Today

- More about functions
  - Parameters
  - Return values
  - Call stacks
  - Recursion
  - Variable Scope

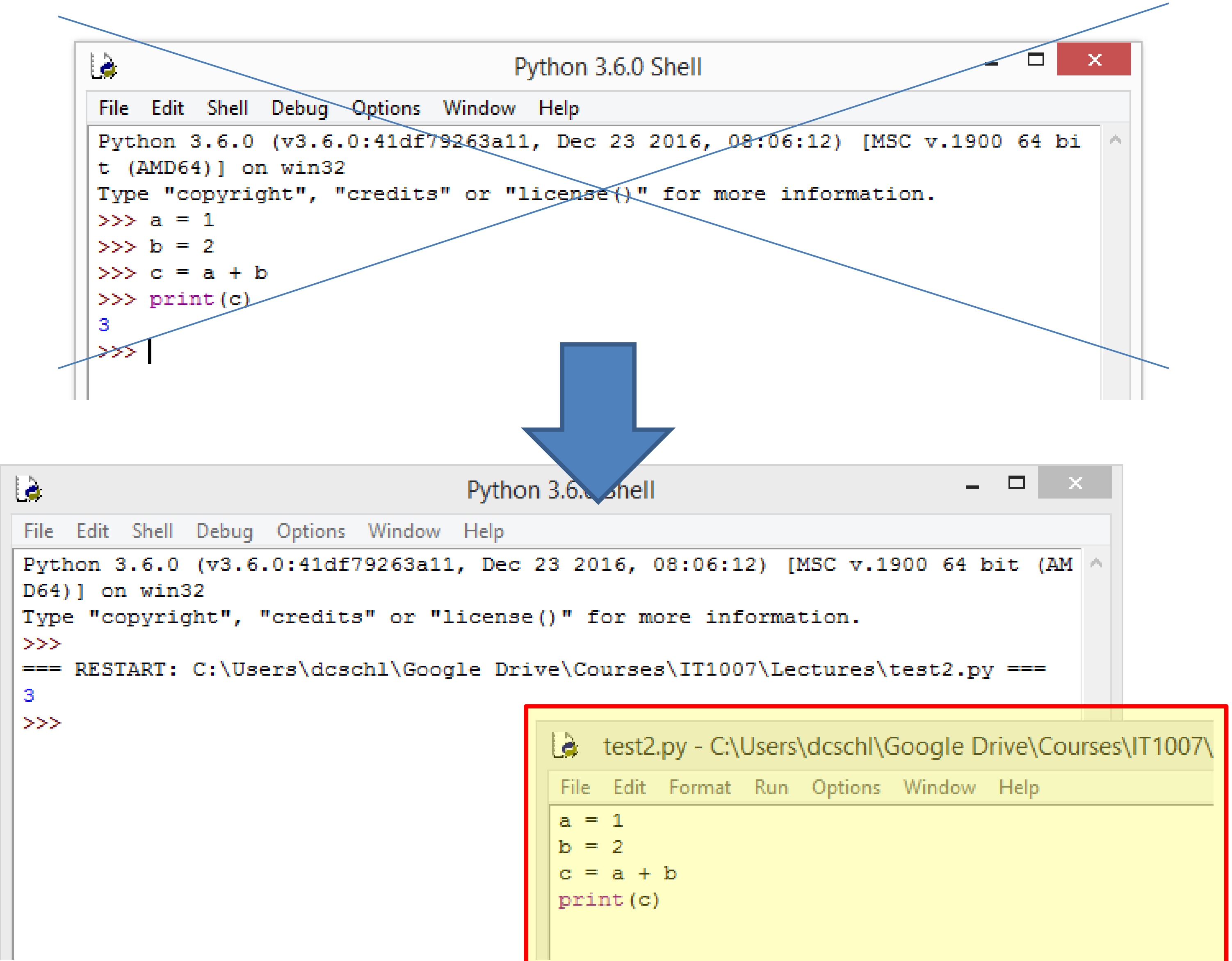
Observation from Tlabs

- Attendance is VERY high
  - And good interactions
  - A lot of questions asked/answered
- Faced REAL programming problem
  - Learn REAL things
- One third of us can finish Part A in the two hours
  - There are a few can even finish Part B

