Functions; File Input

About parentheses

- So far, we've used parentheses for:
  - Grouping symbols in equations

```
number = (x + y) * z
```

  - Creating tuples

```
myTuple = (2, 5)
```

  - Calling functions (today's topic)

```
size = len(leet)
```

About functions

- In our programs we've had several chunks of code listing from top to bottom
- All that code started becoming lengthy and possibly confusing
- Ideally, we want to separate our code into several logical parts in our program. This makes it easier to follow, maintain, and repeat

Defining functions

- A function allows us to break up a program into named sub-sections – you can think of a function as a variable that has code instead of just a value

- Functions often take in one or more values as input (called parameters) – though they aren't required to

- Functions often give back (or return) a value – though they aren't required to

Using functions

- Python has many built-in functions we've been using since day one

- Here are some stand-alone functions
  - `len`
  - `print`
  - `max`

- Using a function is also referred to as ***calling a function***

Let's describe len

- What are `len`'s input parameters?
  - A sequence
- What are `len`'s side effects?
  - What is a side effect? (https://en.wikipedia.org/wiki/Side_effect_(computer_science)#:~:text=In%20computer%20science%2C%20an%20operation,the%20invoker%20of%20the%20operation.)
  - It doesn't display anything or move data around, so it doesn't have any
- What is `len`'s return value?
  - An integer holding the number of elements in the inputted sequence

```
print(len("What does this do?"))
```

```
18
```

Let's describe print

- What are `print`'s parameters?
  - So far, we know `print` can take any number of basic data types (string, number, list, etc.) and some formatting parameters
- What are `print`'s side effects?
  - It displays to the console whatever is passed as input
- What is `print`'s return value?
  - Let's find out!

```
print(print("What does this do?"))
```

```
What does this do?
None
```

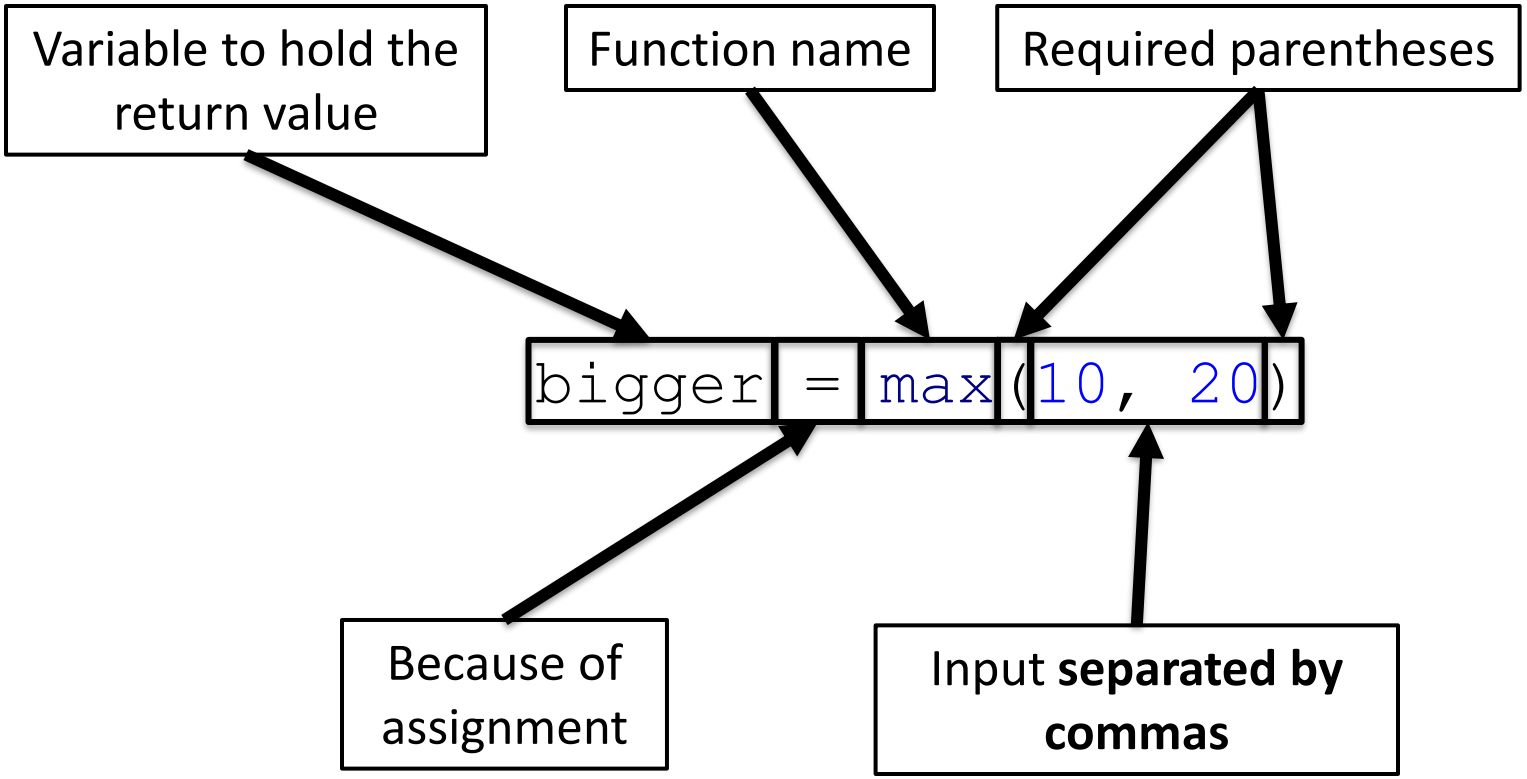
Let's describe max

- What is `max`'s parameters?
  - At least 2 numbers, separated by commas
- What are `max`'s side effects?
  - It doesn't display anything or move data around
- What is `max`'s return value?
  - An integer holding the largest number inside the parenthesis

```
print(max(10, 20))
```

```
20
```

The max function



Calling functions that do not return a value

- A function that does not return a value should be used on a line by itself
- It's return type is "NoneType"
- Python cannot convert that into something else

```
x = 3 + print("Hello")
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Calling functions that return a value

- A function that returns a value can be used anywhere that a value type is valid

- So, if this is valid:

```
x = 3 + 5
print(x)
```

```
8
```

- Then so is this:

```
x = 3 + max(2, 5)
print(x)
```

```
8
```

Calling functions that return a value

- When nesting functions, the inner functions are called first…

```
myName = "Bob Jarvis"
hisName = "Bobbernobby Hobbercobbins"
# maxLen is now 25
maxLen = max(len(myName), len(hisName))
```

- The `max` function can't run until the two `len` functions run – the values `max` will compare are dependent on the output of the `len` functions

- This is handled transparently for you – Python knows it needs to call `len` first!

Two steps to functions

1. Defining the function to Python
   - So far, we've been using functions defined by Python or in modules
   - Soon we'll write our own functions

2. Using the function
   - Calling a function tells Python to execute the function (as described in the definition of the function)

Summary so far …

- We talked about the concept of functions
- A function allows us to break up the program into named sub-sections
- Functions often take in one or more values as input (called parameters) – though they aren't required to
- Functions often give back (or return) a value – though they aren't required to

A program with a custom function

```
def sayHello():
    print("Hello!")


print("Calling my function")
sayHello()
print("Function called!")
```

```
Calling my function
Hello!
Function called!
```

Function definition

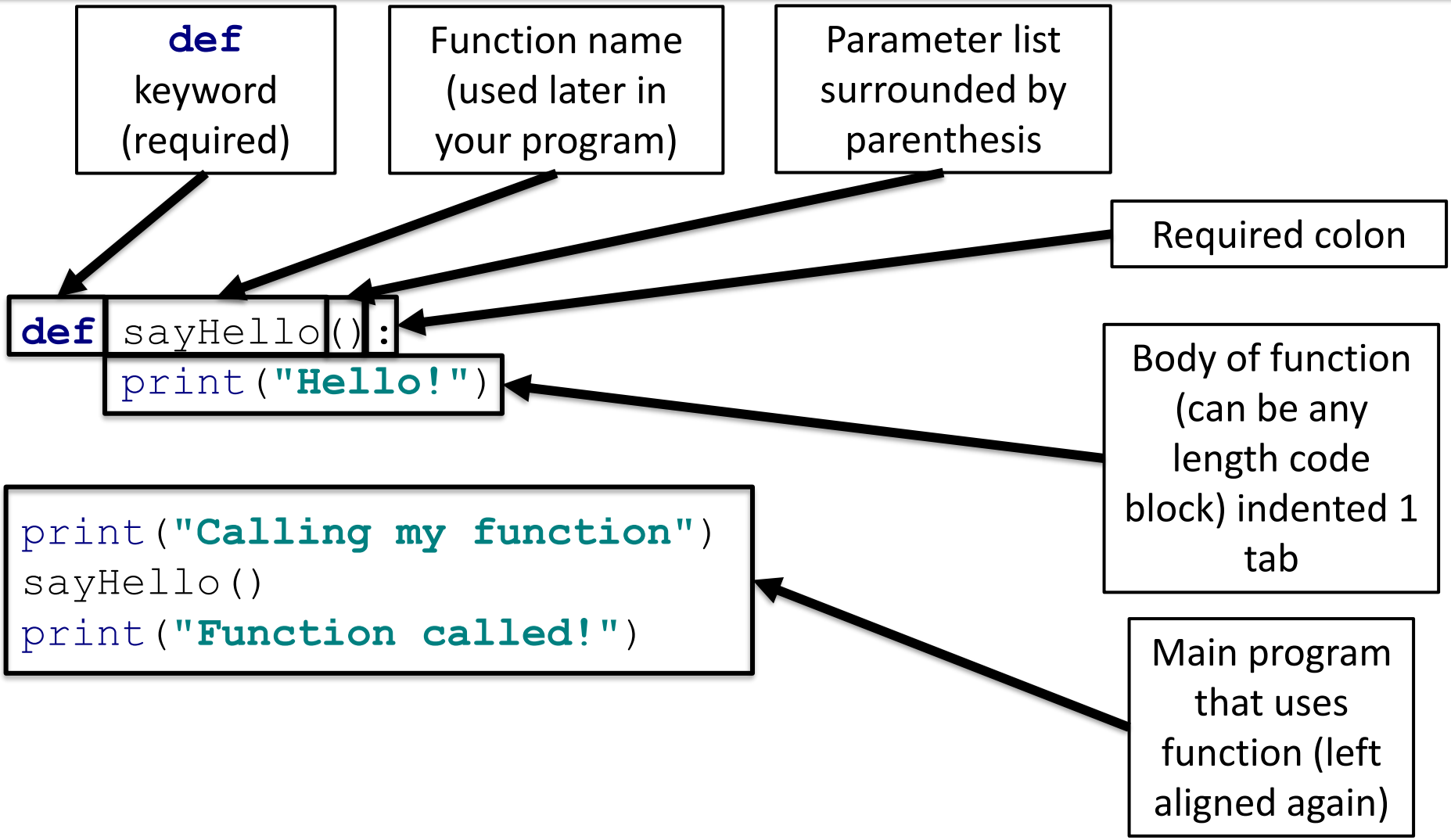
- A function definition is what describes a custom function
- It tells Python what the function does!

- In our example our function definition is

```
def sayHello():
    print("Hello!")
```

- A function must be defined before it can be used – like a variable
- It is a GoodIdea® to define all your functions at the beginning of your program (unless you use a main function – more on that soon)

Function definition syntax



Let's describe sayHello

- What are its parameters?
  - None
- What are its side effects?
  - Displays "Hello!" to the console
- What are sayHello's return values?
  - None

Function headers

- It's good coding practice to always put comments right before the declaration of a function – and **now required for your labs and assignments**

```
# Function: sayHello
# Purpose: Displays "Hello!" to the console
# Parameters: None
# Side effects: prints to the console
# Returns: Nothing
def sayHello():
    print("Hello!")


print("Calling my function")
sayHello()
print("Function called!")
```

What if we reversed the order?

```
print("Calling my function")
sayHello()
print("Function called!")

