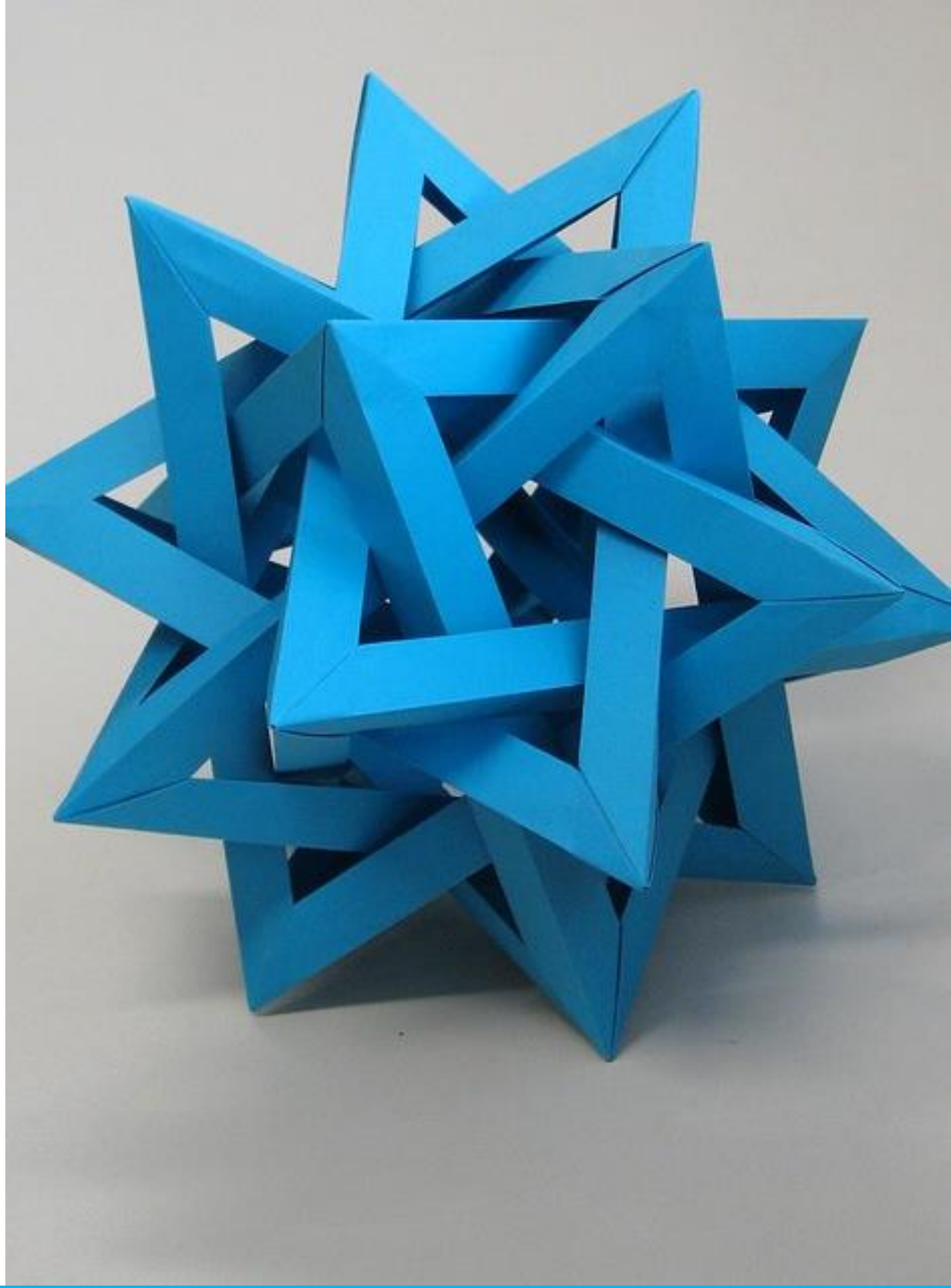
POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Unit P5: Functions

STRUCTURING YOUR CODE, FUNCTIONS, PARAMETERS, RETURN VALUE

Chapter 5



This Photo by Unknown Author is licensed under CC BY-SA

Unit 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

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

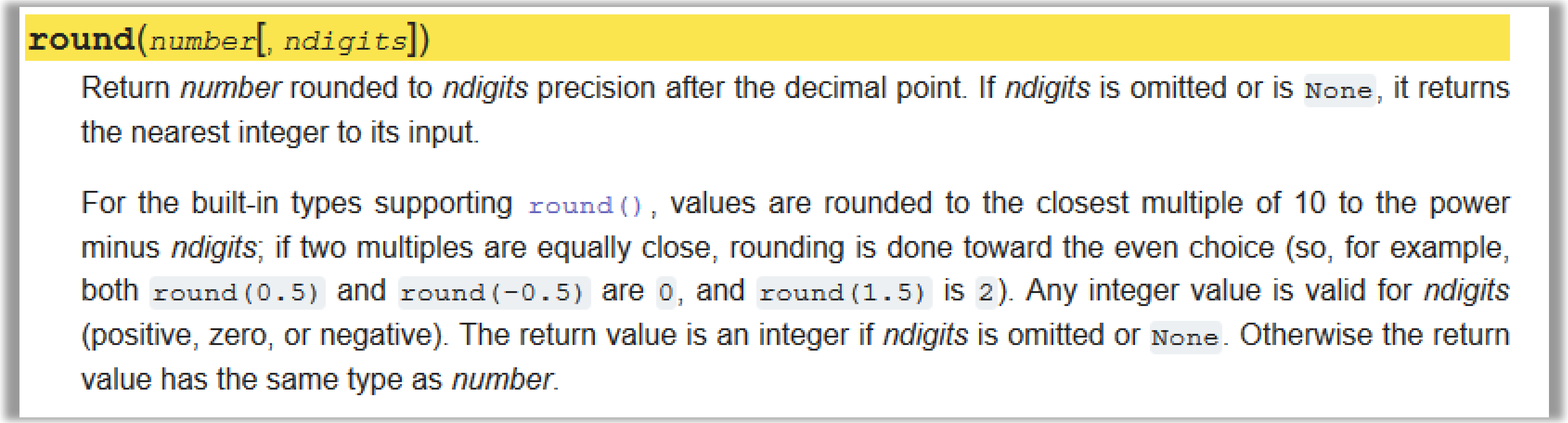
Using the process of stepwise refinement, you will be able to break up complex tasks into sets of cooperating functions

Functions as Black Boxes

📘 5.1

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



round(*number*[, *ndigits*])

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise the return value has the same type as *number*.