Observation from Tlabs

- Some confused with C syntax
  - I saw someone wrote
    - `for (i = 0; i < n; i ++)`
- Some are still used to the console coding

Let’s Move Out of the Console



Python 3.6.0 Shell
File Edit Shell Debug Options Window Help

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bi
t (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 1
>>> b = 2
>>> c = a + b
>>> print(c)
3
>>> 
```

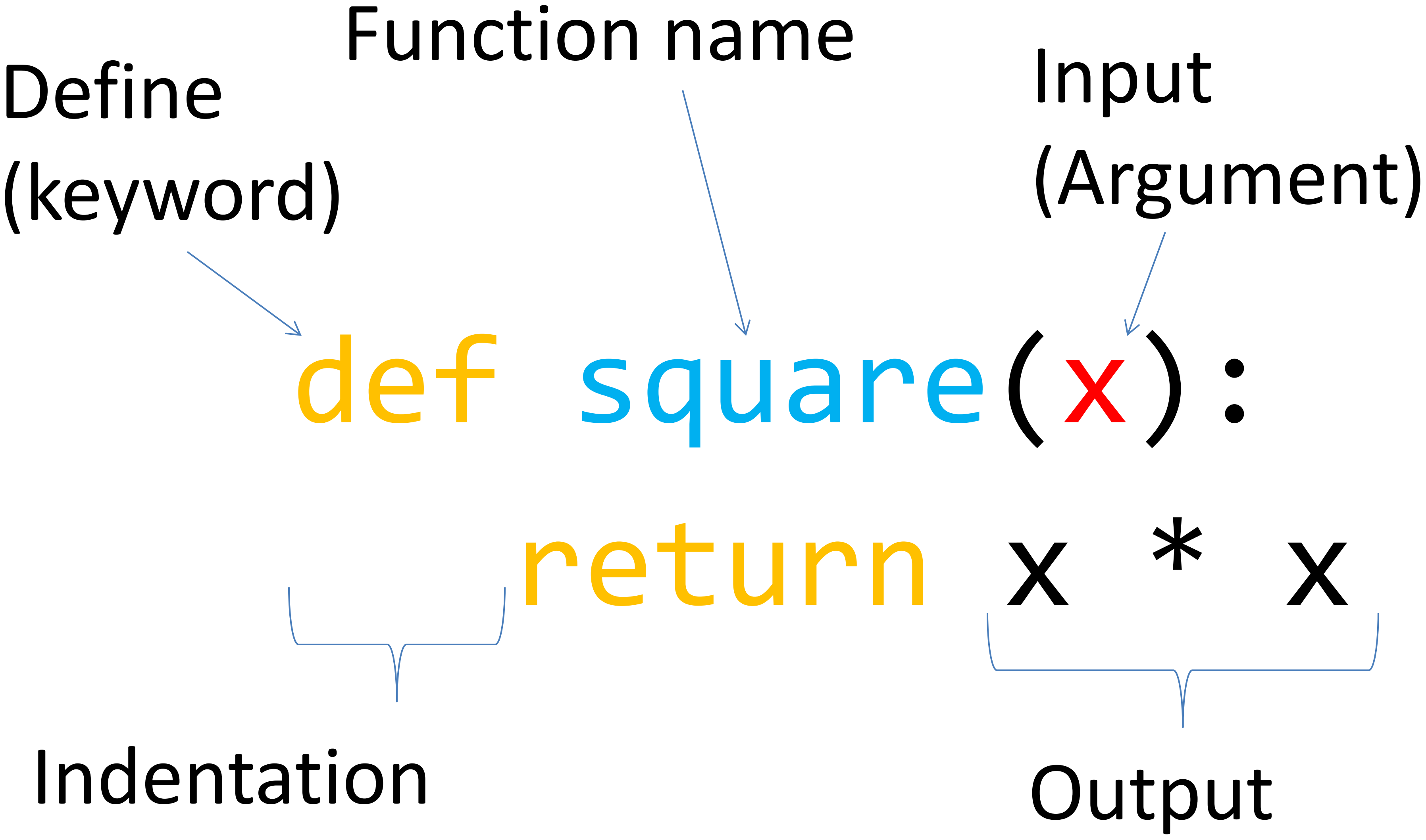
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
=== RESTART: C:\Users\dcschl\Google Drive\Courses\IT1007\Lectures\test2.py ===
3
>>> 
```

