Functions (II)

CSE 160 Summer 2026

Questions?

Announcements

- **Homework Assignment 1** due tonight at 11:59 PM
- **Programming Practice 1** due tonight at 11:59 PM
- **Homework Assignment 2** due Monday, July 13th at 11:59 PM
- **Programming Practice 2** due Sunday, July 12th at 11:59pm

Code Quality

cs.uw.edu/160/resources/code_quality/

Code is meant to be read and understood by humans(!), so maintaining good code style (or '**code quality**') is an important part of programming.

The CSE 160 Code Quality Guide (i.e. expectations for code style) is on the website.

`flake8` is a tool for automatically checking code style in Python files.

```
jovyan@jupyter-netid:~/…/$ flake8 python_file.py
```

use this command to run the `flake8` style checker!

`flake8_demo.py`

```
1  x=1
2  print("There's an extra space on this line")
3
```

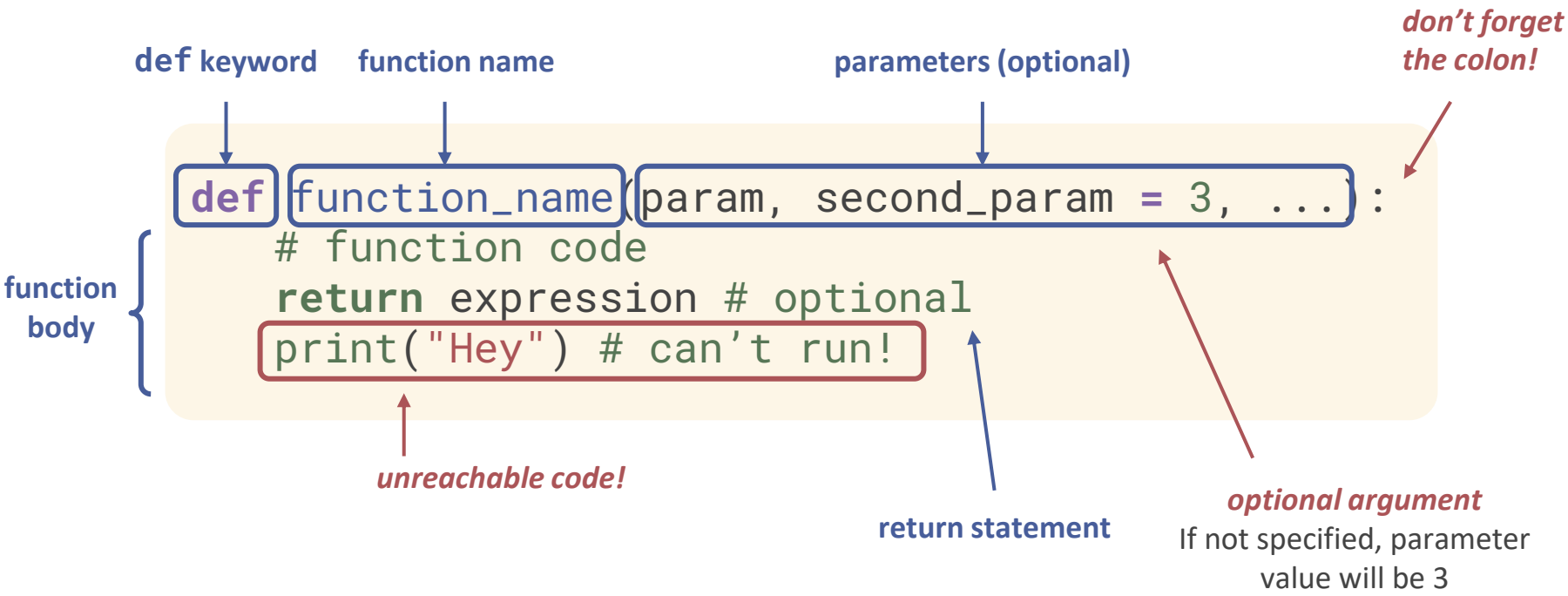
```
flake8_demo.py:1:2: E225 missing whitespace around operator
flake8_demo.py:2:45: W291 trailing whitespace
```

Today's Roadmap

1. Functions Recap
2. Evaluating a Function Call
3. Practice with Functions
4. Variable Scope

Jupyter Hub

All About Functions



Think Pair Share: Even / Odd

```
"""
Write a function named is_even.

Arguments:

    num: a number

Returns:

    True if `number` is even,
    False otherwise
"""
```

The **modulo** (**%**) operator calculates the remainder of a number divided by another number.

