Functions: Usage

CPSC 217: Introduction to Computer Science for Multidisciplinary Studies I
Jul 2021 - CBE

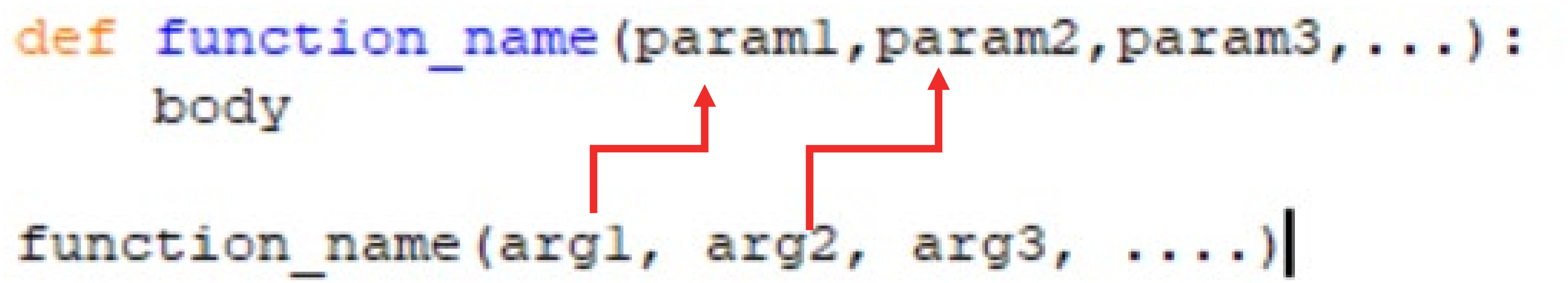
Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Wednesday, June 2, 2021

UNIVERSITY OF
CALGARY

Function calling review

Select a descriptive name for your function



```
def function_name(param1,param2,param3,...):
    body

function_name(arg1, arg2, arg3, ....)
```

Use brackets when calling functions even if you are not passing any arguments

At least one statement needs to be in a function.

Functions must be defined before they are called!

How to use a function?

- Call function
- Pass valid inputs
- Store the result in a variable

If function returns a value:

```
returnedValue = functionName(values/variables)
```

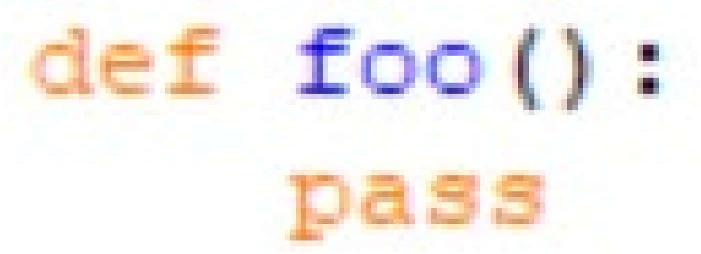
If no value is returned

```
functionName(values/variables)
```

Functions that do nothing

Functions have to have one line of code in them

- Only way to make pythons syntax parsing that is looking for indentation happy
- (Once you put something indented in function the rest of indentation has to match)
  - This is also true for conditionals and loop indentation
- Can use pass keyword to do nothing

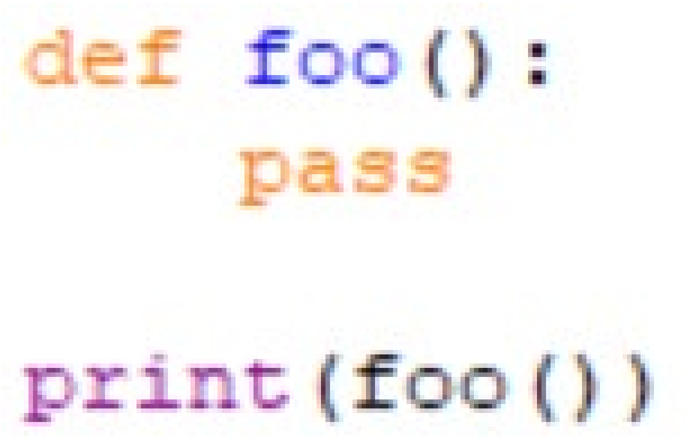


```
def foo():
    pass
```

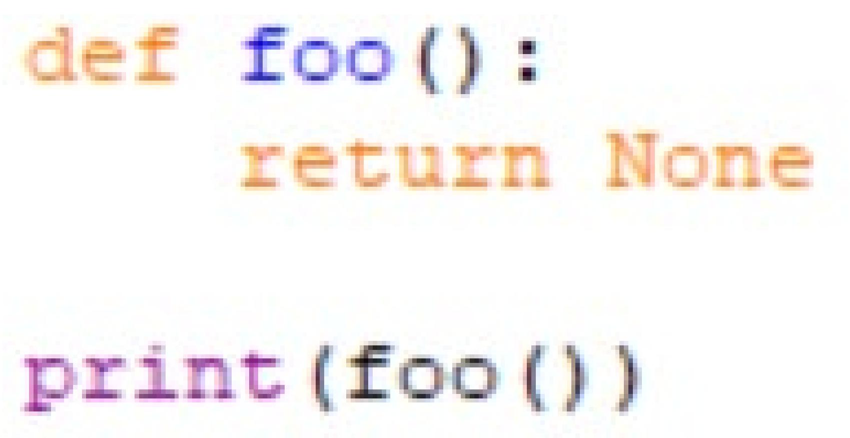
Functions return None by default

Functions in python always return something

- That something is by default nothing or None
- None is a special keyword
- (We often use None in other places in our code to show nothing has been stored in a variable yet)



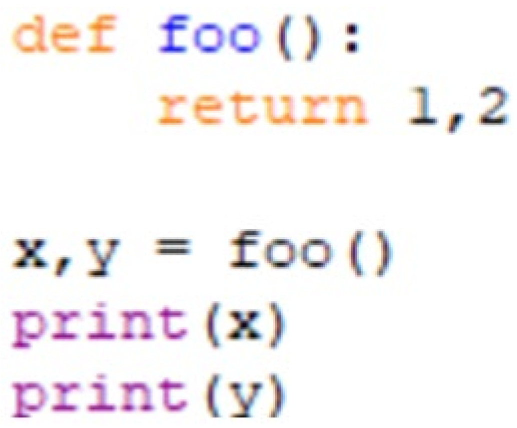
```
def foo():
    pass

print(foo())
```



```
def foo():
    return None

print(foo())
```

Return multiple things

Functions can return multiple things



```
def foo():
    return 1,2

x,y = foo()
print(x)
print(y)
```

Return values

- Format

```
def <function name> (param1, param2, ...):
        body
        return var1, var2, ...
```

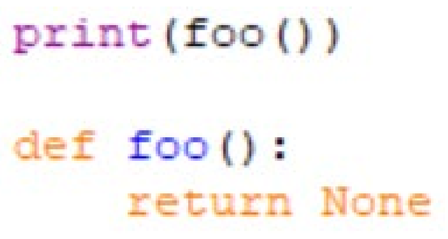
- The results can be stored into variables for later use

```
var1, var2, ... = <function name> (arg1, arg2, ...)
```

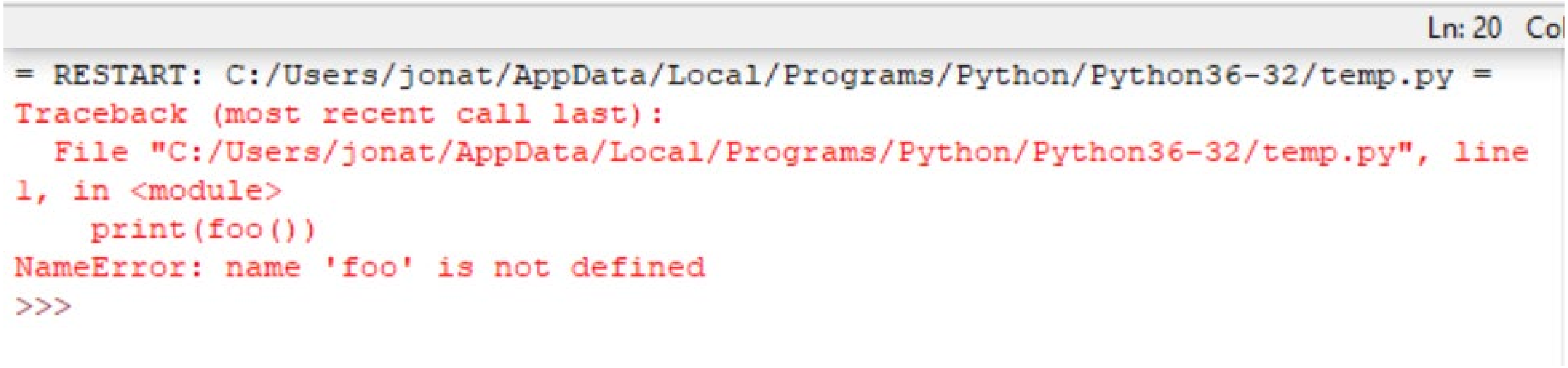
Namespace

Must define functions before use

Functions must be declared before use



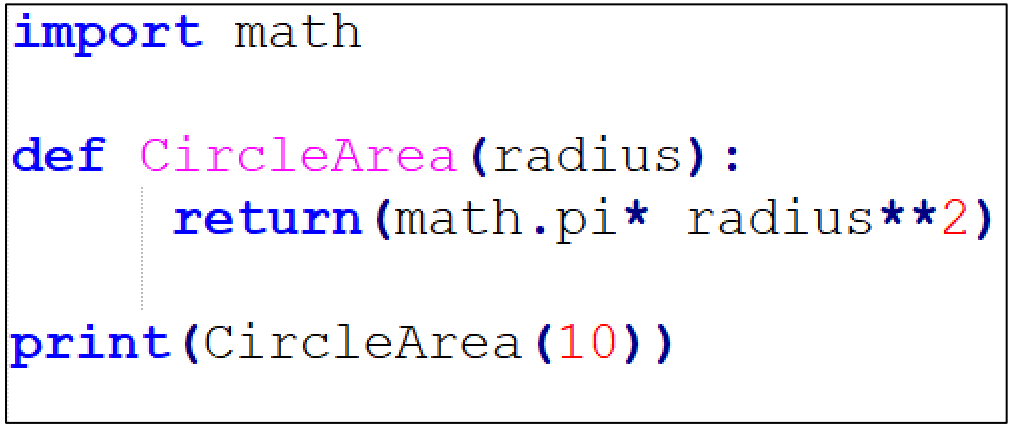
```python
print(foo())

def foo():
    return None
```



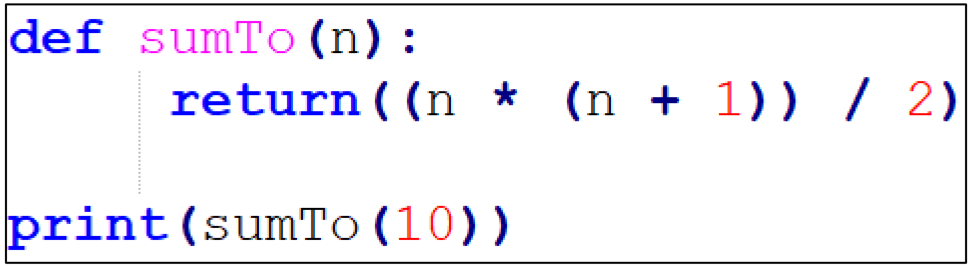
```
= RESTART: C:/Users/jonat/AppData/Local/Programs/Python/Python36-32/temp.py =
Traceback (most recent call last):
  File "C:/Users/jonat/AppData/Local/Programs/Python/Python36-32/temp.py", line
1, in <module>
    print(foo())
NameError: name 'foo' is not defined
>>>
```

Examples

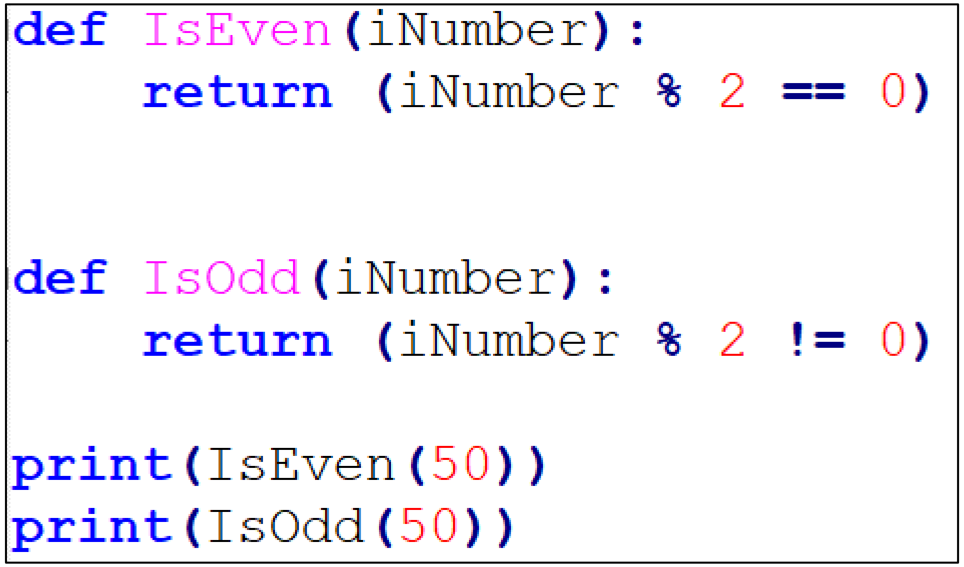
Some simple functions



```
import math

def CircleArea(radius):
    return(math.pi* radius**2)

print(CircleArea(10))
```



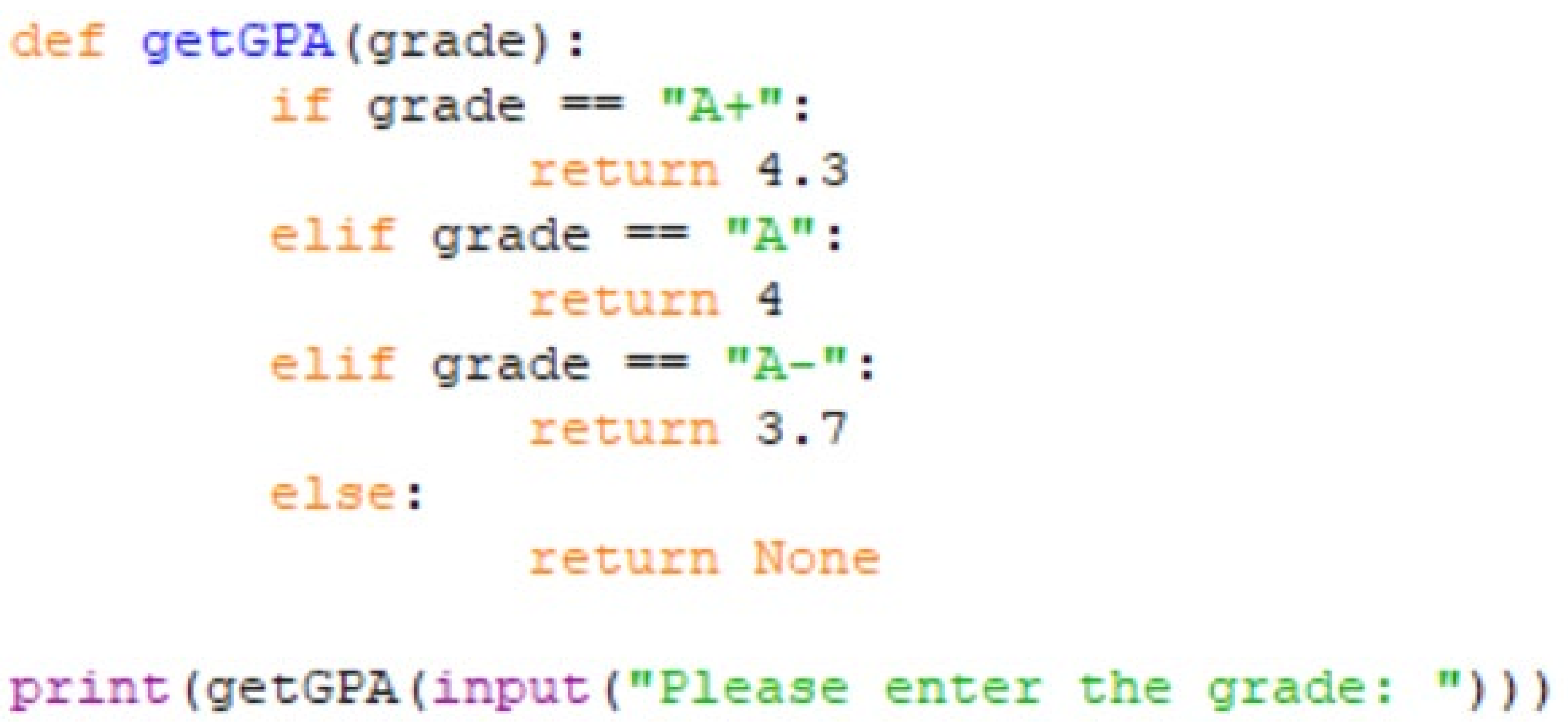
```
def sumTo(n):
    return((n * (n + 1)) / 2)

print(sumTo(10))
```



```
def IsEven(iNumber):
    return (iNumber % 2 == 0)


def IsOdd(iNumber):
    return (iNumber % 2 != 0)

print(IsEven(50))
print(IsOdd(50))
```

Design

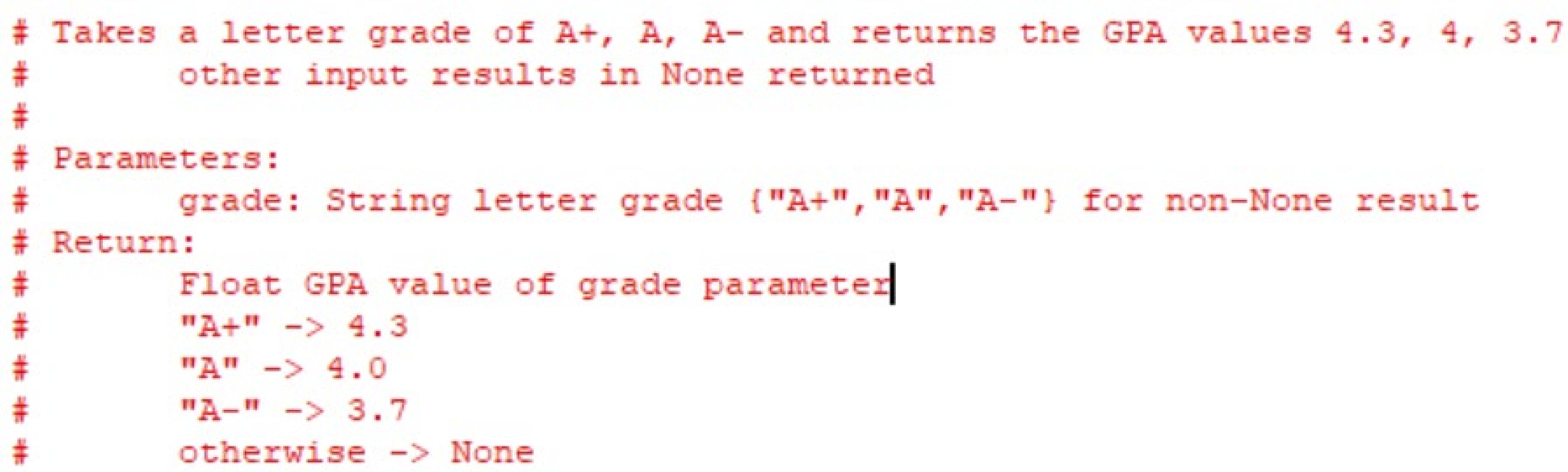
There are challenges in defining a function



```python
def getGPA(grade):
        if grade == "A+":
                return 4.3
        elif grade == "A":
                return 4
        elif grade == "A-":
                return 3.7
        else:
                return None

print(getGPA(input("Please enter the grade: ")))
```

User-Defined Functions - Commenting

- A good function always contains explicit comments that describe the purpose of the function, the parameters, and returned values.