test2.py - C:\Users\dcschl\Google Drive\Courses\IT1007\
File Edit Format Run Options Window Help

```
a = 1
b = 2
c = a + b
print(c)
```

Recap: Simple Functions



Function name

Define (keyword)

Input (Argument)

```
def square(x):
    return x * x
```

Indentation

Output

Parameters of Functions

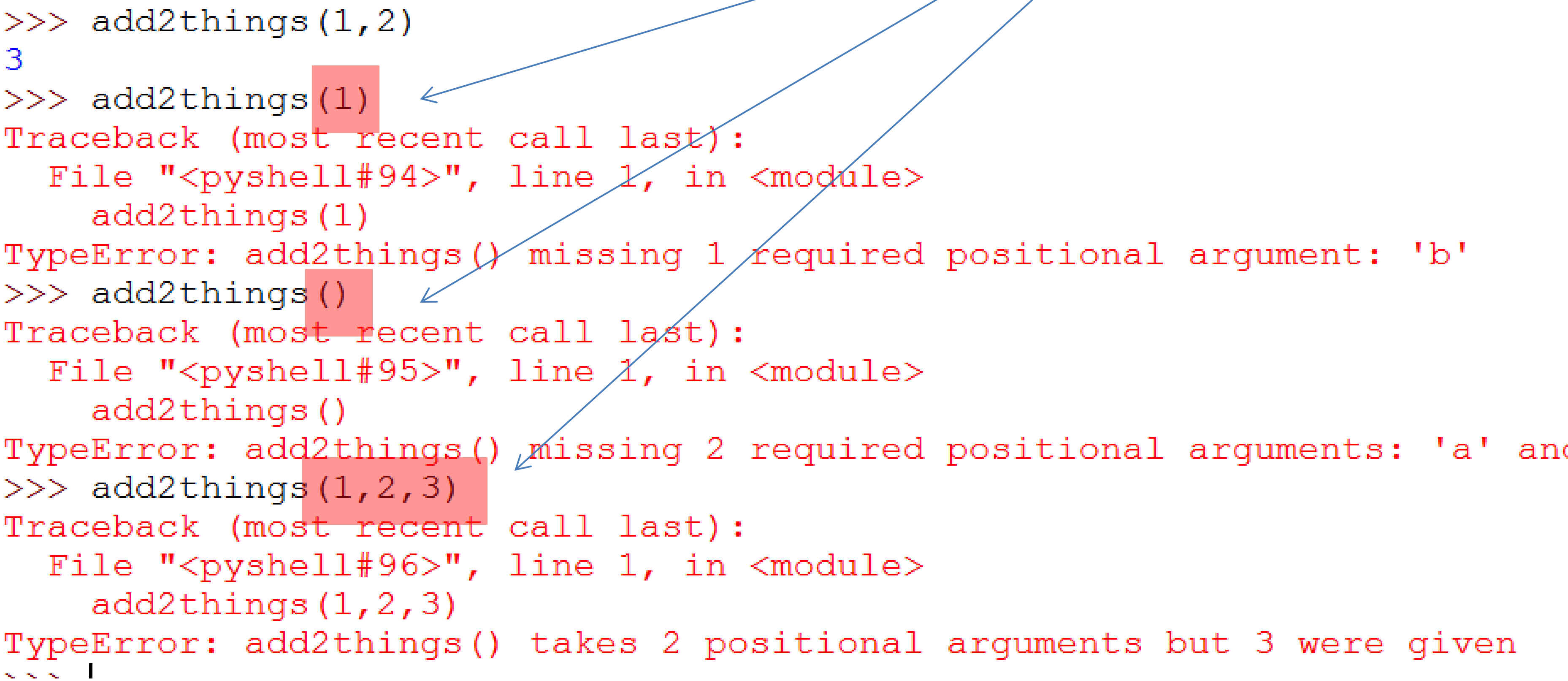
= input

= arguments

Input Parameters

```
def add2things(a,b)
    return a + b
```

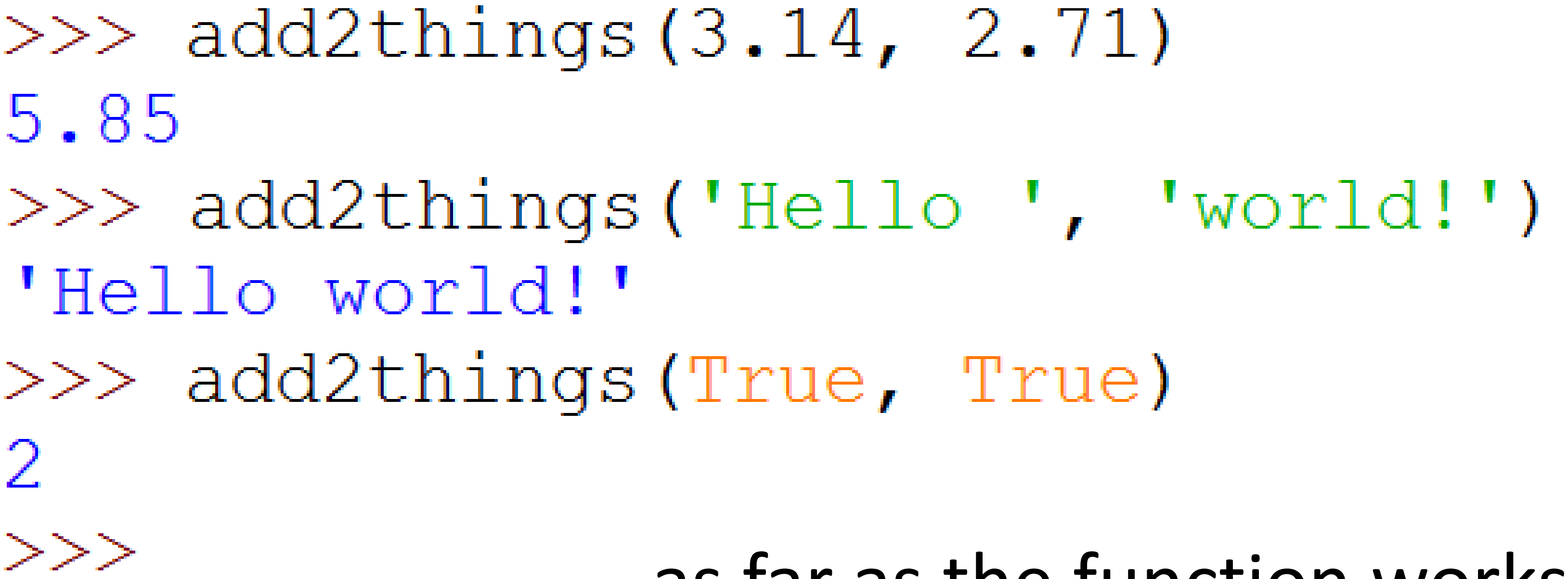
Must be the same number of items



```
>>> add2things(1,2)
3
>>> add2things(1)
Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    add2things(1)
TypeError: add2things() missing 1 required positional argument: 'b'
>>> add2things()
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in <module>
    add2things()
TypeError: add2things() missing 2 required positional arguments: 'a' and
>>> add2things(1,2,3)
Traceback (most recent call last):
  File "<pyshell#96>", line 1, in <module>
    add2things(1,2,3)
TypeError: add2things() takes 2 positional arguments but 3 were given
>>>
```