Evaluating a Function Call

What happens when we call a function?

```
def my_function(a, b):
    c = a * b
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```

```
rv = my_function(10 / 5, 3)
```

Evaluating a Function Call: *Step 1*

What happens when we call a function?

1. Evaluate **arguments**

```
def my_function(a, b):
    c = a * b
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```

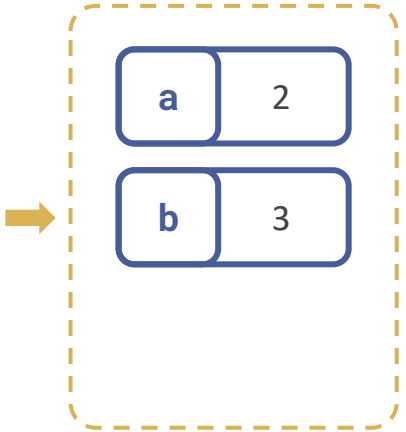
```
rv = my_function( 2  3
```

Evaluating a Function Call: *Step 2*

What happens when we call a function?

1. Evaluate **arguments**
2. Assign argument values to **parameters**

```
def my_function(a, b):
    c = a * b
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```



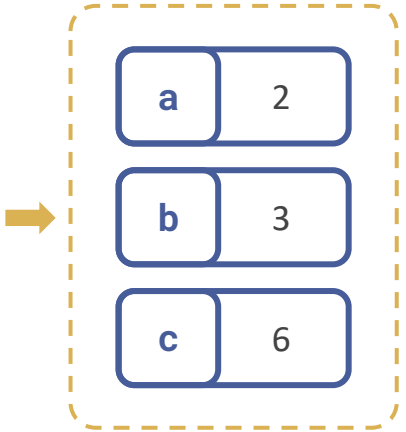

```
rv = my_function(2, 3)
```

Evaluating a Function Call: *Step 3*

What happens when we call a function?

1. Evaluate **arguments**
2. Assign argument values to **parameters**
3. Evaluate **function body**

```
def my_function(a, b):
    c = a * b      6
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```




```
rv = my_function(10 / 5, 3)
```

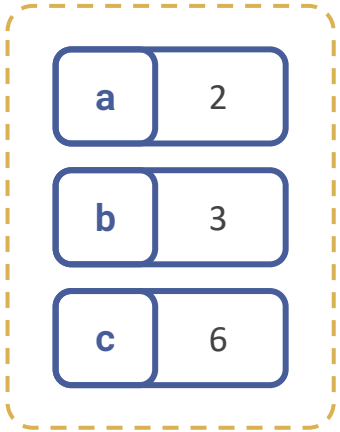
Evaluating a Function Call: *Step 4*

What happens when we call a function?

1. Evaluate **arguments**
2. Assign argument values to **parameters**
3. Evaluate **function body**
4. At a **return statement**, evaluate the expression, skip to step 6

```
def my_function(a, b):
    c = a * b
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```

4




```
rv = my_function(10 / 5, 3)
```

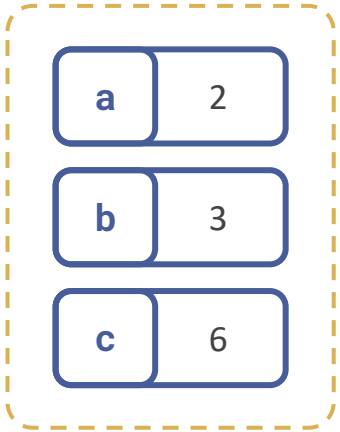
Evaluating a Function Call: *Step 5*

What happens when we call a function?

1. Evaluate **arguments**
2. Assign argument values to **parameters**
3. Evaluate **function body**
4. At a **return statement**, evaluate the expression, skip to step 6
5. No `return` encountered, return value is **None**

```
def my_function(a, b):
    c = a * b
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```

4




```
rv = my_function(10 / 5, 3)
```

Evaluating a Function Call: *Step 6*

What happens when we call a function?