# Function: sayHello
# Purpose: Displays "Hello!" to the console
# Parameters: None
# Side effects: prints to the console
# Returns: Nothing
def sayHello():
    print("Hello!")
```

```
NameError: name 'sayHello' is not defined
```

Order matters in Python!

- Just like variables, functions must be defined before they're used in your program

- There is a caveat…
  - Your function definitions can appear in any order, they just need to appear before you use it in your main program

A main function

- In Python, your program runs from the top to bottom of your file
- In other programming languages only the “main” function runs by default
- To simulate these other languages, we will use a “main” function too – it has no special meaning, it’s just another function!

Example with a main function

```
def main():
    number = int(input("Enter a number: "))
    result = square(number)
    print(str(number) + " squared is " + str(result))

def square(x):
    return x * x

main()
```

```
Enter a number: 5
5 squared is 25
```

Why make a main?

- Notice that now our function definitions can appear in any order
  - Using a main function avoids the declare before you use problem

- By getting into this habit, the only “left aligned” code is a call to main and function order doesn’t matter

- This is also how many other programming language's structure their execution – so it’s GoodToKnow™

Scope and functions

- Each function has its own variable scope
- Variables declared and modified in a function usually don't have their changes persist
- Let's look at an example…

Example function scope

```
def test01():
    x = 10


def test02():
    y = 5


x = 0
y = 0
test01()
test02()
print(x)
print(y)
```

```
0
0
```

Example function scope, visualized

```
def test01():
    x = 10
```

Scope 1

```
def test02():
    y = 5
```

Scope 2

```
x = 0
y = 0
test01()
test02()
print(x)
print(y)
```

Scope 3

About function scopes

- Each scope is a separate world – things that happen in one world are wholly separate from another.

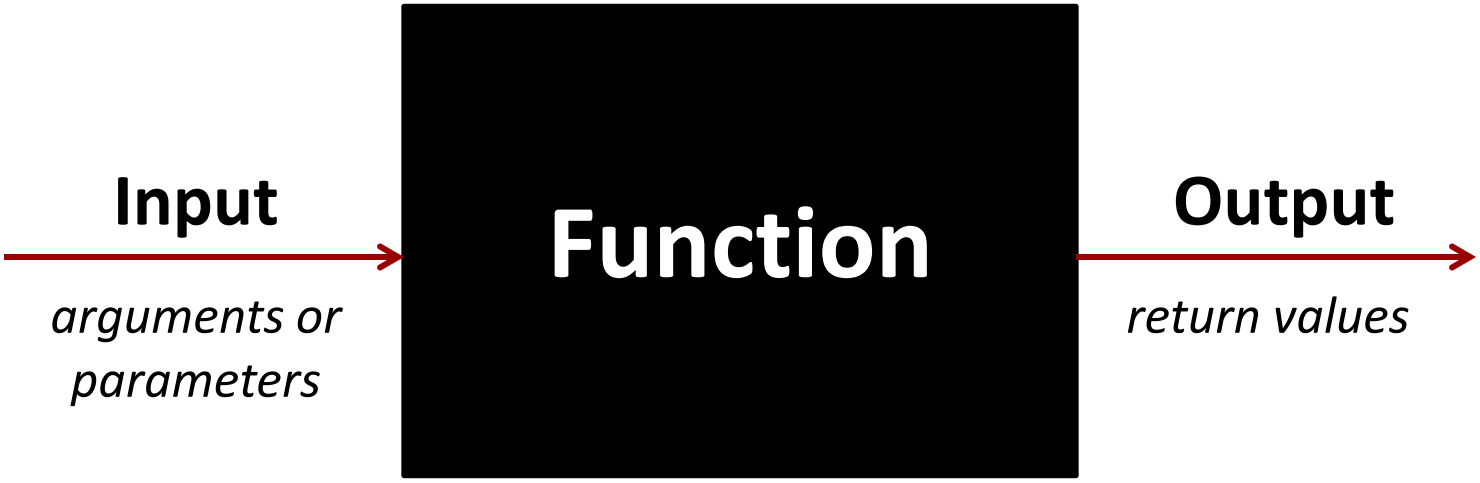
- That means with a slight modification, the previous code won't work at all!

```
def test01():
    x = 10

y = 0
test01()
print(x)
```

```
NameError: name 'x' is not defined
```

Function input and output

- Python allows us to pass data into and receive data from a function in a formal way…



Another program with custom functions

```
def sayHello(name):
    print("Hello " + name + "!")


def displaySum(a, b):
    print(a + b)


displaySum(5, 3)
sayHello("Bob")
```

```
8
Hello Bob!
```

Function syntax (with input)

```
def sayHello(name):
    print("Hello " + name + "!")


def displaySum(a, b):
    print(a + b)


displaySum(5, 3)
sayHello("Bob")
```

Now we have a parameter! This parameter is called `name`

The value `"Bob"` gets put into `name` and is used in the function above

Positional parameters

- The most common way to pass arguments into functions is with positional parameters

- Parameters get their values based on the position the values are sent

- The 1st parameter gets the 1st value, the 2nd parameter gets the 2nd value, and so on

Positional parameters, visualized

```
def sayHello(name):
    print("Hello " + name + "!")


def displaySum(a, b):
    print(a + b)


displaySum(5, 3)
sayHello("Bob")
```

Returning values

- In Python, any function can return any kind of value
- Return values can be stored in a variable or printed to the console

```
def makeGreeting(name):
    retval = "Hello " + str(name)
    return retval

print(makeGreeting("Bob"))
```

Here I'm returning a modified string

This displays the results from the function

```
Hello Bob!
```

What about data type mismatch?

- When the data types don't match, the results can be unexpected

```
def doubler(x):
    return x * 2

num = doubler(2)
print(num)
print(doubler(2.2))
print(doubler("Hi"))
```

```
4
4.4
HiHi
```

About returns

- If a function is to return nothing, it can with the return statement and nothing following the keyword ‘return’

- However, once a function returns, it’s over and none of the other code in it runs

```
def main():
    print("Main is almost over!")
    return
    print("This doesn't display")

main()
```

```
Main is almost over!
```

About calling functions

- Only one function can be the “current” function
- All other functions are “paused” until the current function finishes
- How is this tracked?

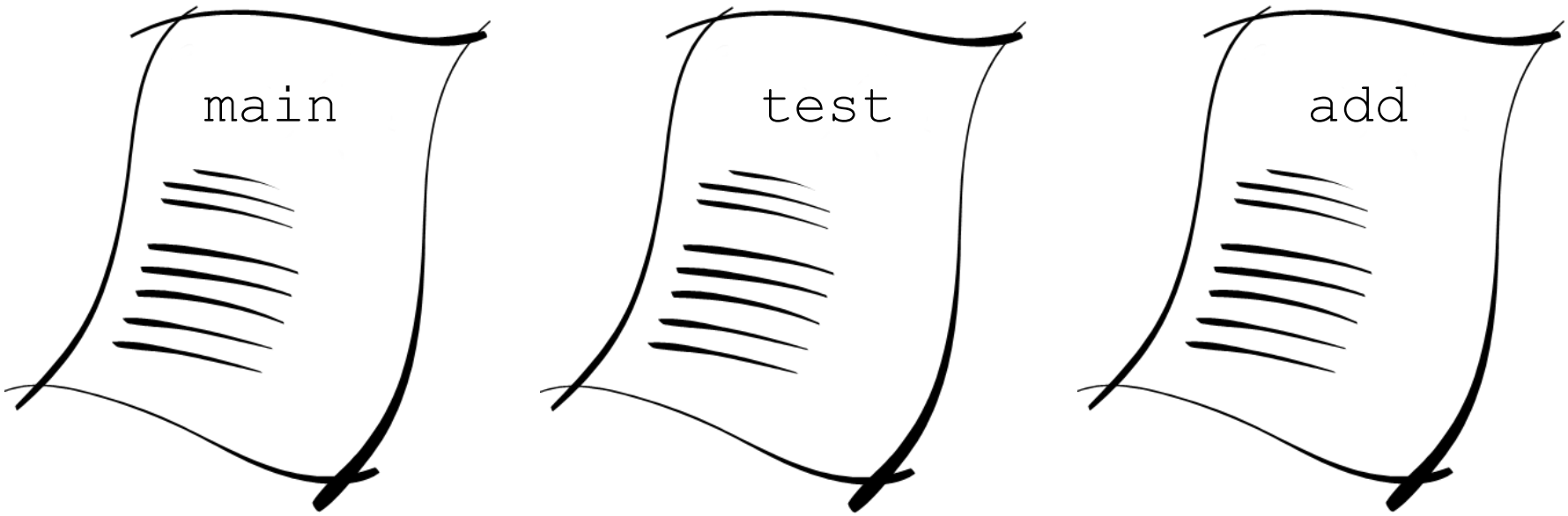
A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

```
Program's starting...
10
Goodbye!
```

Papers

- Imagine you have 3 sheets of paper, each with the name of a function on it:



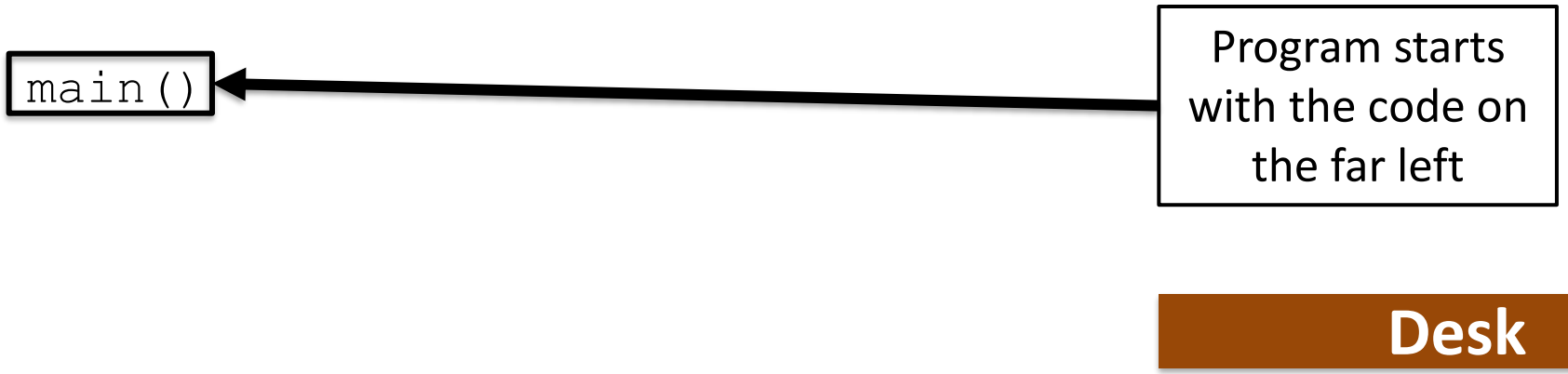
A Desk

- Now imagine you have a desk that you will stack the papers on

- (We will only have one stack of papers on the desk)

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")
```



A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

Then the `main` piece of paper goes on the desk

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

The program runs line by line

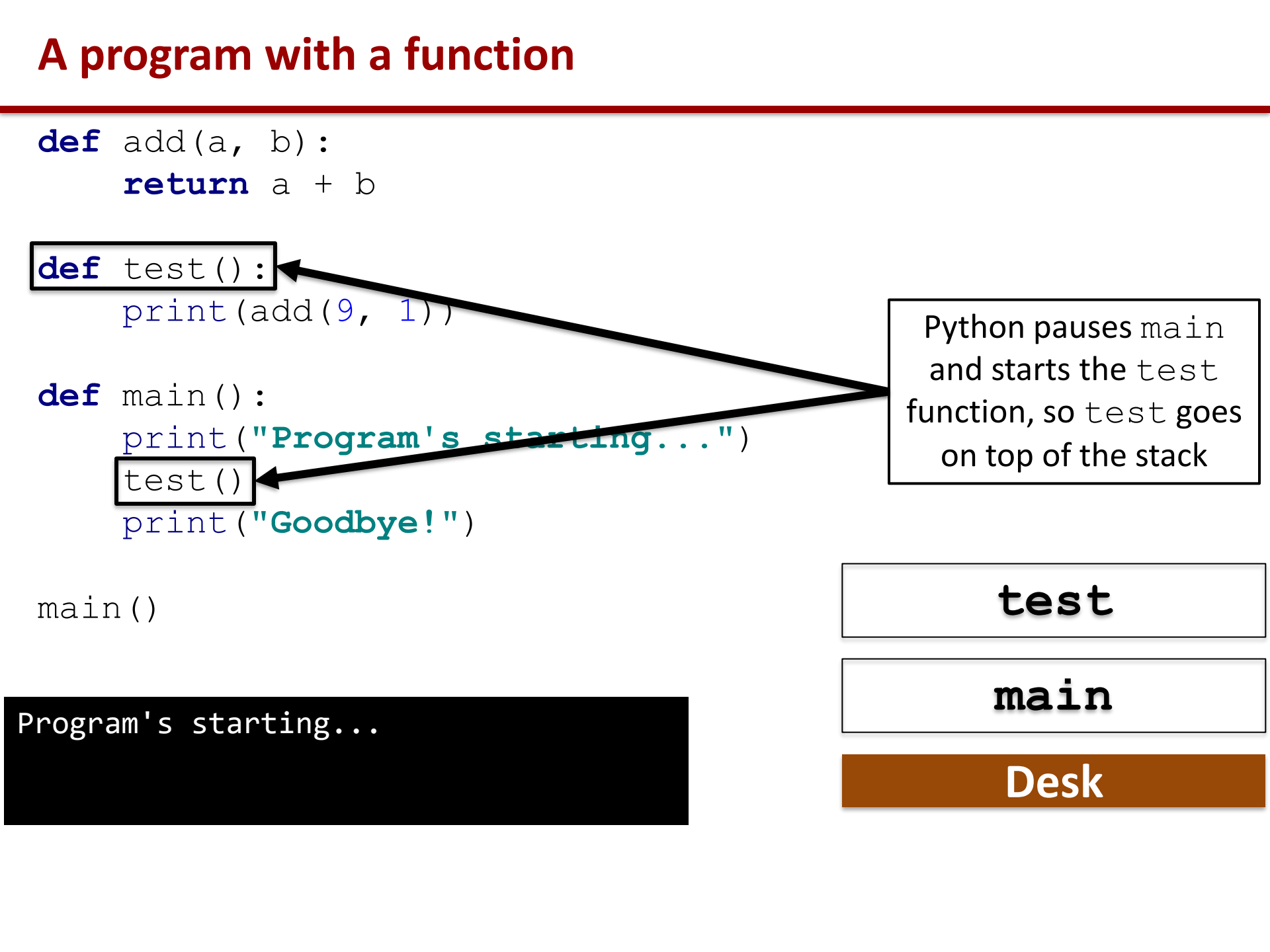
```
Program's starting...
```

main

Desk

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

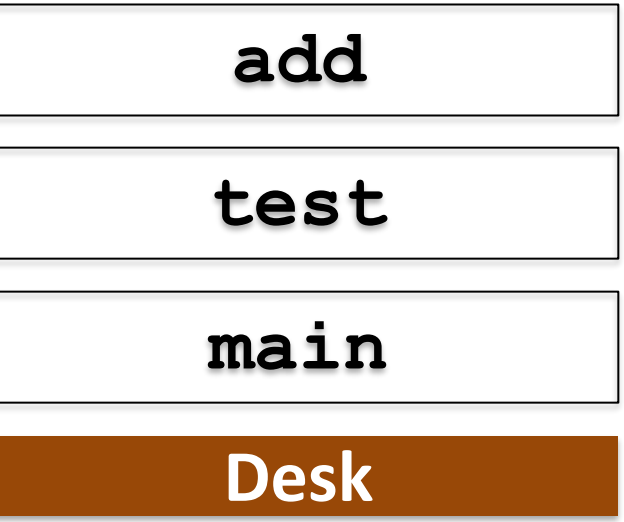



```
Program's starting...
```

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

When `test` calls the `add` function `test` gets paused and `add` goes on top of the stack