https://docs.python.org/3/library/functions.html#round

Calling Functions

- You call a function in order to execute its instructions

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

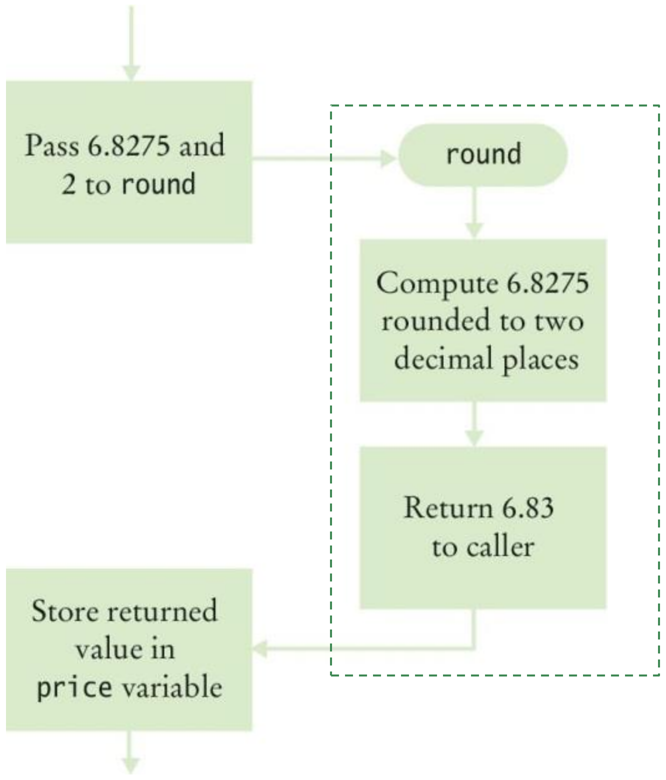
- You call a function in order to execute its instructions

```
price = round(6.8275, 2) # Sets price 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
- When the function terminates, the computed result is **returned** by the function and may be used in an expression (e.g., assigned to `price`)
- After the value has been used, your program resumes execution

Calling Functions (3)

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


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

This is not our code! The function is defined in the standard library.

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

Function Return Value

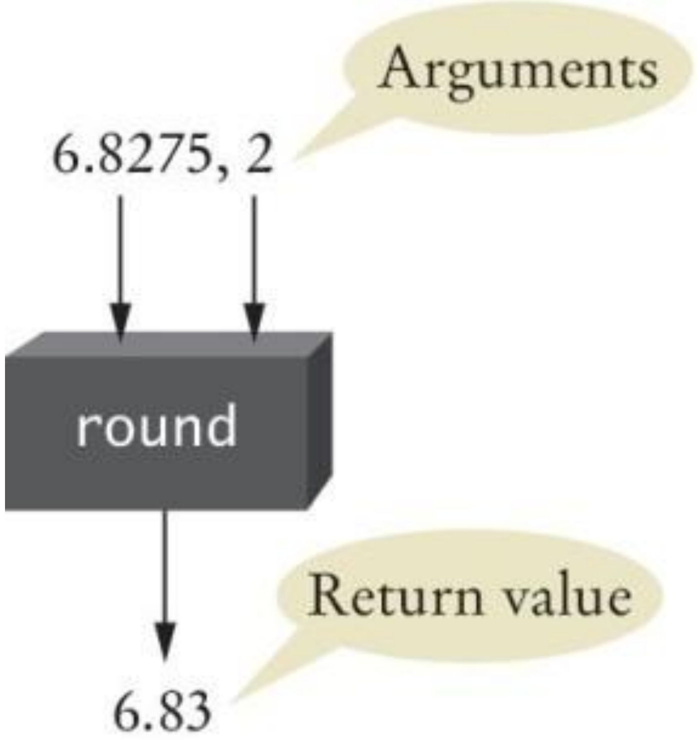
- The “output” that the round function computes is called the return value
- Functions return only one value
  - For multiple values, return a list or a tuple (see later…)
  - Some functions do not return any value
- The return value of a function is returned to the point in your program where the function was called

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

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

The round Function as a Black Box

- Use functions like ‘black boxes’
  - Pass the function what it needs to do its job
  - Receive the answer
- **Don’t need to know how they are implemented!**
- Real-life analogy: A thermostat is a ‘black box’
  - Set a desired temperature
  - Turns on heater or AC (Air Conditioner) 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 AC?



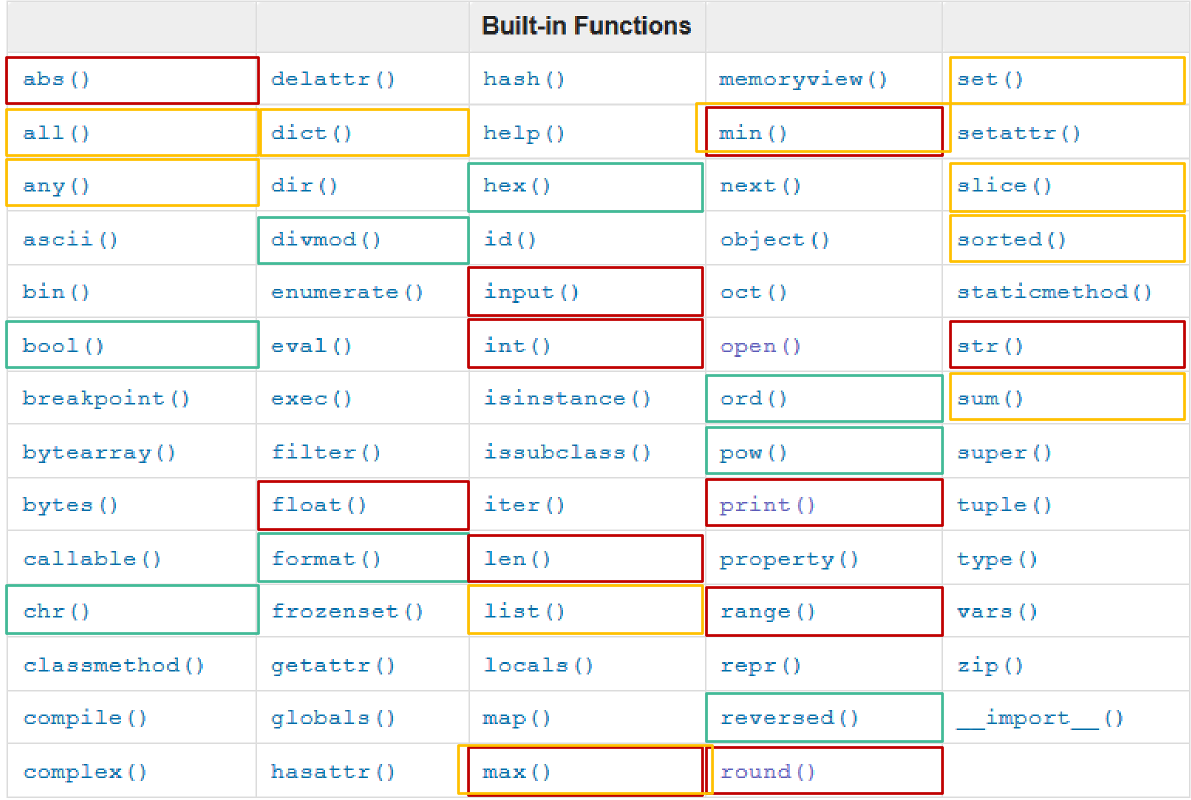
The round Function as a Black Box

- You may wonder… how does the `round` function perform 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

- When you design your own functions, you will want to make them appear as black boxes
    - Even if you are the only person working on a program, you want to use them as simple black boxes in the future, and let other programmers do the same

Where to Find Library Functions (1)

- Built-In functions in the Standard Library
  - https://docs.python.org/3/library/functions.html



		Built-in Functions		
🟥 abs()	delattr()	hash()	memoryview()	🟨 set()
🟨 all()	🟨 dict()	help()	🟥🟨 min()	setattr()
🟨 any()	dir()	🟩 hex()	next()	🟨 slice()
ascii()	🟩 divmod()	id()	object()	🟨 sorted()
bin()	enumerate()	🟥 input()	oct()	staticmethod()
🟩 bool()	eval()	🟥 int()	open()	🟥 str()
breakpoint()	exec()	isinstance()	🟩 ord()	🟨 sum()
bytearray()	filter()	issubclass()	🟩 pow()	super()
bytes()	🟥 float()	iter()	🟥 print()	tuple()
callable()	🟩 format()	🟥 len()	property()	type()
🟩 chr()	frozenset()	🟨 list()	🟥 range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	🟩 reversed()	__import__()
complex()	hasattr()	🟥🟨 max()	🟥 round()	

🟥 Important (already used)

🟩 useful (take a look)

🟨 will use later (with lists, dicts)

Where to Find Library Functions (2)

- Inside Modules in the Standard Library
  - https://docs.python.org/3/library/
  - https://docs.python.org/3/py-modindex.html
  - More than 200 modules, with many functions in each
  - Interesting ones: `string, math, random, statistics, csv, json,` …
- Remember to import:

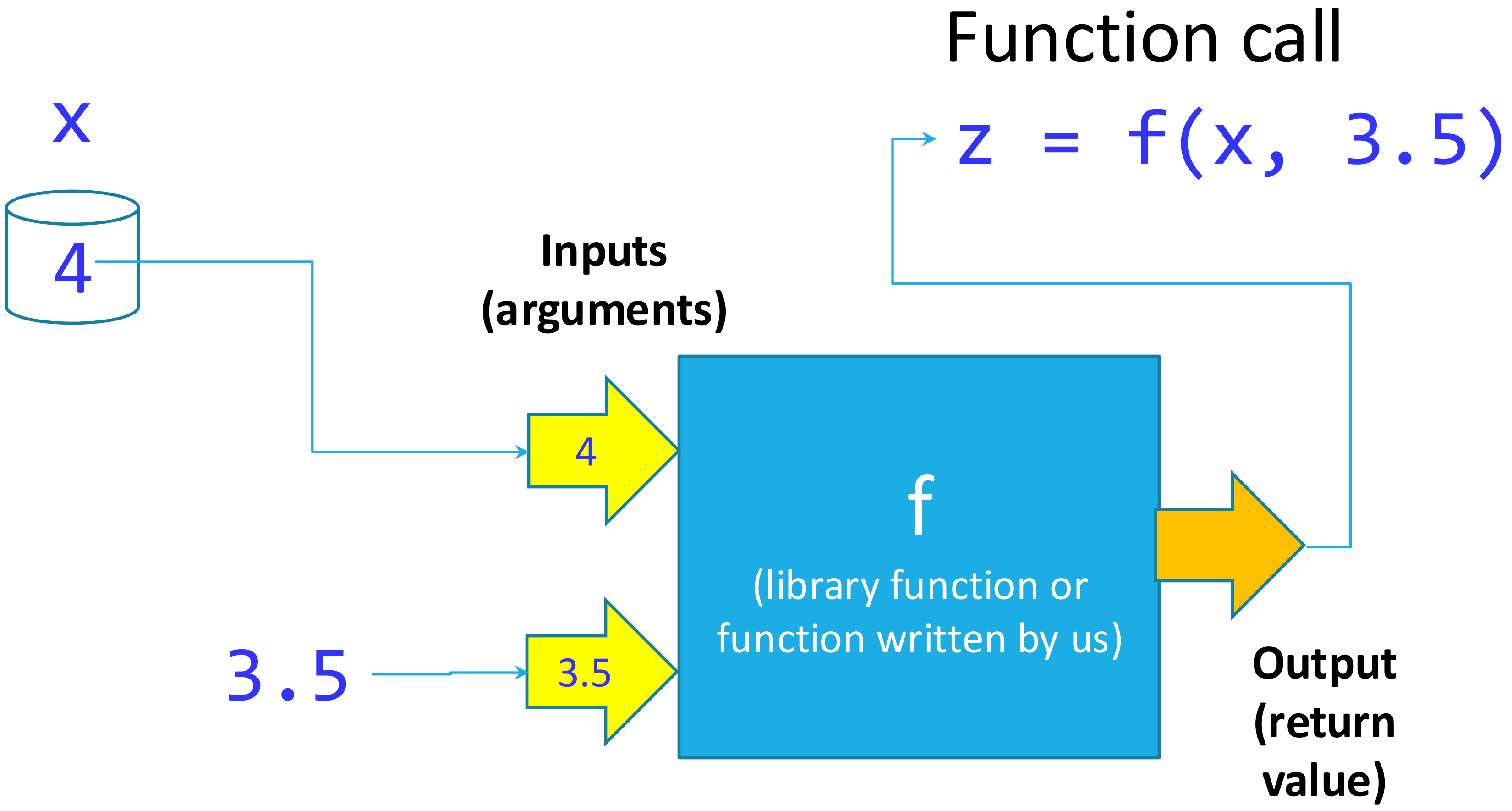
```
import module
```

- Or:

```
from module import function
```

- Thousands of extra modules (not in the standard library) can be **downloaded and installed** if needed (not used in this course…)

Summary


Function call
z = f(x, 3.5)
x
4
Inputs (arguments)
4
3.5
3.5
f
(library function or function written by us)
Output (return value)

Implementing and Testing Functions

📘 5.2

Implementing Functions

- Besides using existing functions, it is useful to be able to create new ones…
- We have to define two key elements
  1. **Interface:**
     - *Function name, arguments, return value*
  2. **Implementation:**
     - ***What does the function do***
     - *Instructions to produce the output given the arguments…*
- Then, we will be able to use (call) the function.

Example:

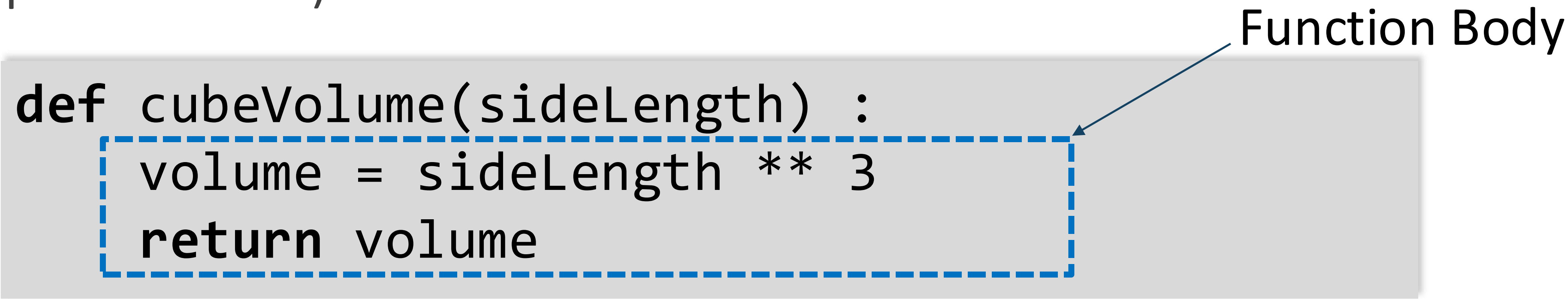
- A function to calculate the volume of a cube
  - What does it need to do its job?
  - What does it answer with?
- When writing (‘defining’) this function
  - Pick a name for the function (`cubeVolume`)
  - Define a variable name for each incoming argument
    - (`sideLength`) – list of parameter variables
  - 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

Implementing the Cube Function

- The def keyword starts a new block of code (compound instruction).
- Inside this block, we will write the instructions that compose the function **body** (implementation)

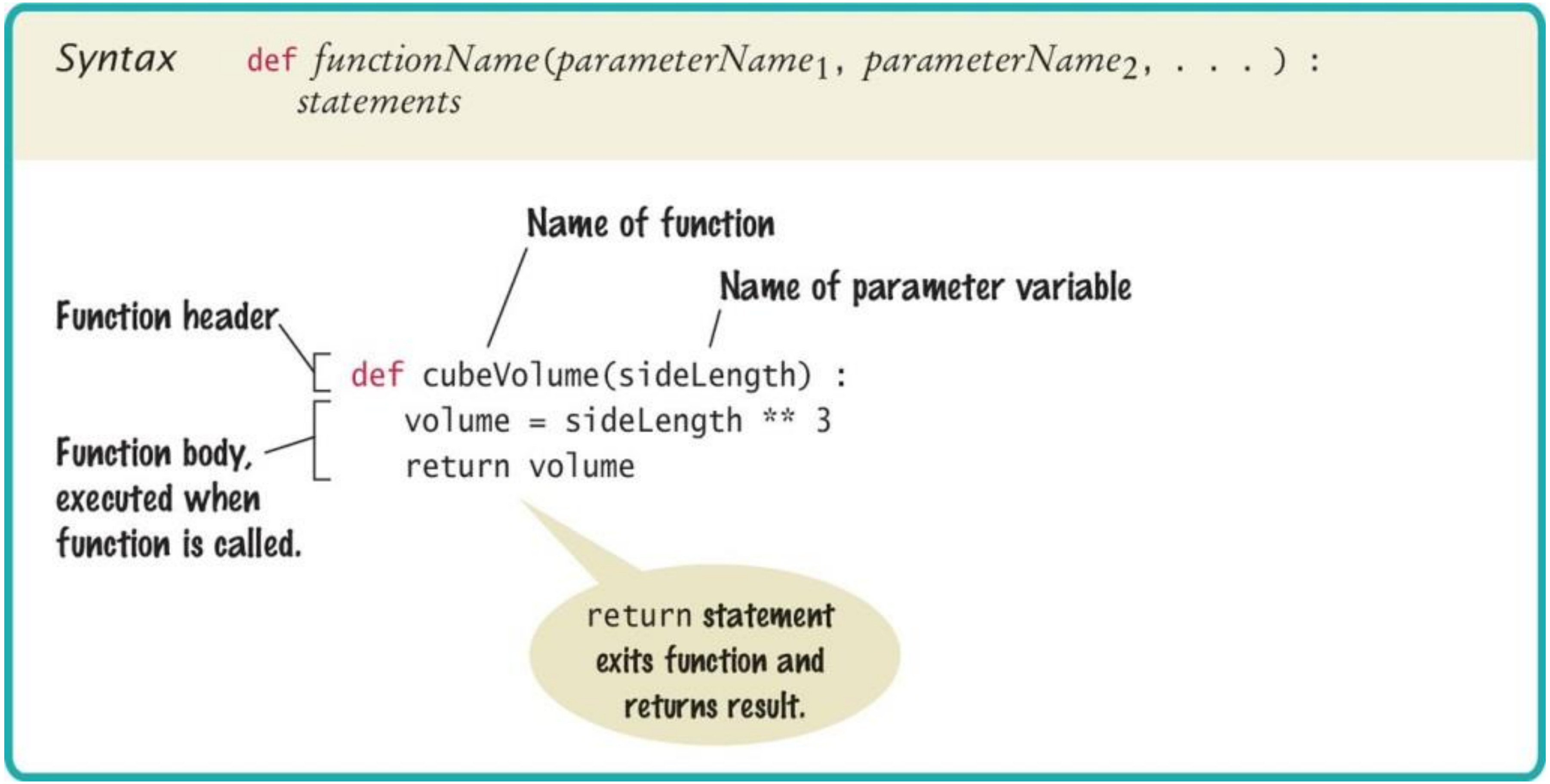