1. Evaluate **arguments**
2. Assign argument values to **parameters**
3. Evaluate **function body**
4. At a **return statement**, evaluate the expression, skip to step 6
5. No **`return`** encountered, return value is **None**
1. **Exit**, function call *evaluates* to return value

```
def my_function(a, b):
    c = a * b
    if c % 2 == 0:
        return c - a
        d = 1 / 0
    return 2 / 0
```

4

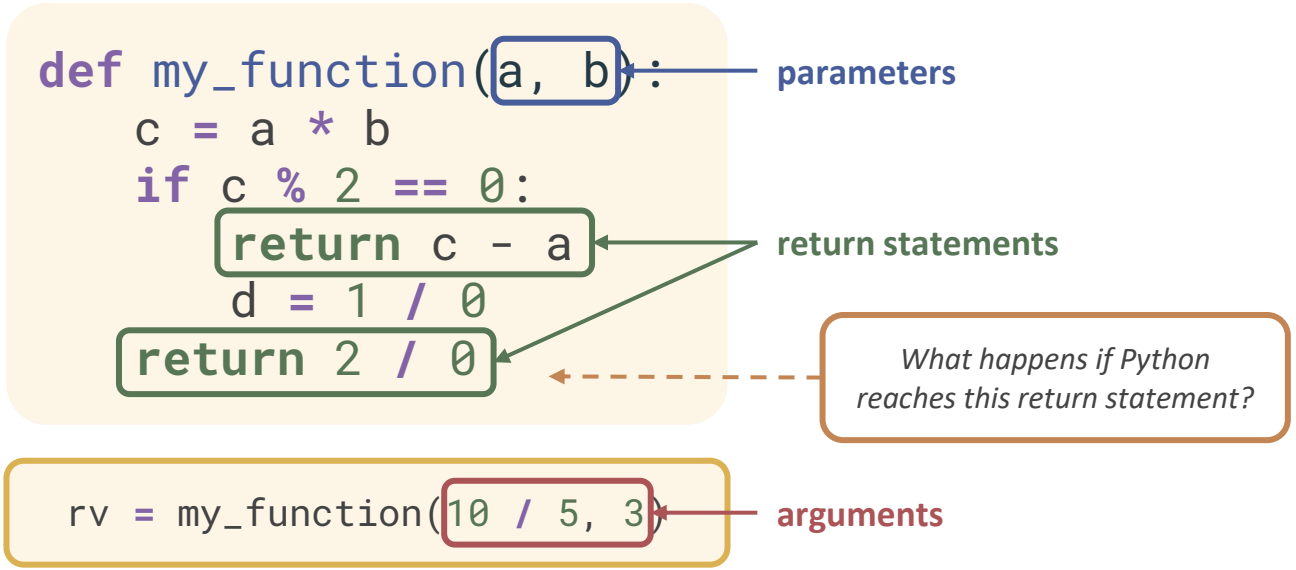
rv = 4

Evaluating a Function Call: *Summary*

What happens when we call a function?

