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

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

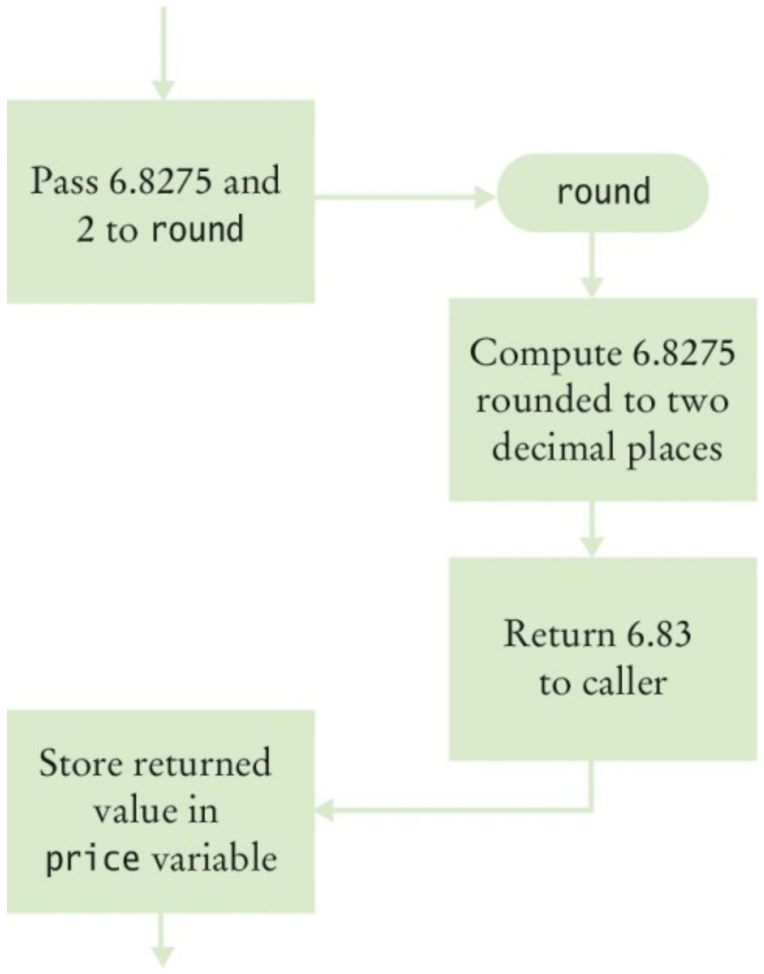
- You *call* a function in order to execute its instructions

```
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)

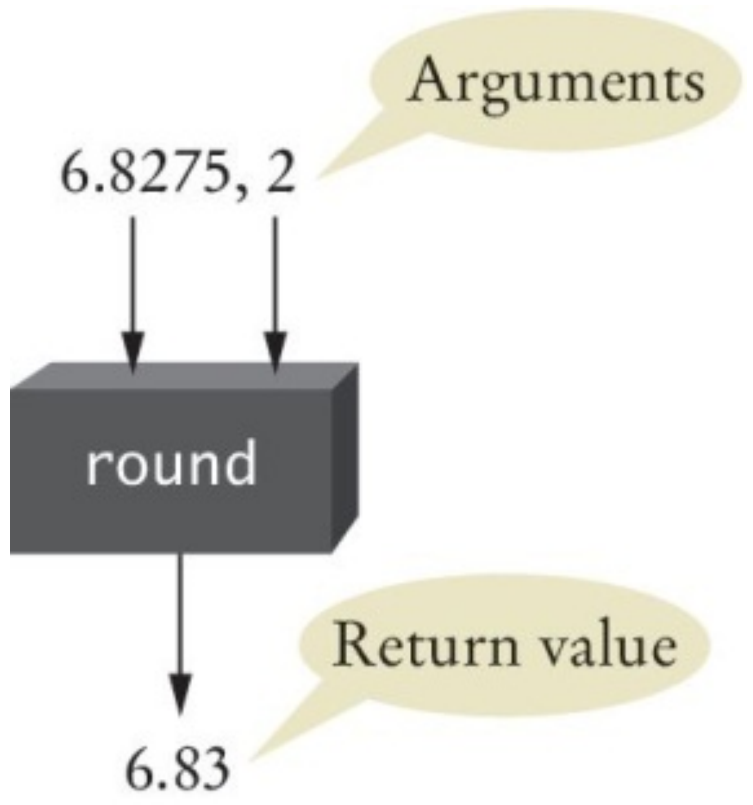
- Do not confuse returning a value with producing program output which is produced when using a `print()` statement

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

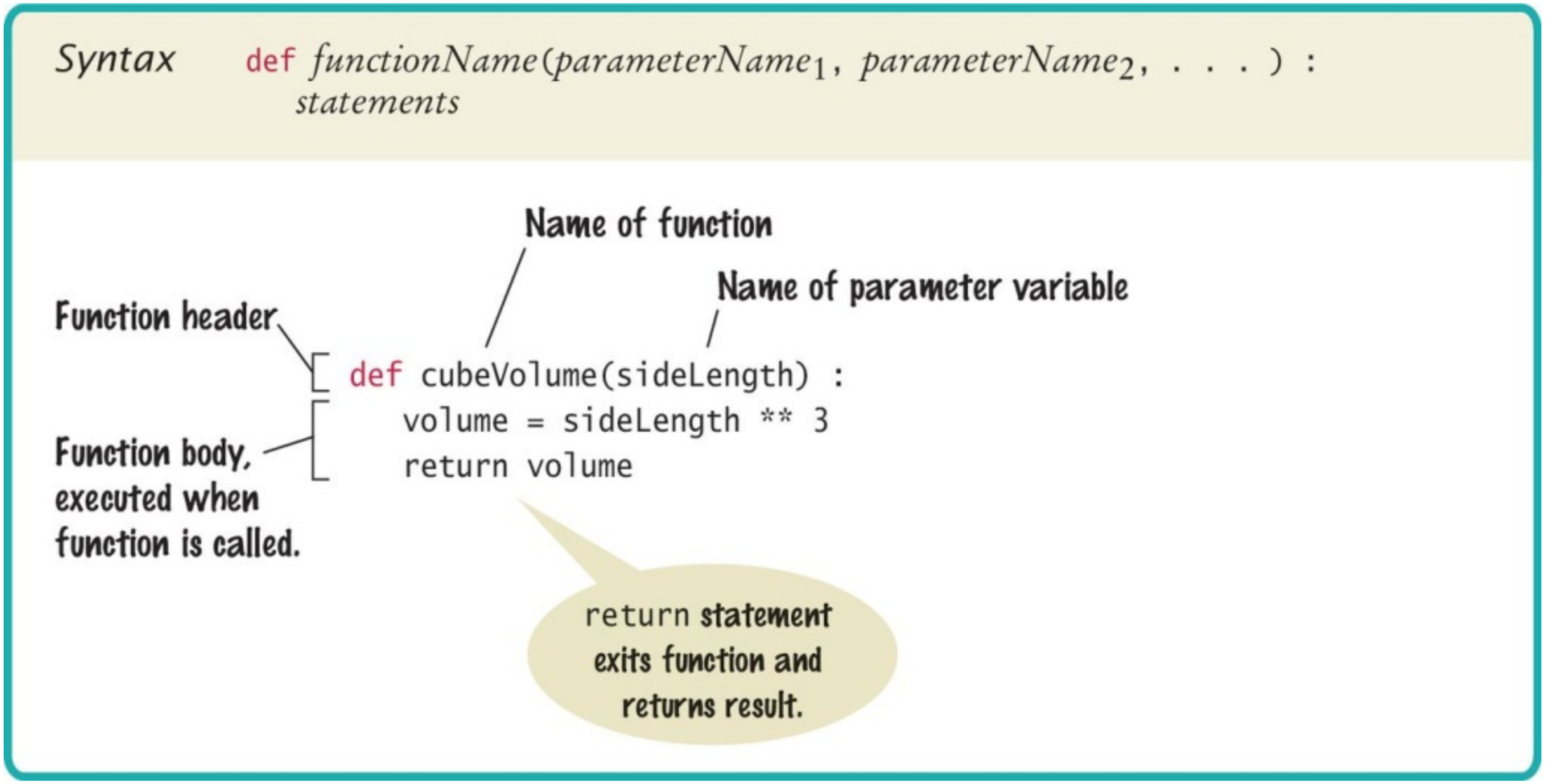
Implementing the function (function definition)

```
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*(*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.