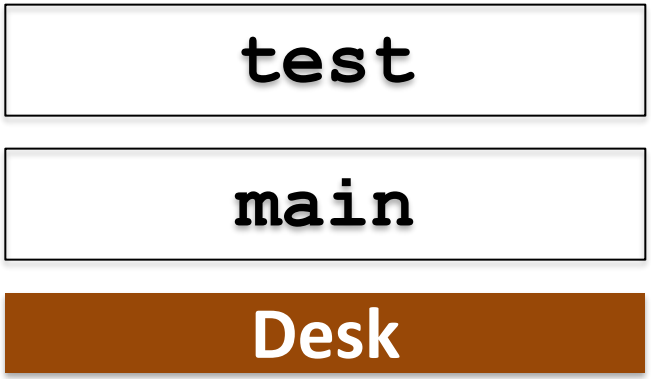

```
Program's starting...
```

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

The **return** ends the add function, so its paper is removed

```
Program's starting...
```

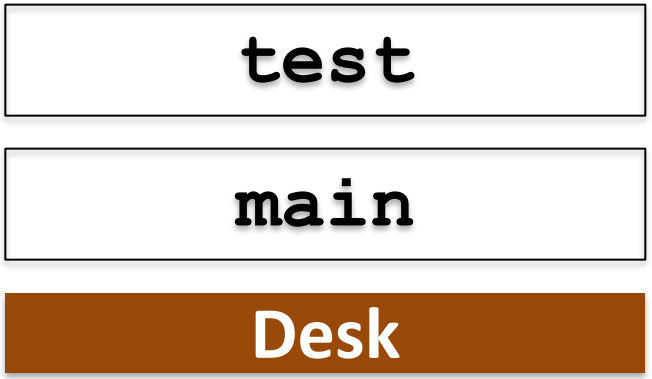


A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

Python completes the `test` function

```
Program's starting...
10
```



A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

When the `test` function is over, Python removes that paper too

```
Program's starting...
10
```

main

Desk

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

Python continues the `main` function from where it was paused

```
Program's starting...
10
Goodbye!
```

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

When that function is over, it's paper is removed too

```
Program's starting...
10
Goodbye!
```

Desk

A program with a function

```
def add(a, b):
    return a + b

def test():
    print(add(9, 1))

def main():
    print("Program's starting...")
    test()
    print("Goodbye!")

main()
```

Now that the desk is empty, the program is over!

```
Program's starting...
10
Goodbye!
```

Desk

Papers Analogy, Summarized

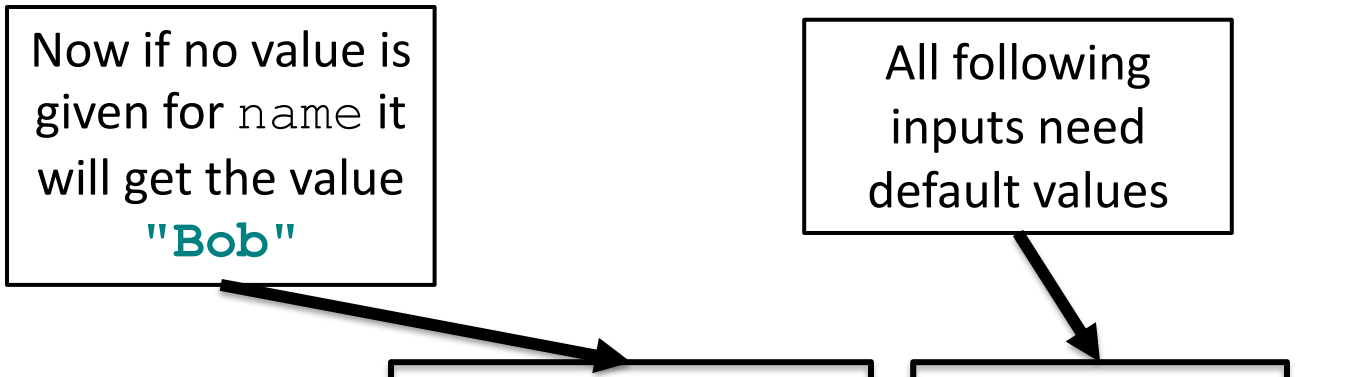
1. The desk is empty right before the program starts
2. When a function starts, its paper gets put on top of the paper stack
3. The "current" function is the paper on top of the stack
4. When a function ends, its paper is removed, and the program continues with the paper below it
5. If there are no papers left on the desk, the program is over

- The papers and desk may seem silly, but conceptually it is *very similar* to how it actually works!

About function input

- Functions can receive 0 or more parameters
- Unless specified, parameters are filled in by position
  - The 1st input is put into the 1st parameter, the 2nd input into the 2nd parameter, and so on
- When defining a function, parameters can be given default values
  - Once you assign a default value to one parameter ALL OTHERS that follow that parameter must also be given default values
- Parameters may also be filled in by name
  - Inside the function call, the variable names (from the function definition) are used to specify what parameters get what input

A function with input (and input default values)




```
def sayHello(name = "Bob", times = 2):
    for i in range(times):
        print("Hello " + name)

sayHello("Nathan", 1)
```

Using the previous code…

```
sayHello("Nathan", 1)
```

```
Hello Nathan
```

```
sayHello()
```

```
Hello Bob
Hello Bob
```

```
sayHello("Nathan")
```

```
Hello Nathan
Hello Nathan
```

```
sayHello(, 1)
```

```
SyntaxError: invalid syntax
```

```
sayHello(times=1)
```

```
Hello Bob
```

```
sayHello(times=1, name="Nathan")
```

```
Hello Nathan
```

```
def sayHello(name = "Bob",
times = 2):
  for i in range(times):
    print("Hello " + name)
```

About function output

- Functions can return 0 or more values
  - Most languages only allow 1 return value
- Returning from a function ends the function
  - Code after a return statement will never run
- When returning multiple items…
  - The items must be comma separated
  - When calling the function, commas are used to capture the different values

A function that returns multiple items

```
def prepForAvg(inList):
    runningSum = 0
    for item in inList:
        runningSum += item
    return runningSum, len(inList)
```

This function returns 2 things

```
someList = [1, 3, 5, 7, 11]
sum, num = prepForAvg(someList)
print("There were {} items".format(num))
print("Their total is {}.".format(sum))
```

The 1st item returned goes into the `sum`, the 2nd item goes into `num`

Functions and data types

- Looking at the previous slide, what if `inList` wasn't a list when the function ran?

- Python won't generate an error until runtime (when you run your program)
  - We'll talk about errors (or exceptions) next week

- But let's talk about how Python stores data…

Variable references

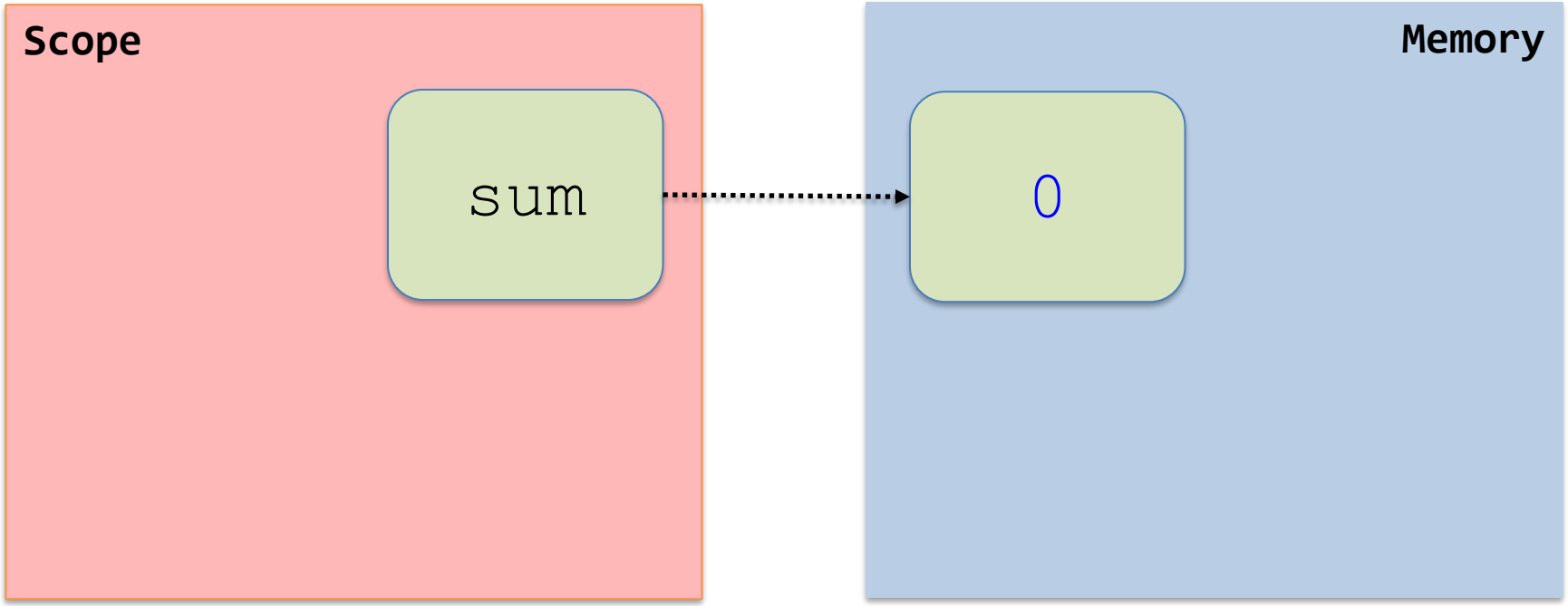
- Python variables do not store data!
- Instead, they store a reference to a piece of your computer's memory where the data is stored
- These references are scope specific!
- Some languages might call these references pointers…
- But Python manages the memory for you, so it's nearly transparent to you

Visualizing references

- When you see

```
sum = 0
```

- You should picture…



References to immutable items

- Immutable items cannot be changed
- When manipulating them, a new immutable object is created with the new value