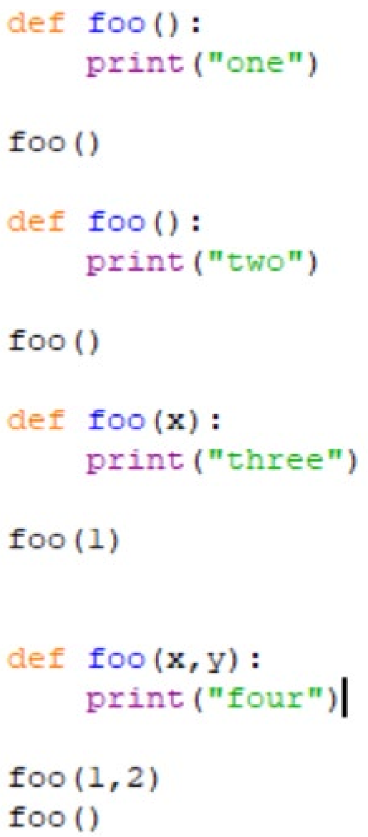
```
# Takes a letter grade of A+, A, A- and returns the GPA values 4.3, 4, 3.7
#       other input results in None returned
#
# Parameters:
#       grade: String letter grade {"A+","A","A-"} for non-None result
# Return:
#       Float GPA value of grade parameter
#       "A+" -> 4.3
#       "A" -> 4.0
#       "A-" -> 3.7
#       otherwise -> None
```

Namespace

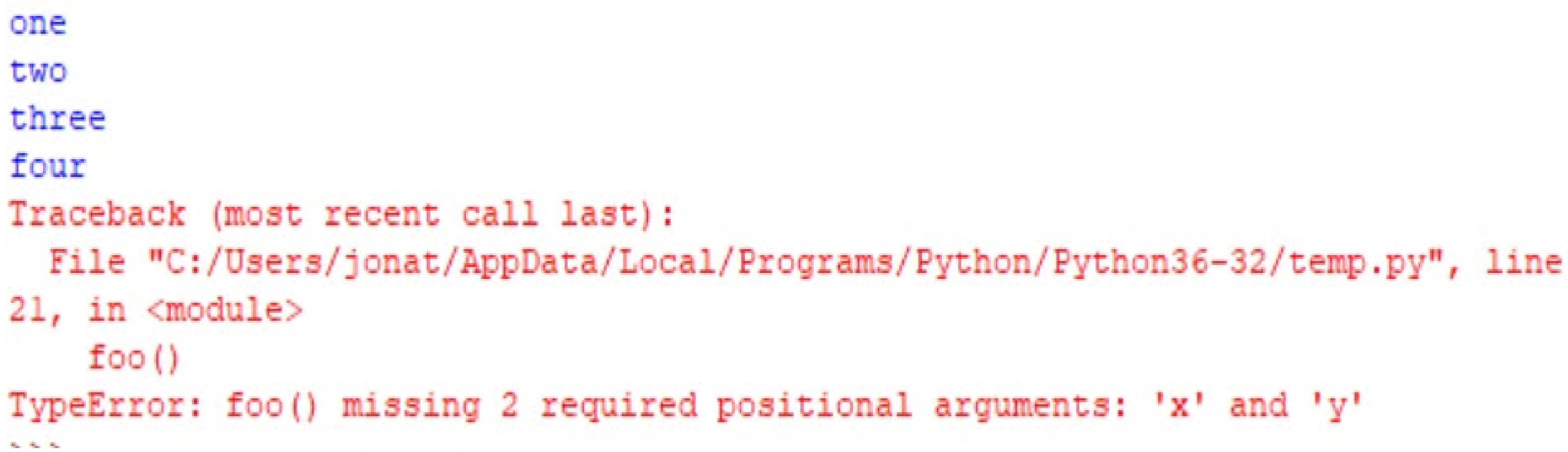
Re-defining functions

Dangers of functions (re-use name)

- Python only lets you have one function per name, but you can override previous usage (ignores parameters unlike other languages)



```
def foo():
    print("one")

foo()

def foo():
    print("two")

foo()

def foo(x):
    print("three")

foo(1)


def foo(x,y):
    print("four")

foo(1,2)
foo()
```



```
one
two
three
four
Traceback (most recent call last):
  File "C:/Users/jonat/AppData/Local/Programs/Python/Python36-32/temp.py", line
21, in <module>
    foo()
TypeError: foo() missing 2 required positional arguments: 'x' and 'y'
```

Parameter order

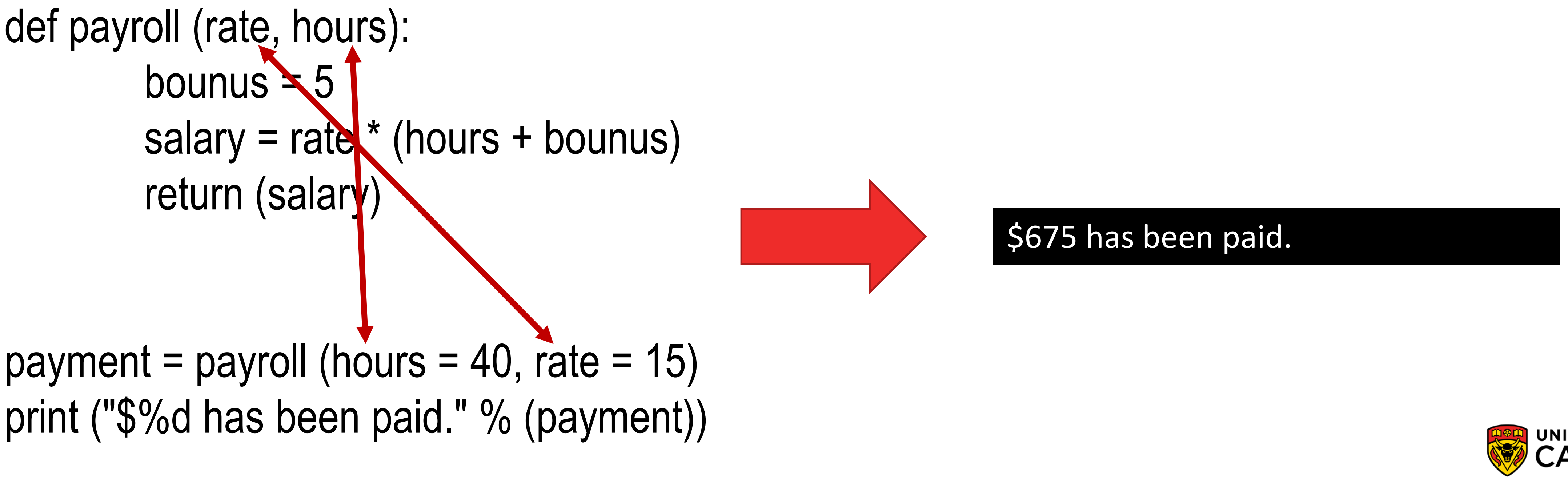
Calling Functions - Order of Parameters

- Function parameters are position sensitive.
- When calling a function that accepts parameters, make sure your arguments are in the same order of the parameters.
- **WARNING:** Not following the order of the parameters will result in parameters having wrong values, which may lead to semantic and runtime errors.

```
def printbar(char, num = 10):
    bar = ''
    for i in range(num + 1):
        bar = bar + char
    print(bar)

printbar(20, '=')
```

Keyword parameters

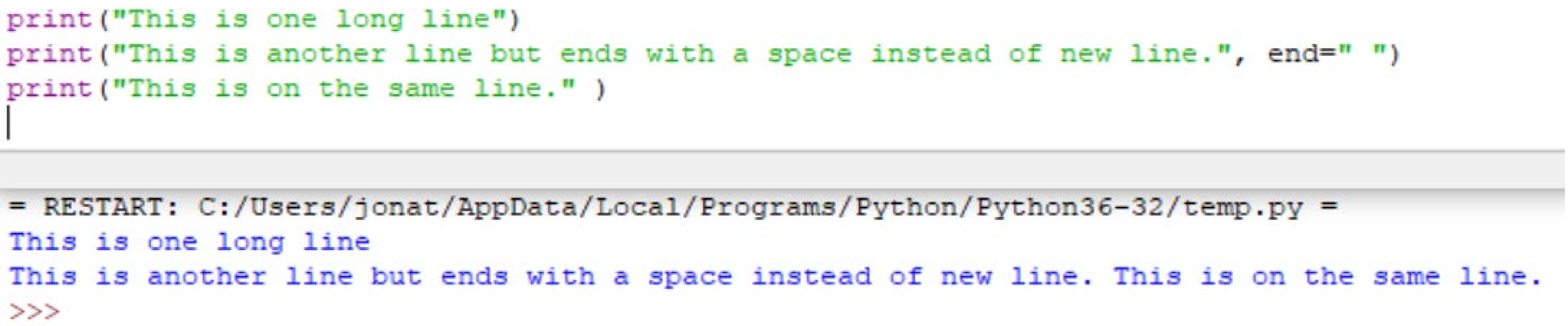
- Keyword parameters allow us to match arguments with parameters by name, instead of positions



```
def payroll (rate, hours):
          bounus = 5
          salary = rate * (hours + bounus)
          return (salary)

payment = payroll (hours = 40, rate = 15)
print ("$%d has been paid." % (payment))
```

```
$675 has been paid.
```

We can do this with functions you already use



```
print("This is one long line")
print("This is another line but ends with a space instead of new line.", end=" ")
print("This is on the same line." )
```

```
= RESTART: C:/Users/jonat/AppData/Local/Programs/Python/Python36-32/temp.py =
This is one long line
This is another line but ends with a space instead of new line. This is on the same line.
>>>
```

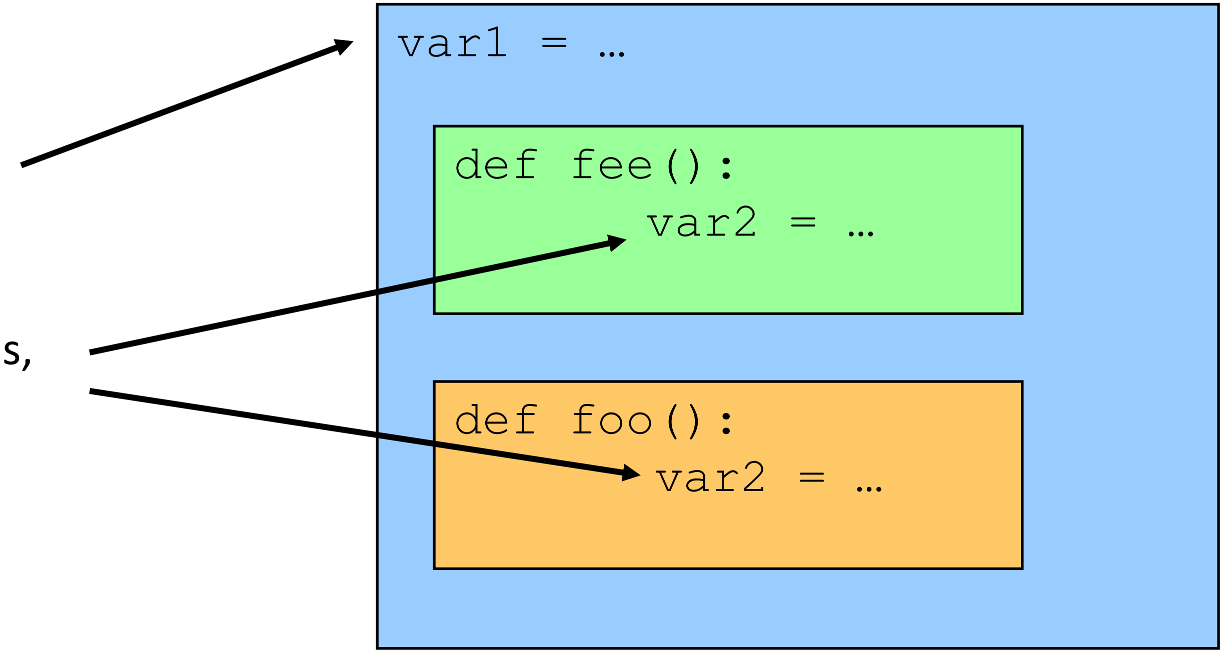
Scope

Scope of Variables

- Variables are memory locations that are used for the temporary storage of data
- The scope of a variable is the section of code in which it is accessible

The global scope:
Accessible by both functions

Local scopes:
Two different memory spaces,
Accessible only within their
functions



```
var1 = …

    def fee():
            var2 = …

    def foo():
            var2 = …
```

Scope of Variables - Local Variables

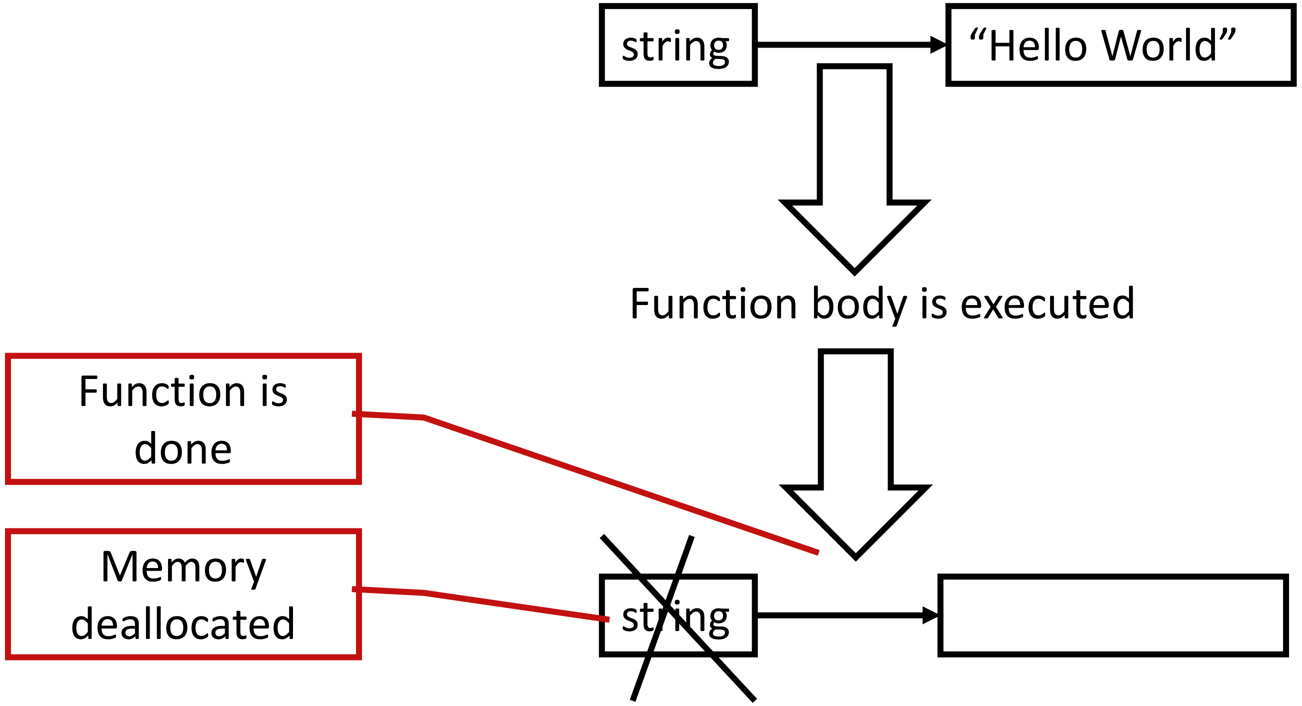
- Local variables are only accessible to the function where they are defined.
- The memory for local variables is only allocated (reserve the memory) when the function is running and deallocated (free up the memory) when the function reaches the end.

- Local variables are defined (memory allocated and value stored) each time the function is called.

Scope of Variables - Local Variables

```
def foo():
    string = "Hello World!"
    print(string)
```

string is a local variable



Function body is executed

Function is done

Memory deallocated

Scope of Variables - Global Variables

- Variables that are declared within the body of a function have a **local scope** → Accessible from inside the function only
  - This includes the parameters

- Variables that are declared outside the body of a function have a **global scope** → Accessible from anywhere in the program

- In Python, global variables can only be modified in global scope.
- They cannot be modified in local scope unless the global keyword is used:
  - **global variableName**

Scope of Variables - Global Variables

```
def failedChange():
        someGlobalVar = "Without Using Global Keyword"

def successfulChange():
        global someGlobalVar
        someGlobalVar = "Using Global Keyword"


someGlobalVar = "I am Global"
print(someGlobalVar)

failedChange()
print(someGlobalVar)

successfulChange()
print(someGlobalVar)
```

```
I am Global
```

```
I am Global
```

```
Using Global Keyword
```

Scope of Variables - Variable lifetime

- The lifetime of a variable is the time that a variable is allocated a memory space.
- The memory is allocated at the time of variable declaration

- **Global variables** exist until the program terminates
- **Local variables** exist until the function containing it finishes

Memory

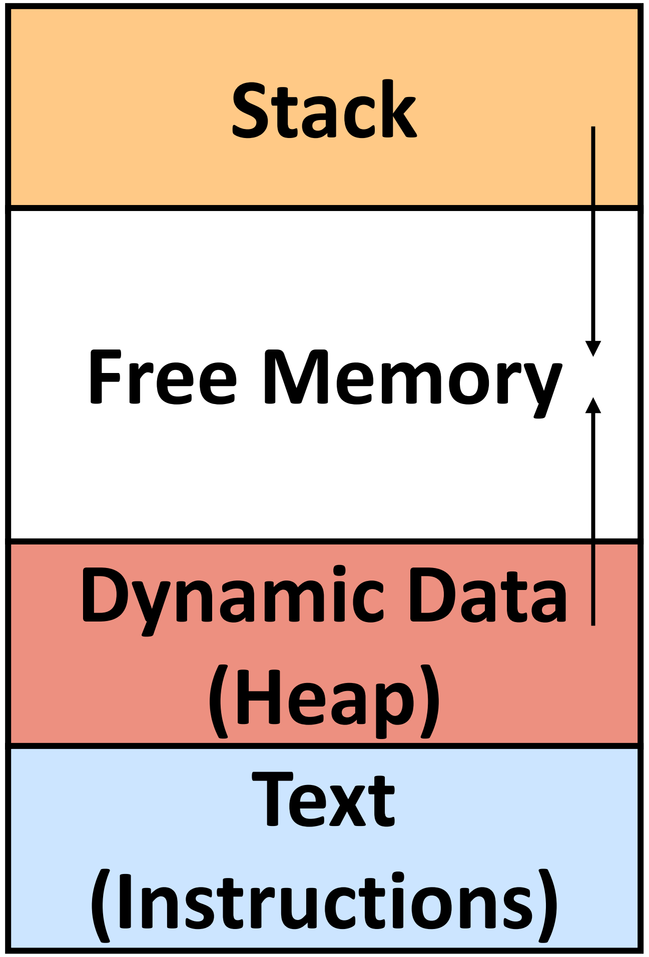
Memory Organization

- The memory for a program is organized into three regions
  - Text (Instructions)
  - Dynamic Data (Heap)
  - Stack

Memory Organization

- The memory for a program is organized into three regions
  - **Text (Instructions)**: holds program instructions. Contrary to what the name suggests, code is in binary machine code (not human-readable). Generally read-only.
  - **Dynamic Data (Heap)**: objects allocated as the program runs
  - **Stack**: information about function calls, including all pointers for local variables. Very common to have pointers from the stack into the heap (not common the other way around).

