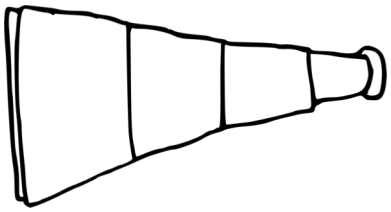
Variable Scope
Functions Calling Functions

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

Variable Scope

- The ***scope*** of a variable is the portion of your program in which the variable can be accessed.
- We need to distinguish between:
  - ***local*** variables: limited to a particular function
  - ***global*** variables: can be accessed anywhere



Local Variables

```
def mystery(x, y):
    b = x - y        # b is a local var of mystery
    return 2*b       # we can access b here

c = 7
mystery(5, 2)
print(b + c)         # we can't access b here!
```

- When we assign a value to a variable inside a function, we create a *local variable*.
  - it "belongs" to that function
  - it can't be accessed outside of that function
- The parameters of a function are also limited to that function.
  - example: the parameters x and y above

Global Variables

```
def mystery(x, y):
    b = x - y
    return 2*b + c    # works, but not recommended

c = 7                 # c is a global variable
mystery(5, 2)
print(b + c)          # we can access c here
```

- When we assign a value to a variable *outside* of a function, we create a *global variable*.
  - it belongs to the *global scope*
- A global variable can be used anywhere in your program.
  - in code that is outside of any function
  - in code inside a function (but this is not recommended)
- Neither globals nor locals exist until they are assigned a value!

Different Variables With the Same Name!

```
def mystery(x, y):
    b = x - y          # this b is local
    return 2*b         # we access the local b here

b = 1                  # this b is global
c = 7
mystery(5, 2)
print(b + c)           # we access the global b here
```

- The program above has two different variables called b.
  - one local variable
  - one global variable
- When this happens, the *local* variable has priority inside the function to which it belongs.

What is the output of this code?

```
def mystery2(a, b):
    x = a + b
    return x + 1

x = 8
mystery2(3, 2)
print(x)
```

A. 5

B. 6

C. 8

D. 9

E. none of these, because an error is produced

What is the output of this code?

```
def mystery2(a, b):
    x = a + b
    return x + 1

x = 8
mystery2(3, 2)
print(x)
```

A. 5

B. 6

C. 8

D. 9

E. none of these, because an error is produced

What is the output of this code?

```
def mystery2(a, b): # there are two different x's!
    x = a + b       # this x is local to mystery2
    return x + 1

x = 8               # this x is global
mystery2(3, 2)
print(x)
```

A. 5

B. 6

C. 8

D. 9

E. none of these, because an error is produced

Follow-up question:
Why don't we see the following?
6
8

`mystery2(3, 2)` returns 6,
but we don't print the return value.
We essentially "throw it away"!

What is the output of this code? (version 2)

```
def mystery2(a, b):
    x = a + b
    return x + 1

x = 8
mystery2(3, 2)
print(x)
```

A. 5

B. 6

C. 8

D. 9

E. none of these, because an error is produced

What is the output of this code? (version 2)

```
def mystery2(a, b):
    x = a + b
    return x + 1

x = 8
mystery2(3, 2)
print(x)        # the only x belongs to mystery2,
                # so we can't access it here.
```

A. 5

B. 6

C. 8

D. 9

E. **none of these, because an error is produced**

A Note About Globals

- It's not a good idea to access a global variable inside a function.
  - for example, you shouldn't do this:

```
def average3(a, b):
    total = a + b + c    # accessing a global c
    return total/3

c = 8
print(average3(5, 7))
```

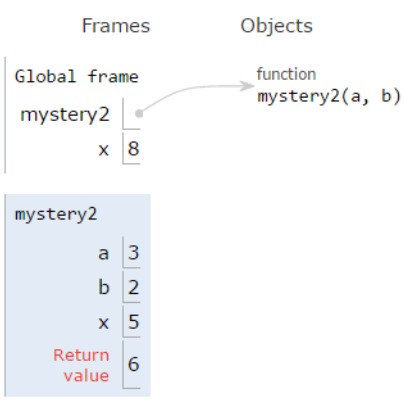
- Instead, you should pass it in as a parameter/input:

```
def average3(a, b, c):
    total = a + b + c    # accessing input c
    return total/3

c = 8
print(average3(5, 7, c))
```

Frames and the Stack

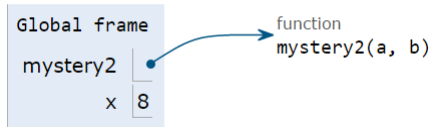
- Variables are stored in blocks of memory known as *frames.*
- Each function call gets a frame for its local variables.
  - goes away when the function returns
- Global variables are stored in the global frame.
- The *stack* is the region of the computer's memory in which the frames are stored.
  - thus, they are also known as *stack frames*



Visualizing How Functions Work
(pythontutor.com/visualize.html)

- Before the call to `mystery2`:

```
1  def mystery2(a, b):
2      x = a + b
3      return x + 1
4
5  x = 8
6  mystery2(3, 2)
7  print(x)
```