Parameter Types

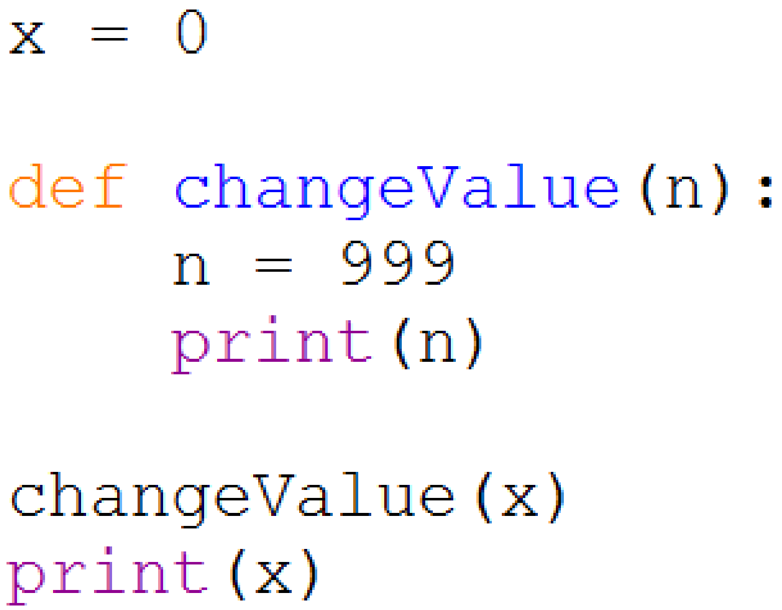
- In Python, parameters have no declared types. We can pass any kind of variable to the function….



```
>>> add2things(3.14, 2.71)
5.85
>>> add2things('Hello ', 'world!')
'Hello world!'
>>> add2things(True, True)
2
>>>
```

…. as far as the function works

Pass by Values

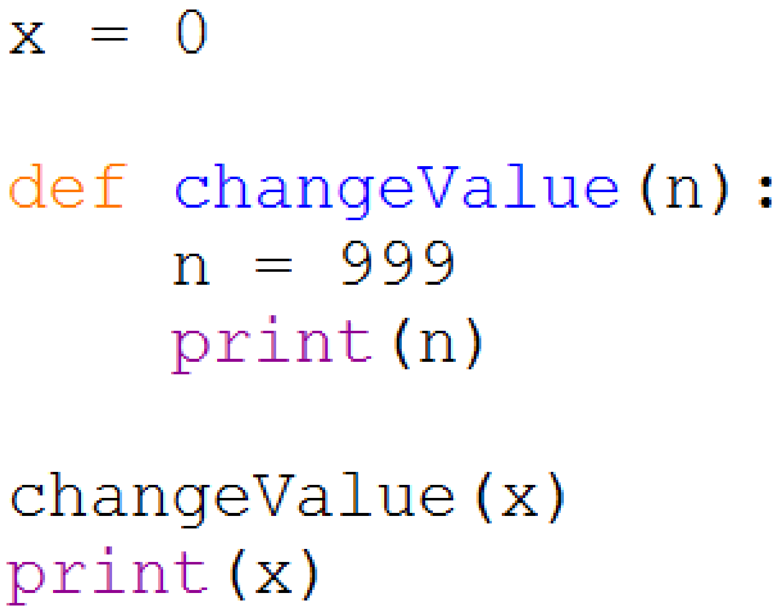


```
x = 0

def changeValue(n):
    n = 999
    print(n)

changeValue(x)
print(x)
```

- The `print` () in “`changeValue`” will print 999
- But how about the last `print(x)?`
  - Will x becomes 999?