Memory Organization



Stack

Free Memory

Dynamic Data (Heap)

Text (Instructions)

writable; not executable

Managed “automatically” by compiler

writable; not executable

Managed by programmer

read-only; executable

Initialized when process starts

Memory Organization

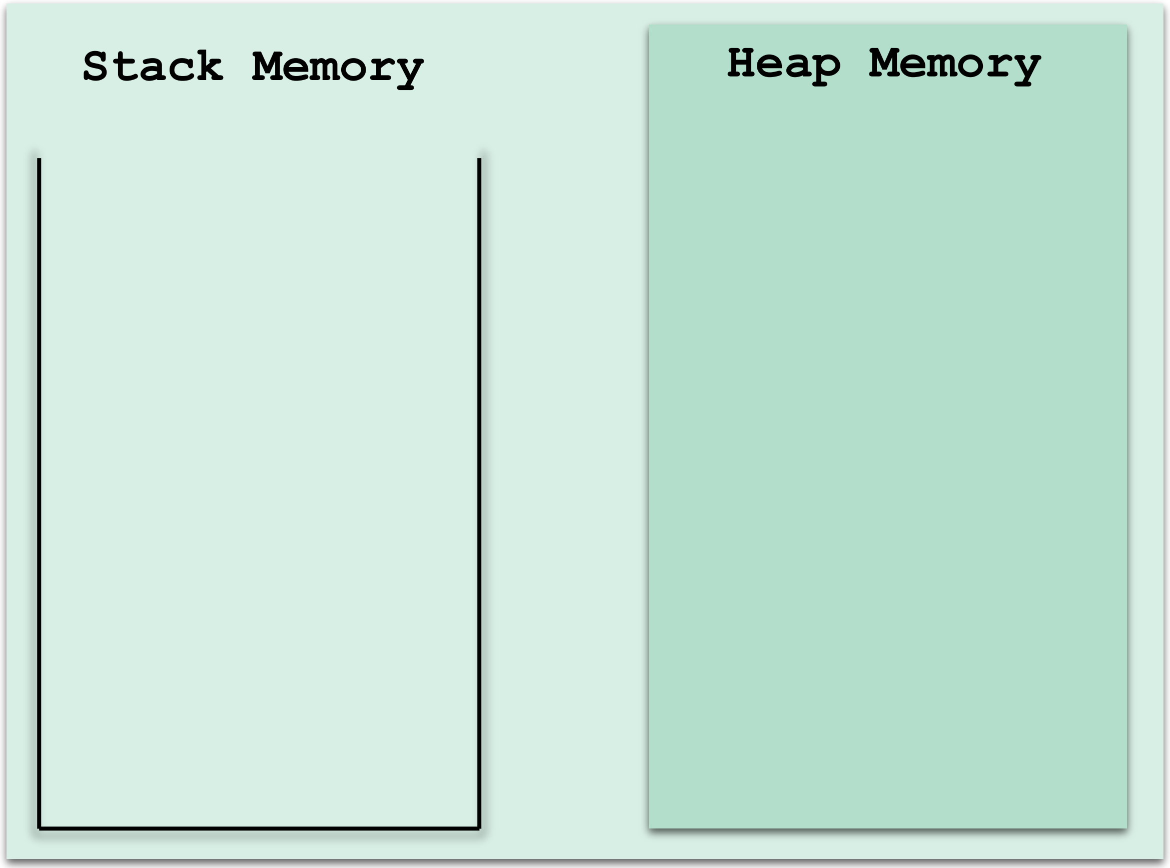
- Everything in Python is an object. Variables are labels that refer to these objects.
- All objects are stored in the heap.
- If the labels (variables) are created in local scope, then the **label** is stored in the stack memory. Otherwise, the label is stored in heap memory.

- Lets run through a simulation of Python’s memory organization to clarify these concepts. This simulation simplifies some aspects for clarity’s sake.

Memory Organization - Walkthrough

Define a global variable *x* by assigning the value 10 to it.

Instructions (code):

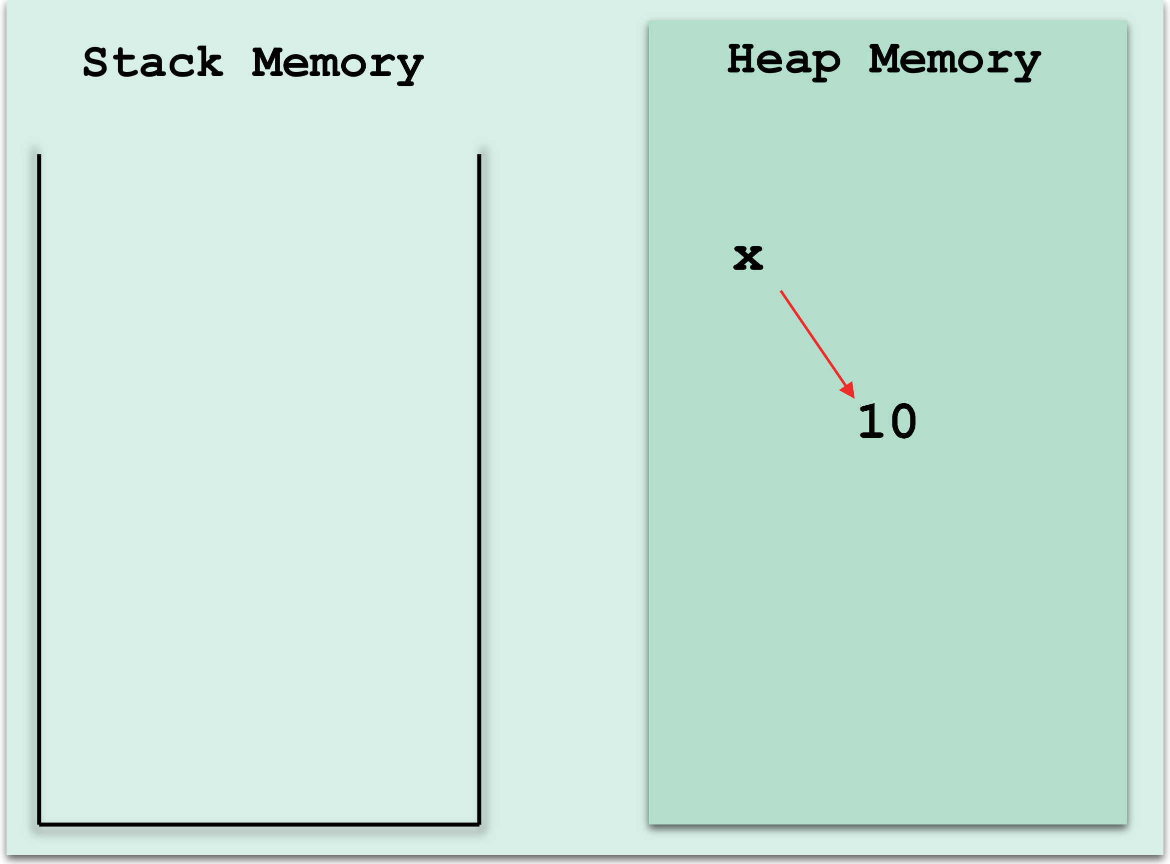


Memory Organization - Walkthrough

The value 10 is an object, so it is stored in heap.
The variable is global, so it is stored in heap as well

Instructions (code):

```
x = 10 #global variable
```

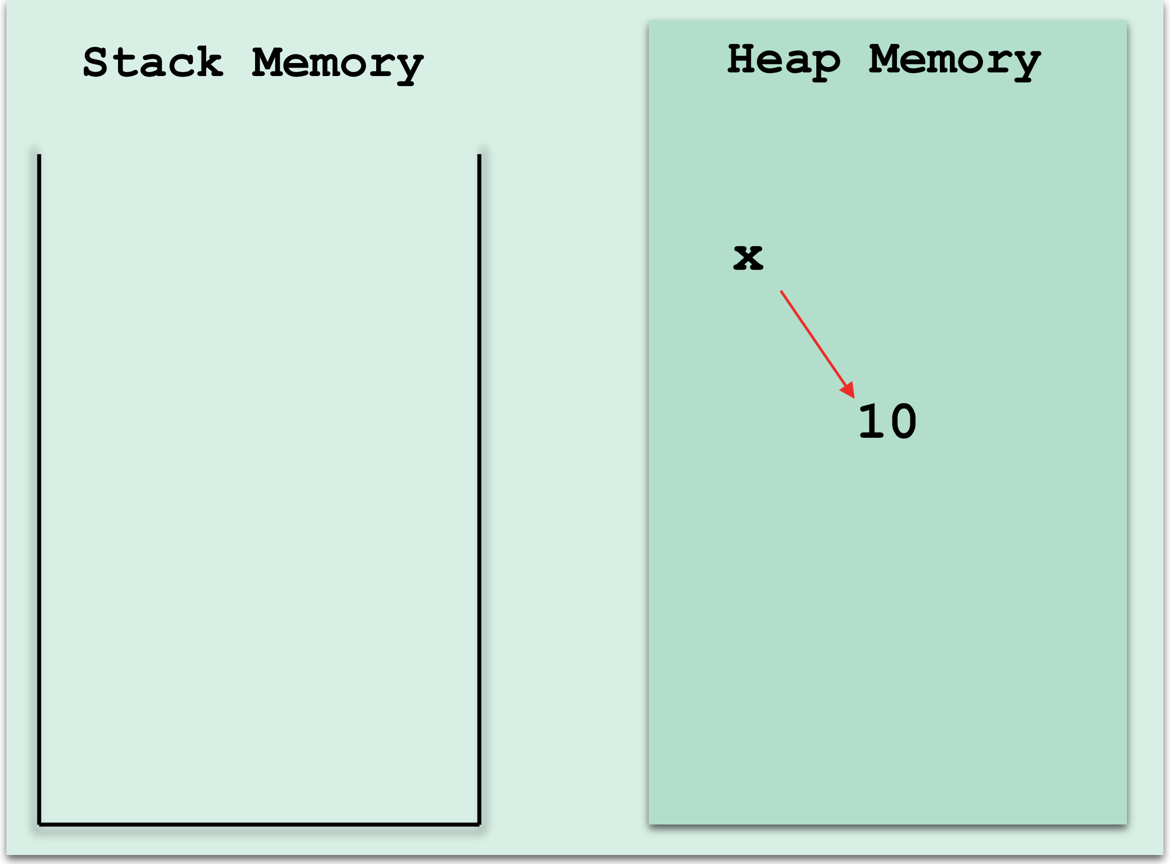


Memory Organization - Walkthrough

Define another variable *y* = 10

Instructions (code):

```
x = 10 #global variable
```



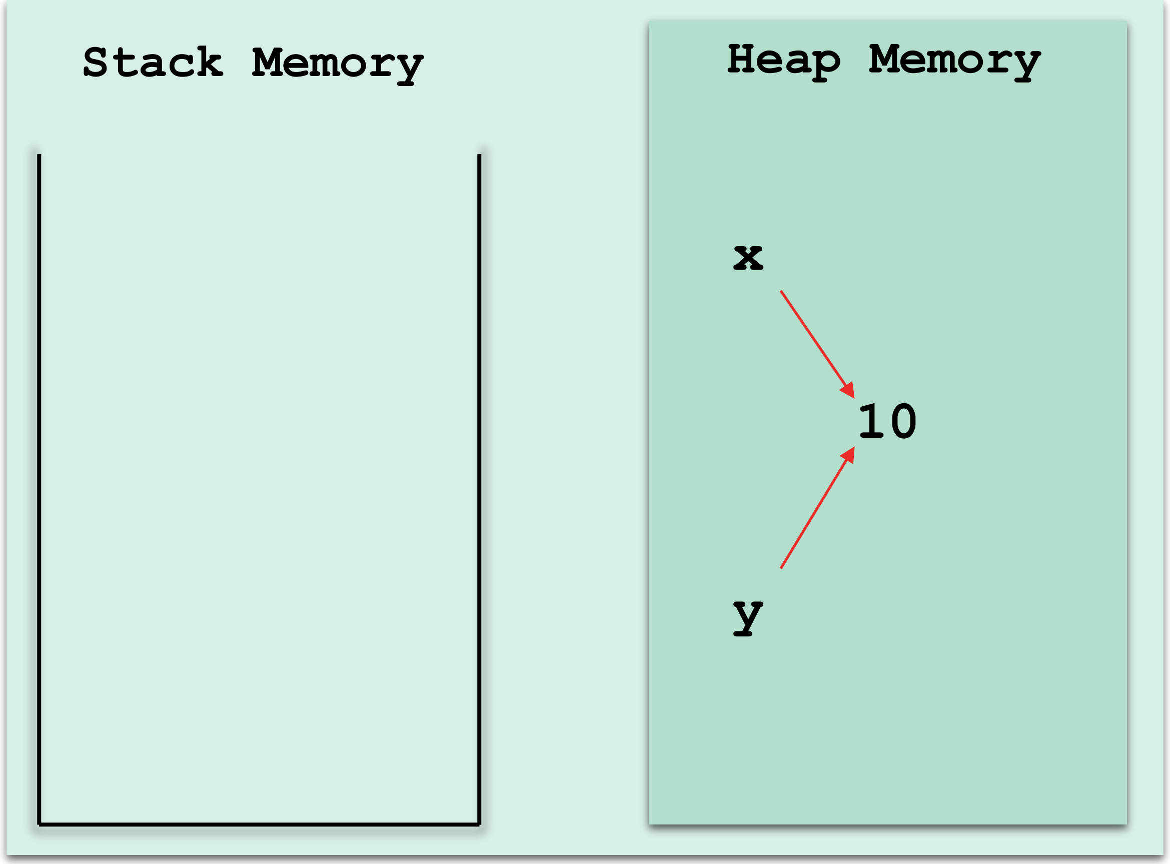
Memory Organization - Walkthrough

Again, *y* is global and 10 is an object, so into the heap they go.

Notice that the object 10 is not recreated; to preserve memory space.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
```

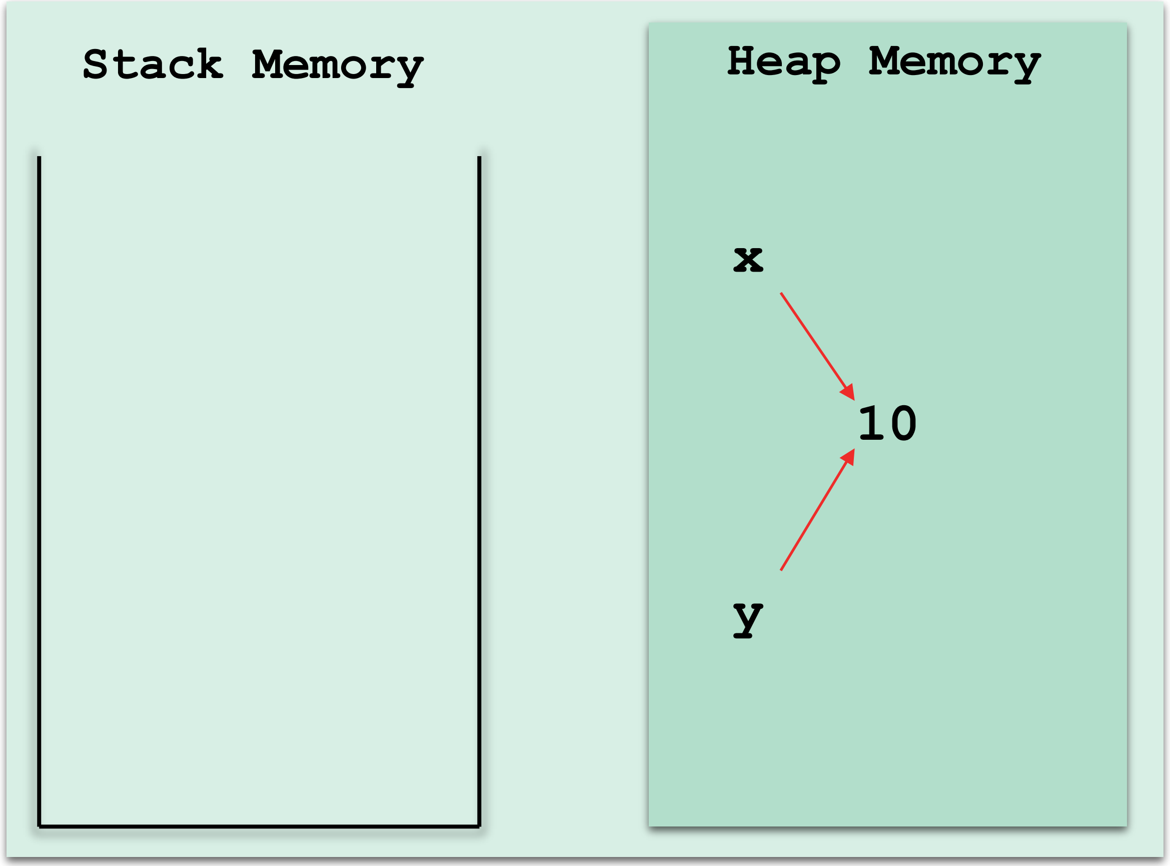


Memory Organization - Walkthrough

Increment *y* by 1.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
```

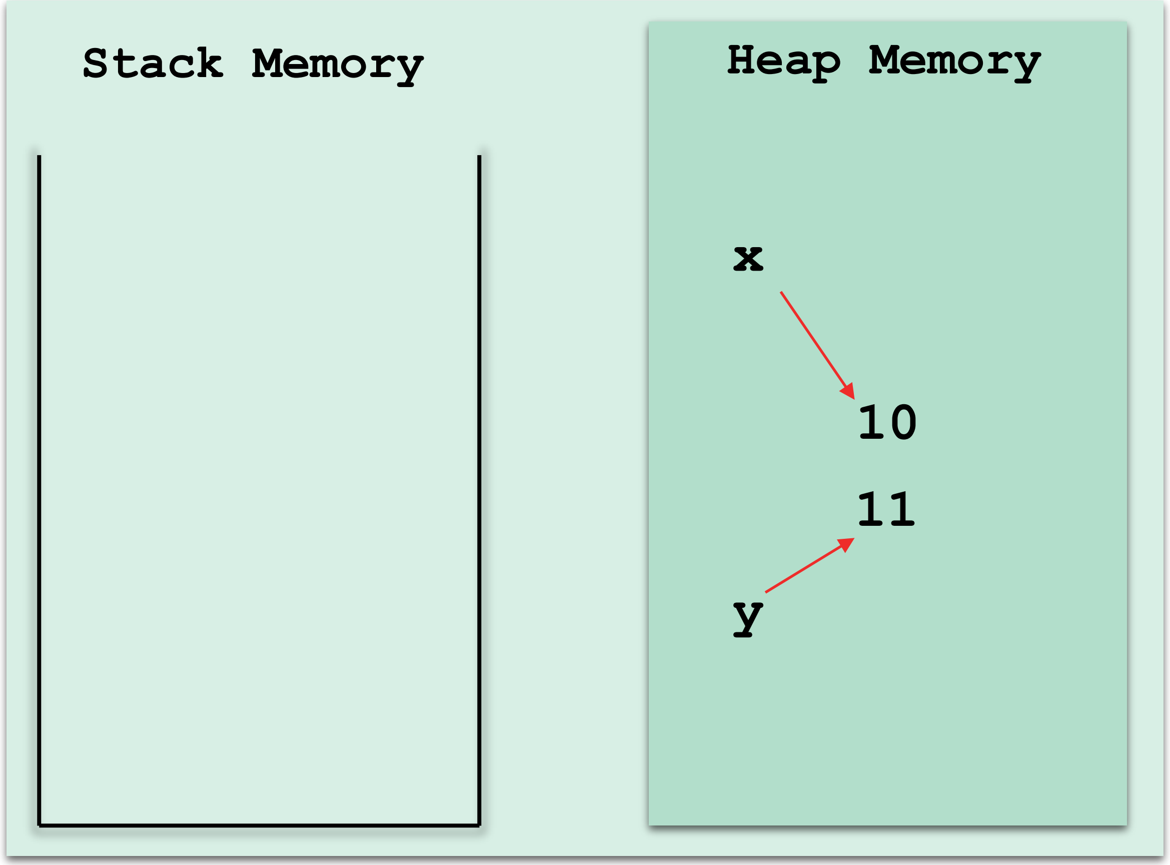


Memory Organization - Walkthrough

A new object, 11, is created and *y* refers (points) to it.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
```

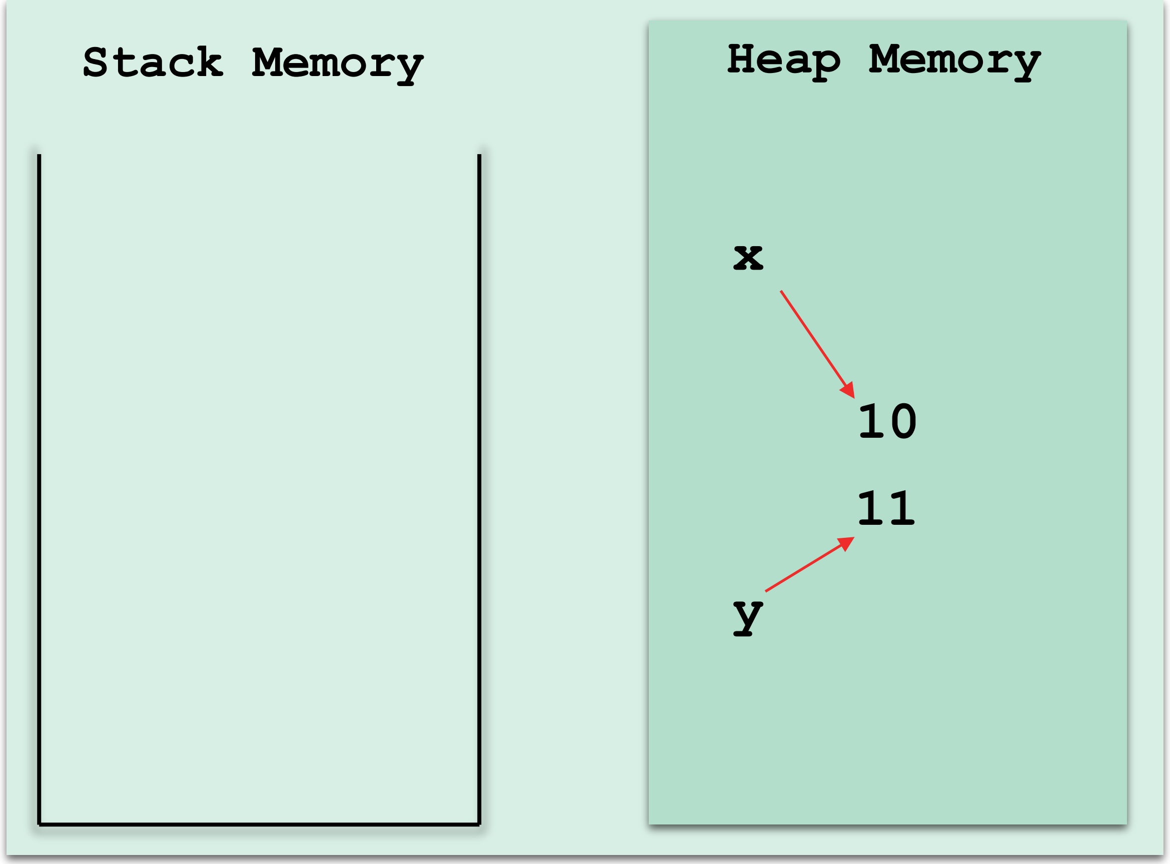


Memory Organization - Walkthrough

Define a function, *increment*, that accepts one argument, a, add one to it and store it in variable *z* then return *z*.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
```