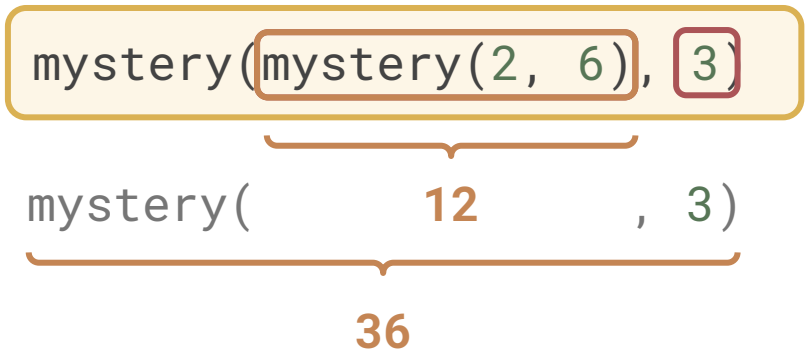
1. Evaluate **arguments**
2. Assign argument values to **parameters**
3. Evaluate **function body**
4. At a **return statement**, evaluate the expression, skip to step 6
5. No `return` encountered, return value is **None**

1. **Exit**, function call *evaluates* to return value



Function Example 1: Mystery

```
def mystery(a, b):
    return a * b
```



What gets printed?

Nothing! (No `print`)

What gets returned?

36

Function Example 2: 

print != return

```
def food():
    print("Tacos :)")
```

```
print(food())
```

```
print( None )
```

What gets printed?

Tacos :)
None

What gets returned?

None

Function Example 3: Variable-ception

```
x = False

def func(x):
    print(x)
```

```
func(True)
```

What gets printed?

True

Function Example 3: Variable-ception (x2)

```
x = False

def func(x):
    print(x)
```

```
func(True)
print(x)
```

What gets printed?