```
def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume
```

- Inside the body, the arguments **can be used as normal variables**.
- If the function needs to return a result to the caller, we add the instruction:

return <value>

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.

Testing (Using) a Function

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

- In order to use the function, your program should contain
  - The definition of the function
  - Statements that **call the function and use the result**

Calling/Testing the Cube Function

- **Implementing** the function (function definition)

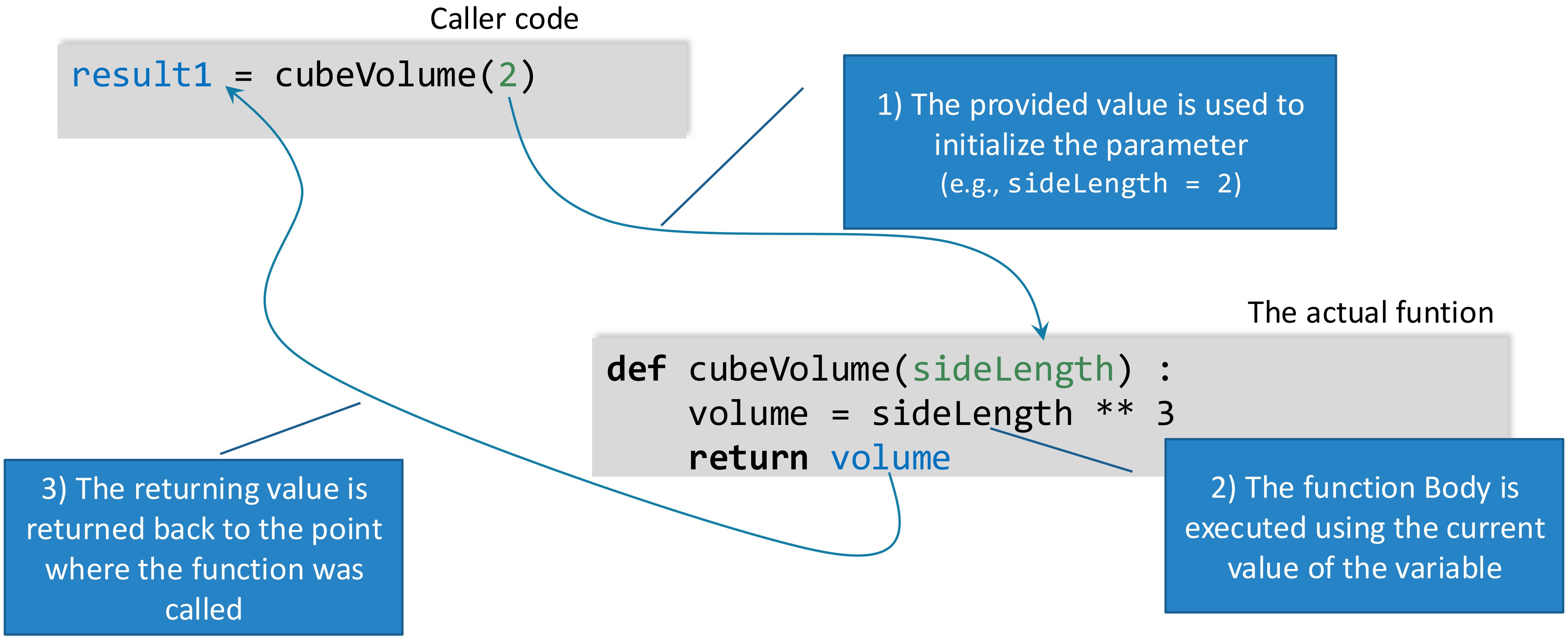
```
def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume
```

- **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)
Print("A cube with side length 4 has volume”, cubeVolume(4))
```