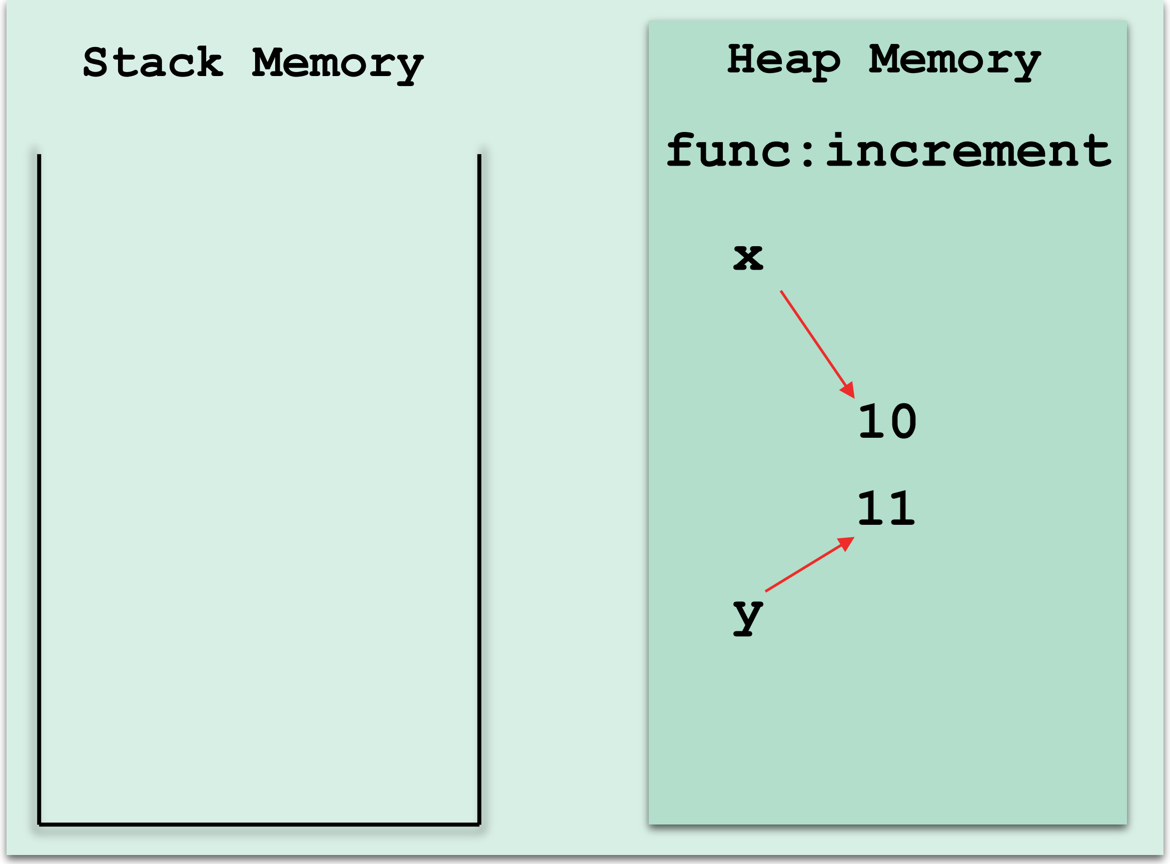
Memory Organization - Walkthrough

The function code is stored in the Text memory. The reference to the function is global so it is stored in the heap.

*This is over simplified for this class's purposes.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
        z = a + 1
        return z
```

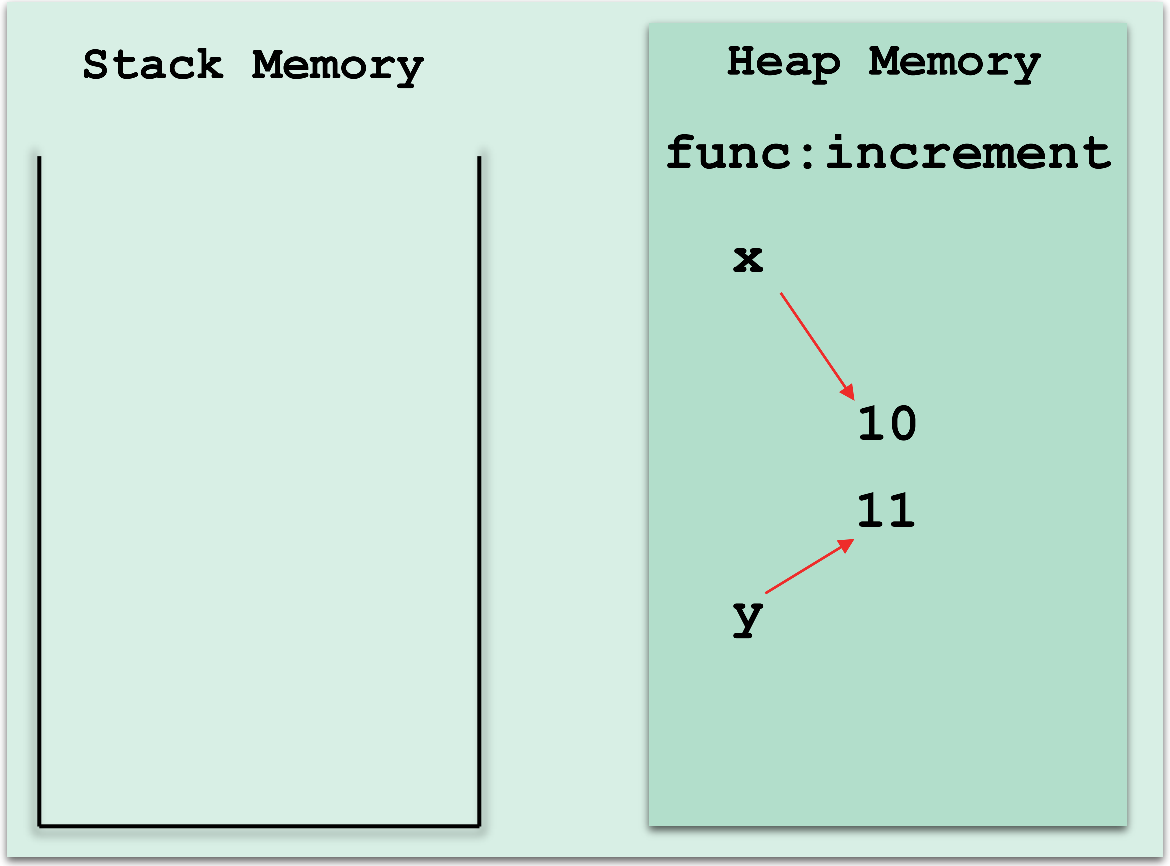


Memory Organization - Walkthrough

Call *increment* and pass *y* to it.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
       z = a + 1
       return z
```



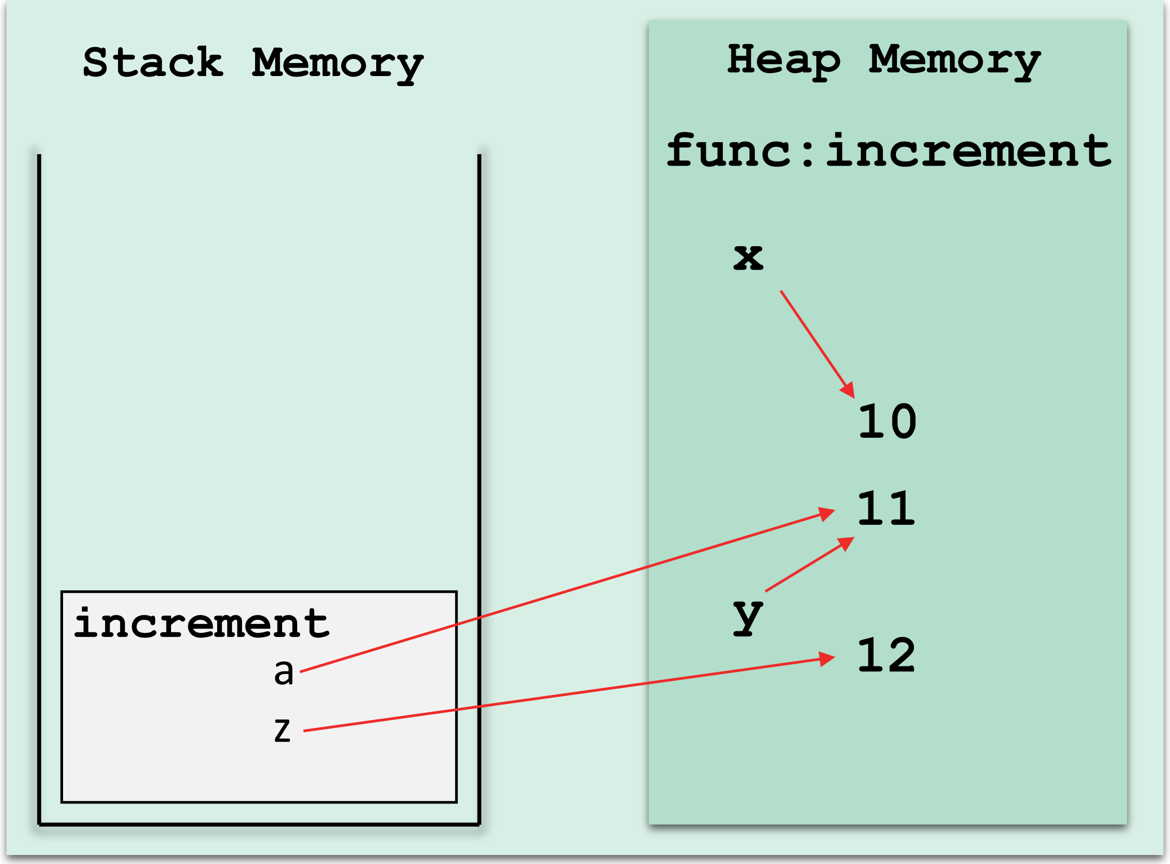
Memory Organization - Walkthrough

The function call gets stored in the stack along with all the local variables.

The parameter *a* points to the 11 object in the heap. A new object 12 is created and *z* points to it.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
        z = a + 1
        return z
z = increment(y)
```

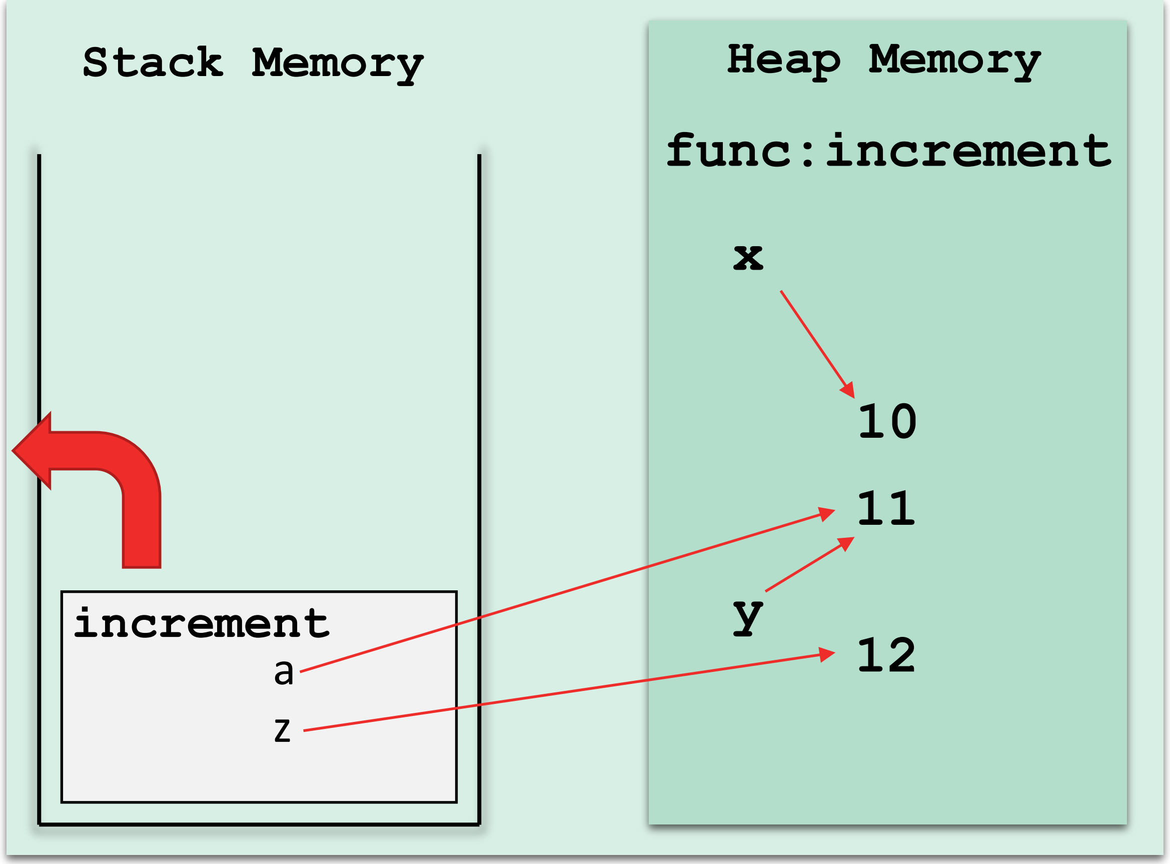


Memory Organization - Walkthrough

When the function execution ends, it gets popped out of the stack; its local variables' reference are deallocated. Its returned value is stored in the caller's scope (global)

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
        z = a + 1
        return z
z = increment(y)
```

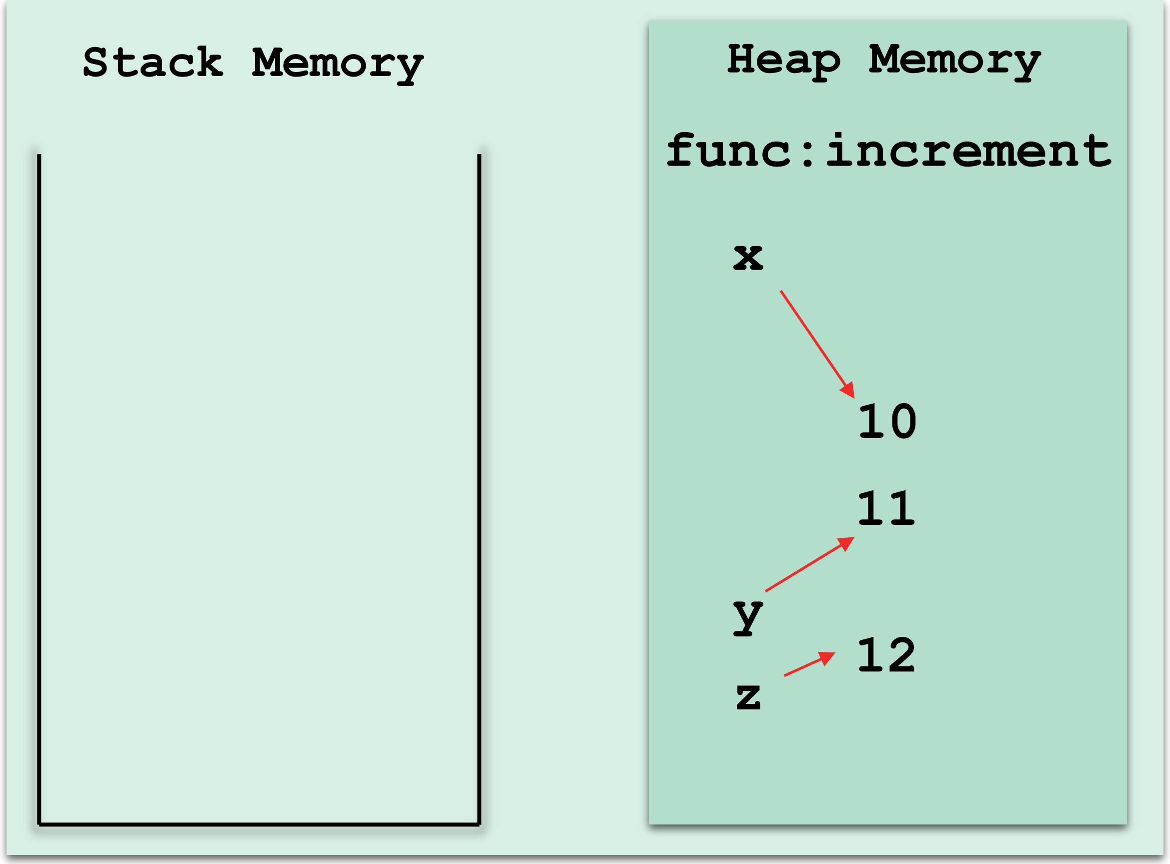


Memory Organization - Walkthrough

When the function execution ends, it gets popped out of the stack; its local variables' reference are deallocated. Its returned value is stored in the caller's scope (global)

Instructions (code):

```python
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
        z = a + 1
        return z
z = increment(y)
```