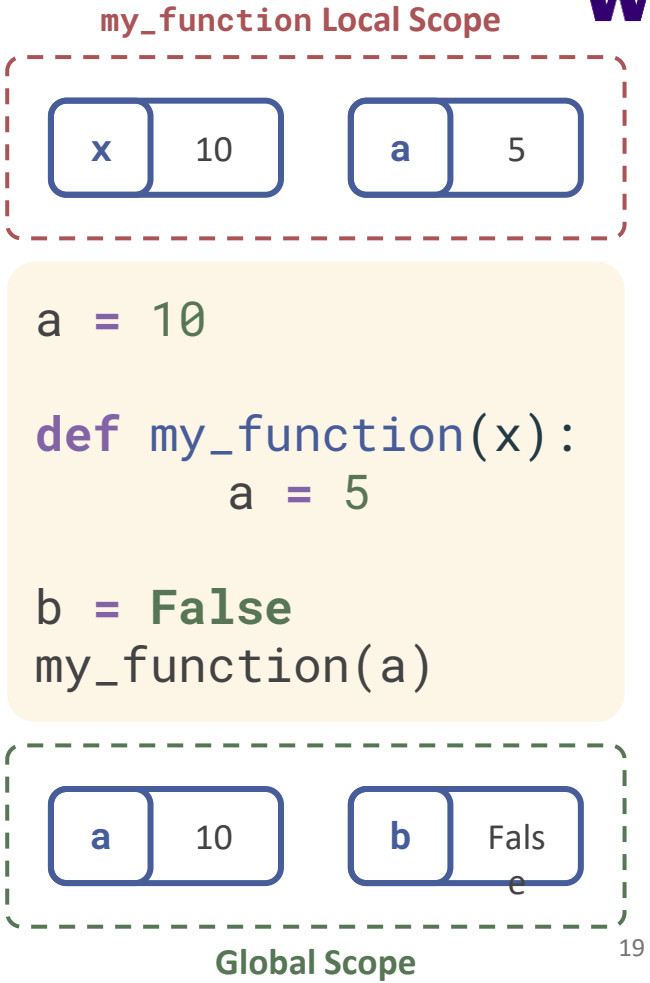
True
False

Variable Scope

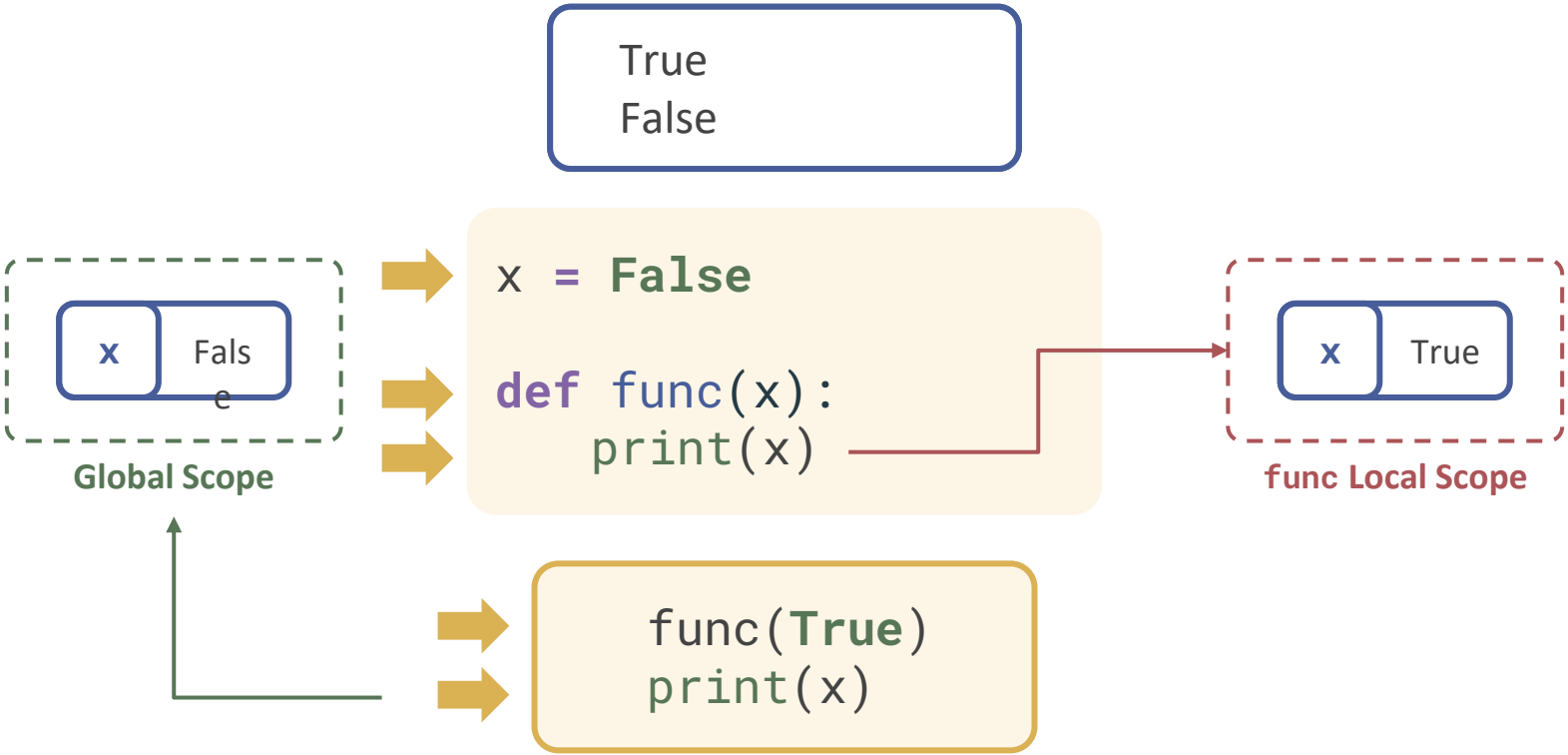
The **scope** of a variable is the area of code in which the variable is defined (has a value).

In Python, there are two scopes:

- **Local Scope** — Variables (including parameters) defined inside a function body
  - Value of variable is accessible only within the function
- **Global Scope** — Everything else
  - Value of variable is accessible anywhere in the program (including inside functions)



Revisiting Function Example 3



Think Pair Share

What gets **printed** when this program is run?

