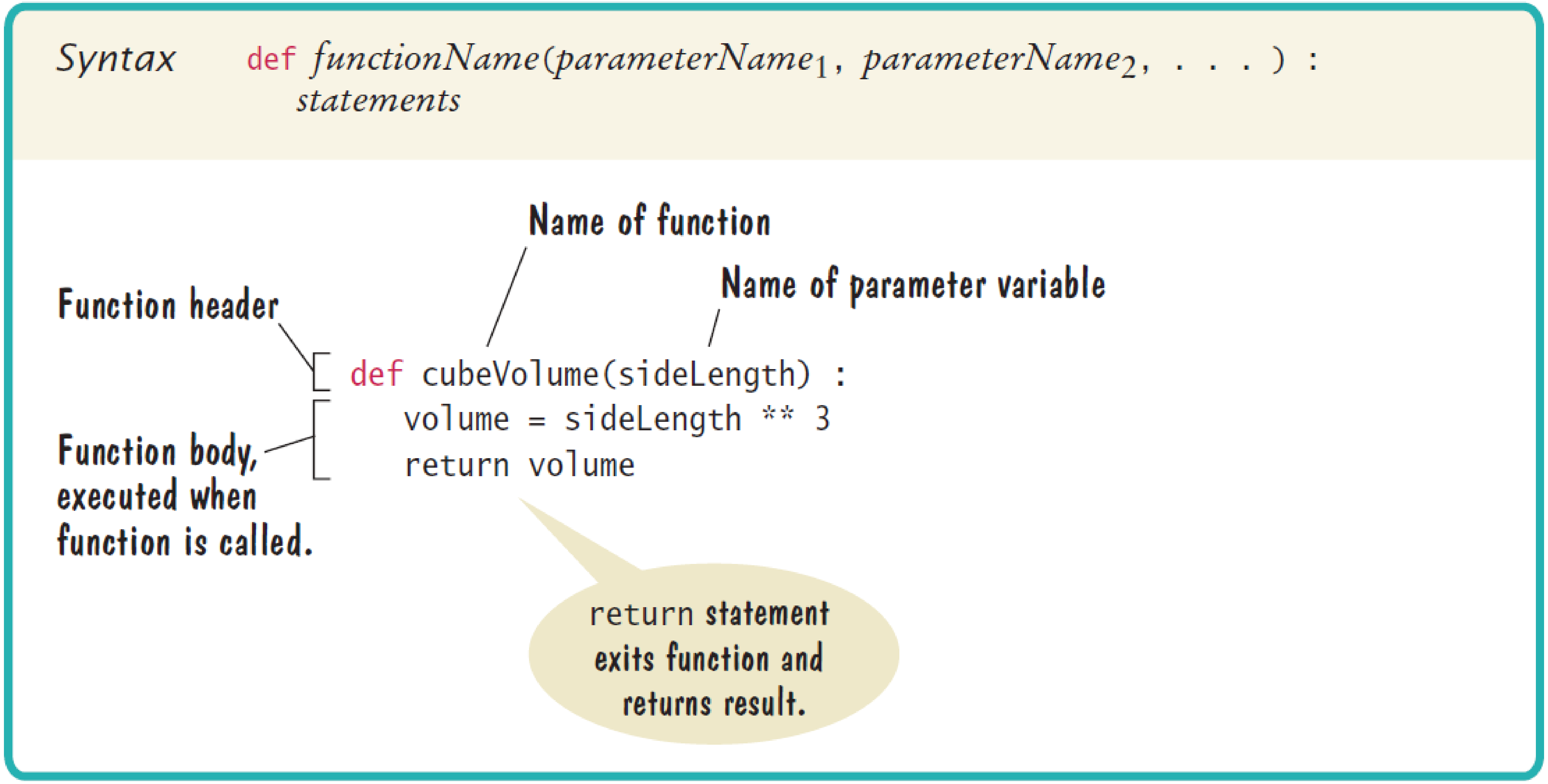
- A **function** is a sequence of instructions with a name. You call a function in order to execute its instructions. For example, the following calls the round function, asking it to round 6.8275 to two decimal digits.