Calling a Function

Let’s test it with the debugger!!



Caller code

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

1) The provided value is used to initialize the parameter (e.g., `sideLength = 2`)

The actual funtion

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

2) The function Body is executed using the current value of the variable

3) The returning value is returned back to the point where the function was called

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
  - Doesn’t know what function to **call**

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)  # 1
    print("A cube with side length 2 has volume",
      result)

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

main() # 2
```

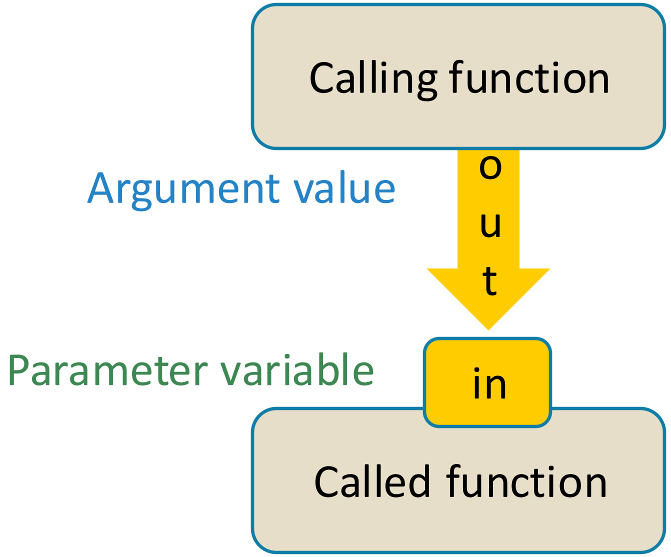
- In #1, the function `main` is just defined (not yet executed). It will be called in #2, that is after the definition of `cubeVolume`.

Parameter Passing

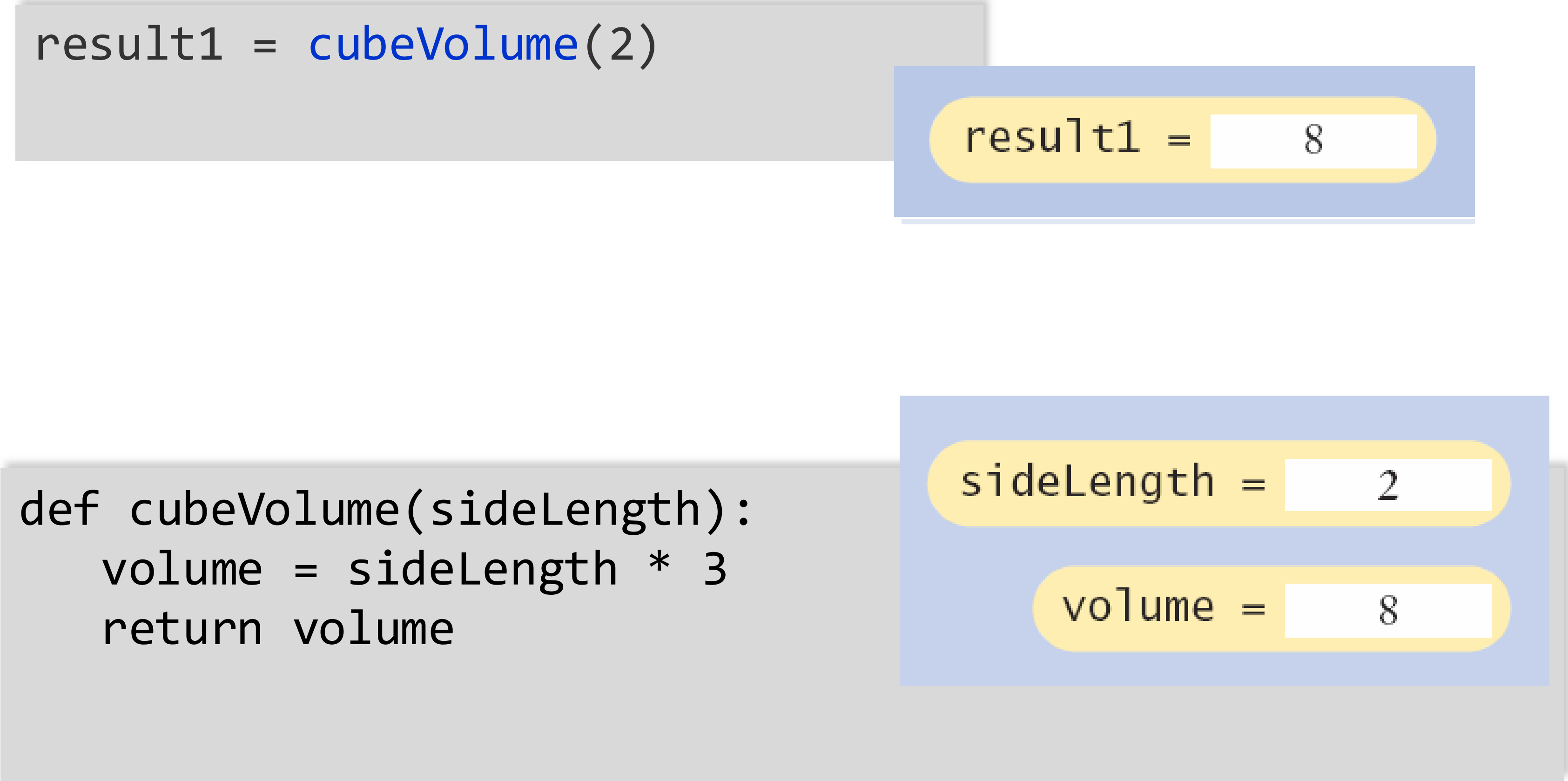
📘 5.3

Parameter Passing

- Parameter variables (called ***formal*** parameters) receive the argument values (***effective*** or ***actual*** parameters) supplied in the function call
- The argument value may be:
  - The current contents of a variable
  - A ‘literal’ value: 2, `3.14`, `'hello'`
- The parameter variable is:
  - *Initialized* with the value of the argument value
  - *Used as a variable* inside the called function



Parameter Passing Steps

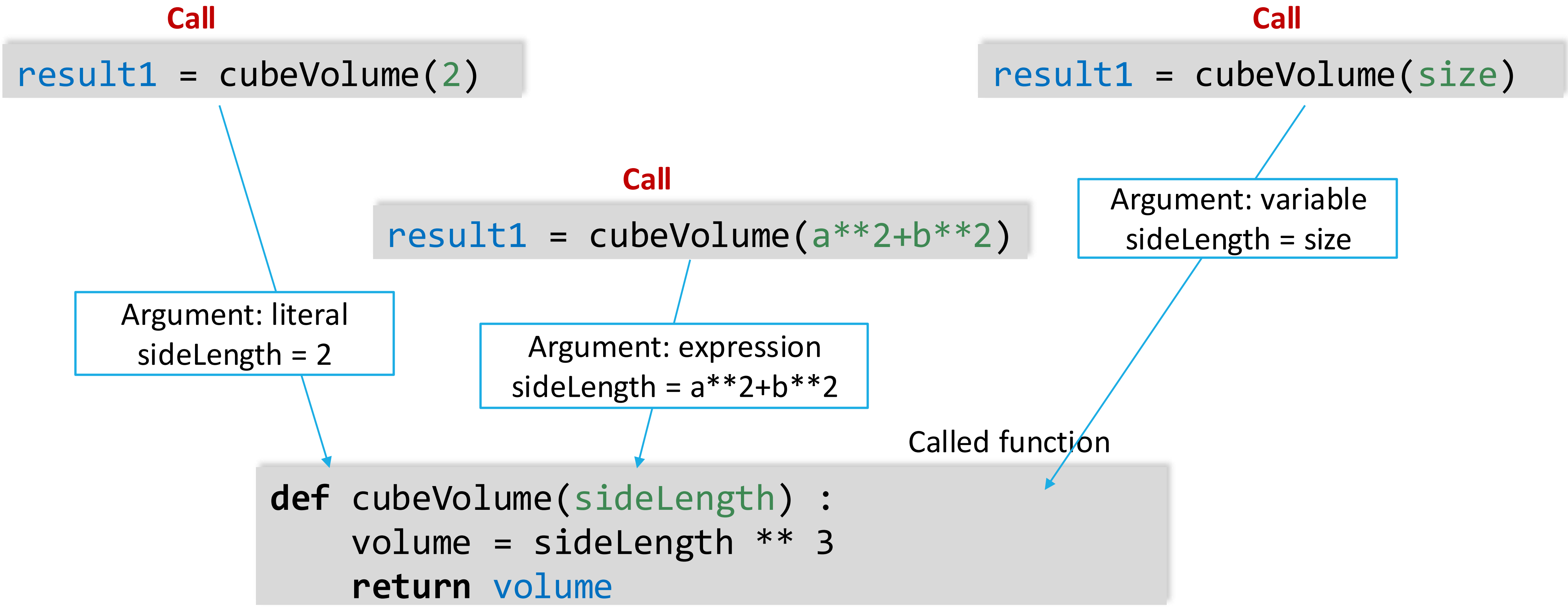