Programming Tip: Function Comments

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

```
## 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

```
##
#  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
```

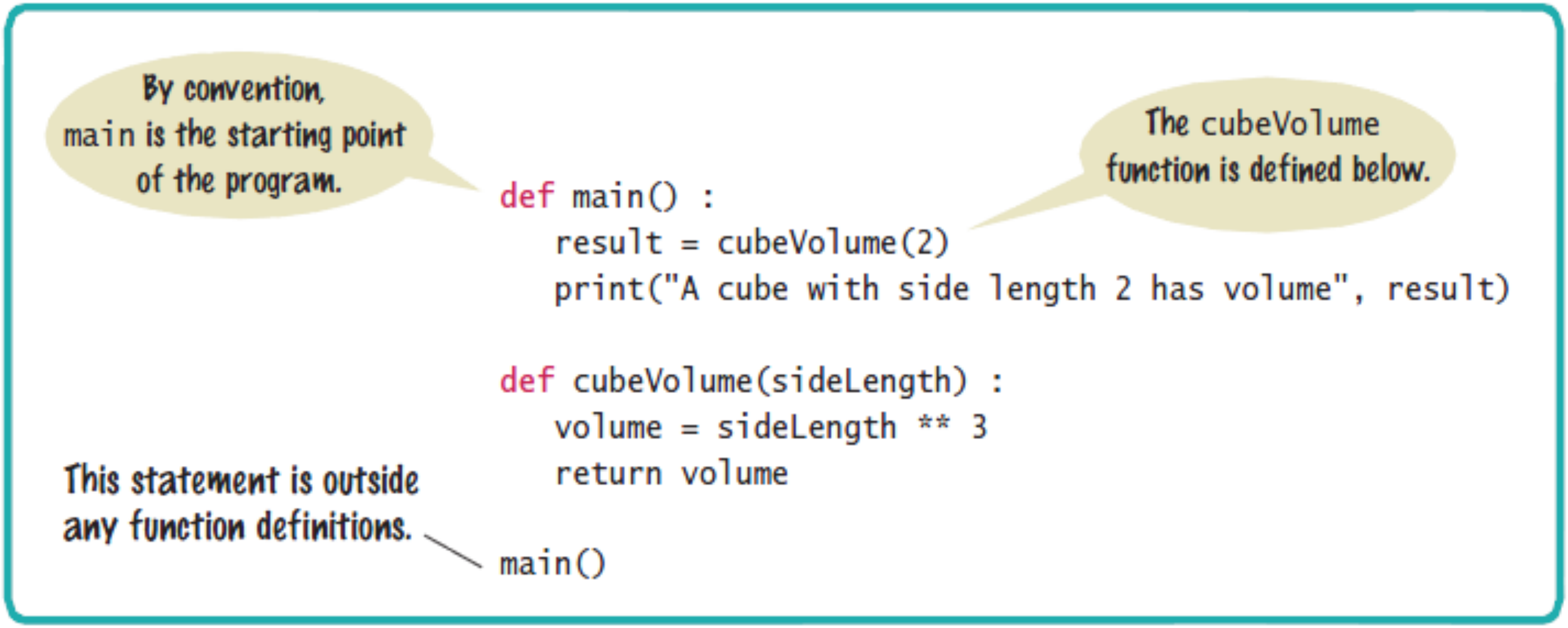
Cubes.py

- Open the file Cubes.py in Wing
- 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.
The cubeVolume function is defined below.
This statement is outside any function definitions.

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

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

main()
```

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:

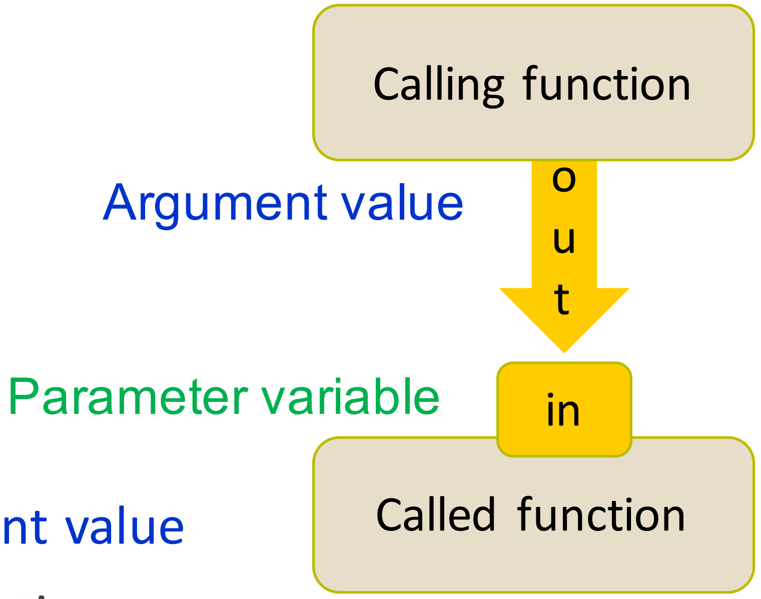
```
def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume",
      result)

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

main()
```

Parameter Passing

SECTION 5.3

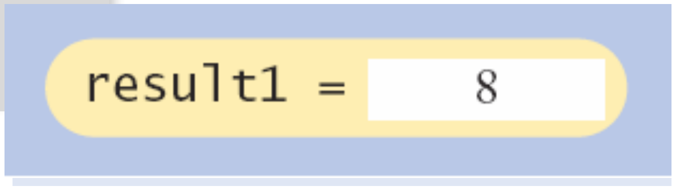
Parameter Passing

- 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’

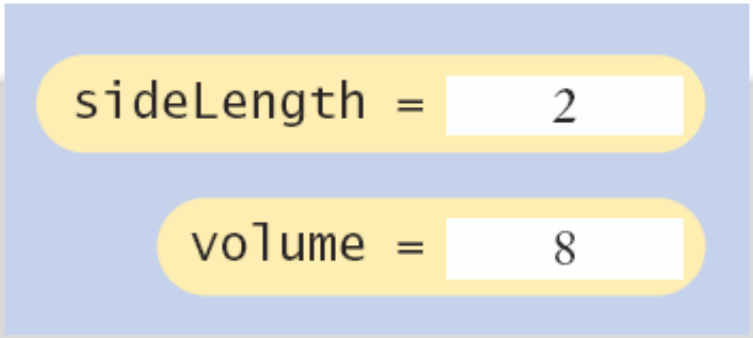


Parameter Passing Steps

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

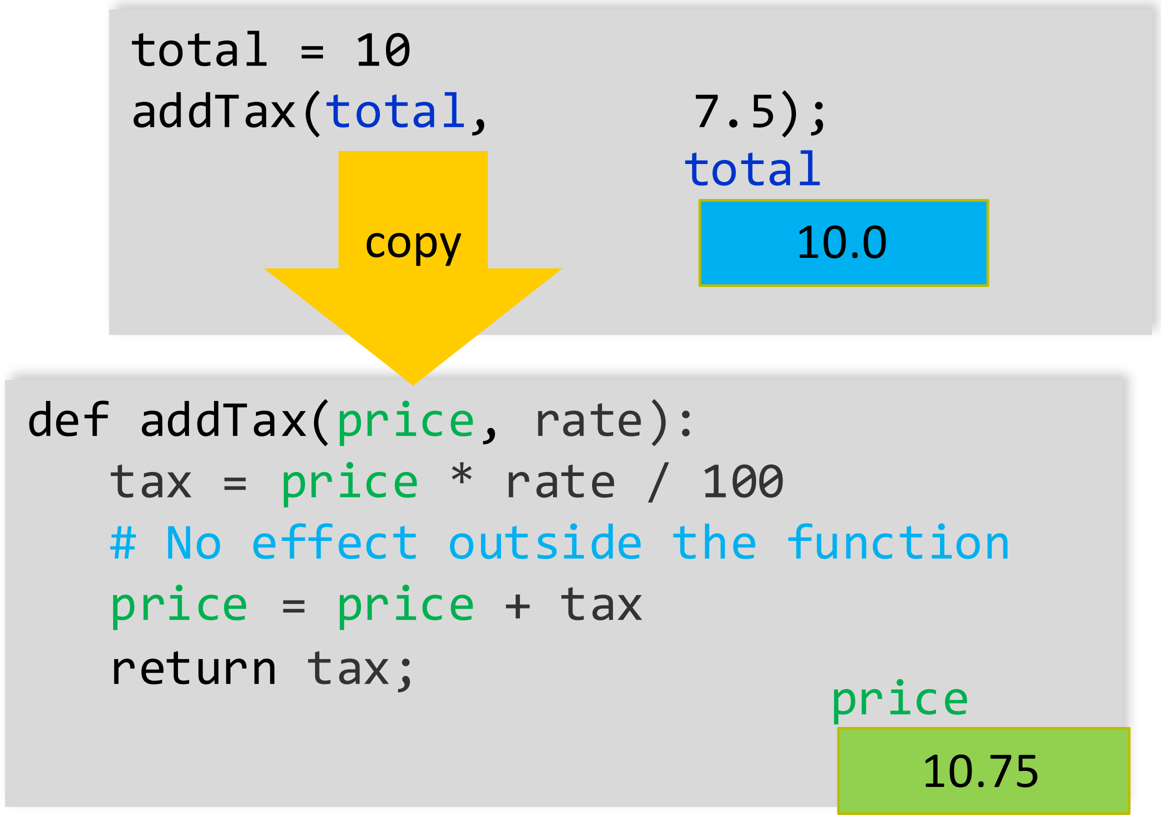


```
def cubeVolume(sideLength):
    volume = sideLength * 3
    return volume
```