A 1

B 2

C 8

D 9

E [*Error*]

Sli.do #cse160

```
x = 1
y = 2
def func1(x):
    y = 1
    return x + y

def func2(x):
    x = 5
    x = func1(x)
    return x + y

x = func2(x)
print("x = " + str(x))
```

Slippery ~~Slope~~ Scope

```
1   x = 1
2   y = 2
3   def func1(x):
4       y = 1
5       return x + y
6
7   def func2(x):
8       x = 5
9       x = func1(x)
10      return x + y
11
12  x = func2(x)
13  print("x = " + str(x))
```

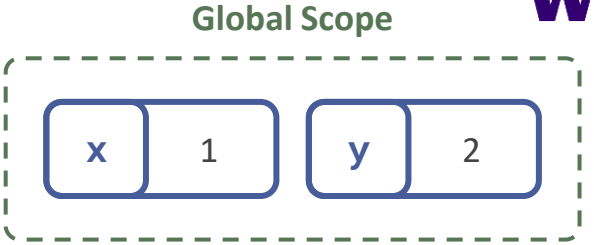
Code Execution

Variables

Slippery ~~Slope~~ Scope (1)

1. [1, 2] **x**, **y** defined in **global scope**

```
1   x = 1
2   y = 2
3   def func1(x):
4       y = 1
5       return x + y
6
7   def func2(x):
8       x = 5
9       x = func1(x)
10      return x + y
11
12  x = func2(x)
13  print("x = " + str(x))
```



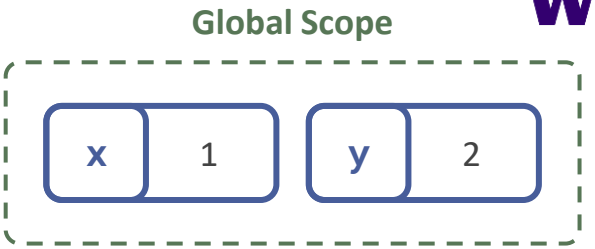
Code Execution

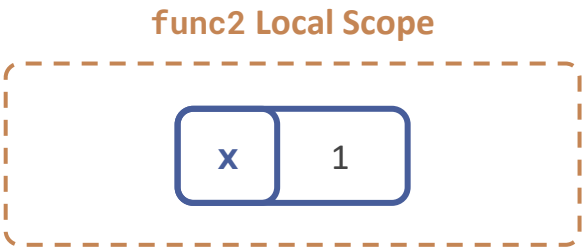
Variables

Slippery ~~Slope~~ Scope (2)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```





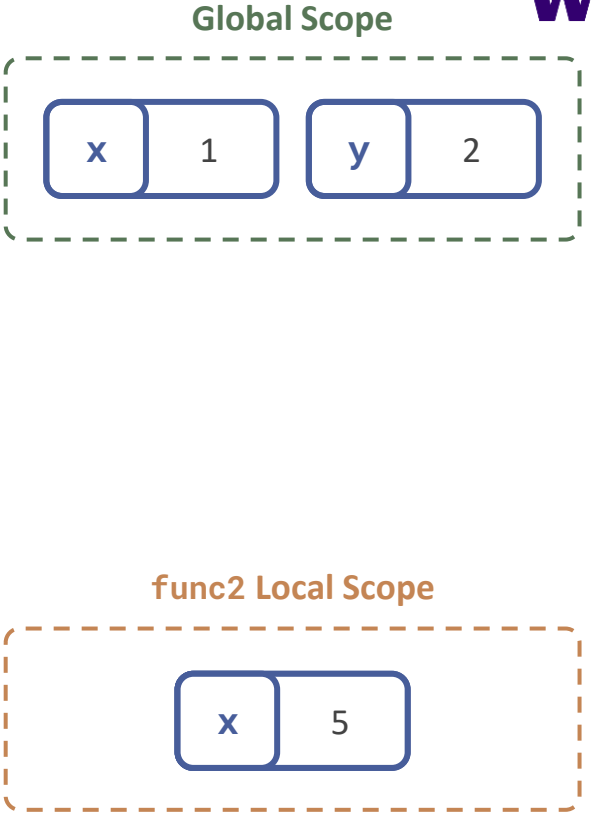
Code Execution

Variables

Slippery ~~Slope~~ Scope (3)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1
1. [8] **x** (**func2** local scope) updated to 5