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

Sets result to 6.83



Functions



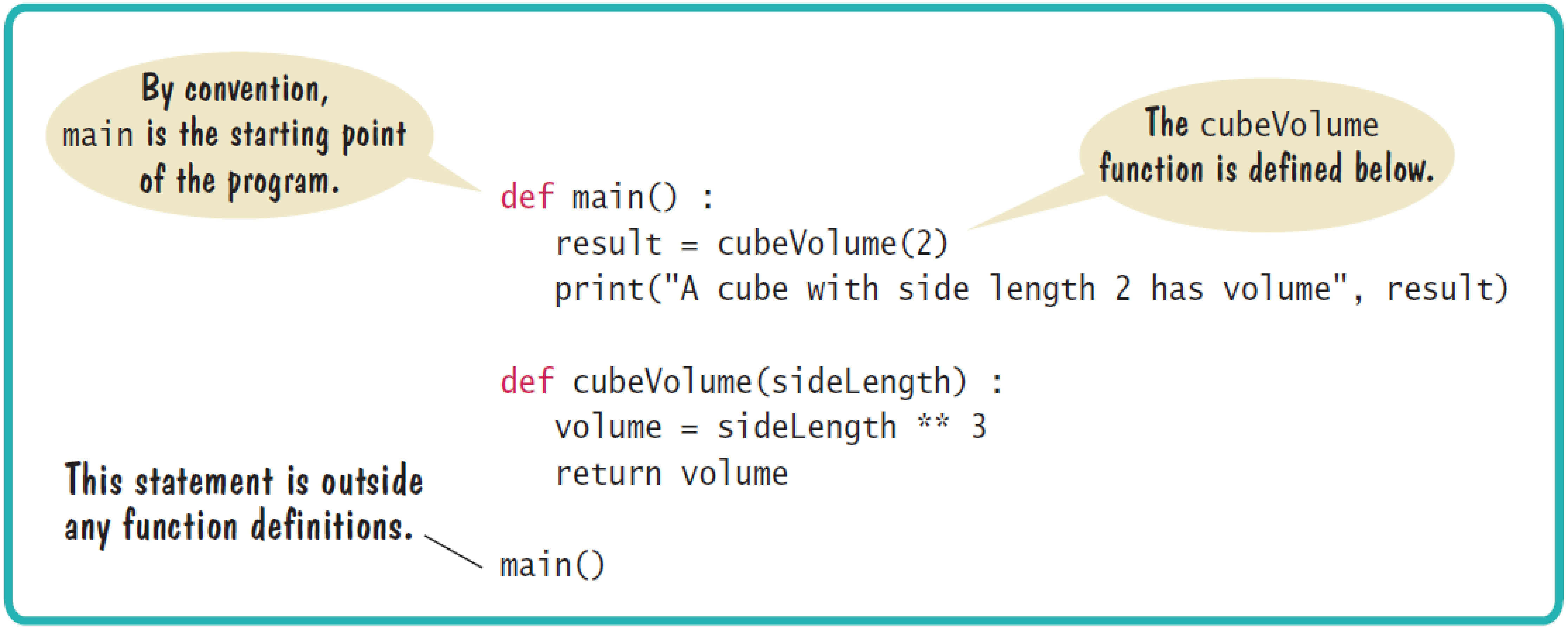
Syntax def *functionName*(*parameterName*$_1$, *parameterName*$_2$, . . .) :
    statements

```
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.

Functions

- A function can be called from within another function before the former has been defined. For example, 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()
```