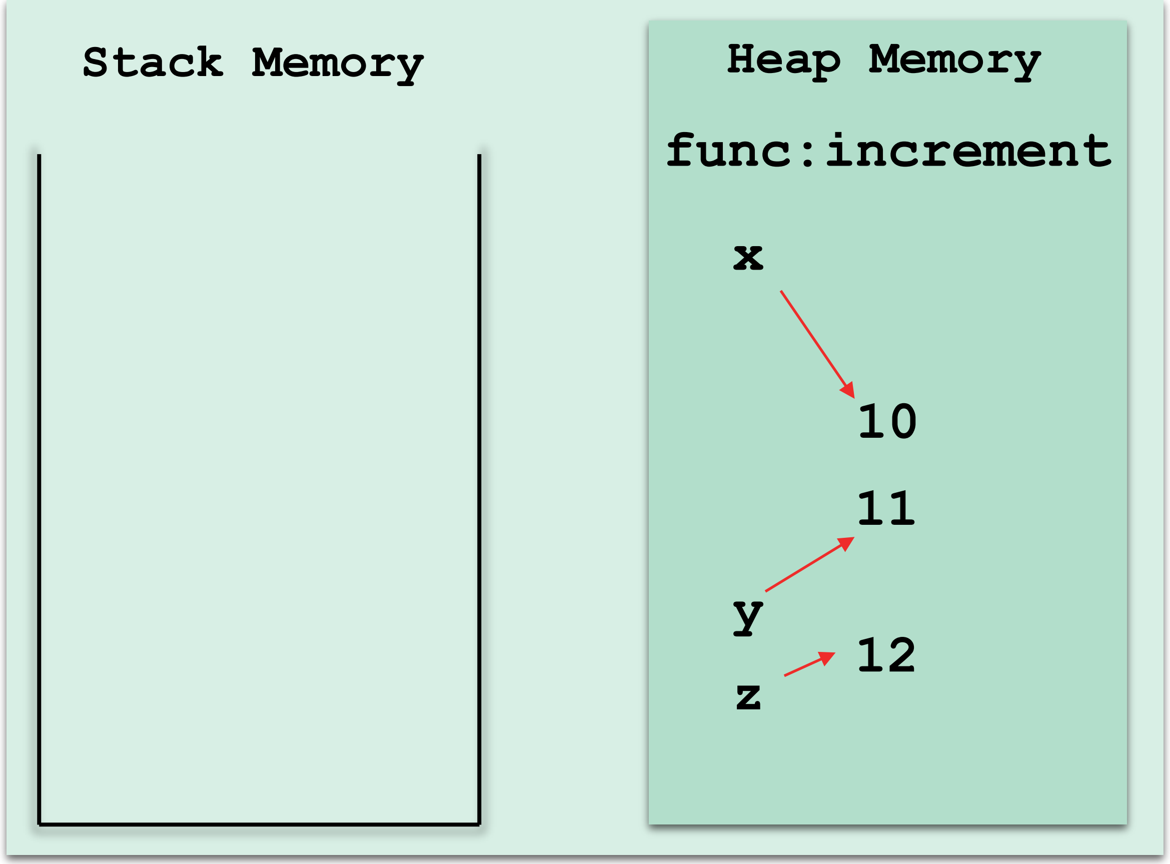


Memory Organization - Walkthrough

Delete the variables y, and z

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
       z = a + 1
       return z
z = increment(y)
```

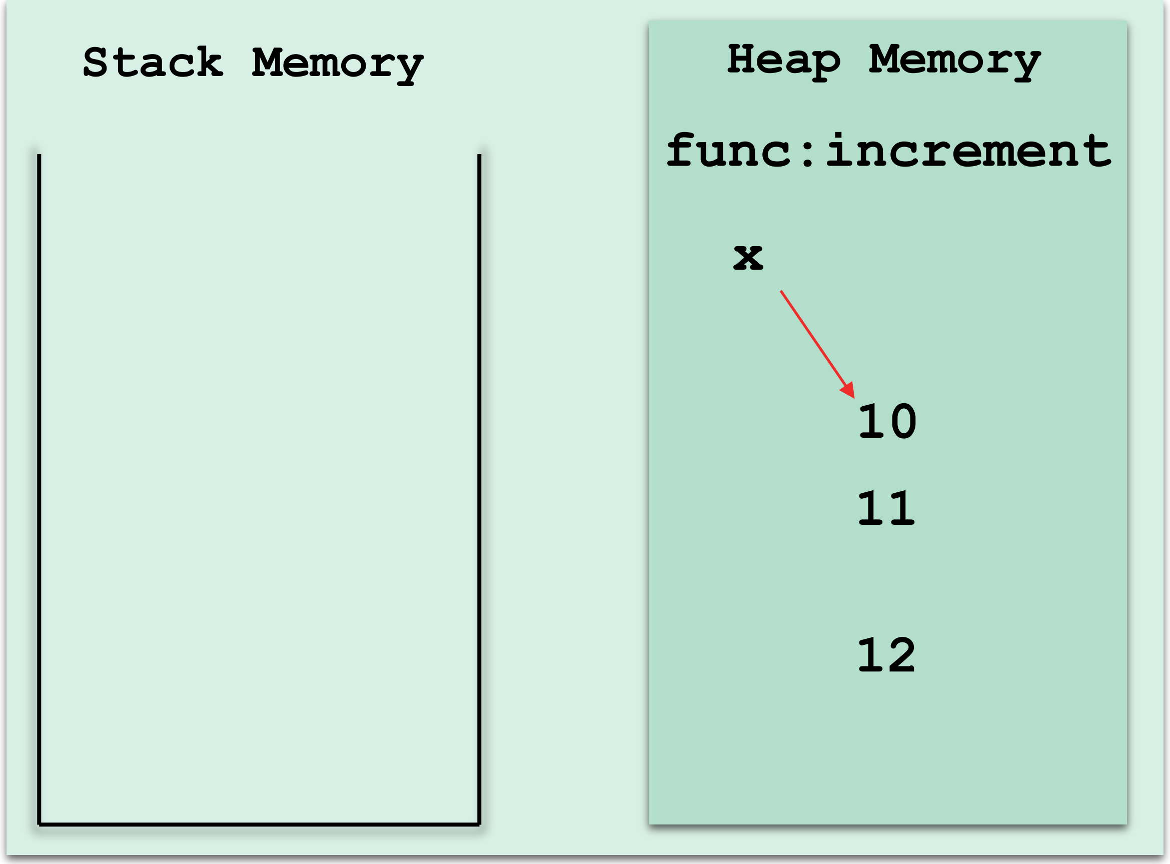


Memory Organization - Walkthrough

The reference is removed. Objects remain in memory.

Instructions (code):

```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
        z = a + 1
        return z
z = increment(y)
del y
del z
```

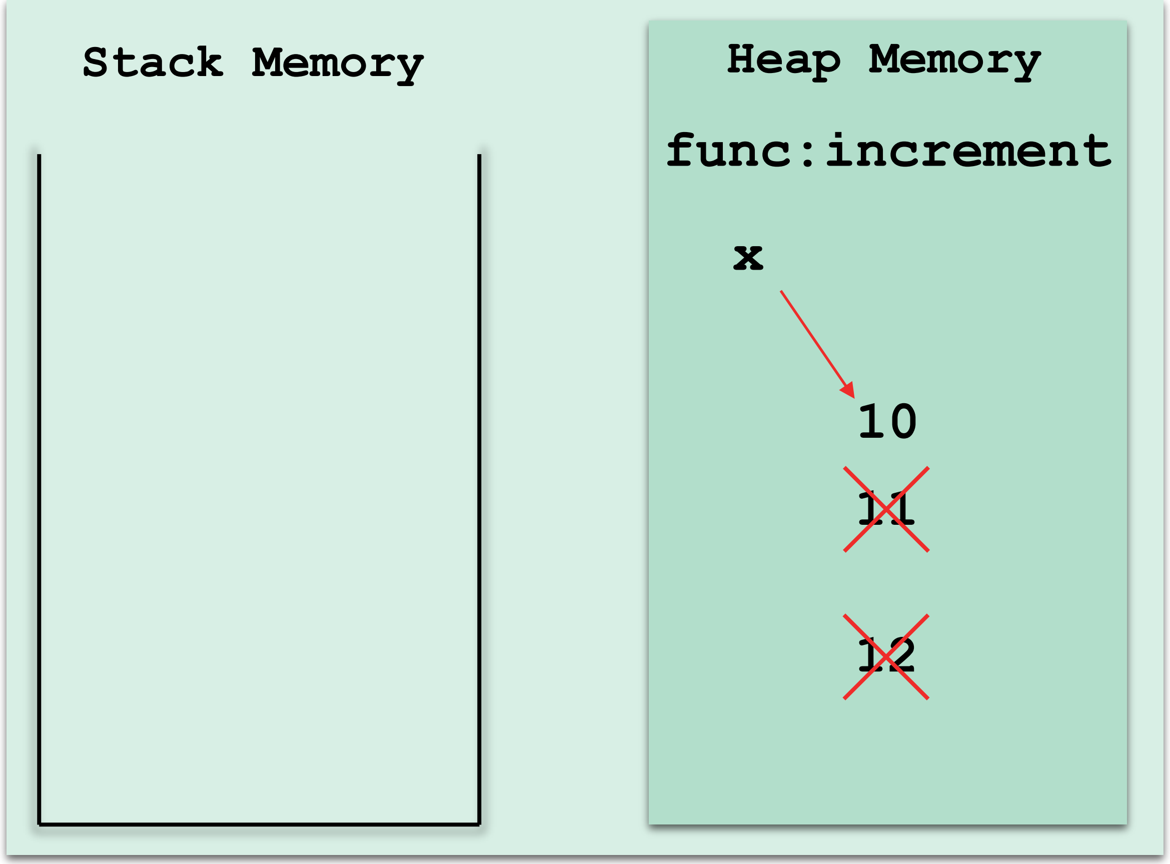


Memory Organization - Walkthrough

If no more references to the object exist, then garbage collection will remove it from memory and free up the space.

Instructions (code):

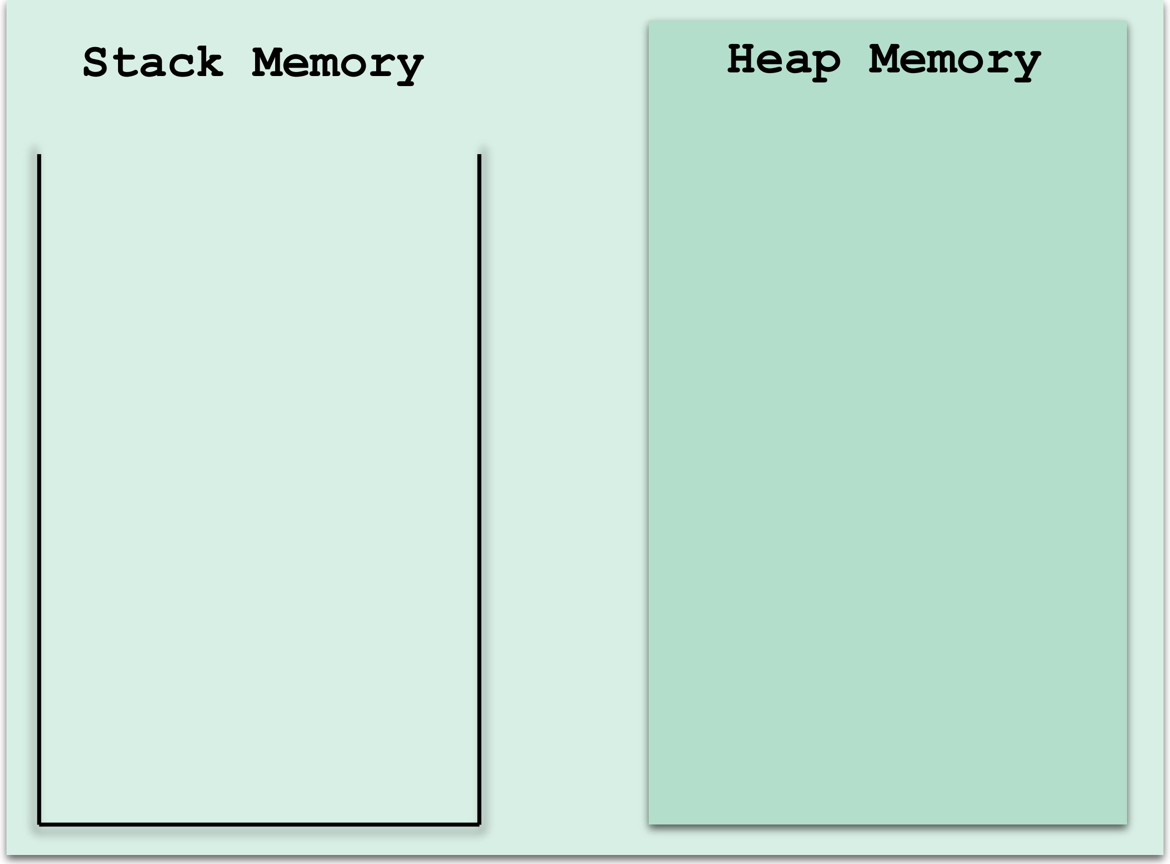
```
x = 10 #global variable
y = 10 #global variable
y += 1 #increment by 1
def increment(a):
       z = a + 1
       return z
z = increment(y)
del y
del z
```



Memory Organization - Walkthrough

Starting fresh…

Instructions (code):



Stack Memory

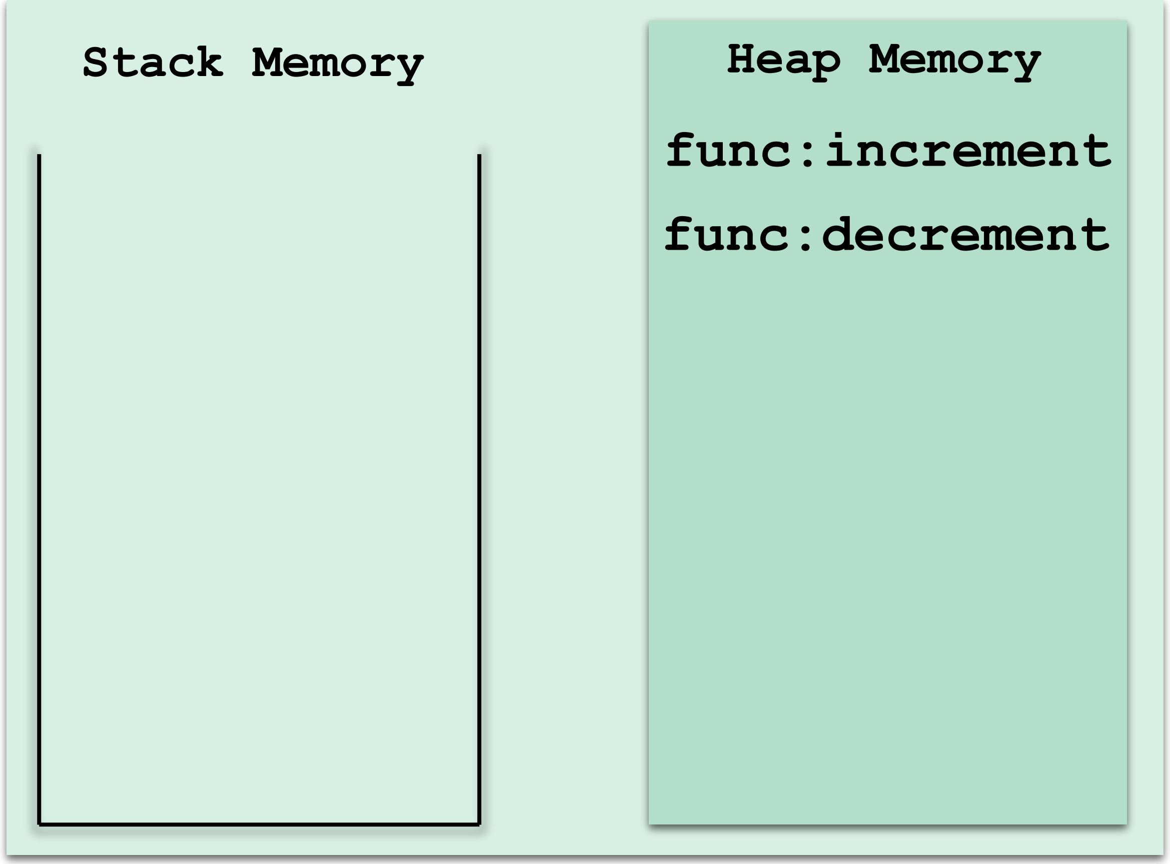
Heap Memory

Memory Organization - Walkthrough

Consider the *increment* function and the new function, *decrement*, which calls it, decrement its output, and return the new value.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
```



Stack Memory

Heap Memory

func:increment

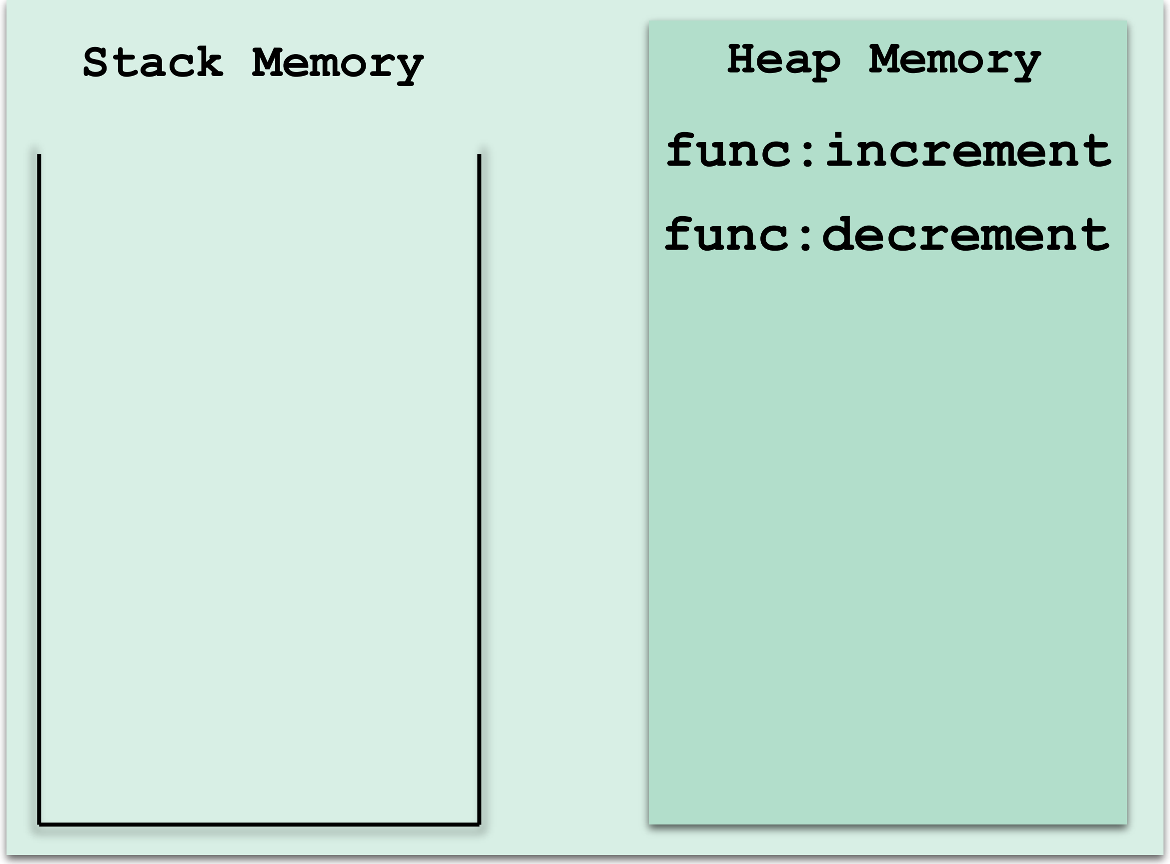
func:decrement

Memory Organization - Walkthrough

Create a global variable *x* = 10.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
```



Stack Memory

Heap Memory

func:increment

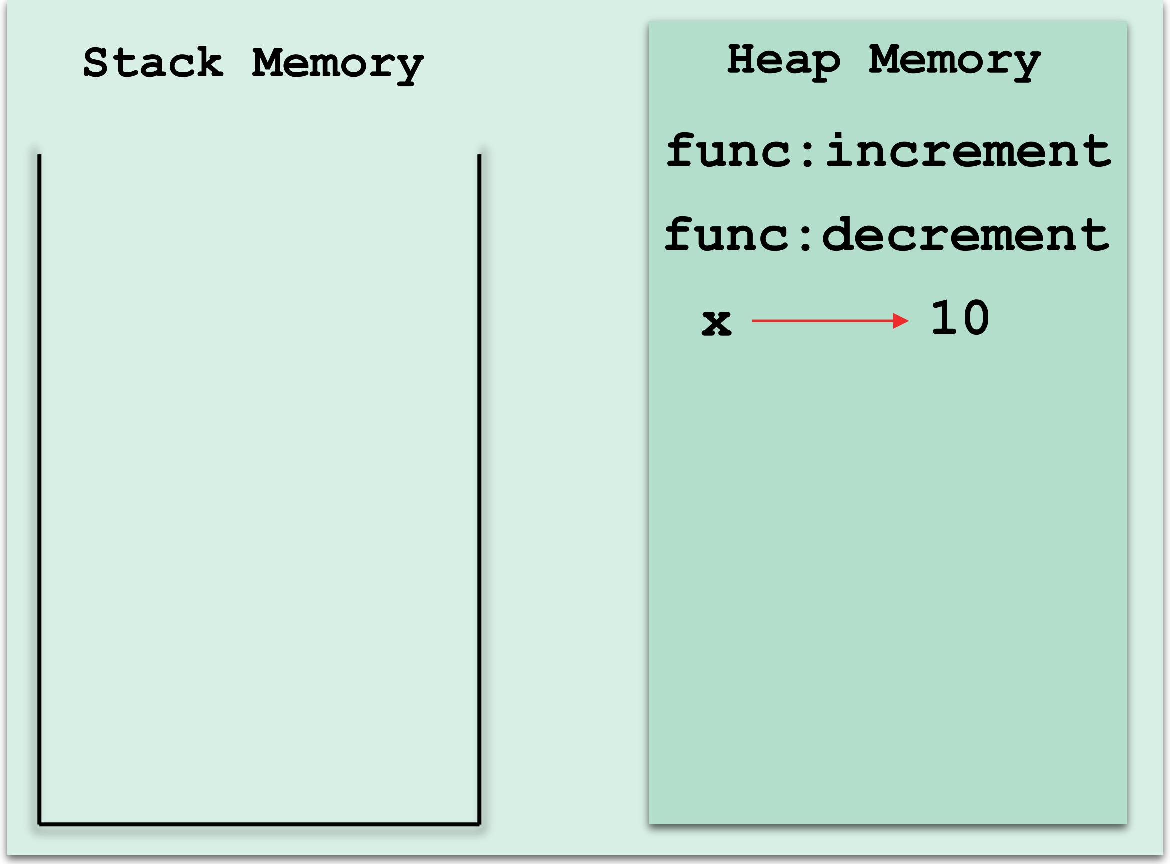
func:decrement

Memory Organization - Walkthrough

Call *decrement*, pass x to it, and store the result in y.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
```