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```



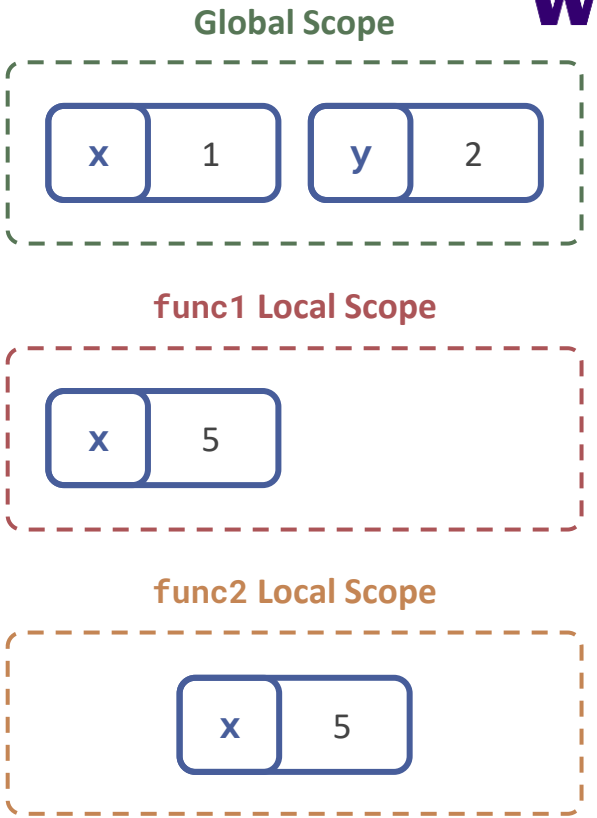
Code Execution

Variables

Slippery ~~Slope~~ Scope (4)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1
1. [8] **x** (**func2** local scope) updated to 5
1. [9] Call **func1**, parameter **x** = 5

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```



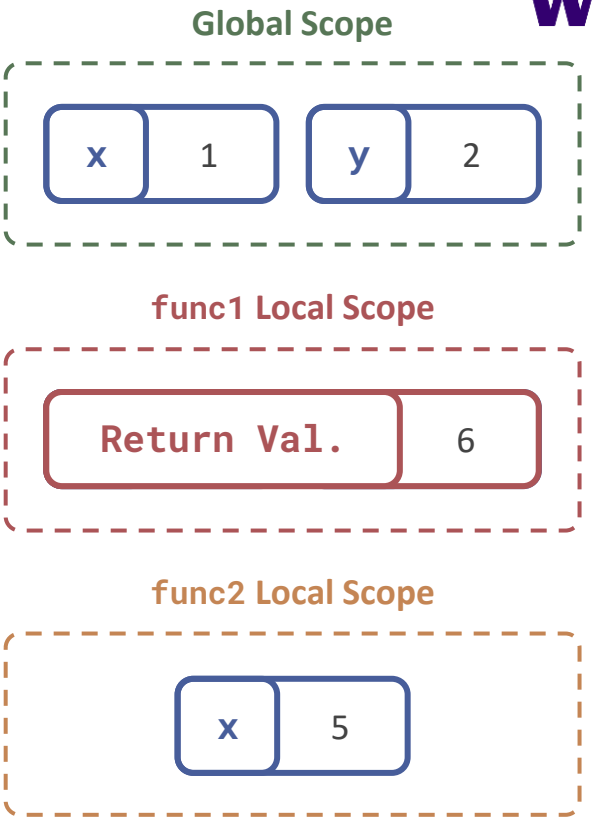
Code Execution

Variables

Slippery ~~Slope~~ Scope (5)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1
1. [8] **x** (**func2** local scope) updated to 5
1. [9] Call **func1**, parameter **x** = 5
1. [4, 5] **y** = 1 (**func1** local scope), return **x** + **y** = 6