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

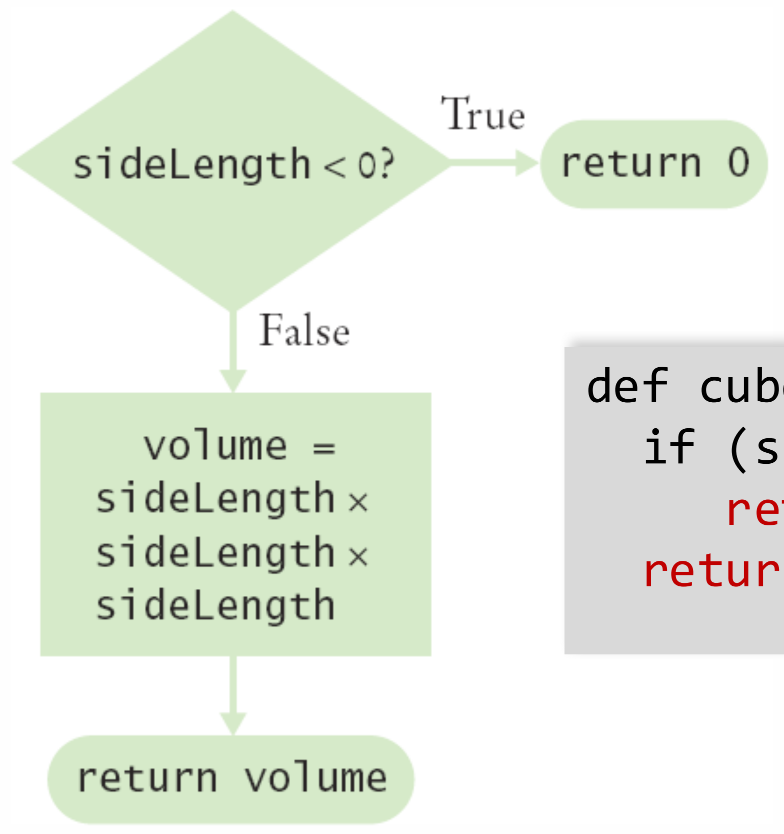
```
def cubeVolume (sideLength):
   volume = sideLength * 3
   return volume
```

$\underbrace{\texttt{return volume}}_{\text{return statement}}$

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
```

Implementing a Function: Steps

1. Describe what the function should do
   i. Provide a simple “liberal arts terms” description of what the functions does
   ii. “Compute the volume of a pyramid with a square base”
2. Determine a list of all of the functions inputs
   i. Make a list of ***all*** of the parameters that can vary
   ii. Do not be overly specific
3. Determine the types of the parameter variables and the return value

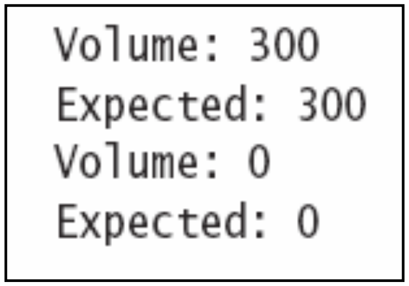
Implementing a Function: Steps

4) Write pseudocode for obtaining the desired result
   i. Express an mathematical formulas, branches and loops in pseudocode
5) Implement the function body

```
def pyramidVolume(height, baseLength) :
    baseArea = baseLength * baseLength
    return height * baseArea / 3
```

Implementing a Function: Steps

6) Test your function
   i. Design test cases and code



```
Volume: 300
Expected: 300
Volume: 0
Expected: 0
```

Pyramids.py

- Open the file pyramids.py
- Look at how the main function is set up to make the calls to `pyramidVolume` and print the expected results

Functions Without Return Values

SECTION 5.5

Functions Without Return Values

- functions are not required to return a value
  - No return statement is required
  - The function can generate output even when it doesn’t have a return value

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

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

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

Using return Without a Value

- You can use the return statement without a value
  - The function will terminate immediately!

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

Reusable Functions

SECTION 5.6

Problem Solving: Reusable Functions

- **Find repetitive code**
  - May have different values but same logic

```
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: "))
```

0 - 23

0 - 59

Write a ‘Parameterized’ Function