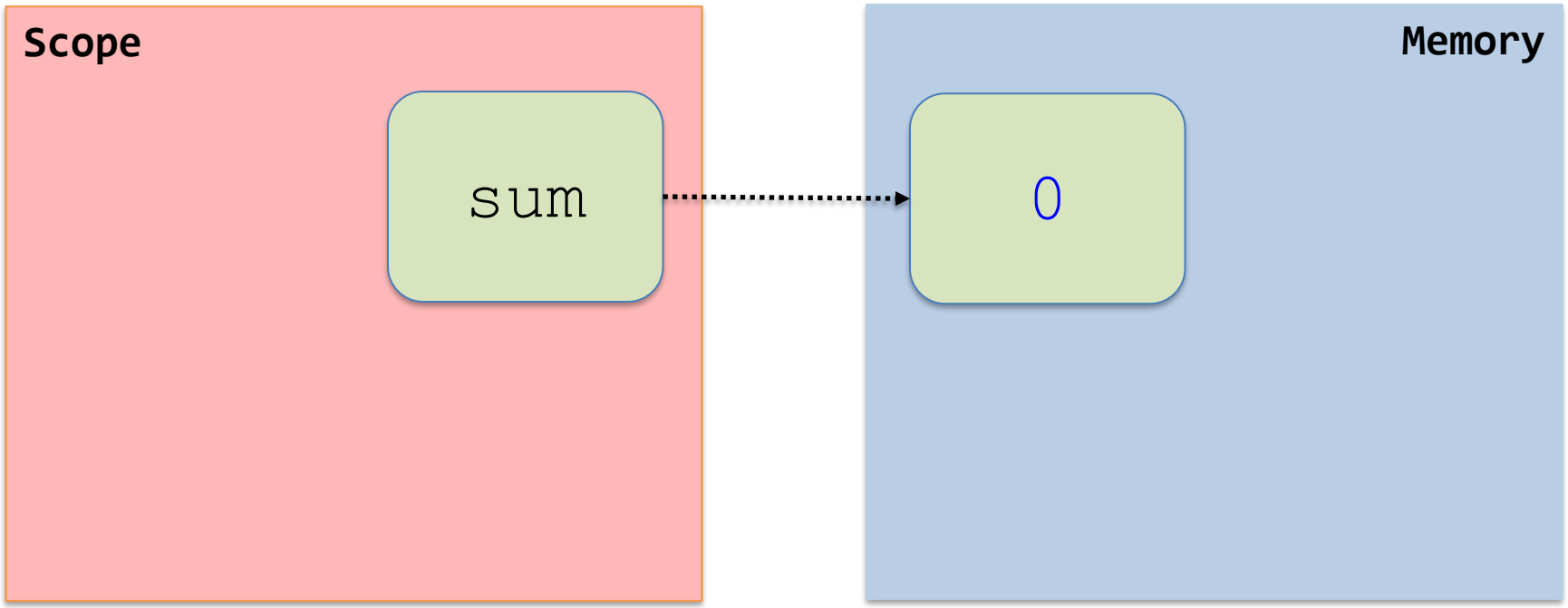
- Immutable objects are
  - strings
  - ints
  - floats
  - tuples

Visualizing manipulating immutables

- When you see

```
sum = 0
sum += 1
```

- You should picture…

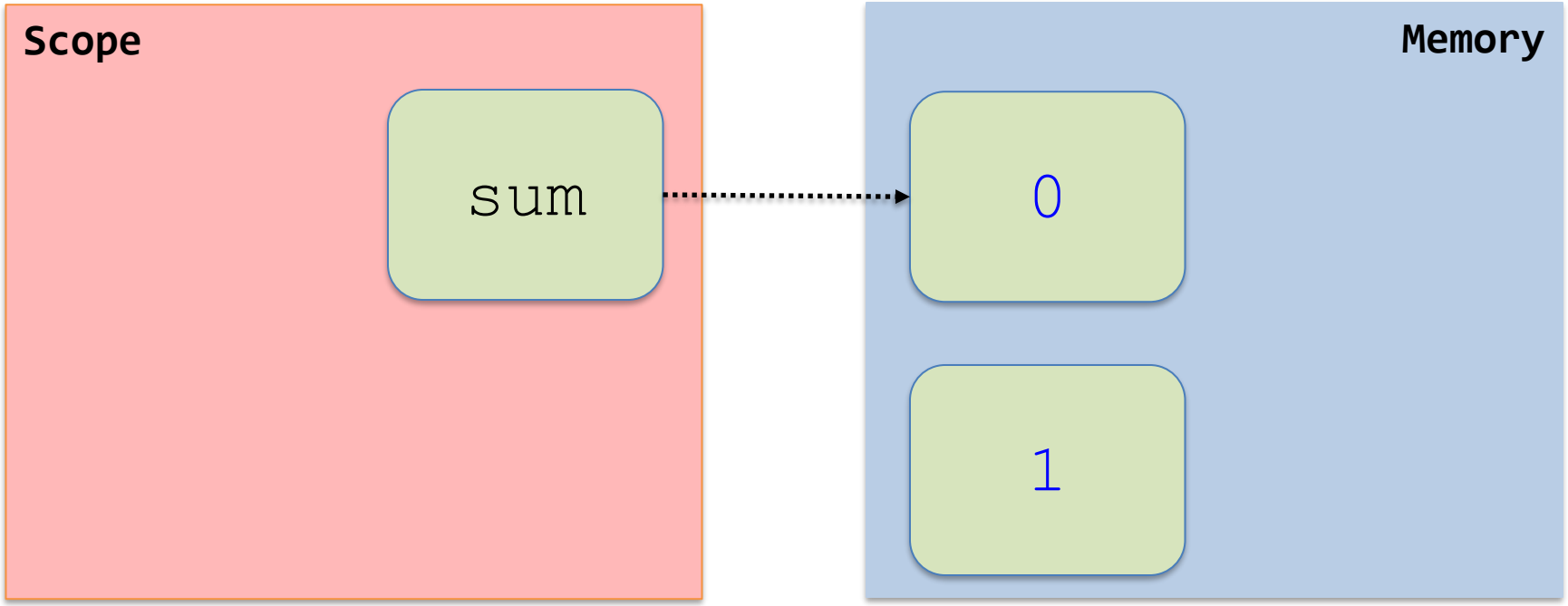


Visualizing manipulating immutables

- When you see

```
sum = 0
sum += 1
```

- You should picture…

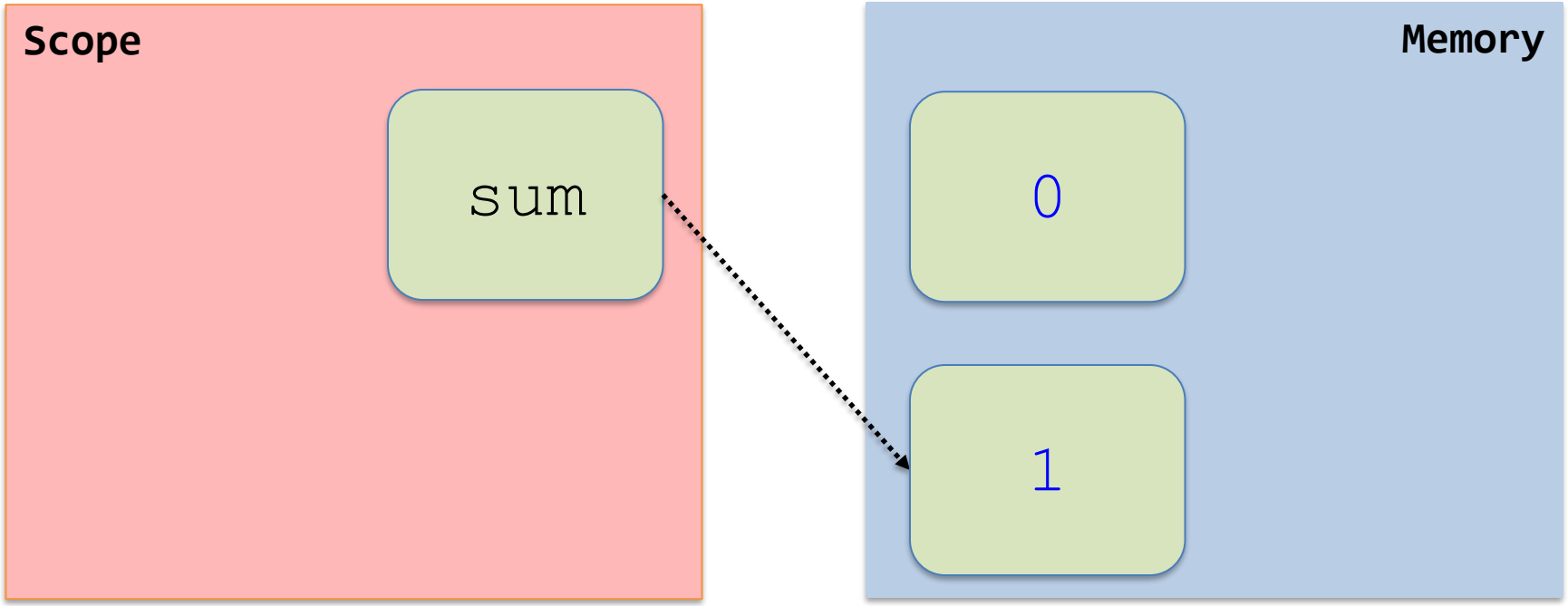


Visualizing manipulating immutables

- When you see

```
sum = 0
sum += 1
```

- You should picture…



Passing immutable things into a function

- When passing an immutable item into a function a copy of the reference is made
  - Both references "point" to the same item in memory
  - But like any immutable item, if someone changes the value then a new item in memory is made and the variable now references the new item