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

Parameter Passing

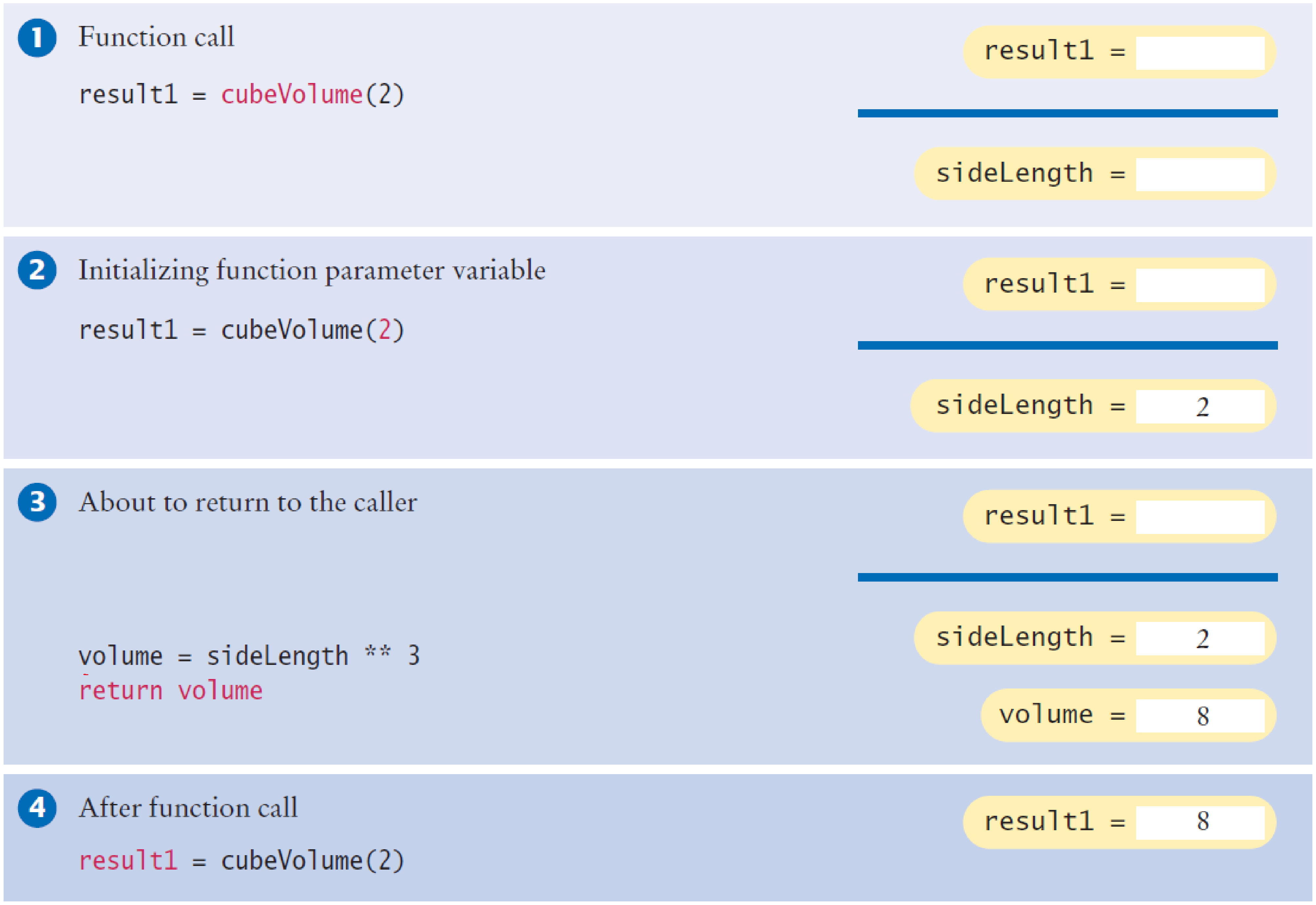
- When a function is called, variables are created for receiving the function's arguments. These variables are called **parameter variables**. (Another commonly used term is **formal parameters**.) The values that are supplied to the function when it is called are the **arguments** of the call. (These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

- Each parameter variable is initialized with the corresponding argument.

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

result1 = cubeVolume (2)
```

Parameter Passing



Return Values

- The **return** statement terminates a function call and specifies the result of a function.

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

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

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

Example

- **Problem Statement:** Suppose that you are helping archaeologists who research Egyptian pyramids. You have taken on the task of writing a function that determines the volume of a pyramid, given its height and base length.

$$\boxed{\text{volume} = 1/3 \times \text{height} \times \text{base area}}$$

$$\text{base area} = \text{base length} \times \text{base length}$$

Functions Without Return Values

- Sometimes, you need to carry out a sequence of instructions that does not yield a value.

- For example, your task is to print a string in a box, like this:

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

```
def boxString(contents) :
   n = len(contents) :
   print("-" * (n + 2))
   print("!" + contents + "!")
   print("-" * (n + 2))
```

```
boxString("Hello")
```

Functions Without Return Values

- If you want to return from a function that does not compute a value before reaching the end, you use a return statement without a value. For example,

```
def boxString(contents) :
   n = len(contents)
   if n == 0 :
      return   # Return immediately
   print("-" * (n + 2))
   print("!" + contents + "!")
   print("-" * (n + 2))
```

Reusable Functions

- When you write nearly identical code or pseudocode multiple times, either in the same program or in separate programs, consider introducing a function. Here is a typical example of code replication:

```
hours = int(input("Enter a value between 0 and 23: "))
while hours < 0 or hours > 23 :
   print("Error: value out of range.")
   hours = int(input("Enter a value between 0 and 23: "))

minutes = int(input("Enter a value between 0 and 59: "))
while minutes < 0 or minutes > 59 :
   print("Error: value out of range.")
   minutes = int(input("Enter a value between 0 and 59: "))
```

Reusable Functions

```python
def readIntUpTo(high) :
   value = int(input("Enter a value between 0 and " + str(high) + ": "))
   while value < 0 or value > high :
      print("Error: value out of range.")
      value = int(input("Enter a value between 0 and " + str(high) + ": "))

   return value
```

```python
hours = readIntUpTo(23)
minutes = readIntUpTo(59)
```

Reusable Functions

- Note that the function can be reused in other programs that need to read integer values. However, we should consider the possibility that the smallest value need not always be zero.

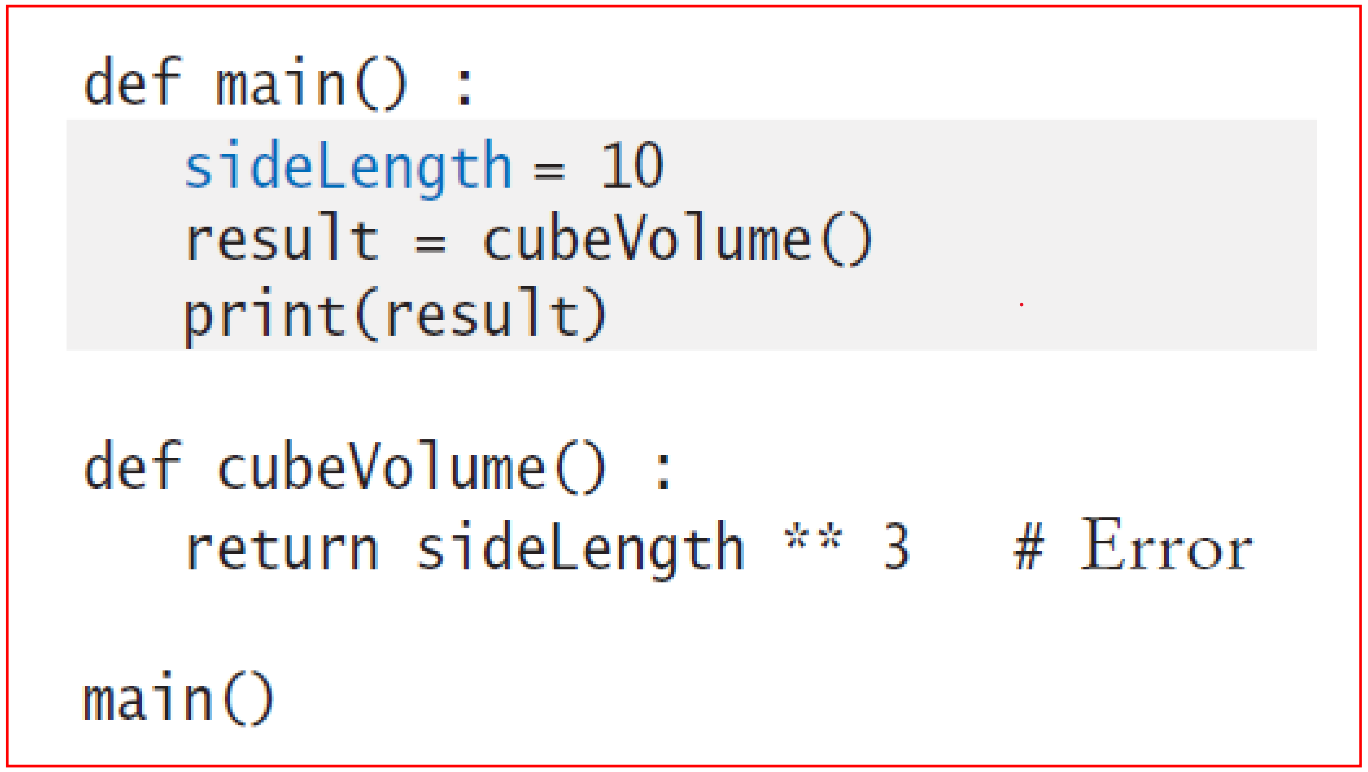
```
def readIntBetween(low, high) :
   value = int(input("Enter a value between " + str(low) + " and " +
                     str(high) + ": "))
   while value < low or value > high :
      print("Error: value out of range.")
      value = int(input("Enter a value between " + str(low) + " and " +
                        str(high) + ": "))
   return value
```