```
## Prompts a user to enter a value up to a given maximum until the user
provides
# a valid input.
# @param high an integer indicating the largest allowable input
# @return the integer value provided by the user (between 0 and high,
inclusive)
#
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
```

Readtime.py

- Open the file readtime.py
- Test the program with several inputs
  - How would you modify your project to use the readInBetween function?

An Alternate If Structure

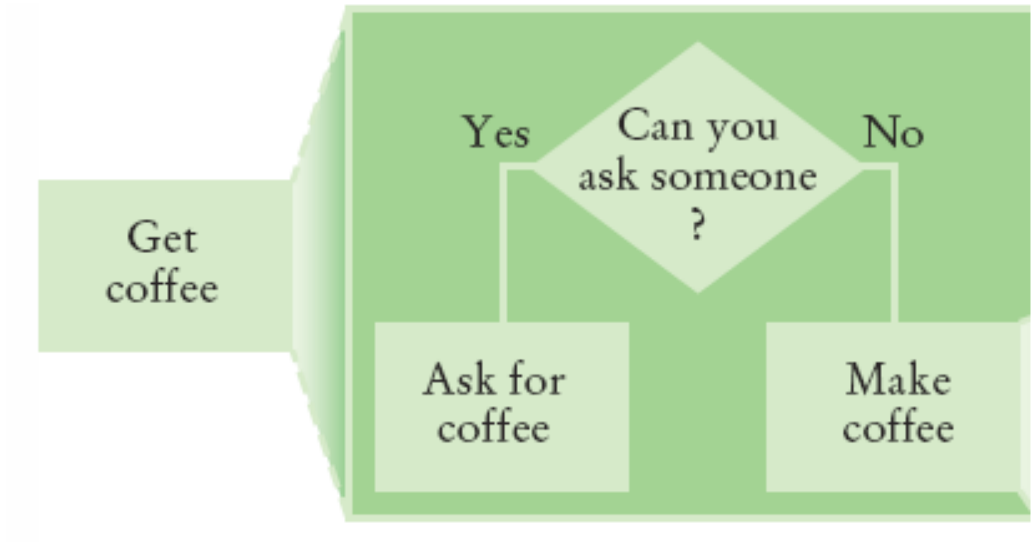
- Open the file earthquake.py
- The file contains two functions that solve the Richter scale problem from earlier this semester
  - The first uses an "`if – elif`" construct
  - The second uses single-line compound statements (Special Topic 5.1, p. 256)
  - This form of an `if` statement is very useful in functions that select and return a single value from a set of values

Stepwise Refinement

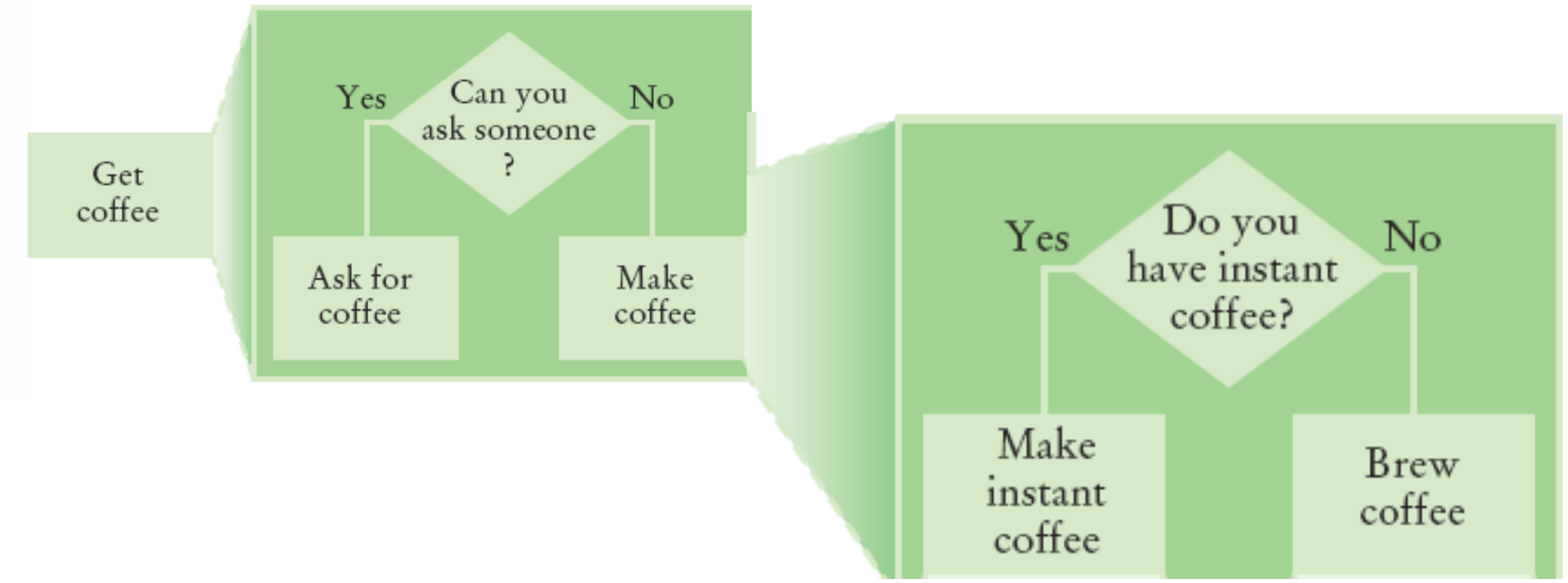
SECTION 5.7

Stepwise Refinement

- To solve a difficult task, break it down into simpler tasks
- Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve



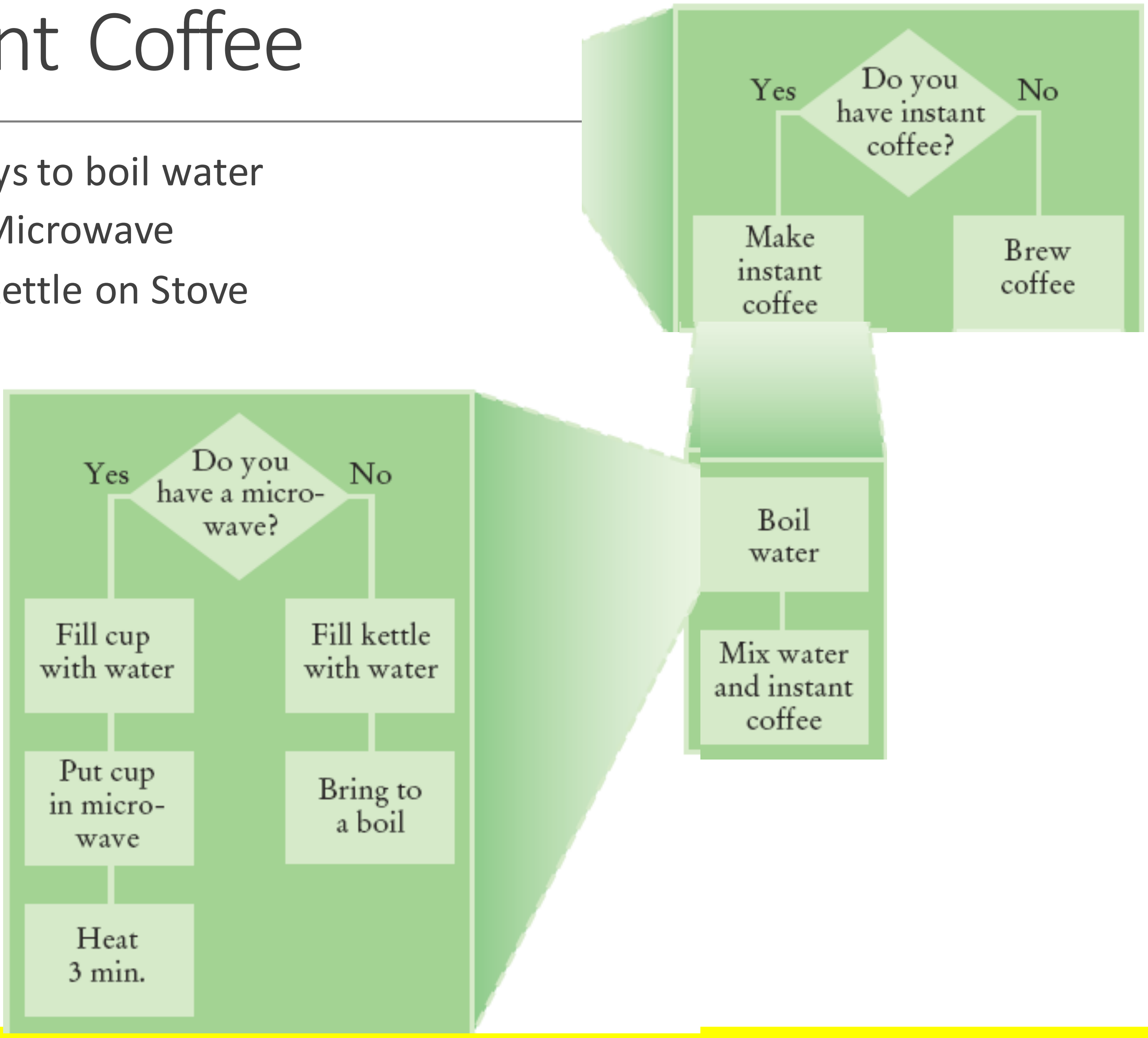
Get Coffee



- If you must make coffee, there are two ways:
  - Make Instant Coffee
  - Brew Coffee

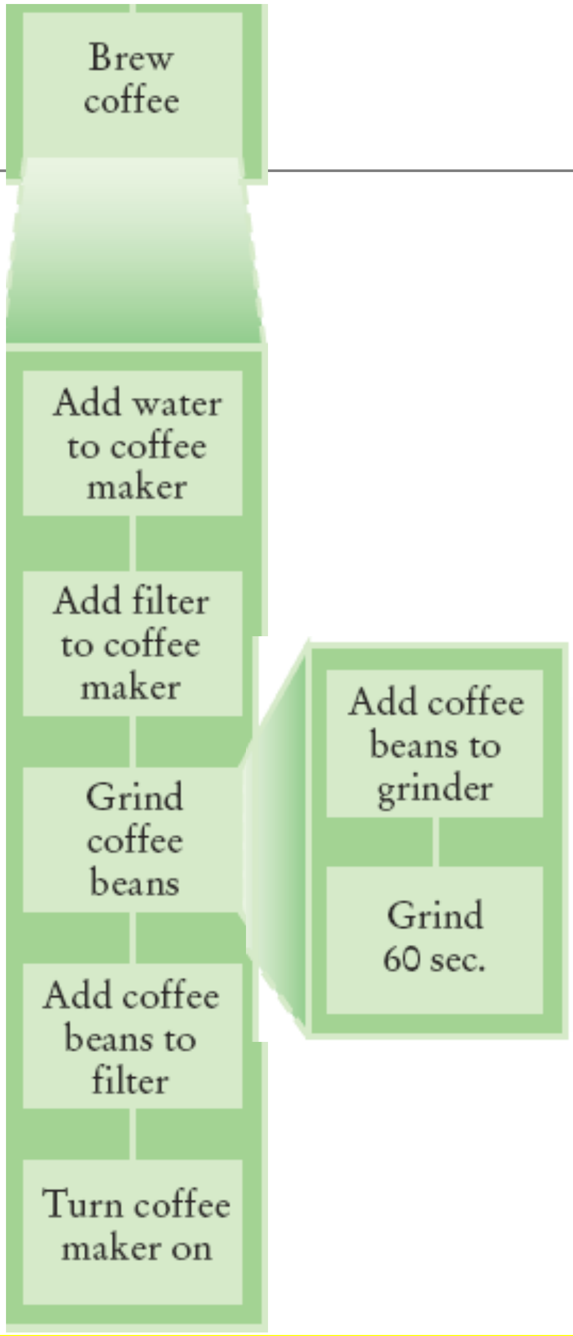
Instant Coffee

- Two ways to boil water
  1) Use Microwave
  2) Use Kettle on Stove



Brew Coffee

- Assumes coffee maker
  - Add water
  - Add filter
  - Grind Coffee
    - Add beans to grinder
    - Grind 60 seconds
  - Fill filter with ground coffee
  - Turn coffee maker on
- Steps are easily done


Brew coffee
Add water to coffee maker
Add filter to coffee maker
Grind coffee beans
Add coffee beans to filter
Turn coffee maker on
Add coffee beans to grinder
Grind 60 sec.

Stepwise Refinement Example

- When printing a check, it is customary to write the check amount both as a number (“$274.15”) and as a text string (“two hundred seventy four dollars and 15 cents”)
- Write a program to turn a number into a text string
- Wow, sounds difficult!
- Break it down
  - Let’s take the dollar part (274) and come up with a plan
  - Take an Integer from 0 – 999
  - Return a String
  - Still pretty hard…

Stepwise Refinement Example

- Take it digit by digit (2, 7, 4) – left to right
- Handle the first digit (hundreds)
  - If empty, we are done with hundreds
  - Get first digit (Integer from 1 – 9)
  - Get digit name (“one”, “two”, “three”…)
  - Add the word “hundred”
  - Sounds easy!
- Second digit (tens)
  - Get second digit (Integer from 0 – 9)
  - If 0, we are done with tens… handle third digit
  - If 1, … may be eleven, twelve... Teens… Not easy!
    - Let’s look at each possibility left (1x-9x)…

Stepwise Refinement Example

- If second digit is a 0
  - Get third digit (Integer from 0 – 9)
  - Get digit name (“”, “one”, “two”…) … Same as before?
  - Sounds easy!
- If second digit is a 1
  - Get third digit (Integer from 0 – 9)
  - Return a String (“ten”, “eleven”, “twelve”…)
- If second digit is a 2-9
  - Start with string “twenty”, “thirty”, “forty”…
  - Get third digit (Integer from 0 – 9)
  - Get digit name (“”, “one”, “two”…) … Same as before
  - Sounds easy!

Name the Sub-Tasks

- digitName
  - Takes an Integer from 0 – 9
  - Return a String (“”, “one”, “two”…)
- tensName (second digit >= 20)
  - Takes an Integer from 0 – 9
  - Return a String (“twenty”, “thirty”…) plus
    - digitName(third digit)
- teenName
  - Takes an Integer from 0 – 9
  - Return a String (“ten”, “eleven”…)

Write Pseudocode