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```



Code Execution

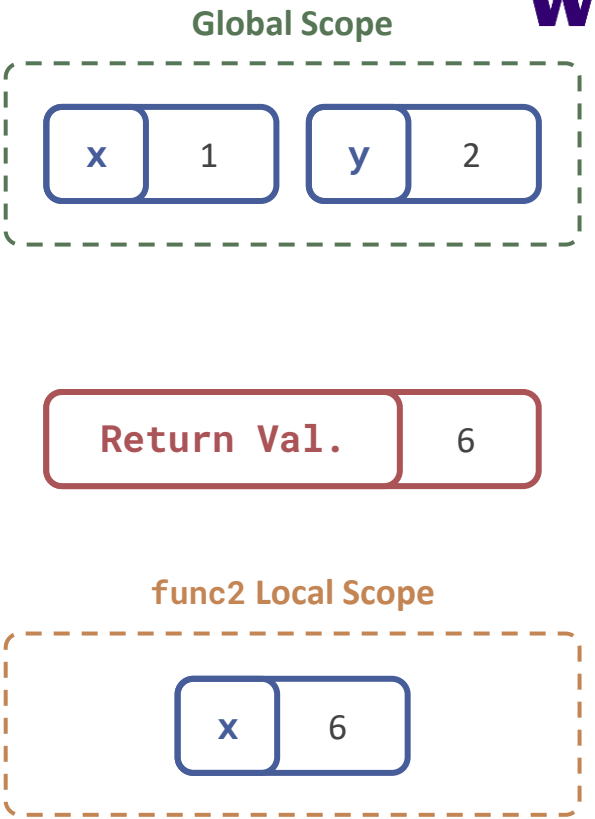
Variables

Slippery ~~Slope~~ Scope (6)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1
1. [8] **x** (**func2** local scope) updated to 5
1. [9] Call **func1**, parameter **x** = 5
1. [4, 5] **y** = 1 (**func1** local scope), return **x** + **y** = 6
1. [9] **x** (**func2** local scope) updated to 6

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```

Code Execution

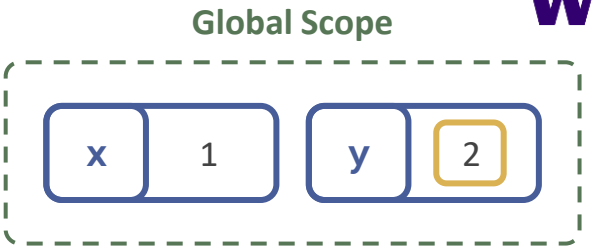


Variables

Slippery ~~Slope~~ Scope (7)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1
1. [8] **x** (**func2** local scope) updated to 5
1. [9] Call **func1**, parameter **x** = 5
1. [4, 5] **y** = 1 (**func1** local scope), return **x** + **y** = 6
1. [9] **x** (**func2** local scope) updated to 6
1. [10] return **x** (local) + **y** (global) → 6 + 2 = 8

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```



Code Execution

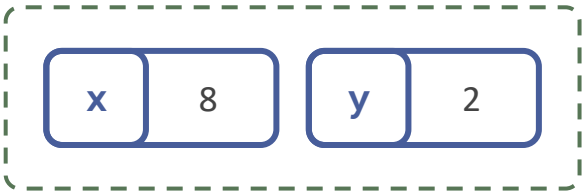
Variables

Slippery ~~Slope~~ Scope (8)

1. [1, 2] **x**, **y** defined in **global scope**
1. [12] Call **func2**, parameter **x** = 1
1. [8] **x** (**func2** local scope) updated to 5
1. [9] Call **func1**, parameter **x** = 5
1. [4, 5] **y** = 1 (**func1** local scope), return **x** + **y** = 6
1. [9] **x** (**func2** local scope) updated to 6
1. [10] return **x** (**local**) + **y** (**global**) → 6 + 2 = 8
1. [12] **x** (**global scope**) updated to 8

Code Execution

```
1  x = 1
2  y = 2
3  def func1(x):
4      y = 1
5      return x + y
6
7  def func2(x):
8      x = 5
9      x = func1(x)
10     return x + y
11
12 x = func2(x)
13 print("x = " + str(x))
```



Variables