```
result1 = cubeVolume(2)
```

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

See it live on PythonTutor:
http://pythontutor.com/live.html#mode=edit

Arguments when calling Functions



Call

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

Argument: literal
sideLength = 2

Call

```
result1 = cubeVolume(a**2+b**2)
```

Argument: expression
sideLength = a**2+b**2

Call

```
result1 = cubeVolume(size)
```

Argument: variable
sideLength = size

Called function

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

Common Error

- Trying to modify parameter variables
- For many types of data, modifying the value of a parameter Variable will **not affect the caller.**
  - **IMMUTABLE:** int, float, string, tuples…
  - **When we modify the value inside the function, we're effectively creating a new copy**
- Example:
  - Called function (`addTax`) modifies `price`
  - The `total` variable in the caller function is unaffected
  - `total` remains equal to `10` after the function call

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

total

10.0

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

price

10.75

But this is different for other types of data like lists (**mutable types**, see later)!!

Programming Tip

- 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:

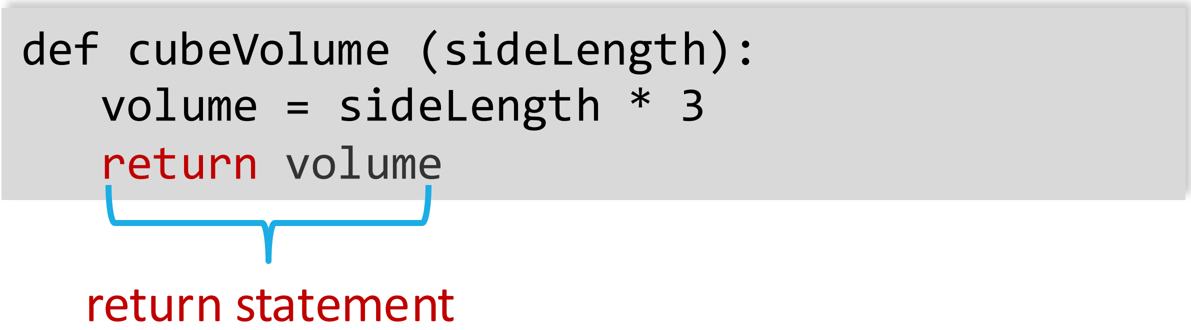
```
def totalCents(dollars, cents) :
    result = dollars * 100 + cents
    return result
```

Return Values

📘 5.4

Return Values

- Functions can return **one value** (or **no values**)
  - Add a `return` statement that returns a value
  - A `return` statement does two things:
    - Immediately terminates the function
    - 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 the result of an expression

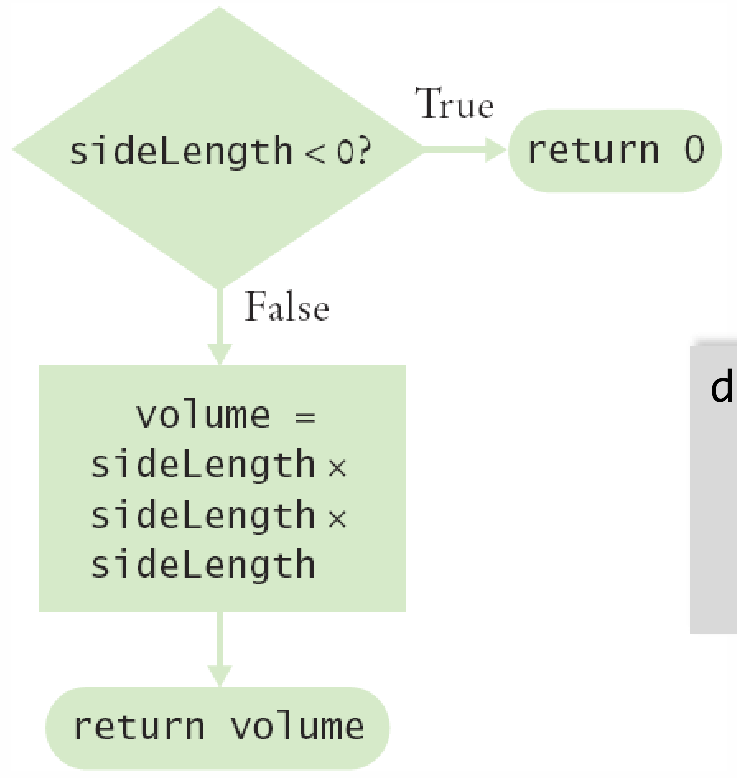
Returning multiple values

- Only **one value** may be returned by a function
- If you need to return more than one value, you may return a **tuple**, containing the values
- Example:
  - `return (x, y)`
  - Build a tuple `(x, y)`
  - Return it

We'll see the details later!!

Multiple return Statements

- A function can use multiple return statements
  - But every branch should lead the function to encounter 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):
  - Storing the function result in a variable
  - Returning the variable’s value 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 less than 0
    - No return statement for this condition
  - 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
```