```
part = number (The part that still needs to be converted)
name = “” (The name of the number)
If part >= 100
  name = name of hundreds in part + " hundred"
  Remove hundreds from part
If part >= 20
  Append tensName(part)  to name
  Remove tens from part
Else if part >= 10
  Append teenName(part)  to name
  part = 0
If (part > 0)
  Append digitName(part)  to name
```

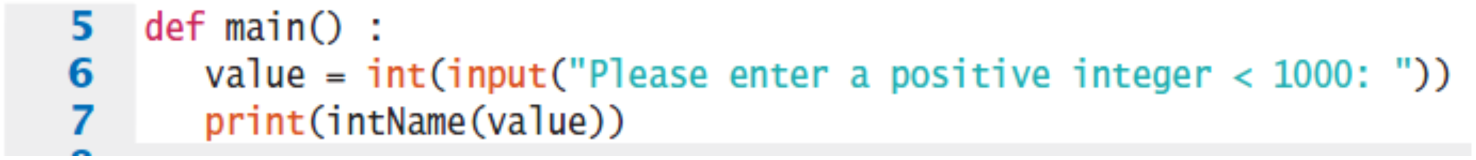
Identify functions that we can use (or re-use!) to do the work

Plan The Functions

- Decide on name, parameter(s) and types and return type
- def intName (number):
  - Turns a number into its English name
  - Returns a String that is the English description of a number (e.g., “seven hundred twenty nine”)
- def digitName (digit):
  - Return a String (“”, “one”, “two”…)
- def tensName (number):
  - Return a String (“twenty”, “thirty”…) plus
    - Return from digitName(thirdDigit)
- def teenName (number):
  - Return a String (“ten”, “eleven”…)

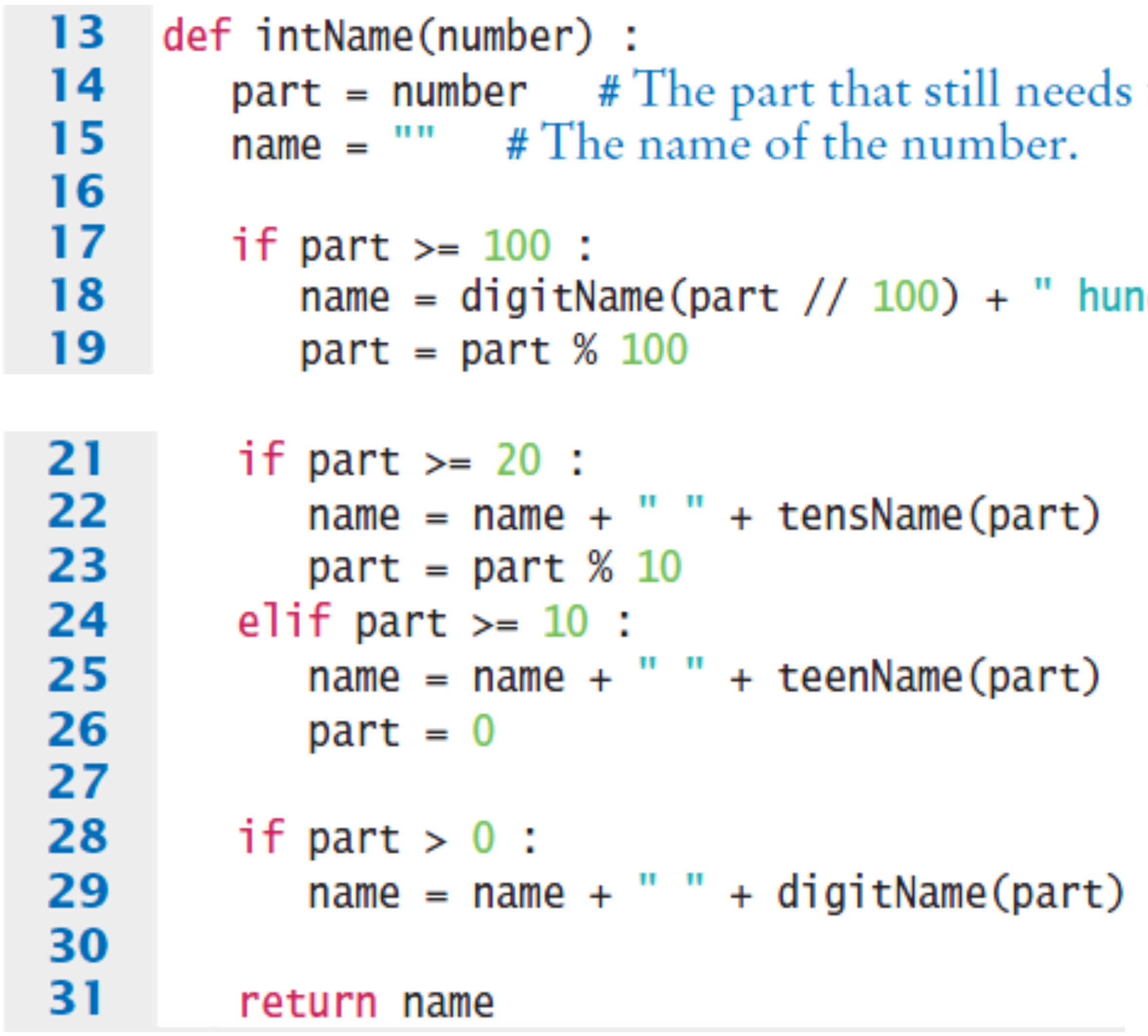
Convert to Python: intName Function

- Open the file intname.py in Wing
- main calls intName
  - Does all the work
  - Returns a String
- Uses functions:
  - tensName
  - teenName



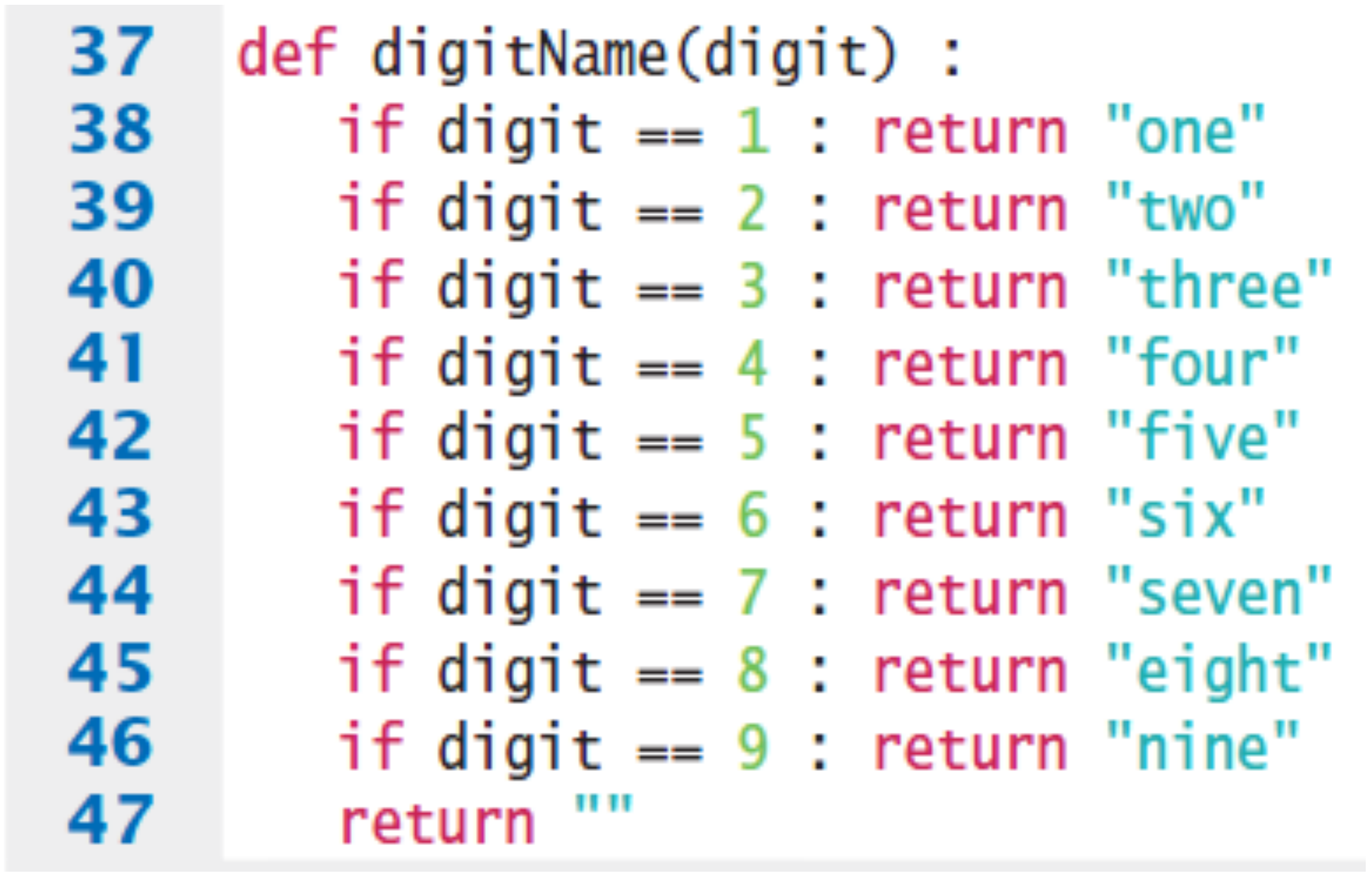
```
def main() :
   value = int(input("Please enter a positive integer < 1000: "))
   print(intName(value))
```

intName