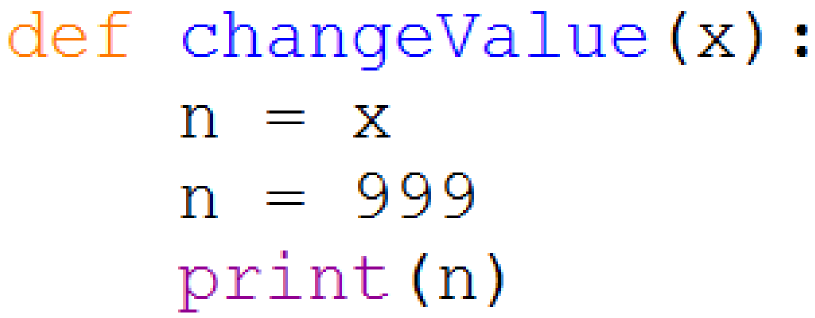
- (So actually this function will NOT change the value of x)

Pass by Values



```
x = 0

def changeValue(n):
    n = 999
    print(n)

changeValue(x)
print(x)
```

- n is another copy of x
- You can deem it as



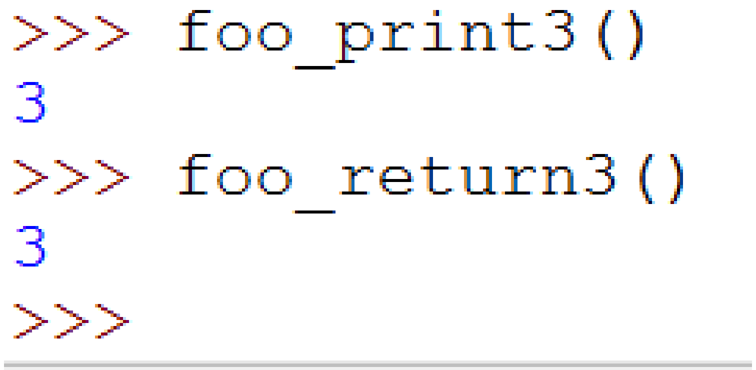
```
def changeValue(x):
    n = x
    n = 999
    print(n)
```

Return Values

Vs “`print()`”

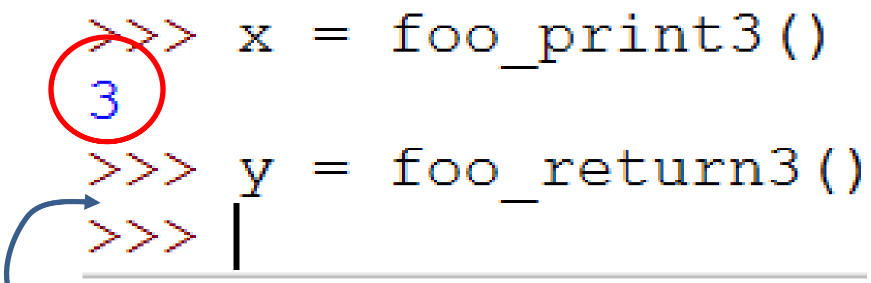
Print vs Return

```
def foo_print3():
    print(3)

def foo_return3():
    return 3
```



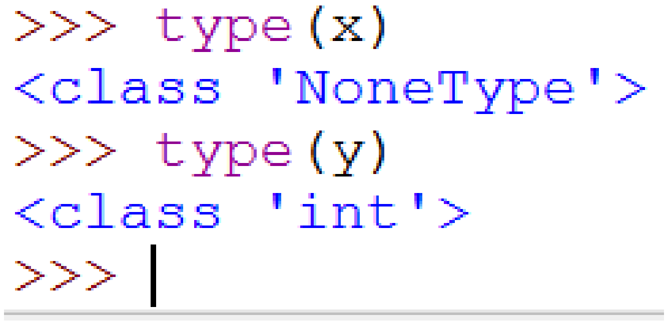
```
>>> foo_print3()
3
>>> foo_return3()
3
>>>
```