Stack Memory

Heap Memory

func:increment

func:decrement

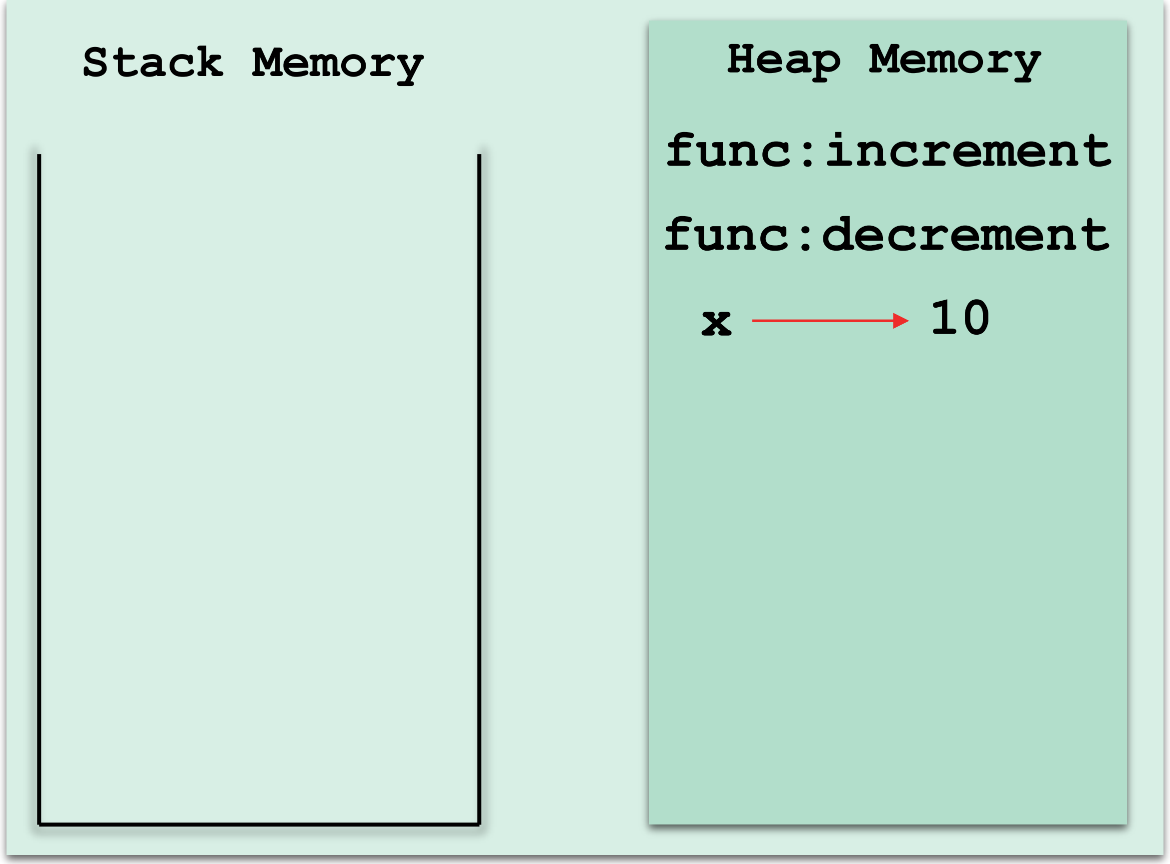
x → 10

Memory Organization - Walkthrough

Lets trace the execution starting the *decrement* function call…

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```

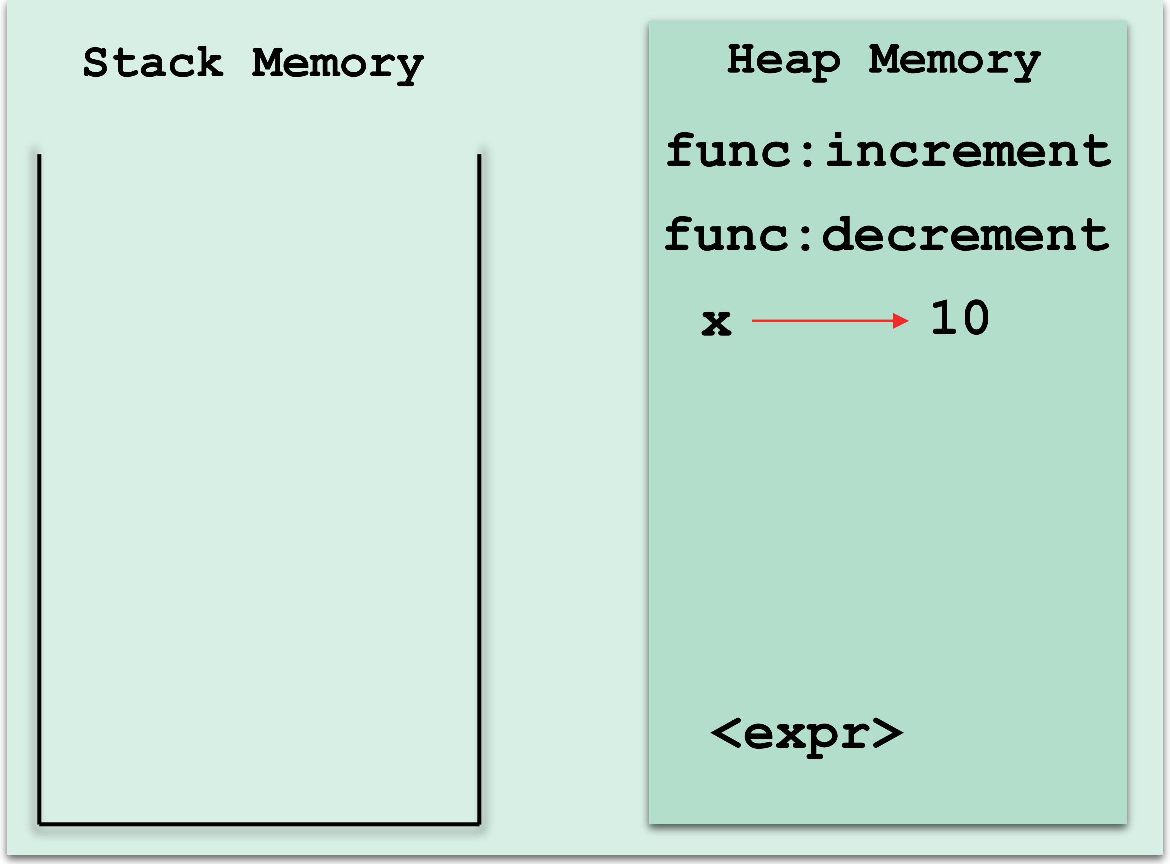


Memory Organization - Walkthrough

Lets trace the execution starting the *decrement* function call…

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```



Stack Memory

Heap Memory

func:increment

func:decrement

x → 10

<expr>

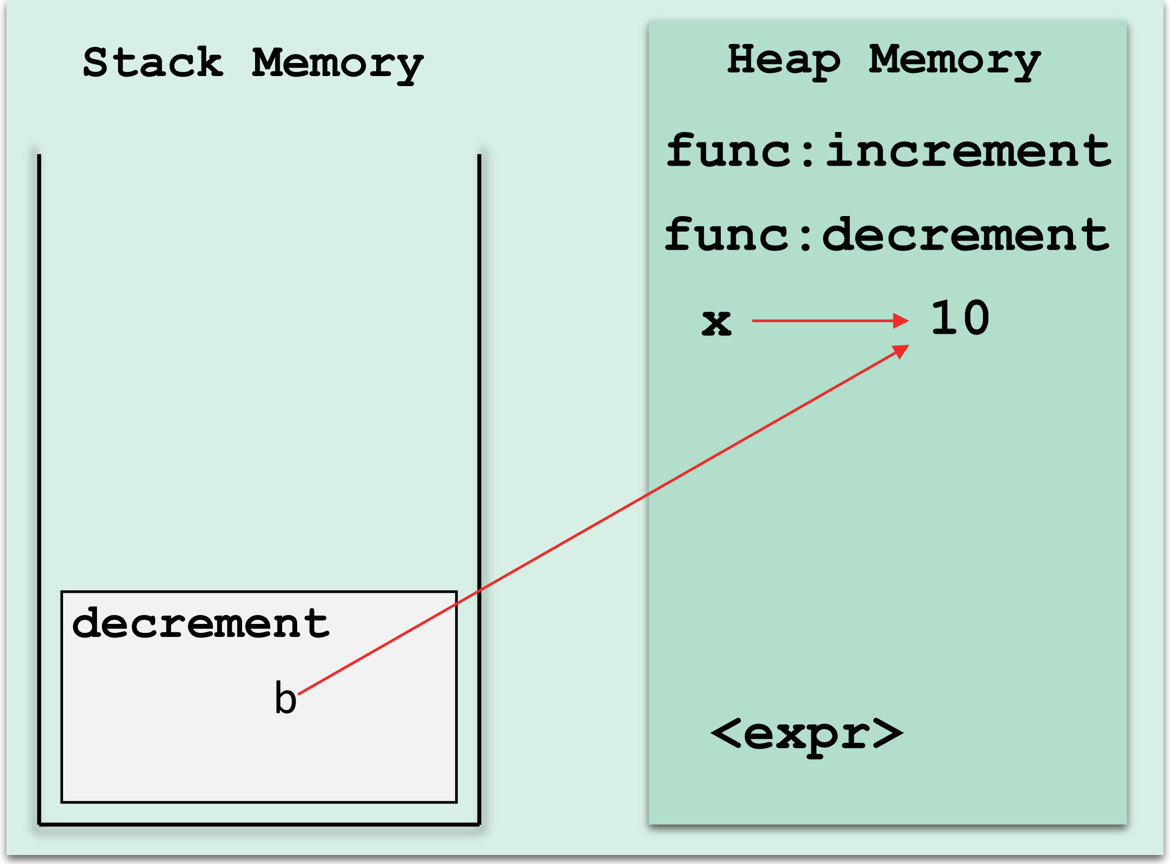
Memory Organization - Walkthrough

When decrement is called, its call is pushed into the stack.

Its parameter is created and points to the object 10.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```

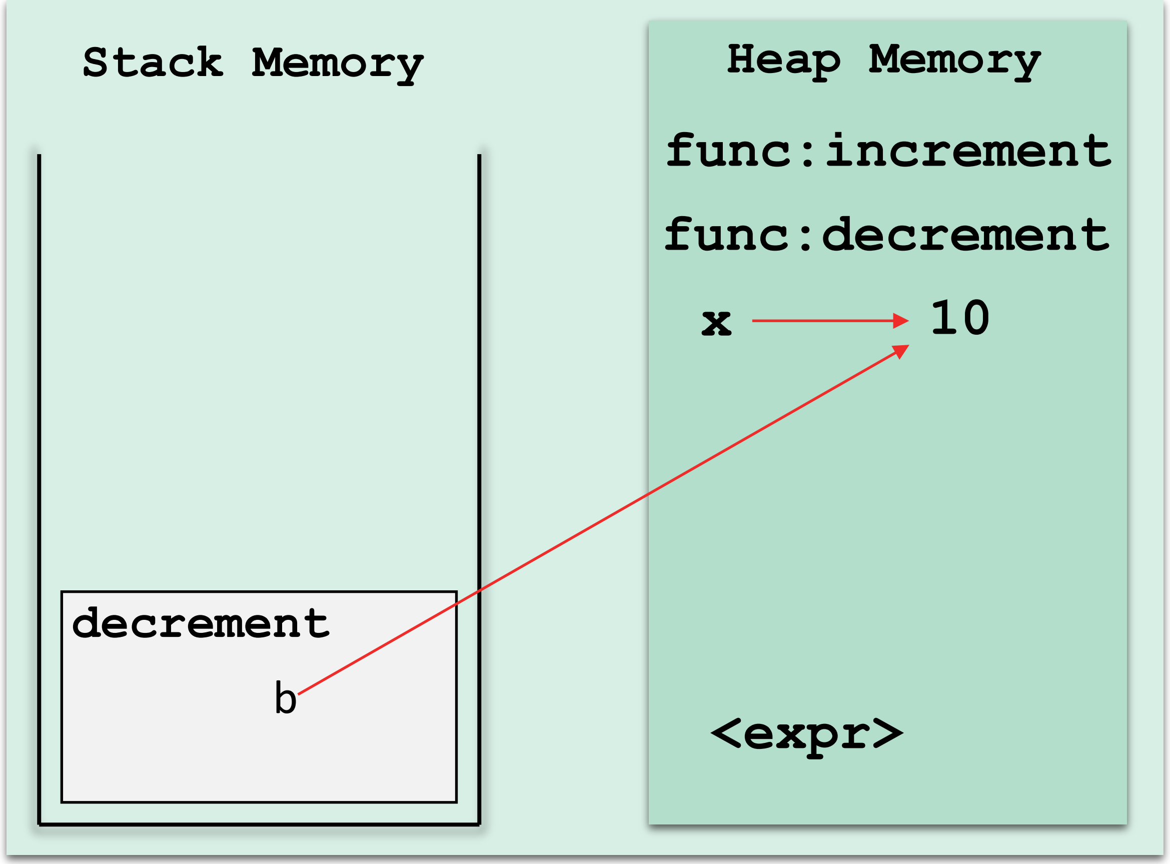


Memory Organization - Walkthrough

The next line is called. It is evaluated as follows: *increment(b)*, then – *1*, then assignment to *c*.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```

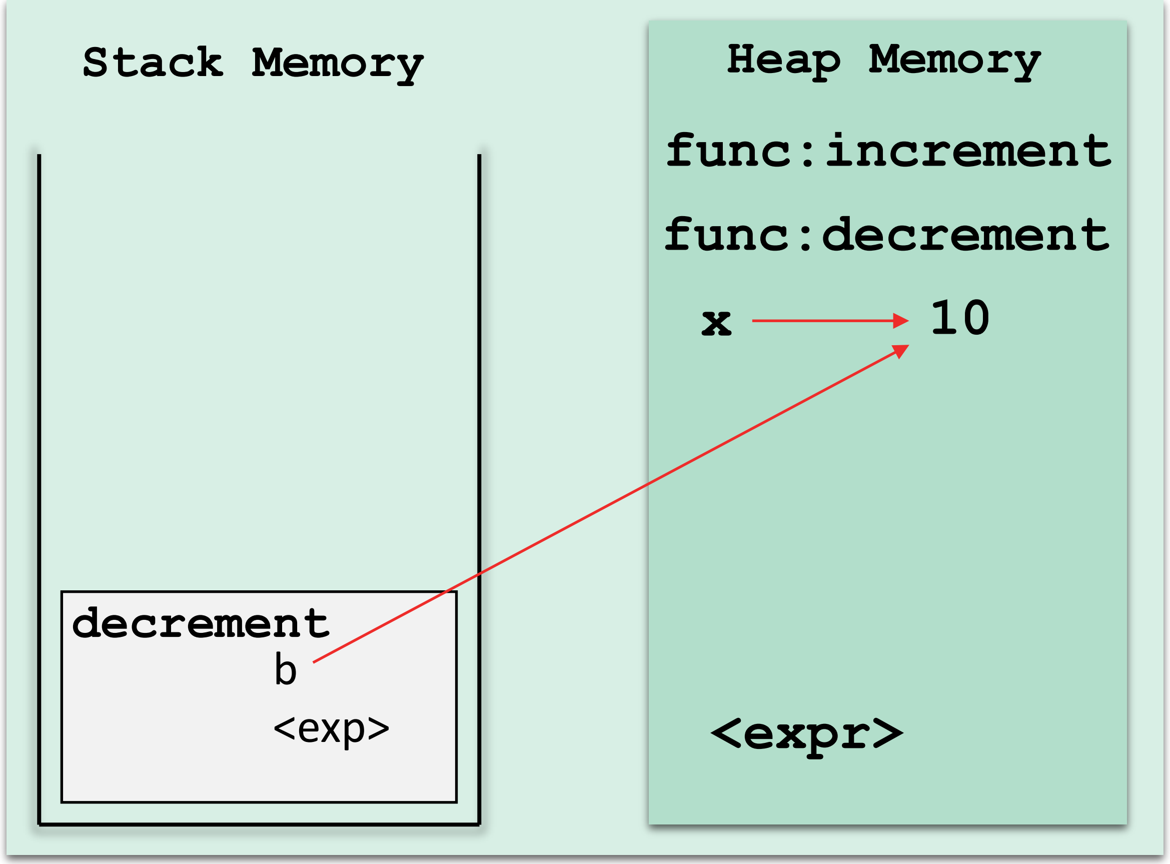


Memory Organization - Walkthrough

The next line is called. It is evaluated as follows: *increment(b)*, then – *1*, then assignment to *c*.

Instructions (code):

```
def increment(a):
    return a + 1
def decrement(b):
    c = increment(b) - 1
    return c
x = 10
y = decrement(x)
```



Stack Memory

decrement
b
<exp>

Heap Memory

func:increment

func:decrement

x → 10

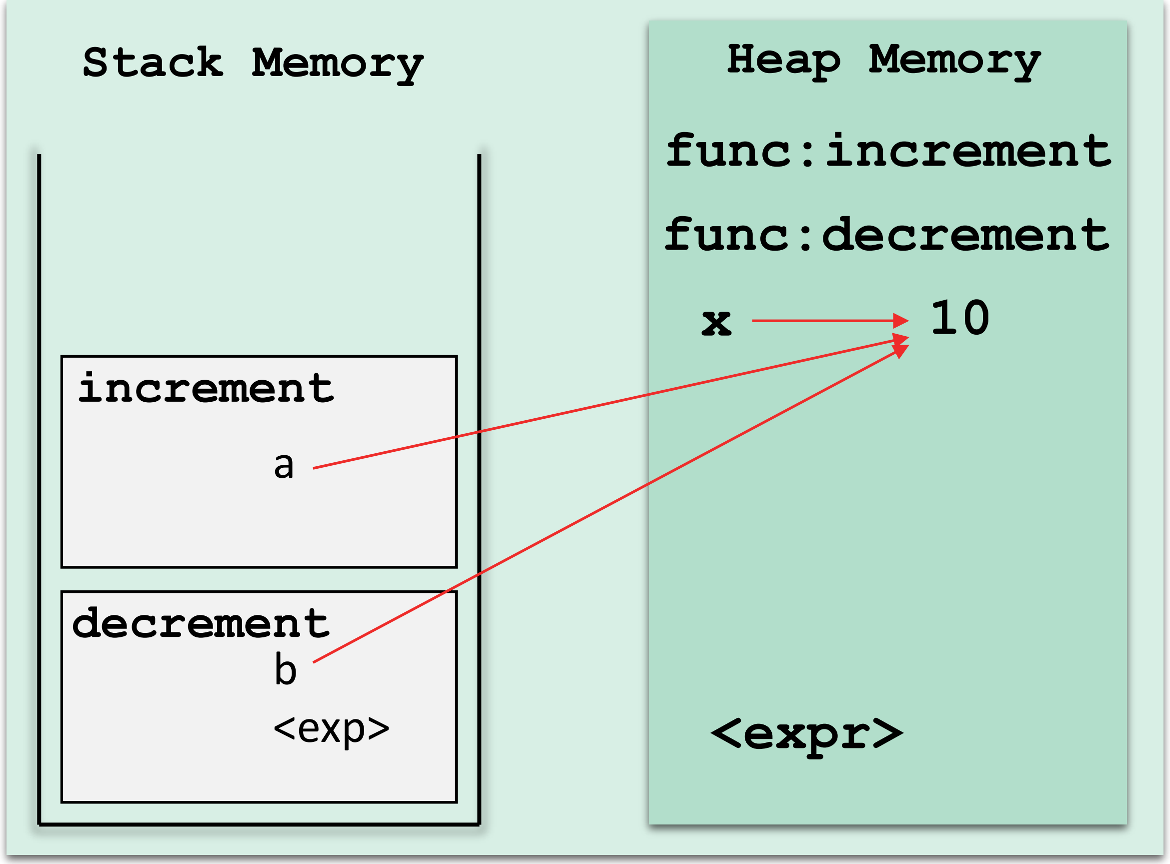
<expr>

Memory Organization - Walkthrough

Increment is called and its parameter *a* points to 10, as well.