```
def intName(number) :
   part = number    # The part that still needs to be converted.
   name = ""    # The name of the number.

   if part >= 100 :
      name = digitName(part // 100) + " hundred"
      part = part % 100

   if part >= 20 :
      name = name + " " + tensName(part)
      part = part % 10
   elif part >= 10 :
      name = name + " " + teenName(part)
      part = 0

   if part > 0 :
      name = name + " " + digitName(part)

   return name
```

digitName

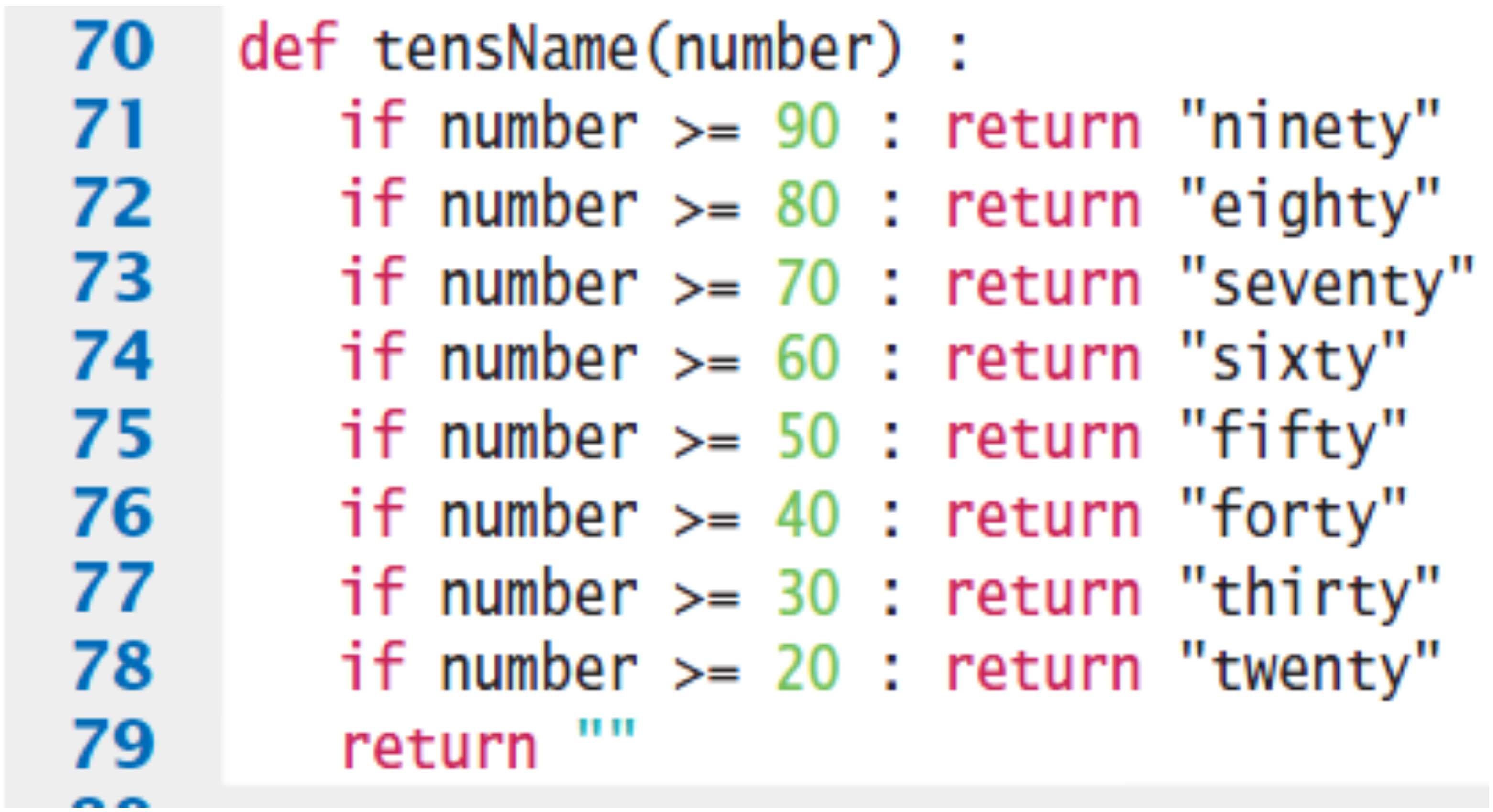


```python
def digitName(digit) :
    if digit == 1 : return "one"
    if digit == 2 : return "two"
    if digit == 3 : return "three"
    if digit == 4 : return "four"
    if digit == 5 : return "five"
    if digit == 6 : return "six"
    if digit == 7 : return "seven"
    if digit == 8 : return "eight"
    if digit == 9 : return "nine"
    return ""
```

teenName