```
hours = readIntBetween(0, 23)
```

```
month = readIntBetween(1, 12)
```

Variable Scope

- The **scope** of a variable is the part of the program in which you can access it. For example, the scope of a function’s parameter variable is the entire function.



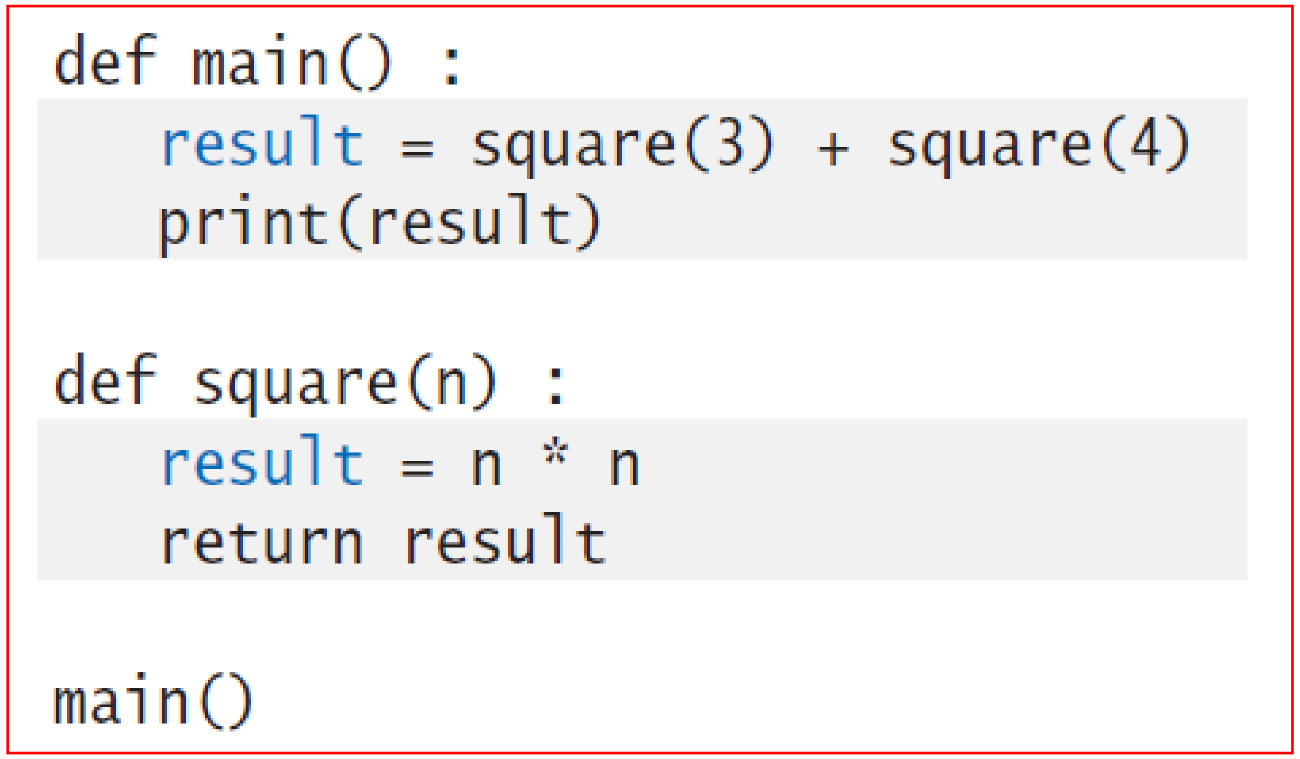
```
def main() :
    sideLength = 10
    result = cubeVolume()
    print(result)

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

main()
```

Variable Scope

- It is possible to use the same variable name more than once in a program. Consider the **result** variables in the following example:



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

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

main()
```

Variable Scope

- **Global Variables:** variables that are defined outside functions. A global variable is visible to all functions defined after it. However, any function that wishes to update a global variable must include a global declaration, like this:

```
balance = 10000    # A global variable

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

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

End of Chapter 5