Instructions (code):

```
def increment(a):
    return a + 1
def decrement(b):
    c = increment(b) - 1
    return c
x = 10
y = decrement(x)
```



Stack Memory

increment

a

decrement

b

<exp>

Heap Memory

func:increment

func:decrement

x

10

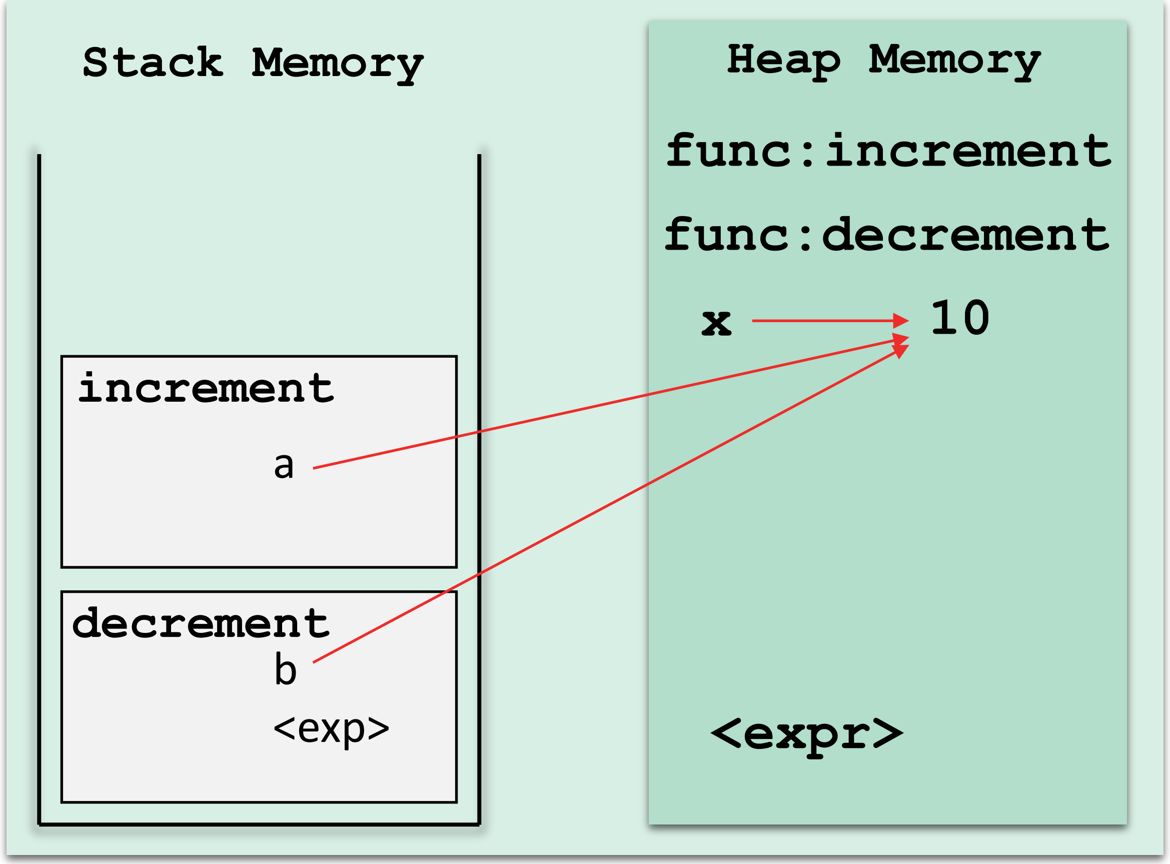
<expr>

Memory Organization - Walkthrough

The ~~return~~ call does two things: 1) increments *a,* and 2) return the reference to the caller.

Instructions (code):

```
def increment(a):
    return a + 1
def decrement(b):
    c = increment(b) - 1
    return c
x = 10
y = decrement(x)
```

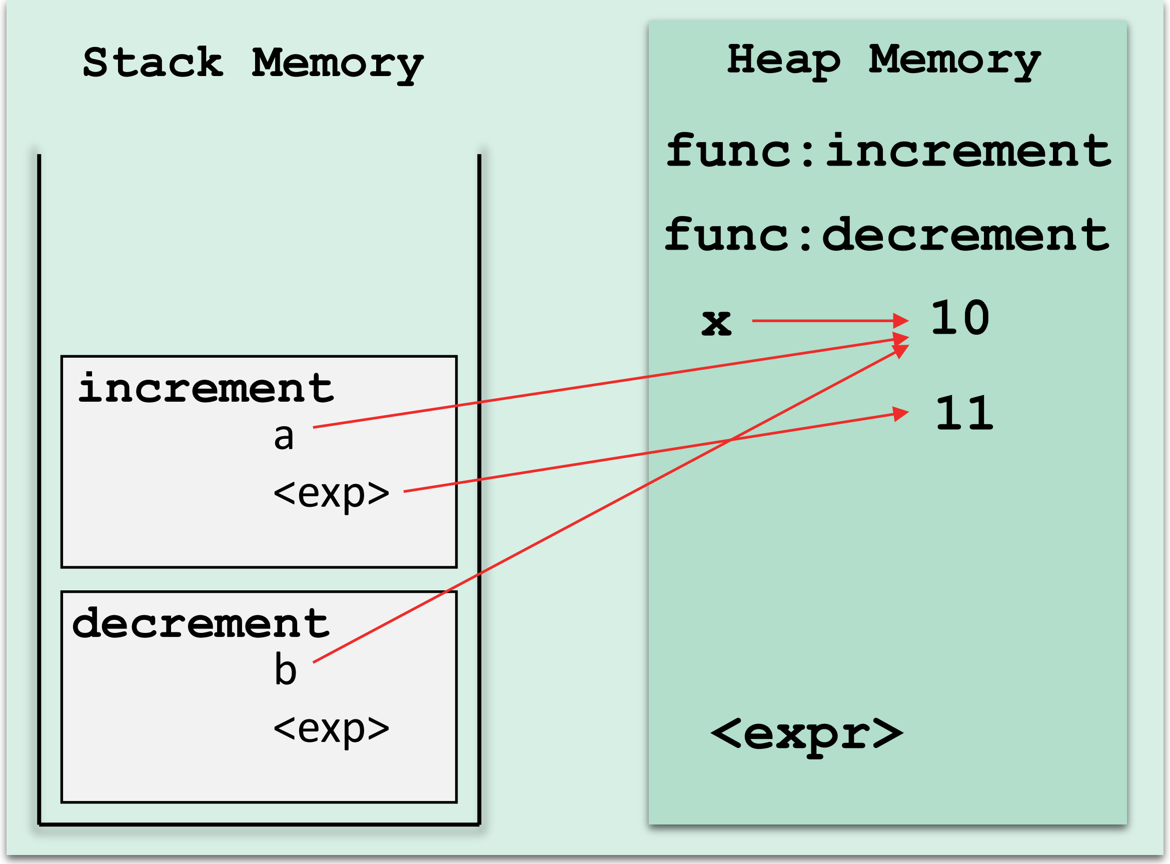


Memory Organization - Walkthrough

~~The return~~ call does two things: 1) increments *a*, and 2) return the reference to the caller.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```

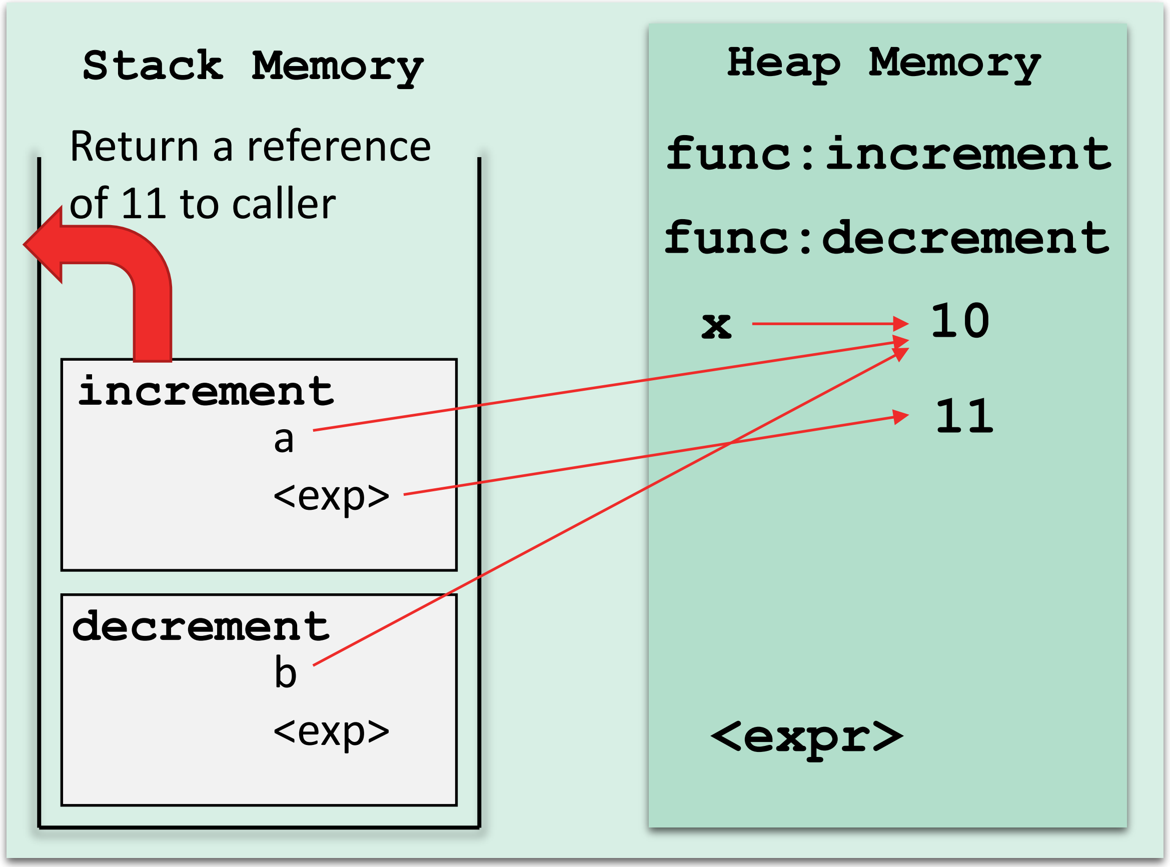


Memory Organization - Walkthrough

The return call does two things: 1) increments *a,* and 2) return the reference to the caller, which ends the *increment*'s execution.

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```



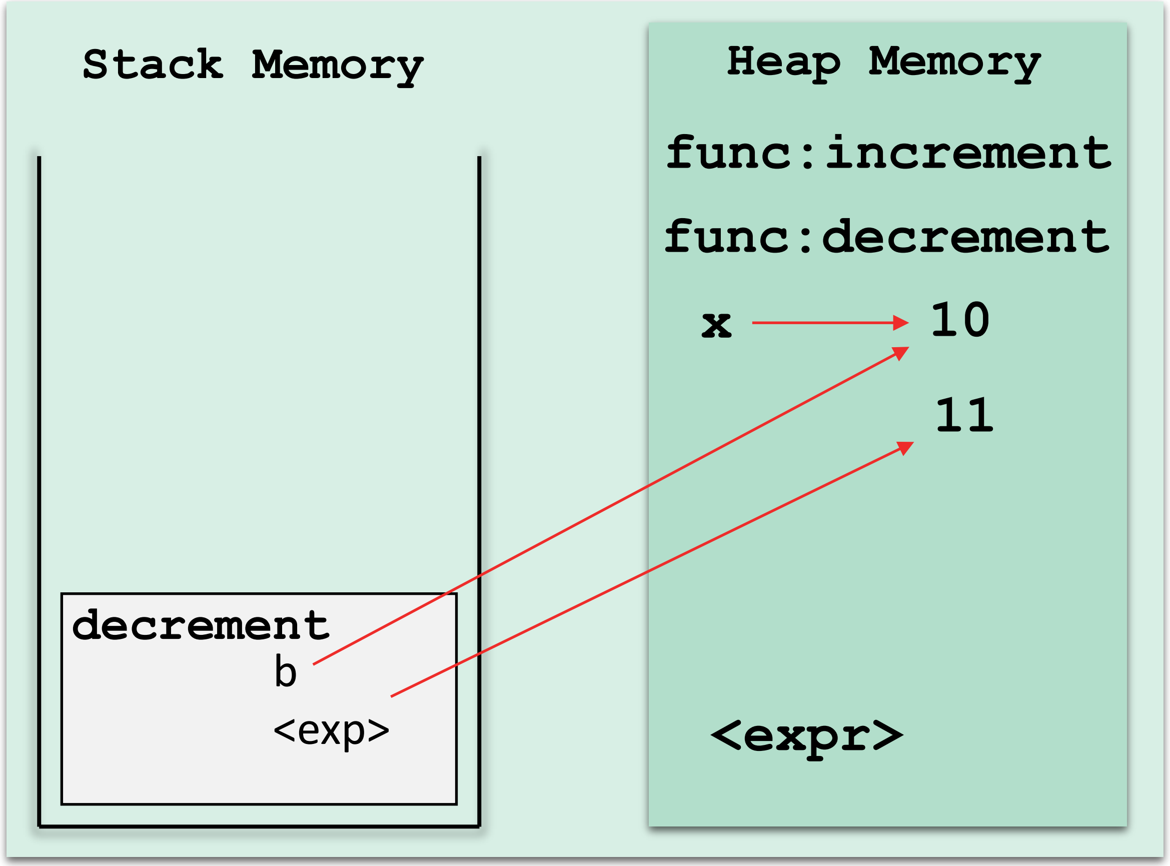
Memory Organization - Walkthrough

Execution is back to the calling function.

The returned value remains in heap and its reference is returned to the expression.

Instructions (code):

```
def increment(a):
      return a + 1
def decrement(b):
      c = increment(b) - 1
      return c
x = 10
y = decrement(x)
```



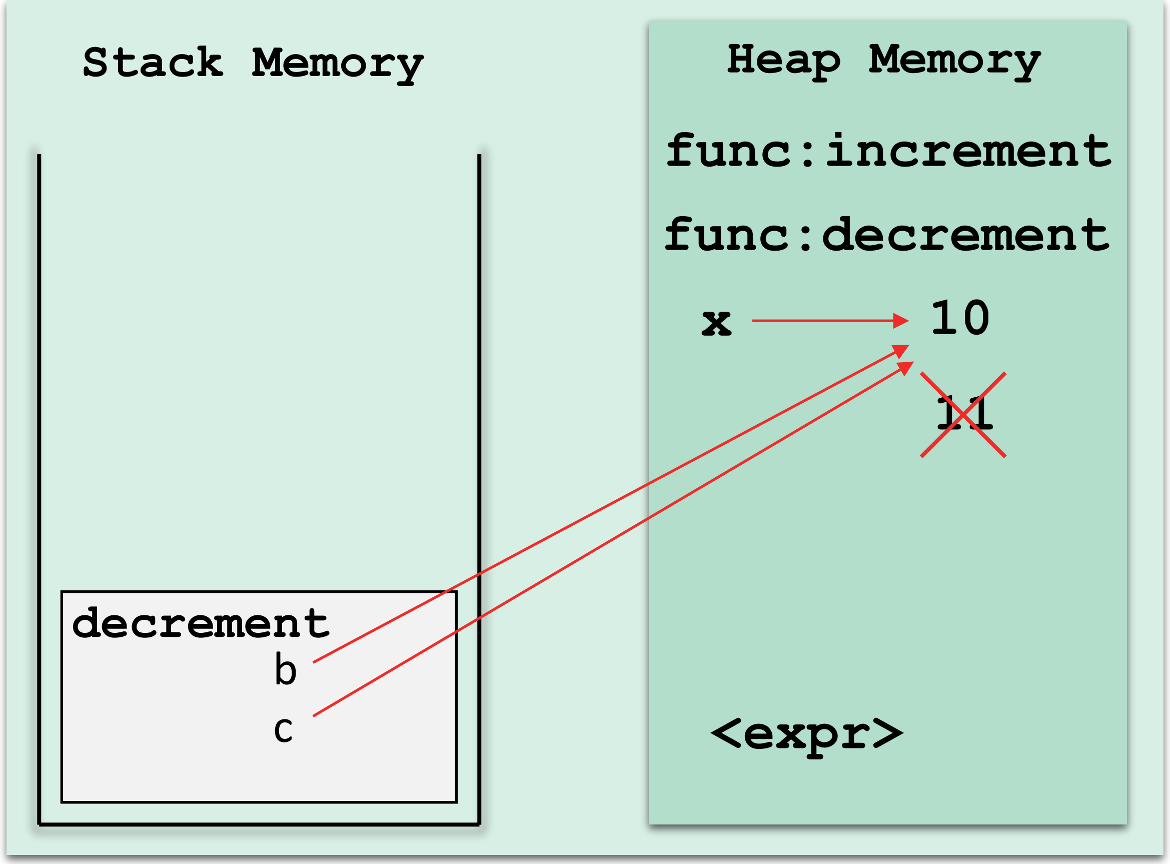
Memory Organization - Walkthrough

The rest of the expression is evaluated. c points to back to 10.

The object 11 has no references, so eventually the garbage collection algorithm will remove it from memory.

Instructions (code):

```
def increment(a):
    return a + 1
def decrement(b):
    c = increment(b) - 1
    return c
x = 10
y = decrement(x)
```

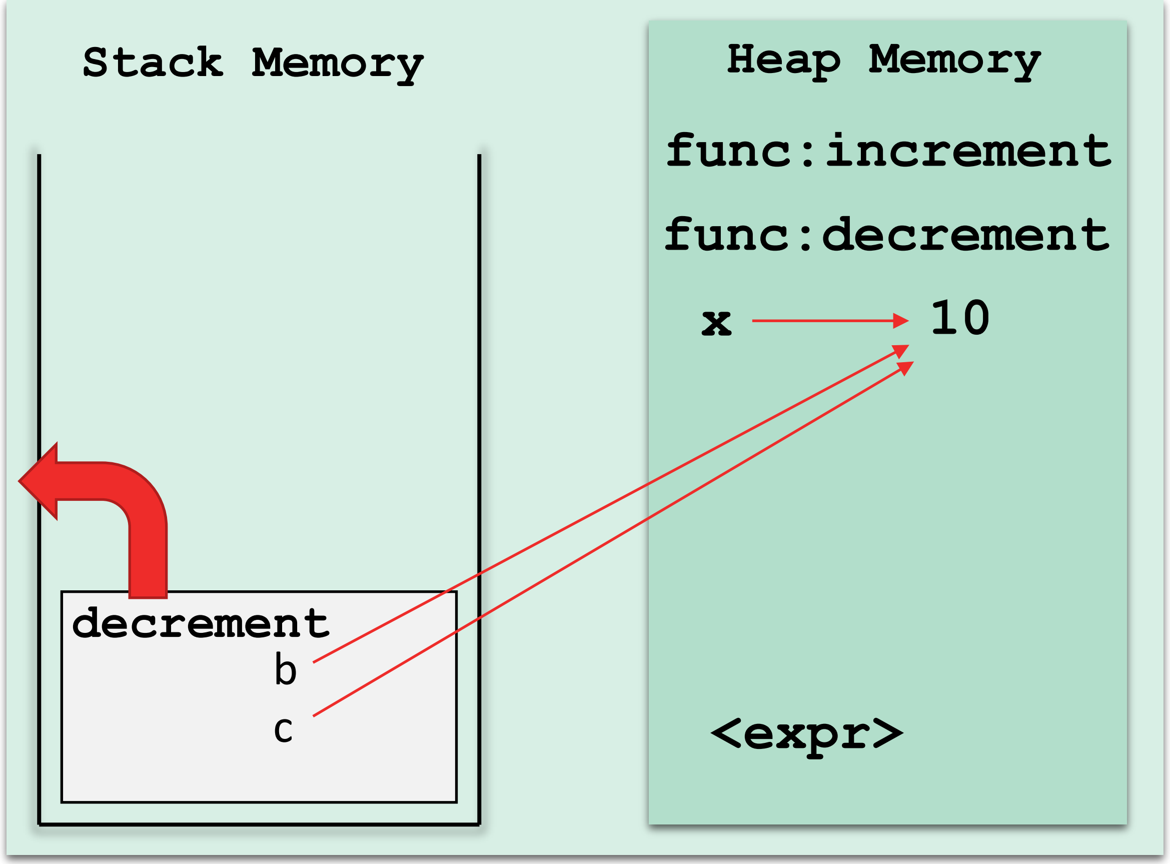


Memory Organization - Walkthrough

c is returned to the caller, which terminates the execution of *decrement*. Local variables are deleted.