```
def teenName(number) :
    if number == 10 : return "ten"
    if number == 11 : return "eleven"
    if number == 12 : return "twelve"
    if number == 13 : return "thirteen"
    if number == 14 : return "fourteen"
    if number == 15 : return "fifteen"
    if number == 16 : return "sixteen"
    if number == 17 : return "seventeen"
    if number == 18 : return "eighteen"
    if number == 19 : return "nineteen"
    return ""
```

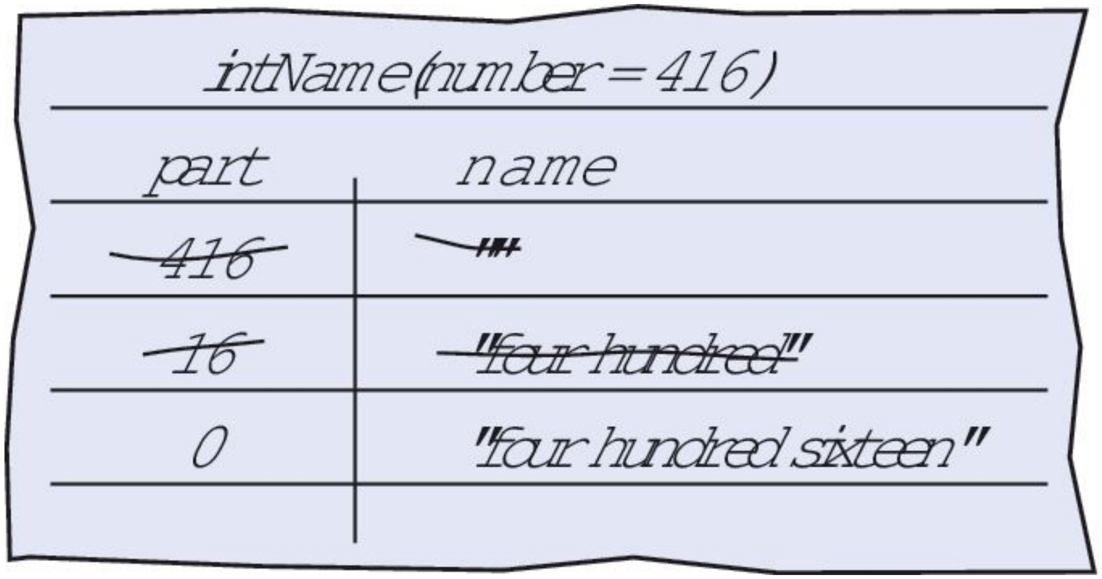
tensName



```
def tensName(number) :
    if number >= 90 : return "ninety"
    if number >= 80 : return "eighty"
    if number >= 70 : return "seventy"
    if number >= 60 : return "sixty"
    if number >= 50 : return "fifty"
    if number >= 40 : return "forty"
    if number >= 30 : return "thirty"
    if number >= 20 : return "twenty"
    return ""
```

Programming Tips

- Keep functions short
  - If more than one screen, break into ‘sub’ functions
- Trace your functions
  - One line for each step
  - Columns for key variables
- Use Stubs as you write larger programs
  - Unfinished functions that return a ‘dummy’ value



intName(number = 416)

part	name
~~416~~	~~""~~
~~16~~	~~"four hundred"~~
0	"four hundred sixteen"

Variable Scope

SECTION 5.8

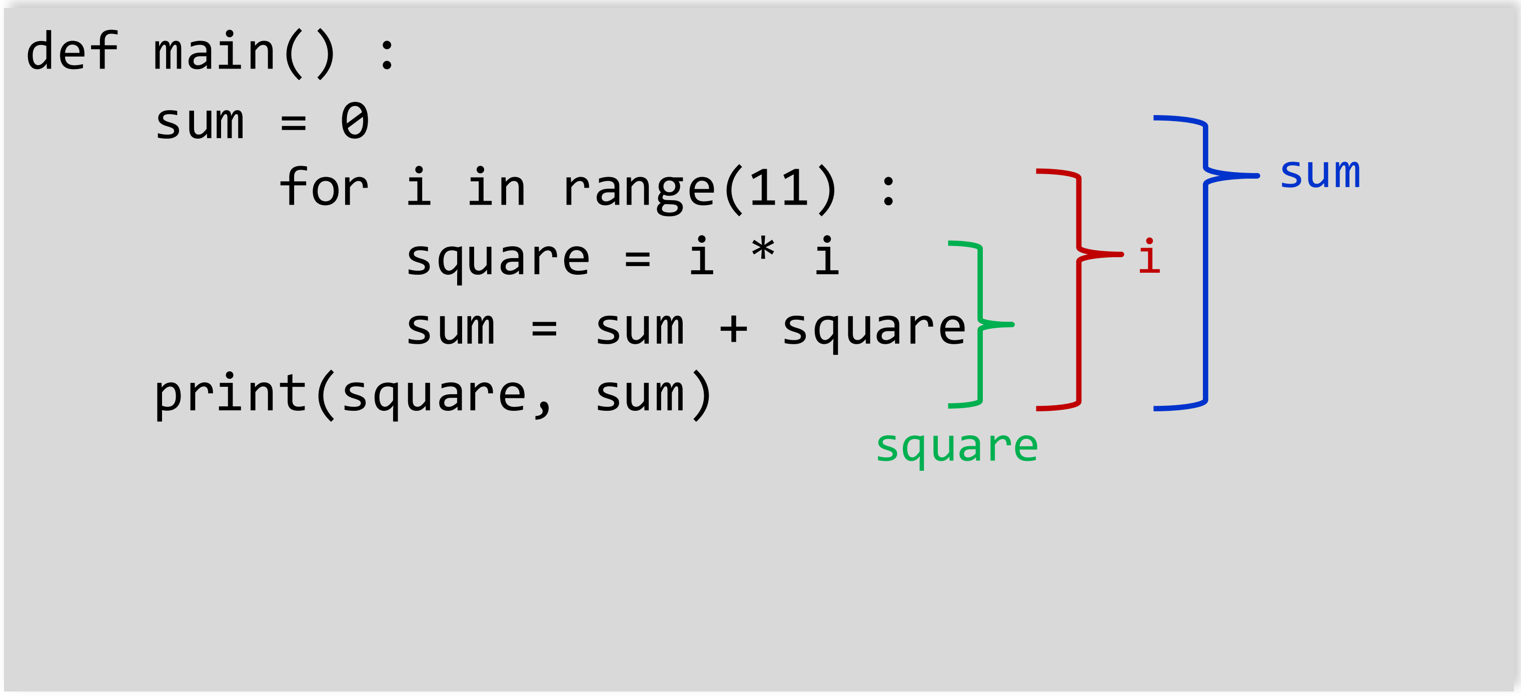
Variable Scope

- Variables can be declared:
  - Inside a function
    - Known as ‘local variables’
    - Only available inside this function
    - Parameter variables are like local variables
  - Outside of a function
    - Sometimes called ‘global scope’
    - Can be used (and changed) by code in any function
- How do you choose?

The scope of a variable is the part of the program in which it is visible

Examples of Scope

- sum, square & i are local variables in main



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

Local Variables of functions

- Variables declared inside one function are not visible to other functions
  - sideLength is local to main
  - Using it outside main will cause a compiler error

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


def cubeVolume():
  return sideLength * sideLength * sideLength # ERROR
```

Re-using Names for Local Variables

- Variables declared inside one function are not visible to other functions
  - result is local to square and result is local to main
  - They are two different variables and do not overlap
  - This can be very confusing