Exercise

- Let’s build a function to determine if its argument is a prime number
- Let’s use the algorithm seen in class and “encapsulate” it in a function called: `prime()`

Functions Without Return Values

📘 5.5

Functions Without Return Values

- Functions are not required to return a value
  - **Typical example: functions that print something to the console**
- In this case, you can use a return statement without value

```
return   # no value specified
```

- Or **omit** the return keyword. If the return keyword is not encountered during the execution of a function, it is equivalent to having an empty `return` after the last statement of the function

Using return Without a Value

- Example:

Definition

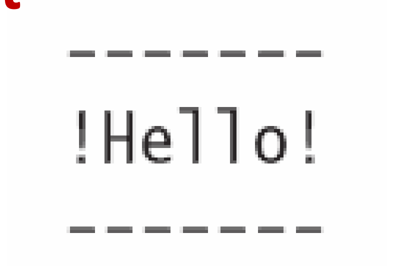
```
def boxString(contents) :
    n = len(contents) :
    print("-" * (n + 2))
    print("!" + contents + "!")
    print("-" * (n + 2))
    return
```

No return in the function is equivalent to a return in the last line.

Call

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

Result



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

Using return Without a Value

- The return keyword without a value can be used also in other places in the function
  - The function will terminate immediately!

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

The `main` function 📘 5.2

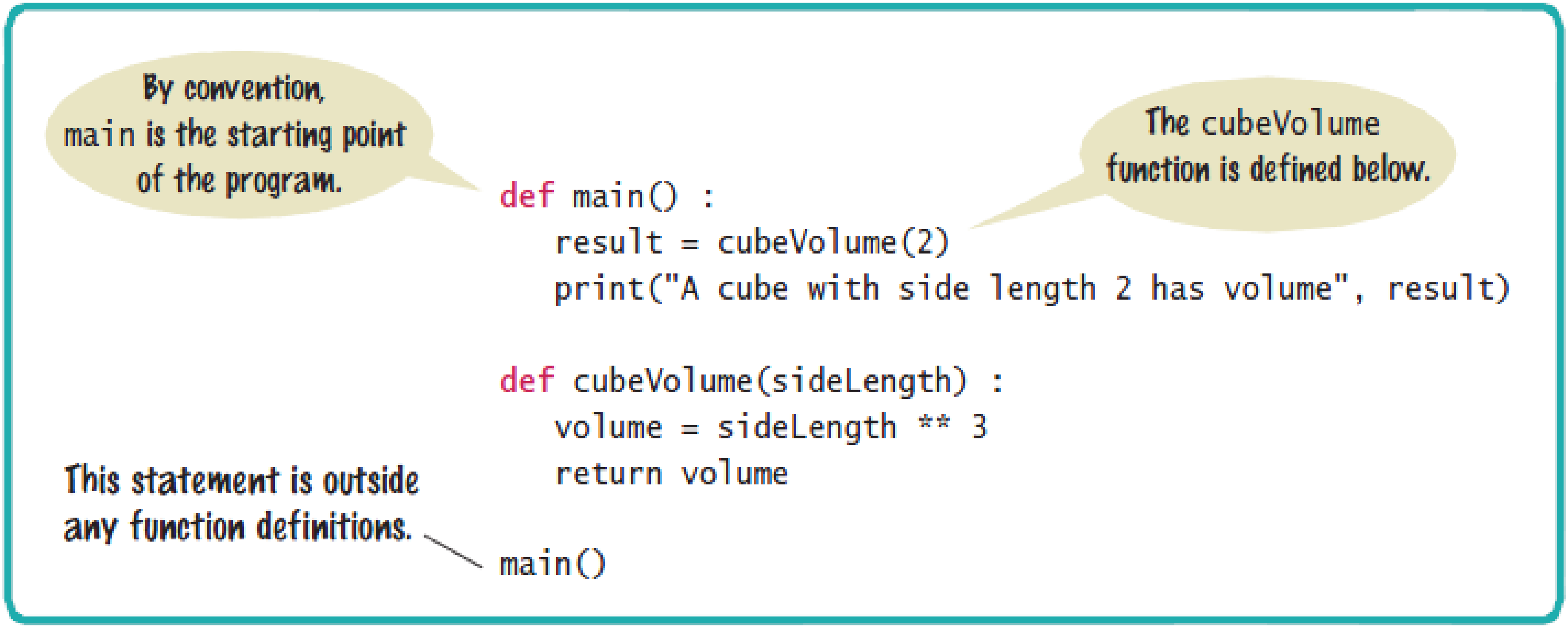
The `main` Function

- 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`’ (**by convention**) since it is the mandatory function name used by other common languages (C/C++)

- 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.