- Lets look at this…

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```

Seeing scopes

```
def addAndReplace(a, b):
    a += b
```

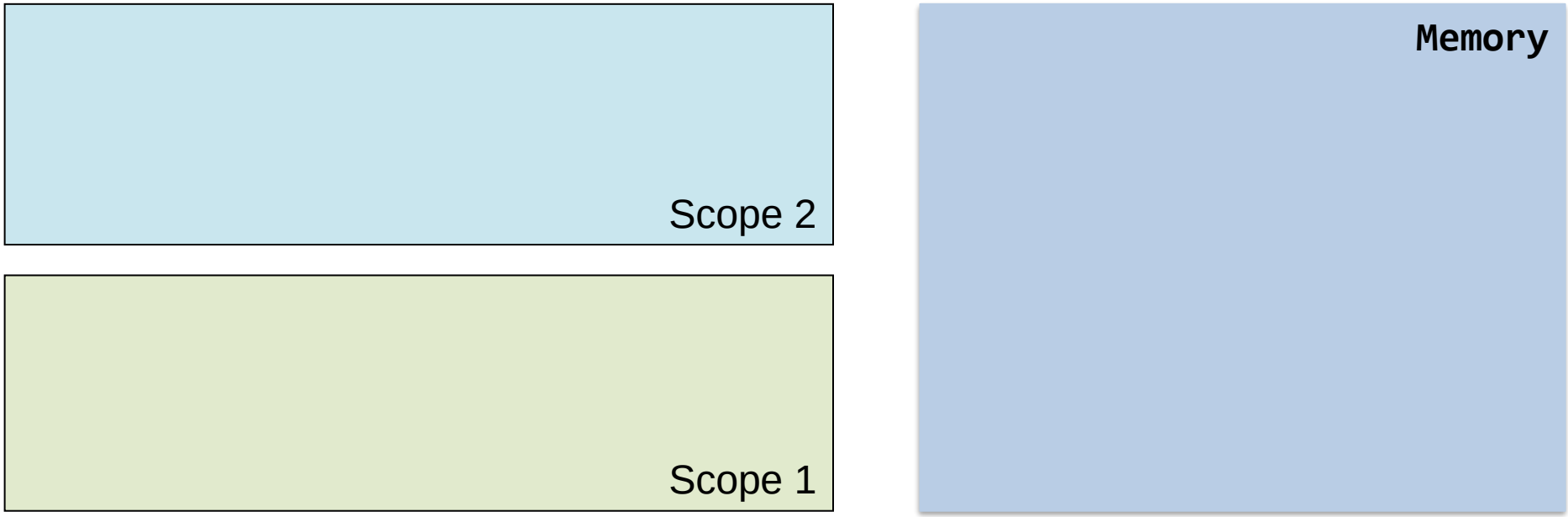
Scope 2

```
x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```

Scope 1

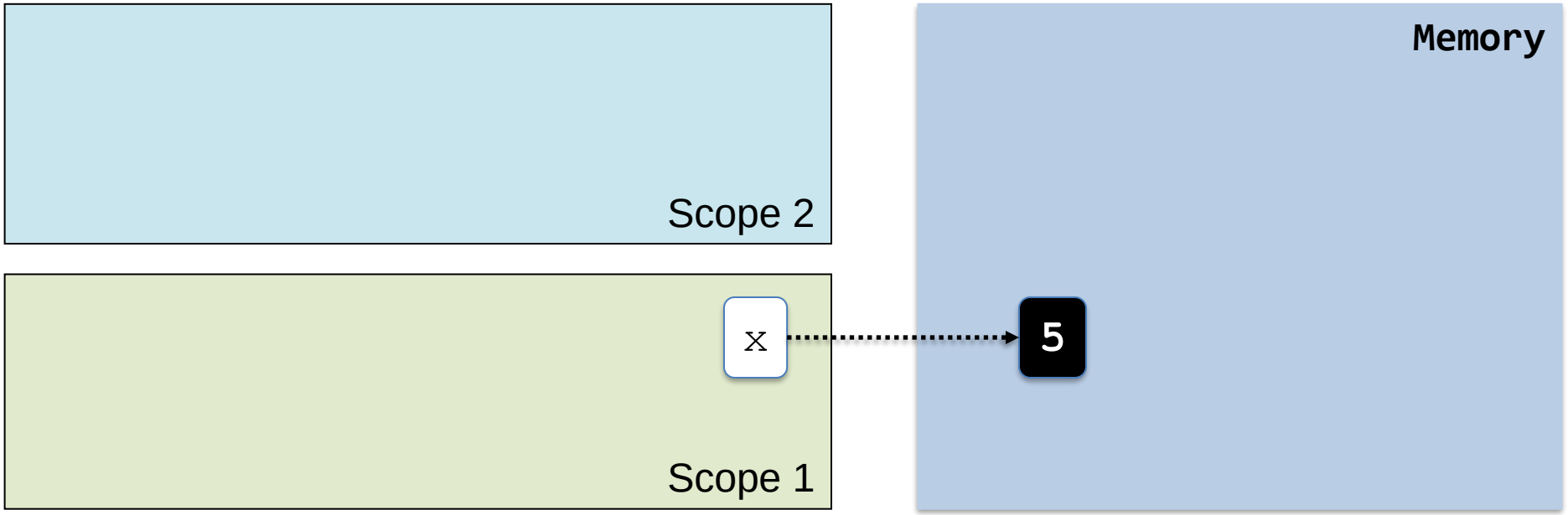
Running our code

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



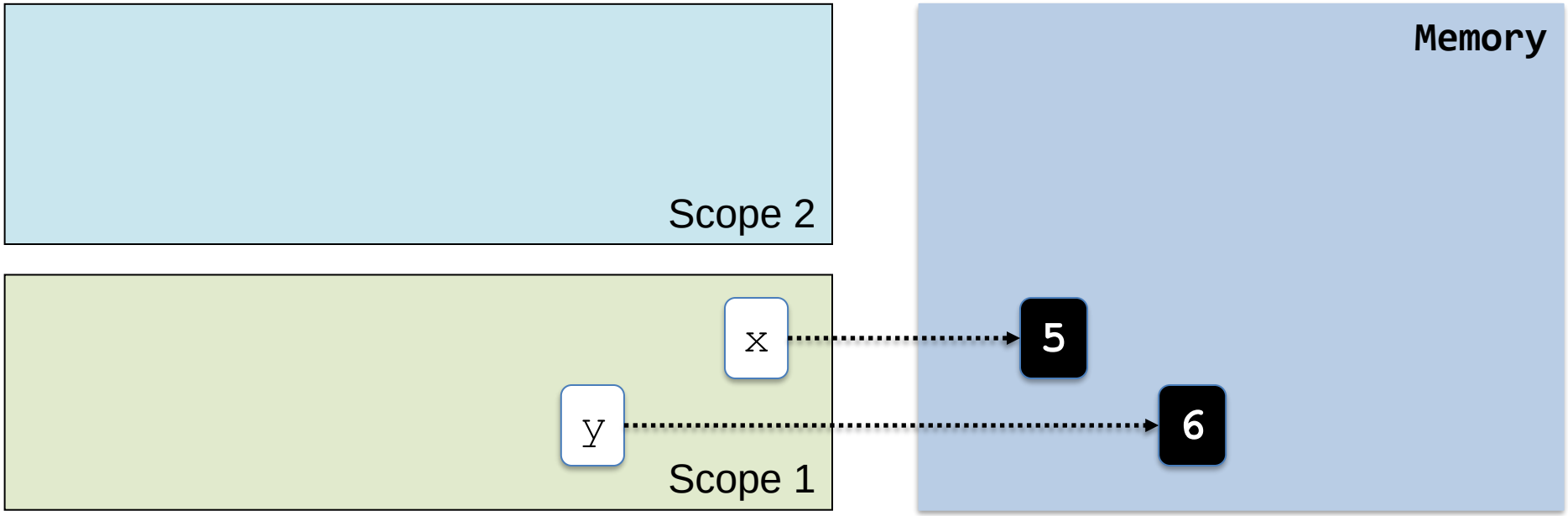
Running our code

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



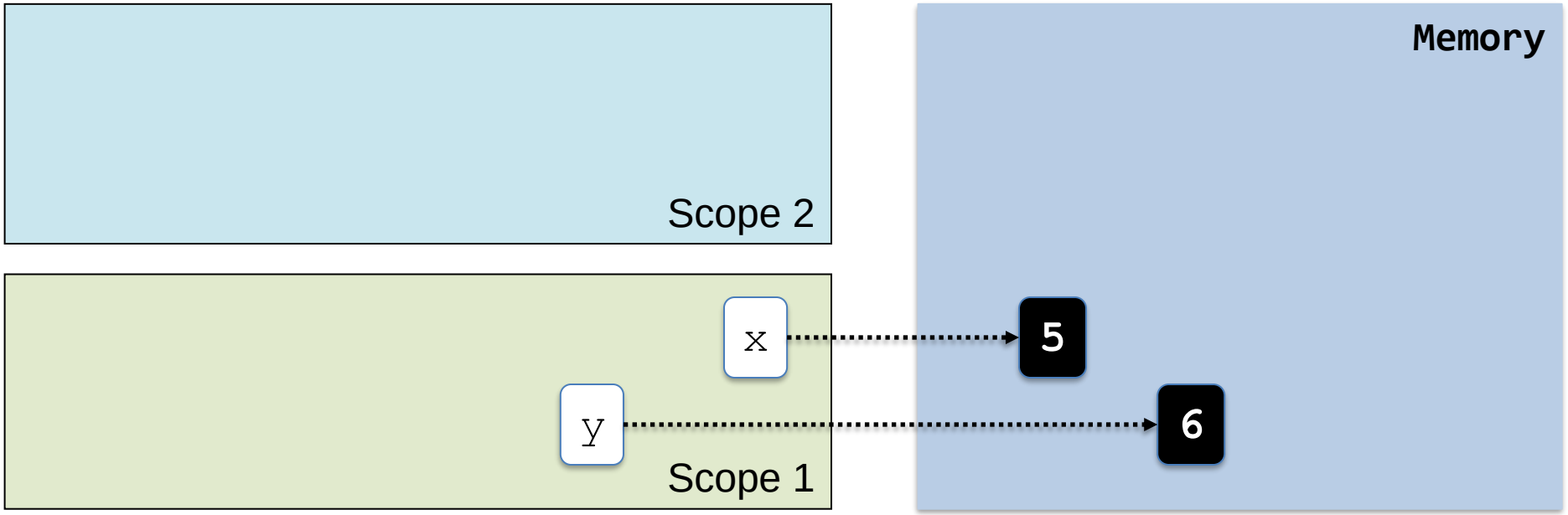
Running our code

```
def addAndReplace(a, b):
    a += b


x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



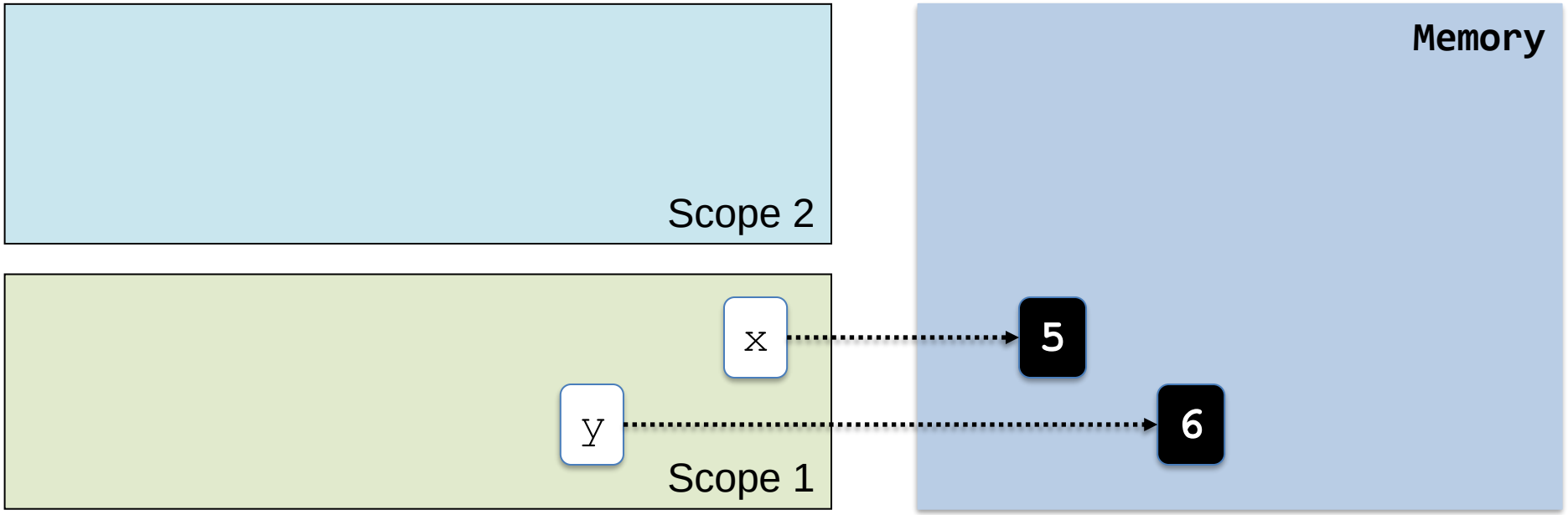
Running our code

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



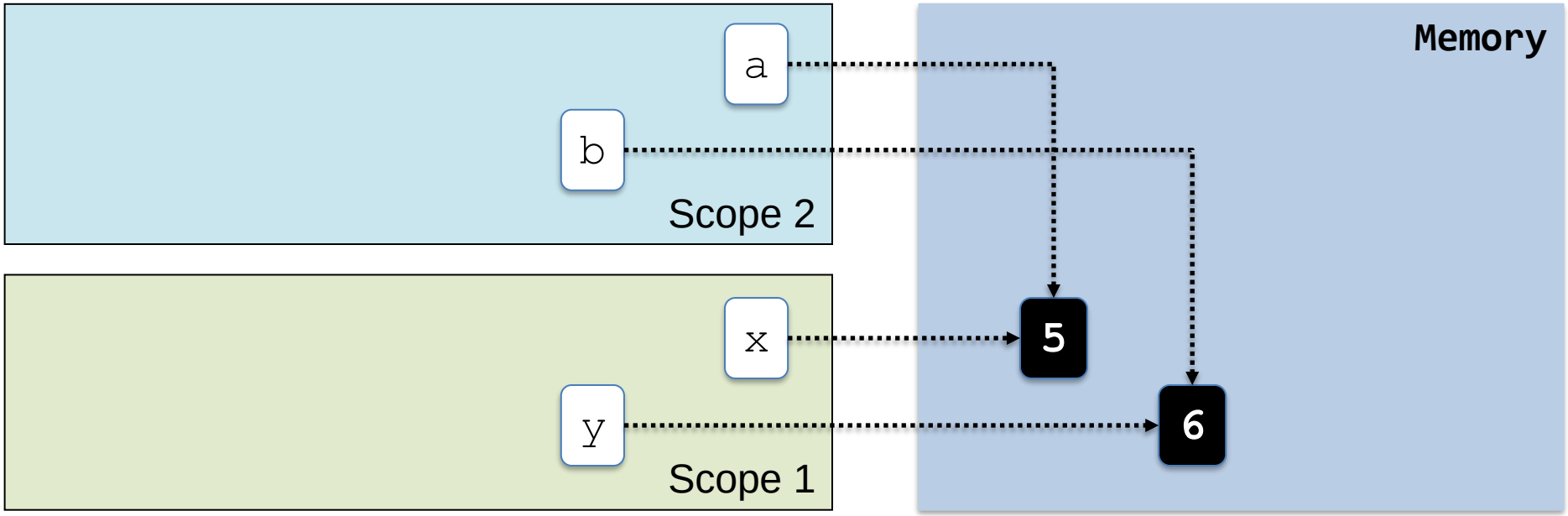
Running our code

```
def addAndReplace(a, b):
    a += b


x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



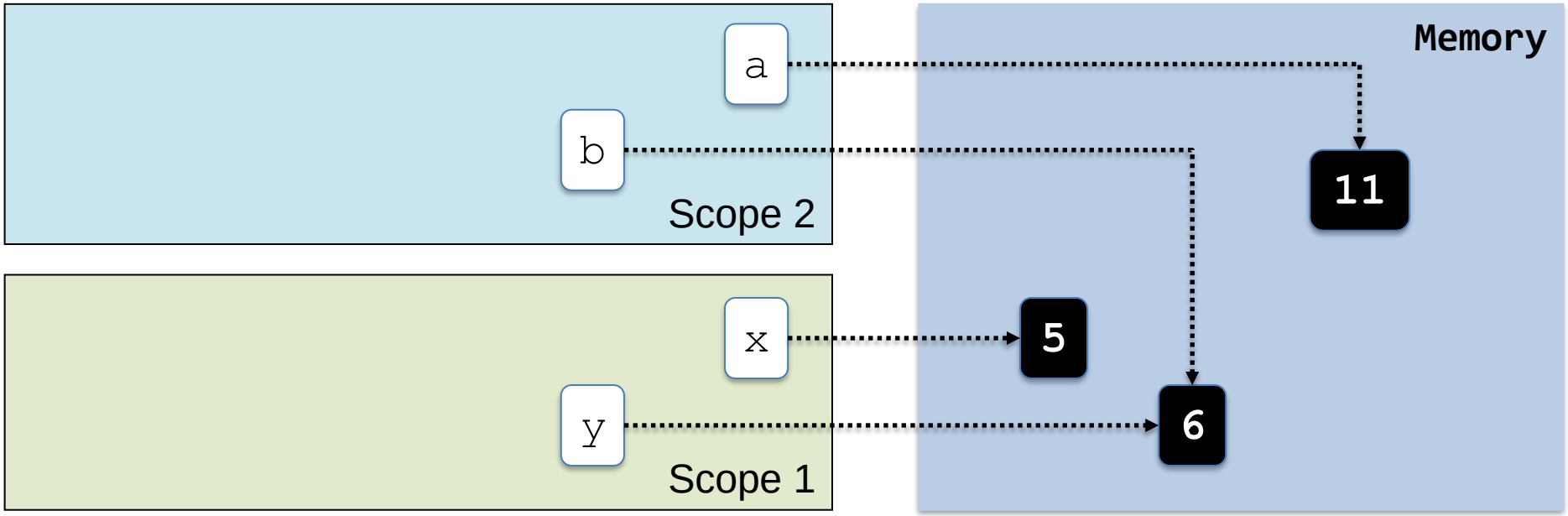
Running our code