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

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

Global Variables

- They are variables that are defined outside functions
- A global variable is visible to all functions that are defined after it
- However, any function that wishes to use a global variable must include a global declaration

Example Use of a Global Variable

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

```
balance = 10000    # A global variable
def withdraw(amount) :
    # This function intends to access the
    # global ‘balance’ variable
    global balance
    if balance >= amount :
        balance = balance - amount
```

Programming Tip

- There are a few cases where global variables are required (such as `pi` defined in the math module), but they are quite rare
- Programs with global variables are difficult to maintain and extend because you can no longer view each function as a “black box” that simply receives arguments and returns a result
- Instead of using global variables, use function parameter variables and return values to transfer information from one part of a program to another

Graphics: Building an Image Processing Toolkit

SECTION 5.9

Building Your Own Tools

- As we learned earlier, Python’s standard libraries contain a large collection of function and classes
  - You import a module (containing functions and / or classes) into your program if you need it
- What do you do when the standard libraries don’t have what you need?
  - You create your own collection of tools and organize them into modules or a *software toolkit*

Creating Your Toolkit

- The tools of functions in the toolkit should all be related
  - The function names should be easy to remember
  - The parameters should be as consistent as possible
- Place the functions in a separate file
- Import them into your programs as needed

Recursive Functions

SECTION 5.10

Recursive Functions

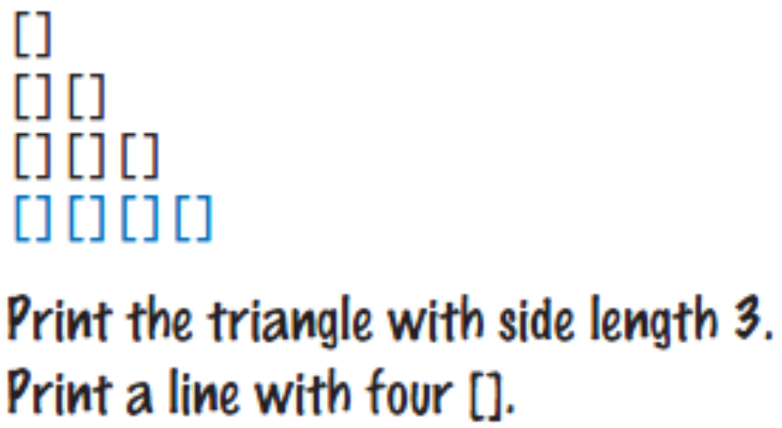
- A recursive function is a function that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs

Recursive Triangle Example

```
def printTriangle(sideLength) :
    if sideLength < 1 : return
    printTriangle(sideLength - 1)
    print("[]" * sideLength)
```

Special Case

Recursive Call

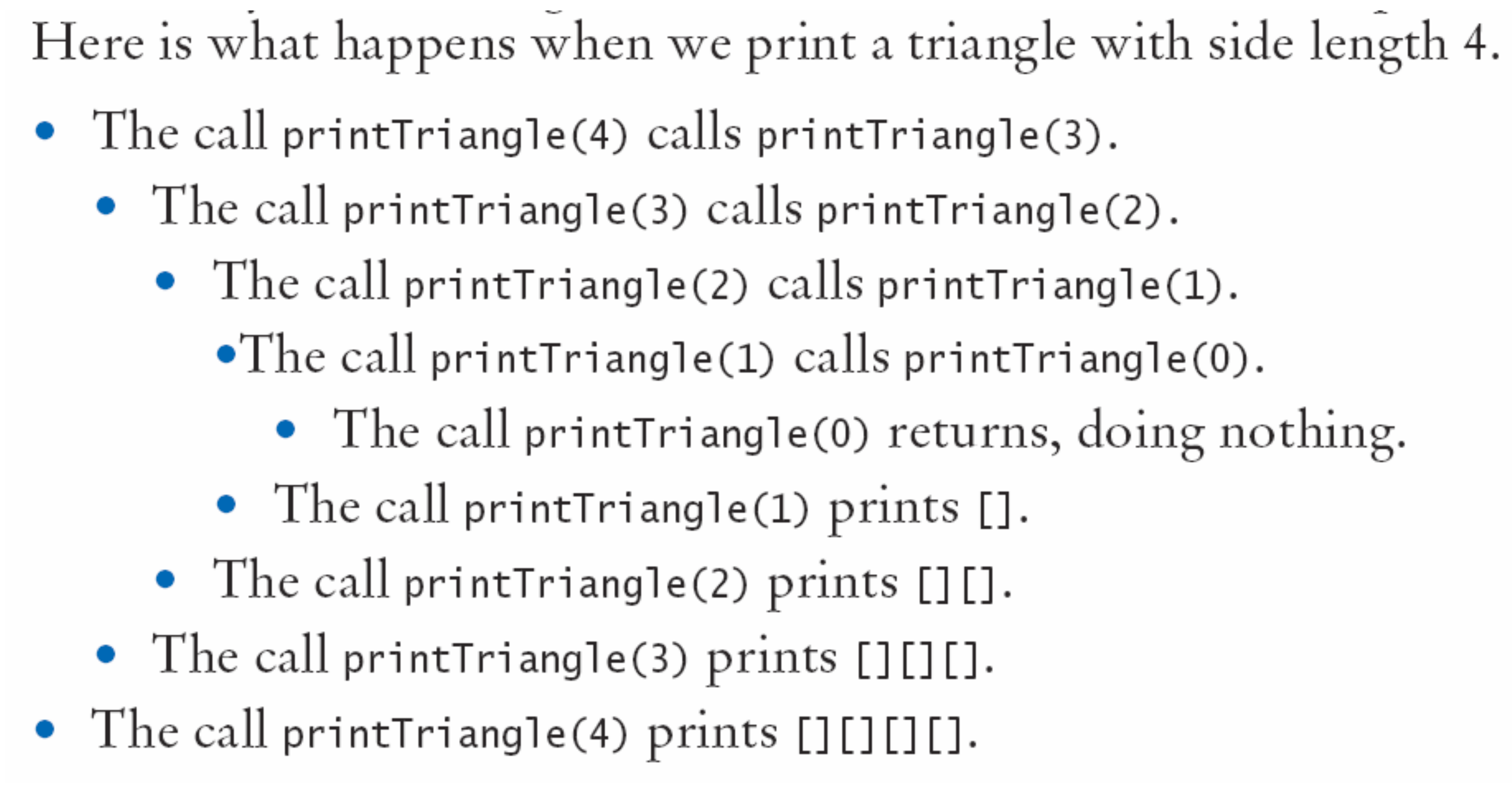


```
[]
[][]
[][][]
[][][][]
```

Print the triangle with side length 3.
Print a line with four [].

- The function will call itself (and not output anything) until sideLength becomes < 1
- It will then use the return statement and each of the previous iterations will print their results
  - 1, 2, 3 then 4

Recursive Calls and Returns


Here is what happens when we print a triangle with side length 4.

- The call `printTriangle(4)` calls `printTriangle(3)`.
    - The call `printTriangle(3)` calls `printTriangle(2)`.
        - The call `printTriangle(2)` calls `printTriangle(1)`.
            - The call `printTriangle(1)` calls `printTriangle(0)`.
                - The call `printTriangle(0)` returns, doing nothing.
            - The call `printTriangle(1)` prints `[]`.
        - The call `printTriangle(2)` prints `[][]`.
    - The call `printTriangle(3)` prints `[][][]`.
- The call `printTriangle(4)` prints `[][][][]`.

A Second Example

- Open the file digits.py
- This program computes the sum of the digits in a number (n)
  - We solved this last chapter in Section 4.2
  - We will use n = 1729 as our example
- Our algorithm was:
  - Remove the last digit by computing n // 10 and add the remainder to our total
  - To use recursion we can use the recursive function:
    - digitsum(n // 10) + n % 10
    - Our special case is n == 0 to terminate the recursion

Summary

Summary: Functions

- 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 value, as well as any special requirements
- Parameter variables hold the arguments supplied in the function call

Summary: Function Returns

- 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: Scope

- 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 scopes do not overlap
  - You can use the same variable name within different functions since their scope does not overlap
  - Local variables declared inside one function are not visible to code inside other functions

Summary: Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
  - For recursion to terminate, there must be special cases for the simplest inputs
  - The key to finding a recursive solution is reducing the input to a simpler input for the same problem
  - When designing a recursive solution, do not worry about multiple nested calls
    - Simply focus on reducing a problem to a slightly simpler one