Instructions (code):

```
def increment(a):
    return a + 1
def decrement(b):
    c = increment(b) - 1
    return c
x = 10
y = decrement(x)
```

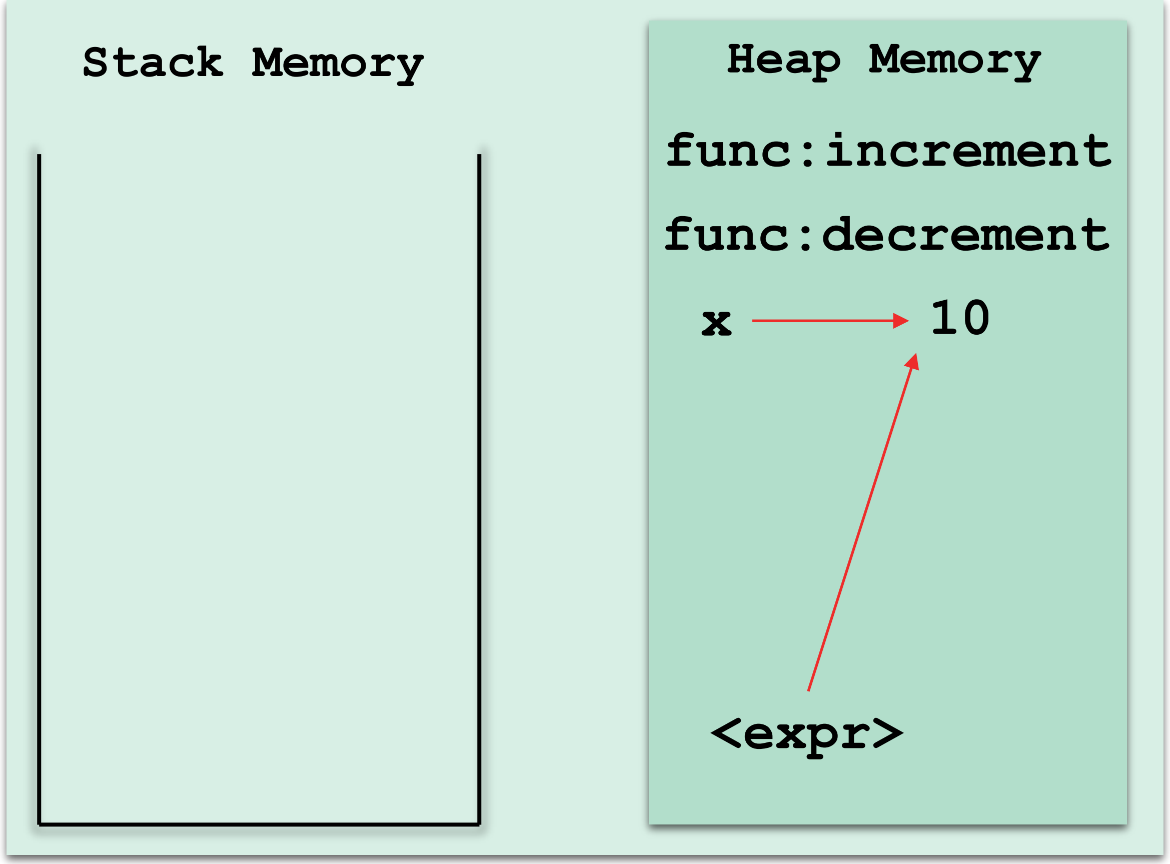


Memory Organization - Walkthrough

The expression is evaluated and the results are stored in y

Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```



Stack Memory

Heap Memory

func:increment

func:decrement

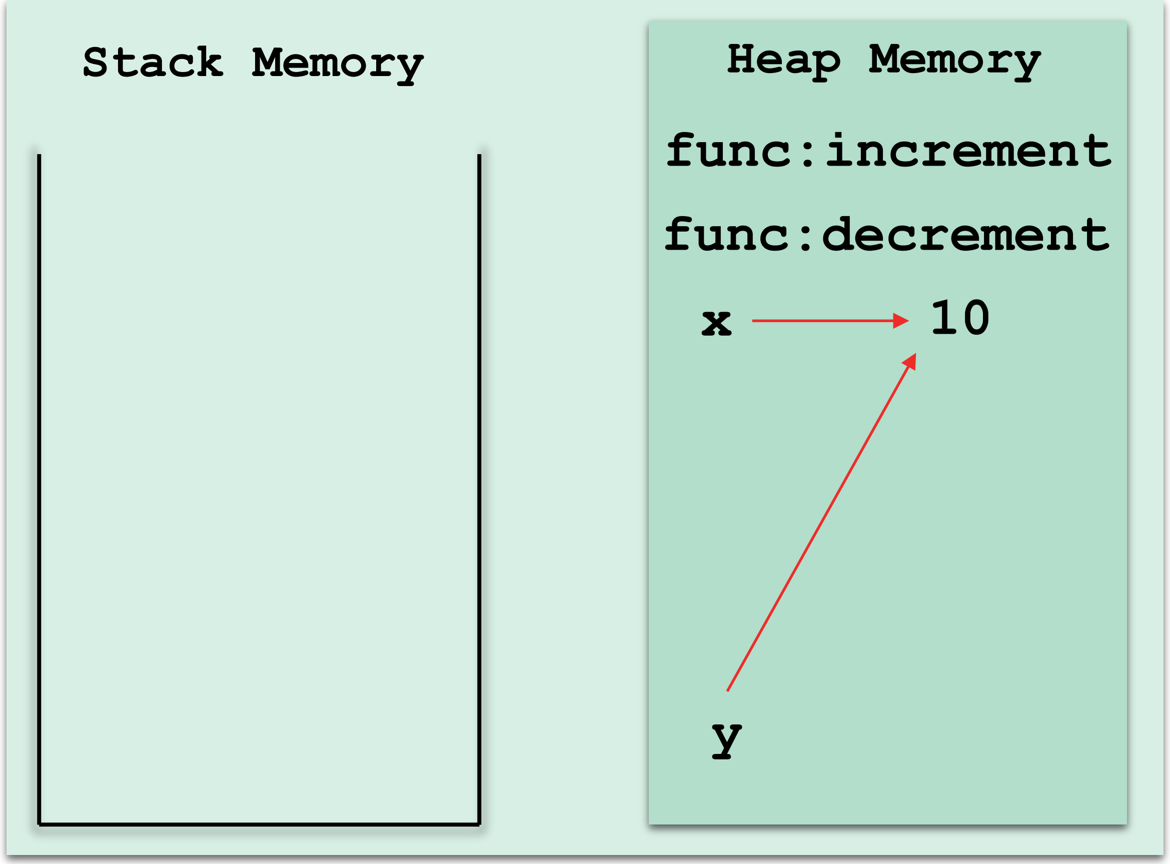
x → 10

<expr>

Memory Organization - Walkthrough

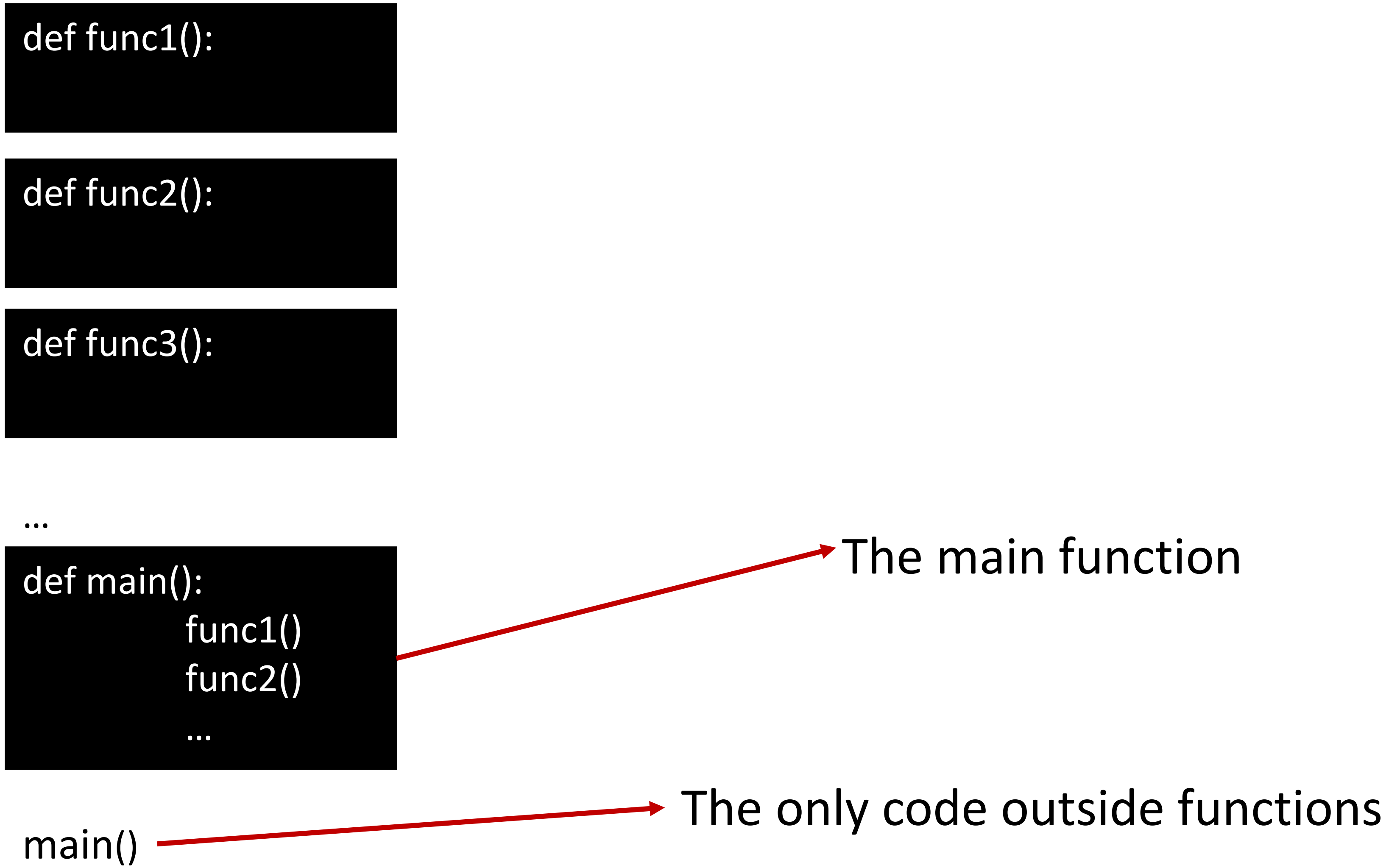
Instructions (code):

```
def increment(a):
        return a + 1
def decrement(b):
        c = increment(b) - 1
        return c
x = 10
y = decrement(x)
```



Program Structure – Functions

Structure



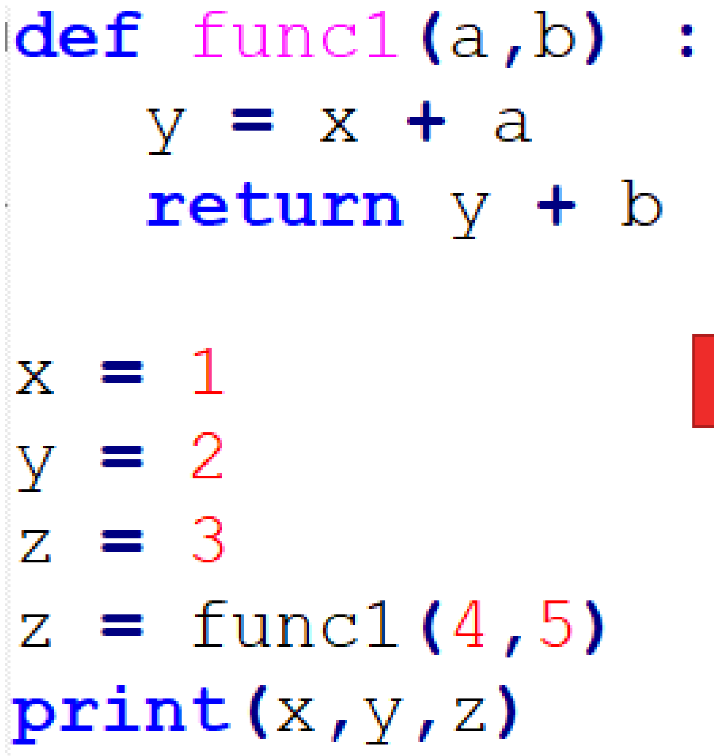
```
def func1():

def func2():

def func3():

…
def main():
            func1()
            func2()
            …

main()
```

The main function

The only code outside functions

Function Tracing

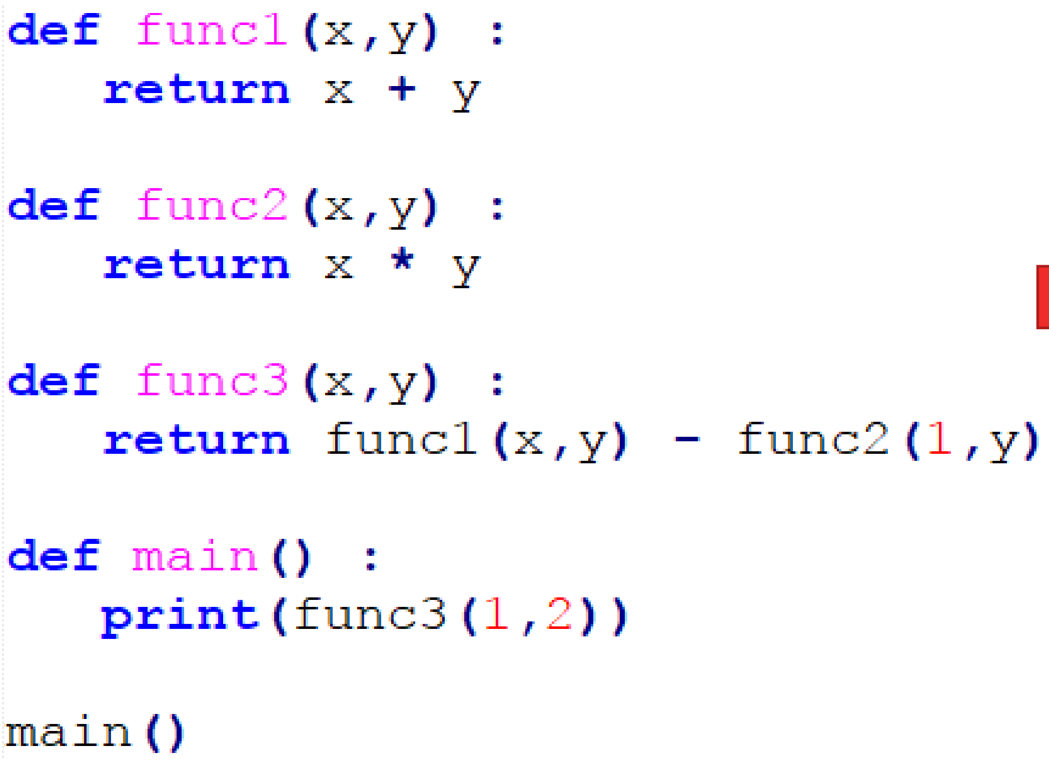
Scope



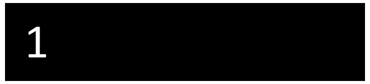
```
def func1(a,b) :
    y = x + a
    return y + b

x = 1
y = 2
z = 3
z = func1(4,5)
print(x,y,z)
```

```
1 2 10
```

Scope

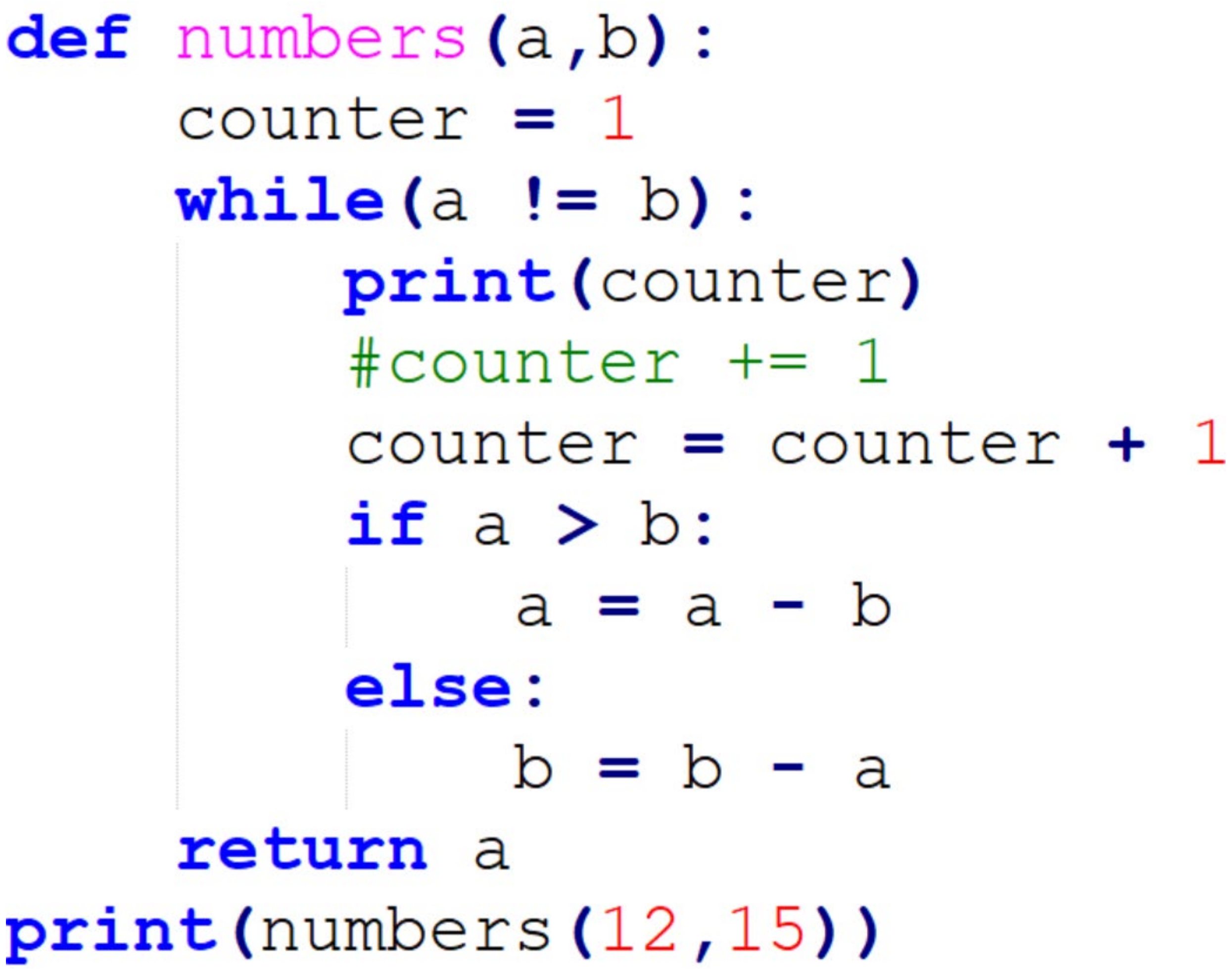


```
def func1(x,y) :
    return x + y

def func2(x,y) :
    return x * y

def func3(x,y) :
    return func1(x,y) - func2(1,y)

def main() :
    print(func3(1,2))

main()
```



```
1
```

Trace the code



```
def numbers(a,b):
    counter = 1
    while(a != b):
        print(counter)
        #counter += 1
        counter = counter + 1
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
print(numbers(12,15))
```

```
1
2
3
4
3
```

Onward to … lists, dictionaries, and strings.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~hudsonj/

UNIVERSITY OF CALGARY