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

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

main()
```

This statement is outside any function definitions.

Exercise

- Given two integers **n, m,** given as input, compute their binomial coefficient C(n, m)
- Analysis:
  - Formula: C(n,m) = n! / (m! * (n-m)!)
  - How do we compute the factorial?
    - Definition n! = n * (n-1) *.... * 1
  - Constraint: we must have **n>m**
- Let's build:
  - A function **fact(x)** to compute the factorial
  - A function **binom(x,y)** to compute the binomial coefficient
  - A **main()** function.

Variable Scope

📘 5.8

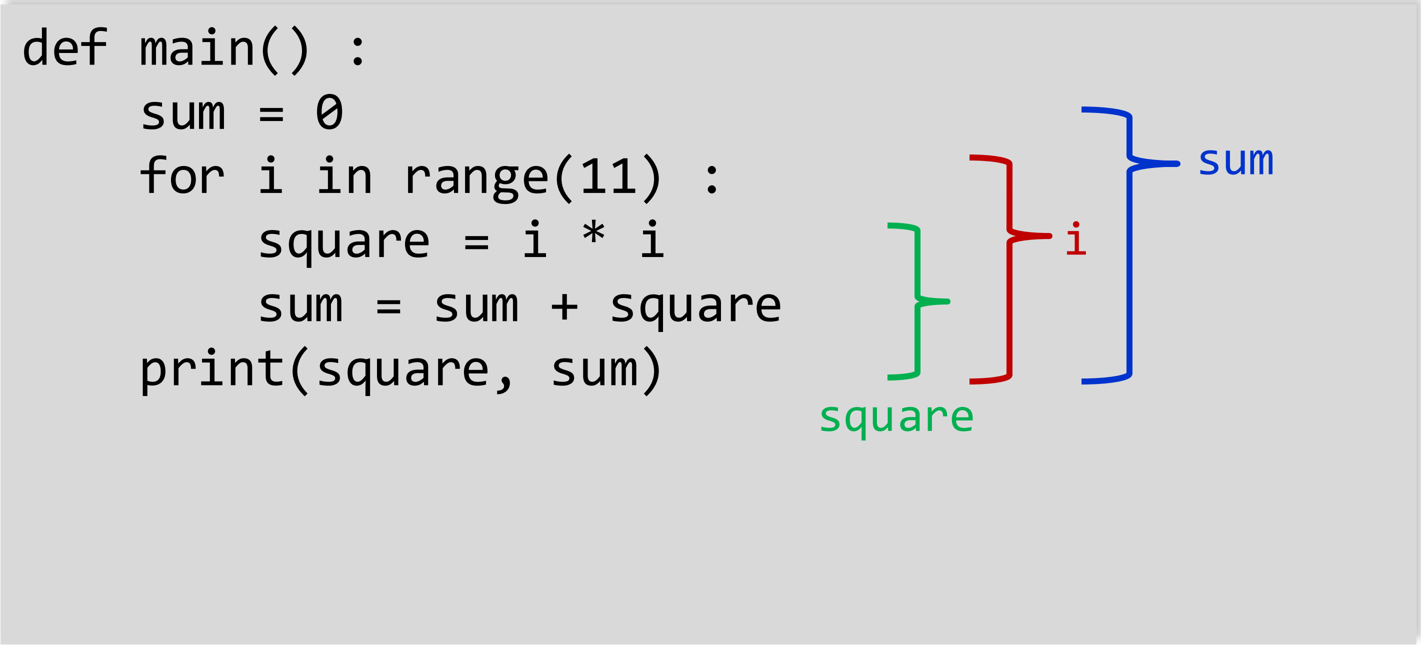
Variable Scope

- 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

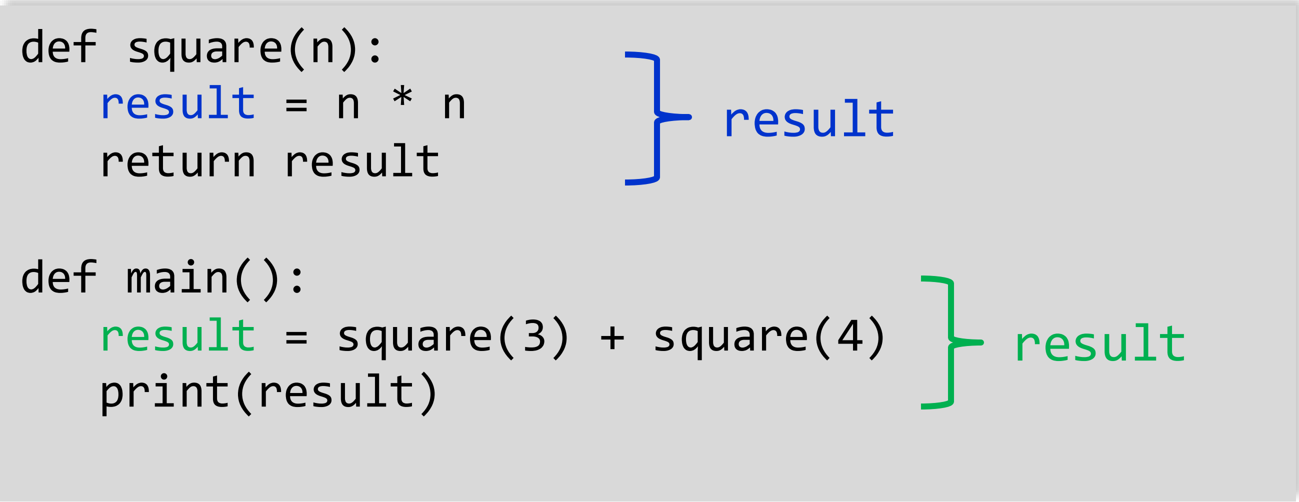
- Variables declared inside one function are not visible to other functions
  - sideLength is local to main
  - Using it outside main will cause a compiler error

```
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, even if they have the same name



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

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

Global Variables

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

Not a good idea…
better avoid it

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
```

Not a good idea… better avoid it

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

Stepwise Refinement

📘 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

Example

- Write a program that **generates a random password** with a specified length
- The password must contain at least 1 digit and 1 special character:
  - For simplicity, let’s decide it will contain **exactly one** digit and **one** special character.
  - Assume that special characters are + – * / ? ! @ # $ % &
- The other characters will be letters, and again for simplicity, we’ll use only **lowercase letters** from the English alphabet.

Program Structure

- How do we proceed?
  - Organize the problem as a set of sub-problems
  - Solve each sub-problem with a function
- **main()**
  - Read the desired length of the password l
  - **Create the password**
  - Print it
- Create the password: **makePassword()**
  - Generate a random sequence of lowercase letters of length **l** → function **initPassword()**
  - Replace one of the characters (in a **random position**) with a **random special character**
  - Replace one of the characters (in a **random position**) with a **random digit**

Example

- Some problems are repeated:
  - Draw a random character from a set (repeated **3 times**)
    - Let’s code a function **randomCharacter()** that selects a random character from a string
    - Use the function 3 times (for letters, digits and special characters)
  - Insert a character in a random position of a string (repeated **2 times**)
    - Function **insertAtRandom()** that:
      - Draws a random index (in the valid range!!!)
      - “Replaces” the character at that index
    - **The most difficult of the functions**

Example

- `initPassword`
  - Create an empty string `password`.
  - Generate `l` random lowercase letters and add them to `password`.

Example

- `randomCharacter`
    - Draw a random character from a string
    - Equivalent to drawing an index (in the valid range) and returning the character at that index.

Example

- `insertAtRandom`
  - Why is it more difficult than it seems?
    - Generate an index in the range [0, len()-1]
    - REPLACE the character at that position with a random one.
  - **But strings are immutable!!!!**
    - I cannot just overwrite characters!
  - Solution:
    - Create a new string
    - Initialize it to the original string up to the selected index
    - Append the new random character
    - Append the rest of the string.

Programming Tips

- Keep functions short
  - If more than one screen, break into ‘sub’ functions

- Use **Stubs** as you write larger programs
  - A function that returns a simply calculated value (even wrong, or always the same), enough to test the function as soon as possible, in the context in which it is called
  - They are used to understand if the sequence of calls "works"

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
- 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

Thanks

- Part of these slides are [edited versions] of those originally made by **Prof Giovanni Squillero** (Teacher of Course 1)

License

- These slides are distributed under the Creative Commons license “Attribution - Noncommercial - ShareAlike 2.5 (CC BY-NC-SA 2.5)”
- You are free:
  - to reproduce, distribute, communicate to the public, exhibit in public, represent, perform and recite this work
  - to modify this work
- Under the following conditions:
  - **Attribution** — You must attribute the work to the original authors, and you must do so in a way that does not suggest that they endorse you or your use of the work.
  - **Non-Commercial** — You may not use this work for commercial purposes.
  - **Share Alike** — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or equivalent license as this one.
- http://creativecommons.org/licenses/by-nc-sa/2.5/it/