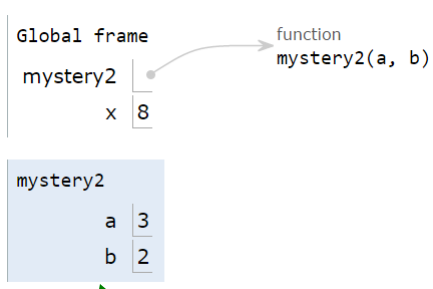
The global frame includes the function names and the global variables.

line that has just executed
next line to execute

Visualizing How Functions Work
(pythontutor.com/visualize.html)

- At the start of the call to `mystery2`:

```
1  def mystery2(a, b):
2      x = a + b
3      return x + 1
4
5  x = 8
6  mystery2(3, 2)
7  print(x)
```



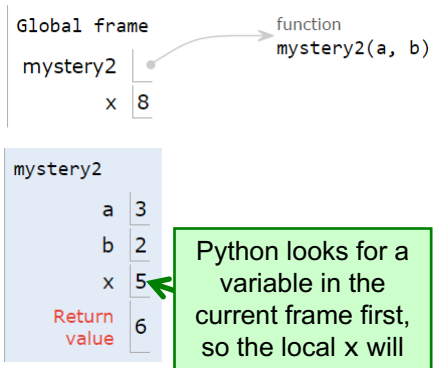
`mystery2(3, 2)` gets its own frame containing the variables that belong to it. `mystery2`'s x isn't shown yet because we haven't assigned anything to it.

line that has just executed
next line to execute

Visualizing How Functions Work
(pythontutor.com/visualize.html)

- When the call to `mystery2` is about to return:

```
1  def mystery2(a, b):
2      x = a + b
3      return x + 1
4
5  x = 8
6  mystery2(3, 2)
7  print(x)
```



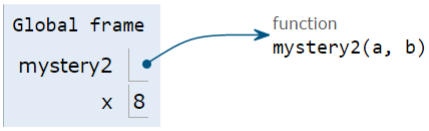
Python looks for a variable in the current frame first, so the local x will be used instead of the global x when returning x + 1.

line that has just executed
next line to execute

Visualizing How Functions Work
(pythontutor.com/visualize.html)

- After the call to `mystery2` has returned:

```
1  def mystery2(a, b):
2      x = a + b
3      return x + 1
4
5  x = 8
6  mystery2(3, 2)
7  print(x)
```



When a function call returns, its frame is removed from memory. Its local variables can no longer be accessed.

- The only x that remains is the global x, so its value is printed.

What is the output of this code?

```
def quadruple(y):
    y = 4 * y
    return y

y = 8
quadruple(y)

print(y)
```

A. 4

B. 8

C. 12

D. 32

E. none of these, because an error is produced

What is the output of this code?

```
def quadruple(y):     # the parameter y is local
    y = 4 * y
    return y

y = 8                 # this y is global
quadruple(y)

print(y)
```

A. 4

B. 8

C. 12

D. 32

E. none of these, because an error is produced

What is the output of this code?

```
def quadruple(y):     # 3. local y = 8
    y = 4 * y
    return y

y = 8                 # 1. global y = 8
quadruple(y)          # 2. pass in global y's value

print(y)
```

A. 4

B. 8

C. 12

D. 32

E. none of these, because an error is produced

What is the output of this code?

```
def quadruple(y):     # 3. local y = 8
    y = 4 * y         # 4. local y = 4 * 8 = 32
    return y          # 5. return local y's value
            32
y = 8                 # 1. global y = 8
quadruple(y)          # 2. pass in global y's value
                      # 6. return value is thrown away!
print(y)
```

A. 4

B. 8

C. 12

D. 32

E. none of these, because an error is produced

What is the output of this code?

```
def quadruple(y):     # 3. local y = 8
    y = 4 * y         # 4. local y = 4 * 8 = 32
    return y          # 5. return local y's value
            32
y = 8               # 1. global y = 8
quadruple(y)        # 2. pass in global y's value
                    # 6. return value is thrown away!
print(y)            # 7. print global y's value,
                    #    which is unchanged!
```

A. 4

B. 8

C. 12

D. 32

E. none of these, because an error is produced

You ***can't*** change the value of a variable by passing it into a function!

How could we change this to see the return value?

```
def quadruple(y):
    y = 4 * y
    return y

y = 8
quadruple(y)
print(y)
```

Seeing the return value (option 1)

```
def quadruple(y):
    y = 4 * y
    return y

y = 8
y = quadruple(y)    # assign return val to global y
print(y)
```

Seeing the return value (option 2)

```
def quadruple(y):
    y = 4 * y
    return y

y = 8
print(quadruple(y))    # print return val
print(y)
```

What is the output of this program?

```
def demo(x):
    return x + f(x)
def f(x):
    return 11*g(x) + g(x//2)
def g(x):
    return -1 * x
print(demo(-4))
```

A. 4

B. 42

C. 44

D. 46

E. none of these

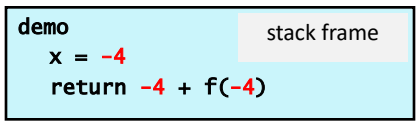
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