```
def addAndReplace(a, b):
    a += b


x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



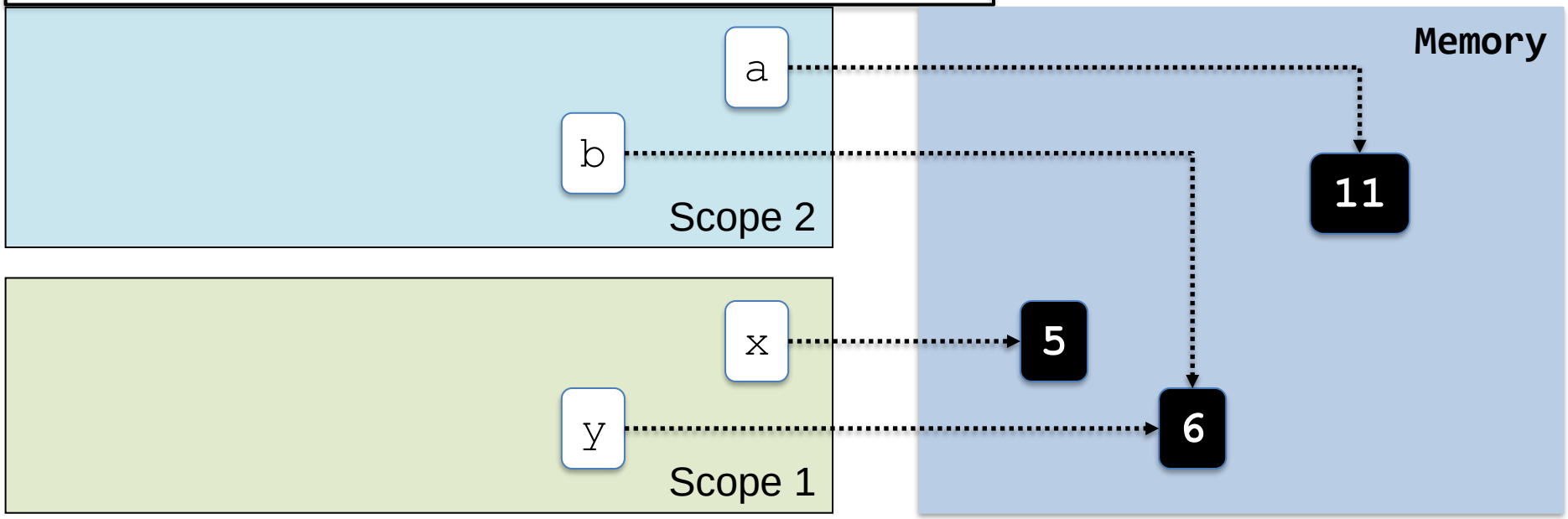
Running our code

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



Running our code

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```



Output

```
def addAndReplace(a, b):
    a += b

x = 5
y = 6
addAndReplace(x, y)
print("x = {}, y = {}".format(x, y))
```

```
x = 5, y = 6
```

Passing mutable things into a function

- When passing a mutable item into a function, the changes in the function will persist after the function finishes
  - Since both references “point” to the same location in memory, any change in one changes the memory item for the other
  - However, if you reassign the variable in your function, it still creates something new

- Let’s look at an example…

Seeing scopes

```
def addName(inList):
    inList.append("Nathan")
```

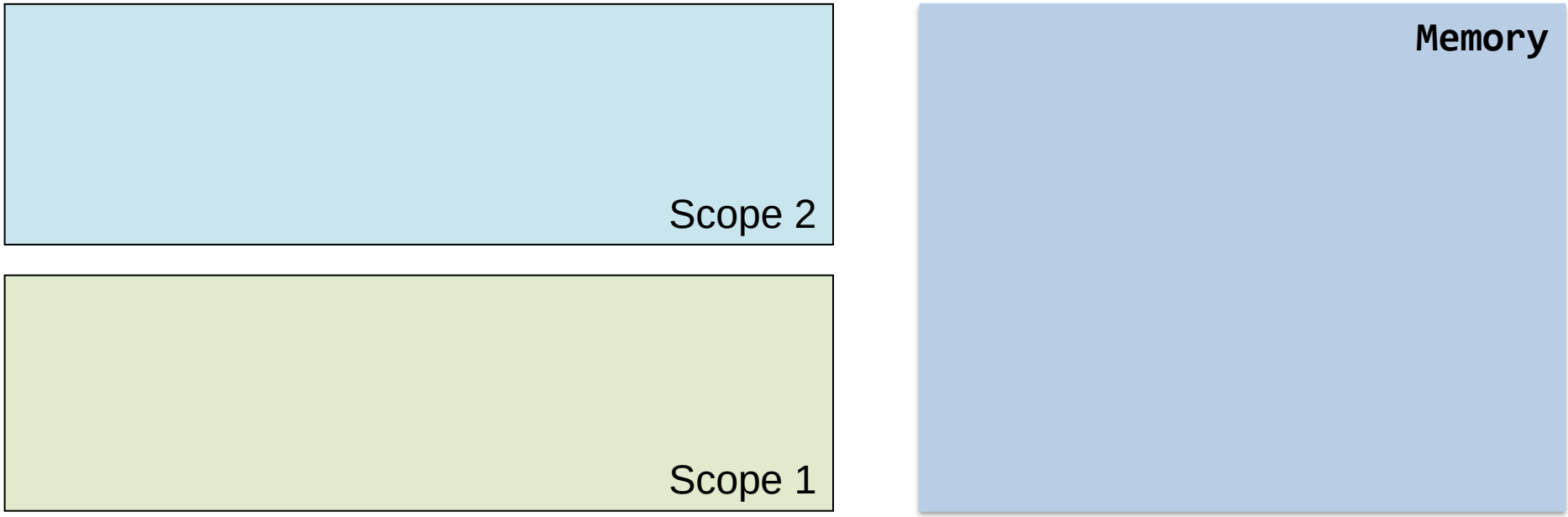
Scope 2

```
myList = ["Hi"]
addName(myList)
print(myList)
```

Scope 1

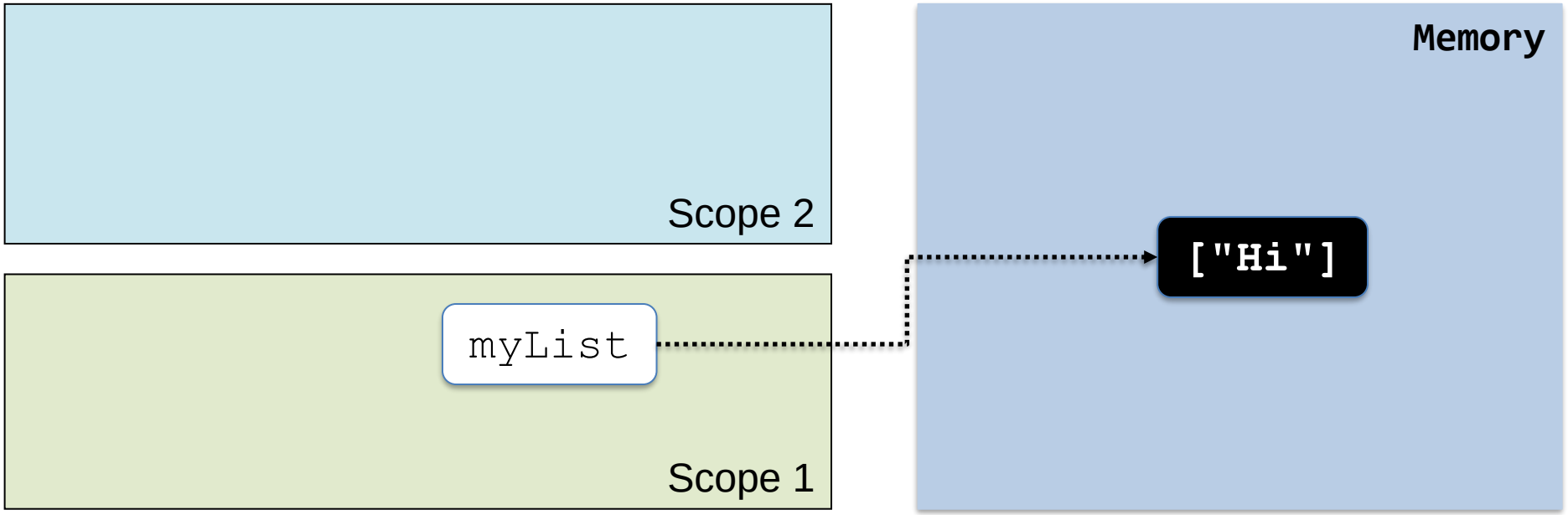
Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



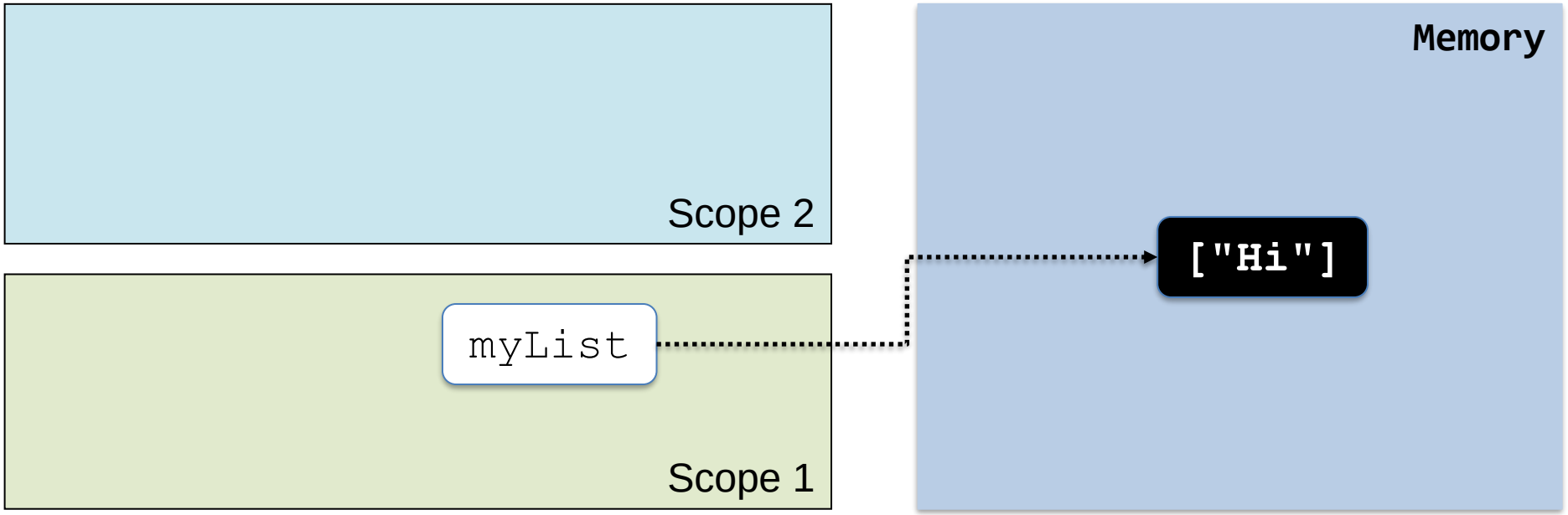
Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



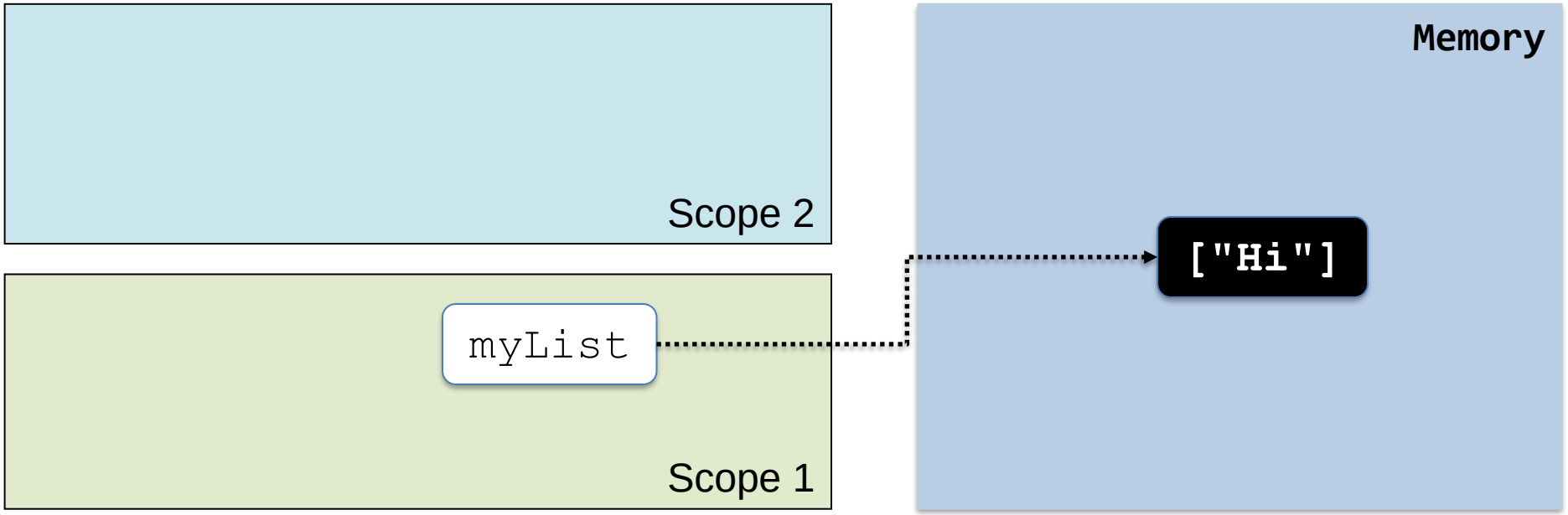
Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



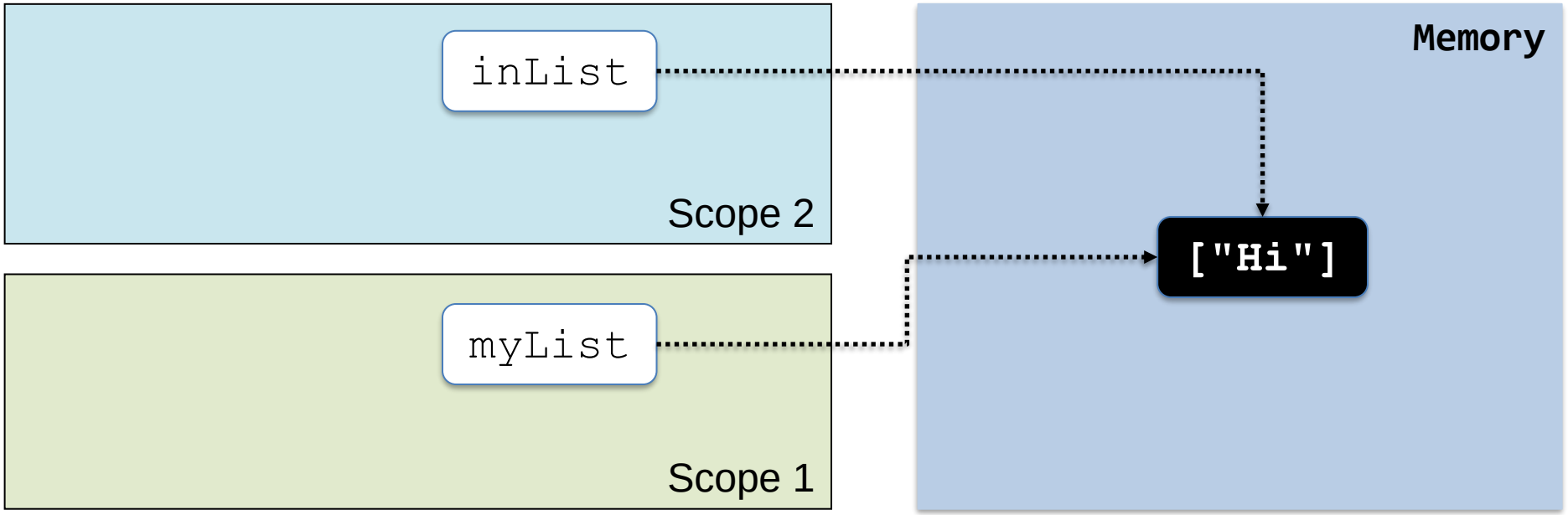
Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



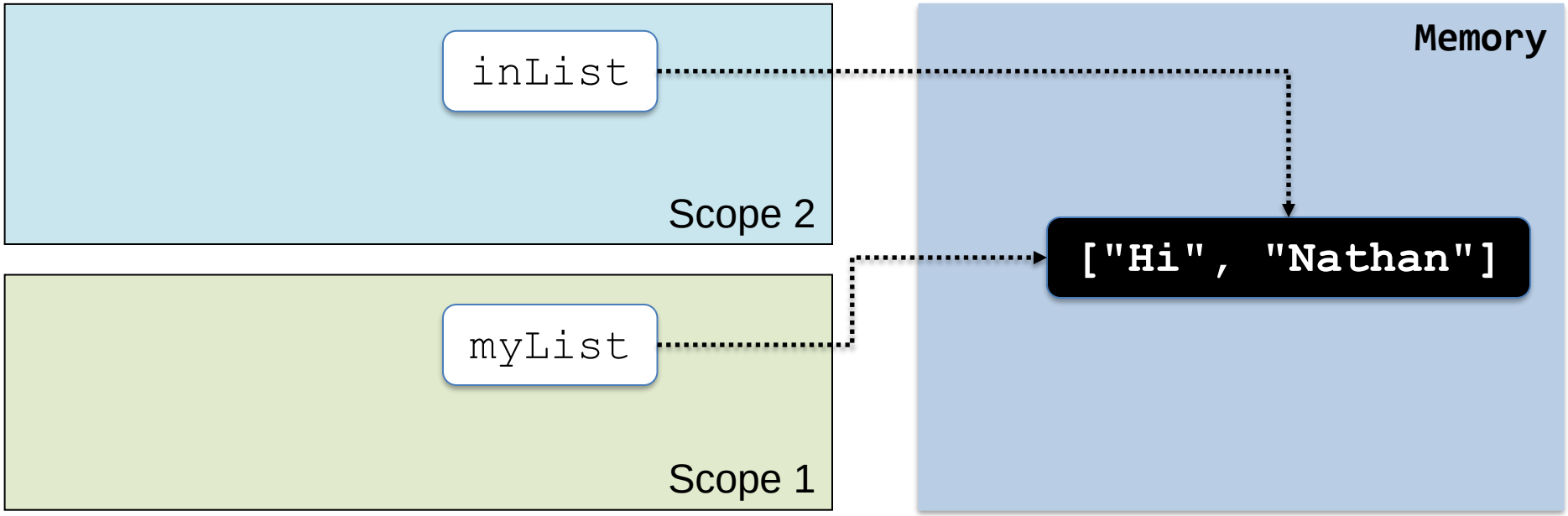
Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



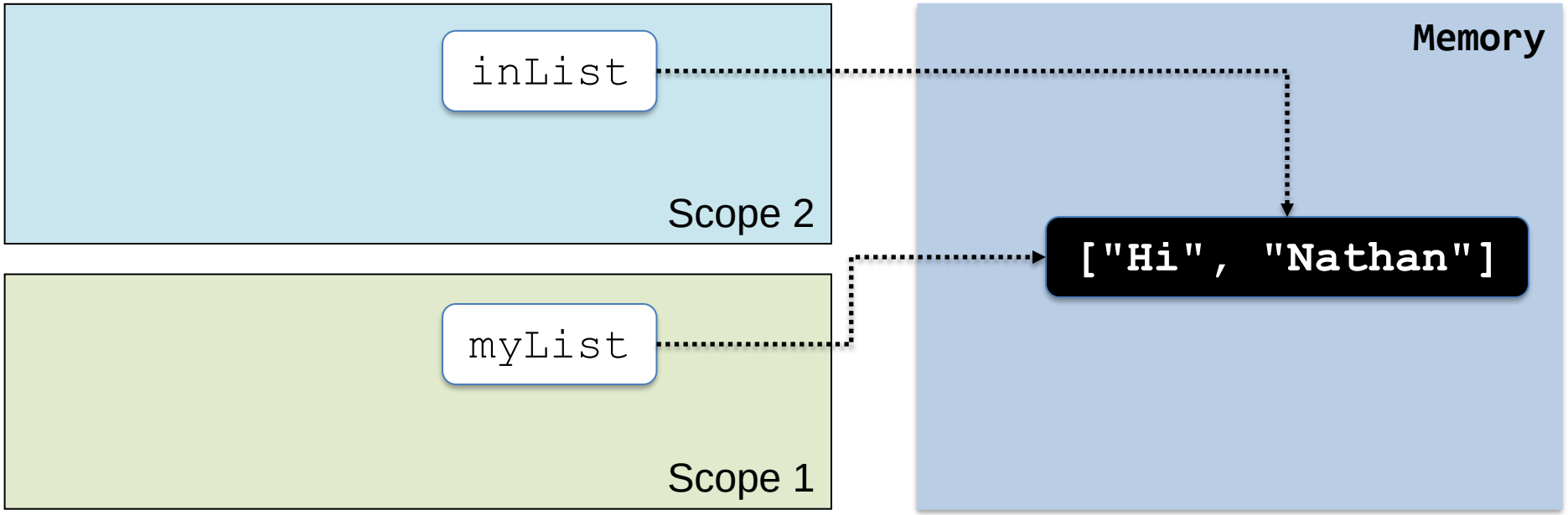
Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



Running our code

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```



Output

```
def addName(inList):
    inList.append("Nathan")

myList = ["Hi"]
addName(myList)
print(myList)
```

```
['Hi', 'Nathan']
```

What about this?

```
def addName(inList):
    inList.append("Nathan")

def newList(inList):
    inList = ["Bob"]

myList = ["Hi"]
addName(myList)
print(myList)
newList(myList)
print(myList)
```

```
['Hi', 'Nathan’]
['Hi', 'Nathan']
```

About assignment

- Assigning an item in Python makes a new value in memory no matter if the item is mutable or not

- Consider the following code

```
myList = [1, 2, 3]
del myList[0]
myList = [5, 6]
```

- The `del` removes the item at index `0` without changing the reference
- The assignment changes the `myList` reference to point to a new list in memory with the contents of `[5, 6]`
- The old contents of `myList` eventually disappear from memory

File input

A bit about files

- A file is a collection of data
- The unit of storage is bytes (or 8 bits)
  - Kilobyte is ~1000 bytes
  - Megabyte is ~1000 kilobyte
  - Gigabyte is ~1000 megabytes
  - Terabyte is ~1000 gigabytes
  - Petabyte is ~1000 terabytes
  - Exabyte is ~1000 petabytes

- Files persist on your computer after your program runs
- Files can be transferred before, during, or after your program runs

File formats

- Files are either stored as text or binary
- A file suffix usually indicates what the format of the file is

- Text files are human-readable and can be viewed and edited without a special program
  - Examples:
    - Simple text (grades.txt)
    - Web pages (index.html)

- Binary files are computer-readable and require a special program to view or edit
  - Examples:
    - Pictures (monalisa.jpg)
    - Music (rickroll.mp3)

Starting with text

- We'll begin with plain text files

- They are cross platform
- They don't require special programs or tools
- They are highly compressible
- They are human readable
  - We can parse them manually to start with

- We will be working with
  - .txt
  - .csv

Python and files

- Using files is a 3-step process
  1. Open the file in the desired mode
  2. Use the file
  3. Close the file

- Think about file input and output (file I/O) as a series of ***pipes***
  - Pipes establish a connection between two things
  - Pipes have directionality

- Some programmers may even say, “Pipe the output to a file”

File modes

- File are opened either in text mode or binary mode
  - “t” for text mode (the default for Python)
  - “b” for binary mode

- Additionally, each file can be opened for..
  - “r” for reading: getting data **from** a file
  - “w” for writing: putting data **into** a file
  - “a” for appending: adding data to the **end** of a file