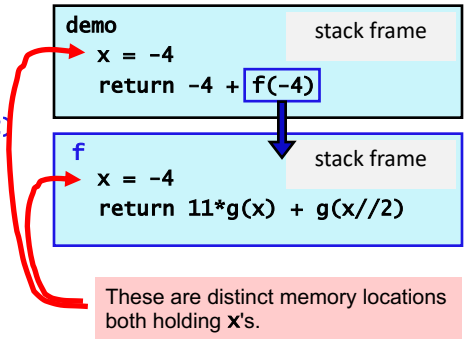
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



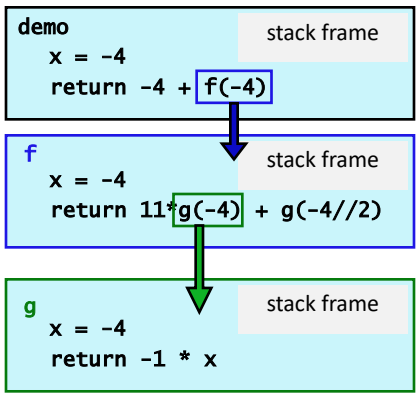
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



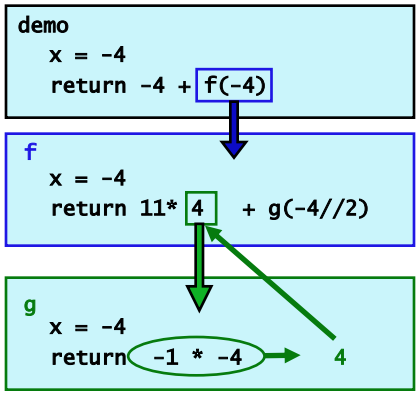
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



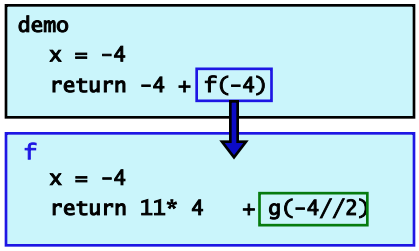
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



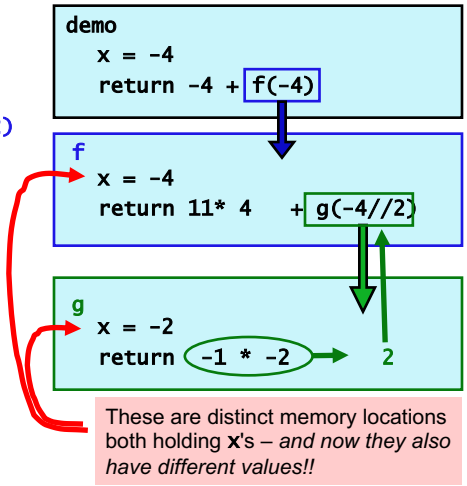
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



These are distinct memory locations both holding **x**'s – *and now they also have different values!!*

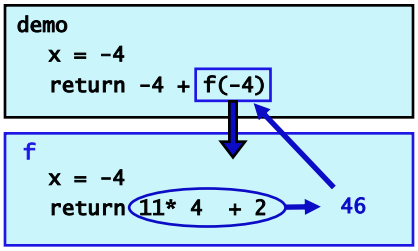
Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```



Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```

```
demo
  x = -4
  return -4 +  46  →  42
```

Functions Calling Other Functions!

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))   # print(42)
```

```
42
```

What is the output of this program?

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x//2)

def g(x):
    return -1 * x

print(demo(-4))
```

A. 4

B. 42

C. 44

D. 46

E. none of these

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo
x | y

global
x | y

output

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0
        0  2
y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

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

foo

x	y

global

x	y
2	0

output

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0
        0  2
y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

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

foo

x	y
0	2

global

x	y
2	0

output

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3

global

x	y
2	0

output

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3

global

x	y
2	0

output

3 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x
```

3

```
x = 2
y = 0
```

3

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3

global

x	y
2	0

output

3 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3

global

x	y
2	0
2	3

output

3 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3

global

x	y
2	0
2	3

output

3 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3

global

x	y
2	0
2	3

output

3 3
2 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)
    2   2
foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3
2	2

global

x	y
2	0
2	3

output

3 3
2 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3
2	2
5	3

global

x	y
2	0
2	3

output

3 3
2 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3
2	2
5	3

global

x	y
2	0
2	3

output

3 3
2 3
5 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x
            5
x = 2
y = 0

y = foo(y, x)
print(x, y)
     5
foo(x, x)     # throw return value away!
print(x, y)

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

foo

x	y
0	2
3	3
2	2
5	3

global

x	y
2	0
2	3

output

3 3
2 3
5 3

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

foo

x	y
0	2
3	3
2	2
5	3

global

x	y
2	0
2	3

output

3 3
2 3
5 3
2 3

What does the rest do?

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

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

foo(x, x)
print(x, y)

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

Trace through the rest on your own!

foo

x	y
0	2
3	3
2	2
5	3

global

x	y
2	0
2	3

output

3 3
2 3
5 3
2 3