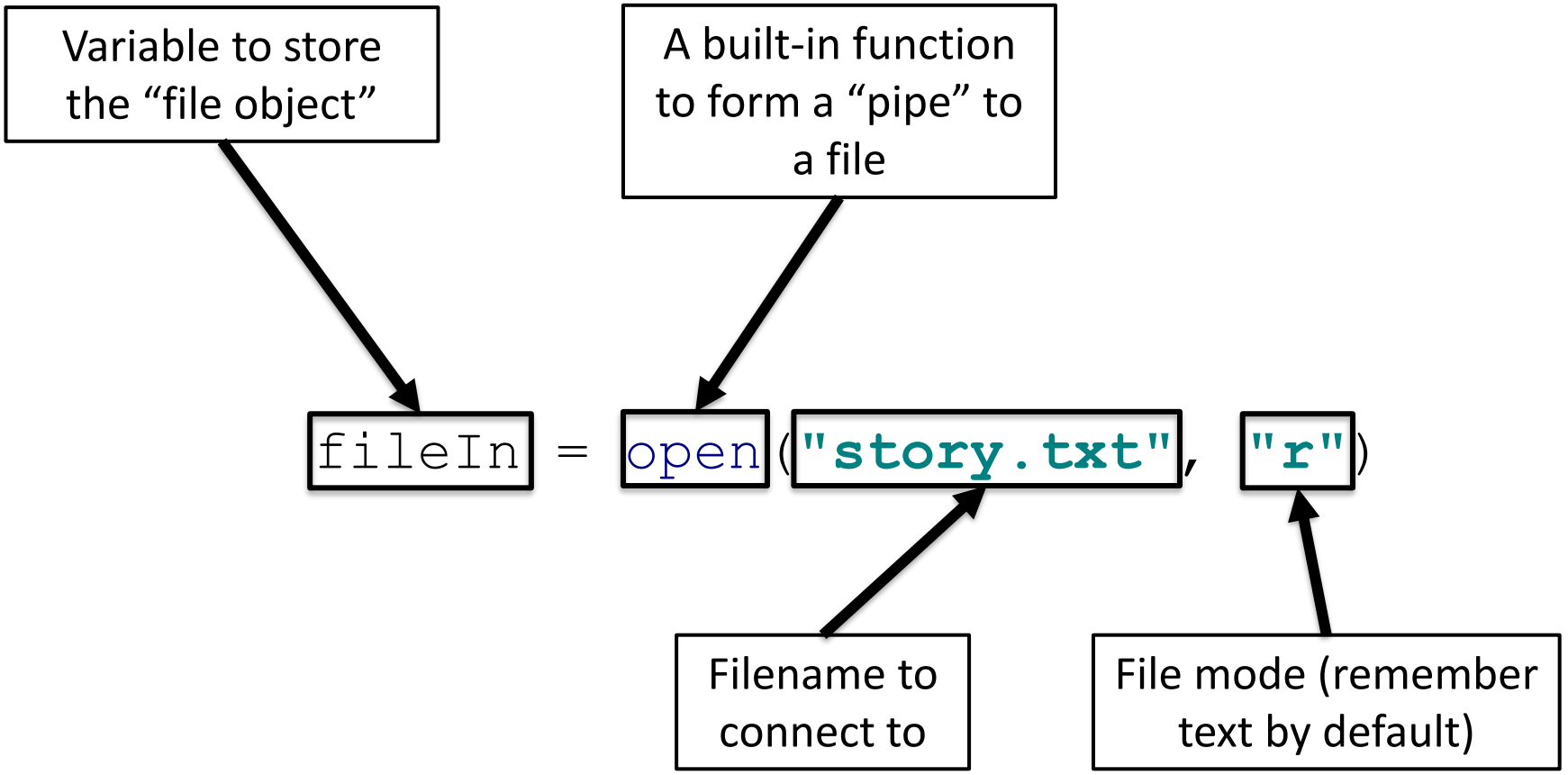
A program that reads a file

```
# 1. Open the file for reading
fileIn = open("story.txt", "r")

# 2. Read the file (line-by-line)
for line in fileIn:
    print(line)

# 3. Close the file
fileIn.close()
```

Opening a file

- Python has built-in functions for file manipulation
- The `open` function will generate a file object
- We'll talk about objects next week – for now it's a data type that's linked to a particular file and has special functions attached to it

Opening a file syntax



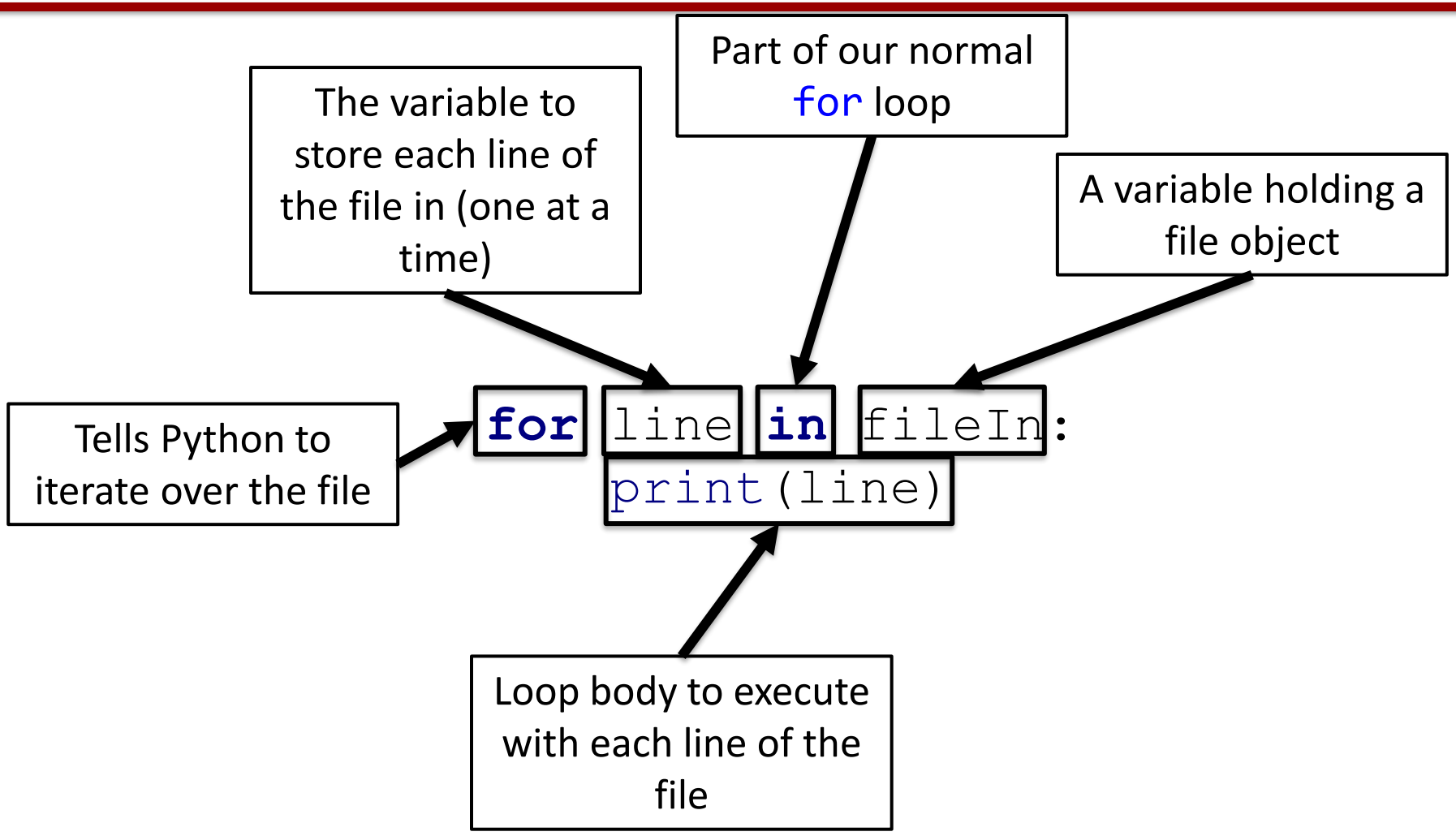
Reading a file line-by-line

- Once open for reading, Python will allow you to loop through the file with a `for in` loop

- Each iteration through the loop will put an entire line of text into the looping variable
  - A line is defined as all the text up to (and including) the next `"\n"`

- You can almost think of the file as a sequence of lines!
- And each item in the sequence is a ***string***

File reading syntax



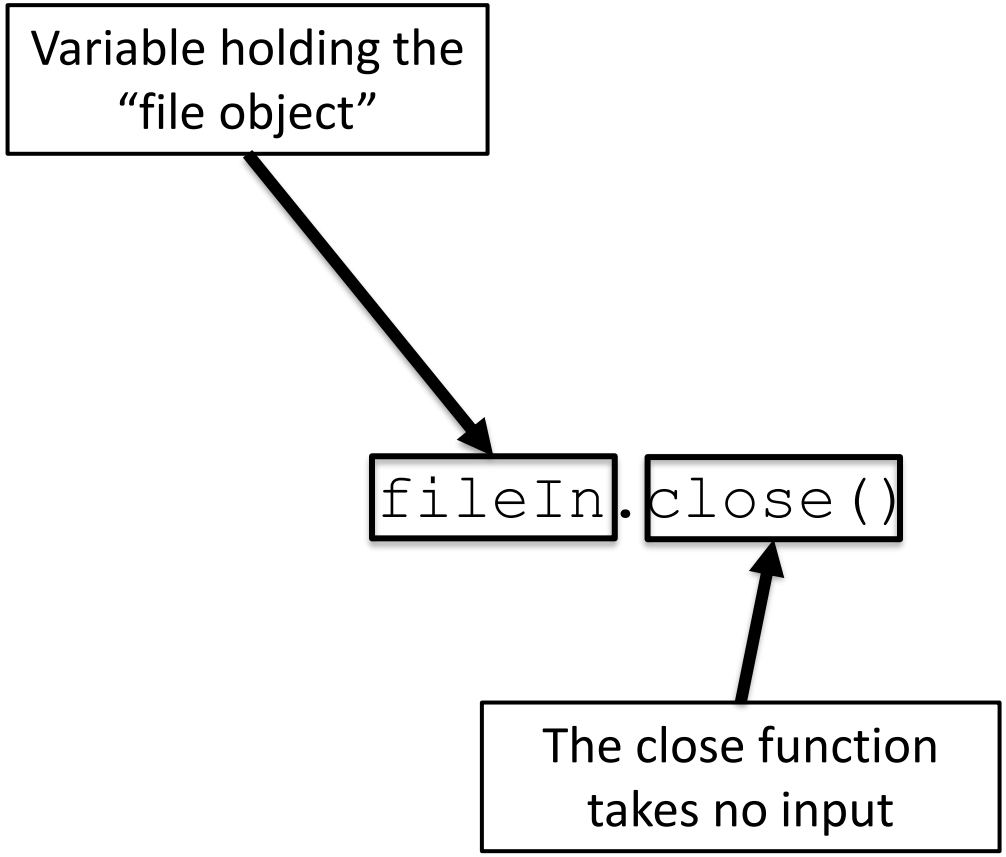
Closing a file

- When done with a file it's a GoodIdea® to close it

- This prevents the file from staying "open" for your entire program, just while you're reading from it

- It prevents file corruption which may occur when 2 programs attempt to read or write the same file

Closing a file syntax



Sample input

- Suppose I have a file called `story.txt` with the following:

```
In West Philadelphia, born and raised
on the playground was where I spent
most of my days. Chillin' out maxin'
relaxin' all cool, and all shooting
some b-ball outside of the school.
```

- What would be the output of the previous program?

Output

```
In West Philadelphia, born and raised

on the playground was where I spent

most of my days. Chillin' out maxin'

relaxin' all cool, and all shooting

some b-ball outside of the school.
```

- ?!?!?!?!?!?!?!

Examining the input file

- Remember, reading a line of a file includes the new line!

```
In West Philadelphia, born and raised
on the playground was where I spent
most of my days. Chillin' out maxin'
relaxin' all cool, and all shooting
some b-ball outside of the school.
```

Examining the input file

- Remember, reading a line of a file includes the new line!

```
In West Philadelphia, born and raised\n
on the playground was where I spent\n
most of my days. Chillin' out maxin'\n
relaxin' all cool, and all shooting\n
some b-ball outside of the school.\n
```

Stripping extra whitespace

- Remove any whitespace from the beginning and end of a string with **strip**

- Whitespace is:
  - A space
  - A tab
  - A newline

- Returns the edited string
  - Doesn’t modify the source string because strings are immutable!

Using strip

- Without strip

```
msg = "\tHello!\nBob\n\n"
print(msg)
```

```
        Hello!
Bob
```

- With strip

```
msg = "\tHello!\nBob\n\n"
print(msg.strip())
```

```
Hello!
Bob
```

- Notice the whitespace between words remains

Lets fix that bug!

```
# 1. Open the file for reading
fileIn = open("story.txt", "r")

# 2. Read the file (line-by-line)
for line in fileIn:
    line = line.strip()
    # Do something with line now

# 3. Close the file
fileIn.close()
```

Warnings about file input

- Remember data comes from a text file as strings – just like the input() function!
- What if we want integers – like from primes.txt?

```
2
3
5
7
11
```

- Just cast to an integer, like with `input`!

Reading in numbers

```
# 1. Open the file for reading
fileIn = open("primes.txt", "r")

# 2. Read the file (line-by-line)
for line in fileIn:
    number = int(line.strip())
    # Do something with number now

# 3. Close the file
fileIn.close()
```

2 ways to do the same thing

```
# Displays every line in the file
fileIn = open("story.txt")
for line in fileIn:
    line = line.strip()
    print(line)
fileIn.close()

# Also displays every line in the file and
# closes the file for us
with open("story.txt") as fileIn:
    for line in fileIn:
        print(line.strip())
```

About using “with”

- The **`with`** keyword implicitly calls the `open` and `close` on the file – as soon as the block is over, the file is closed

- It can work most of the time for basic file input and output, but not always (it is not fixed when the file actually gets closed)

- The 3-step process will ALWAYS work

Lab 3 and assignment 2 are now assigned