Wait…

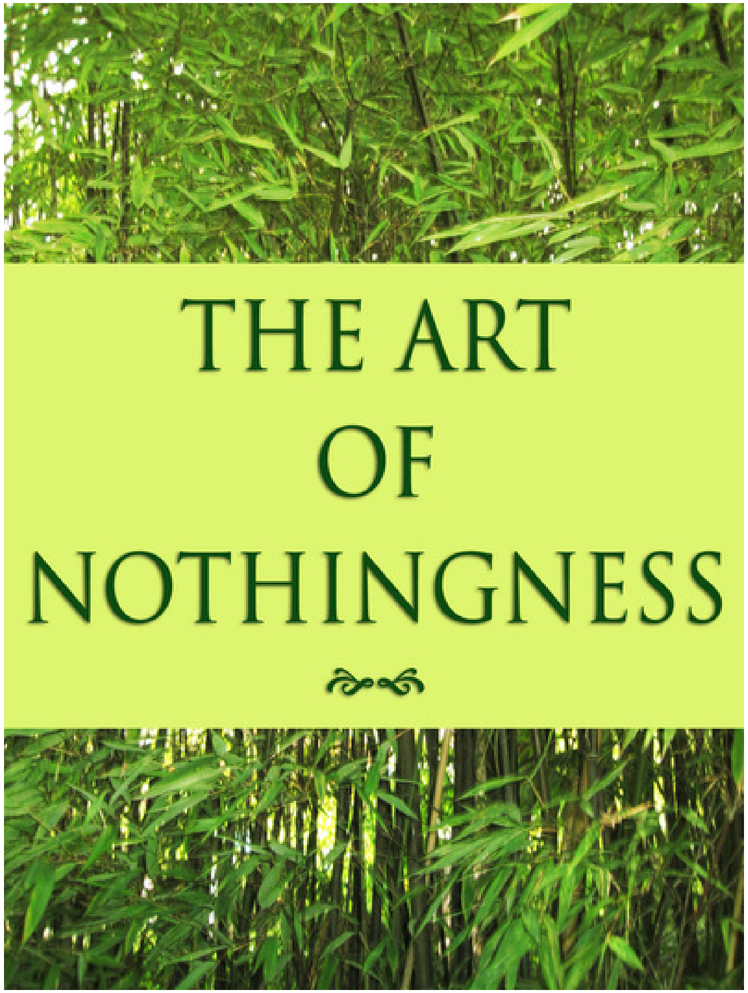


```
>>> x = foo_print3()
3
>>> y = foo_return3()
>>> 
```

Nothing?

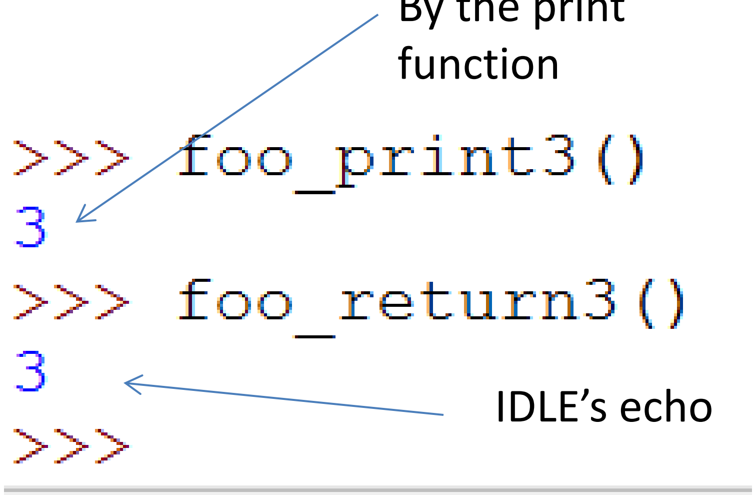


```
>>> type(x)
<class 'NoneType'>
>>> type(y)
<class 'int'>
>>> 
```



Print vs Return

```
def foo_print3():
    print(3)

def foo_return3():
    return 3
```



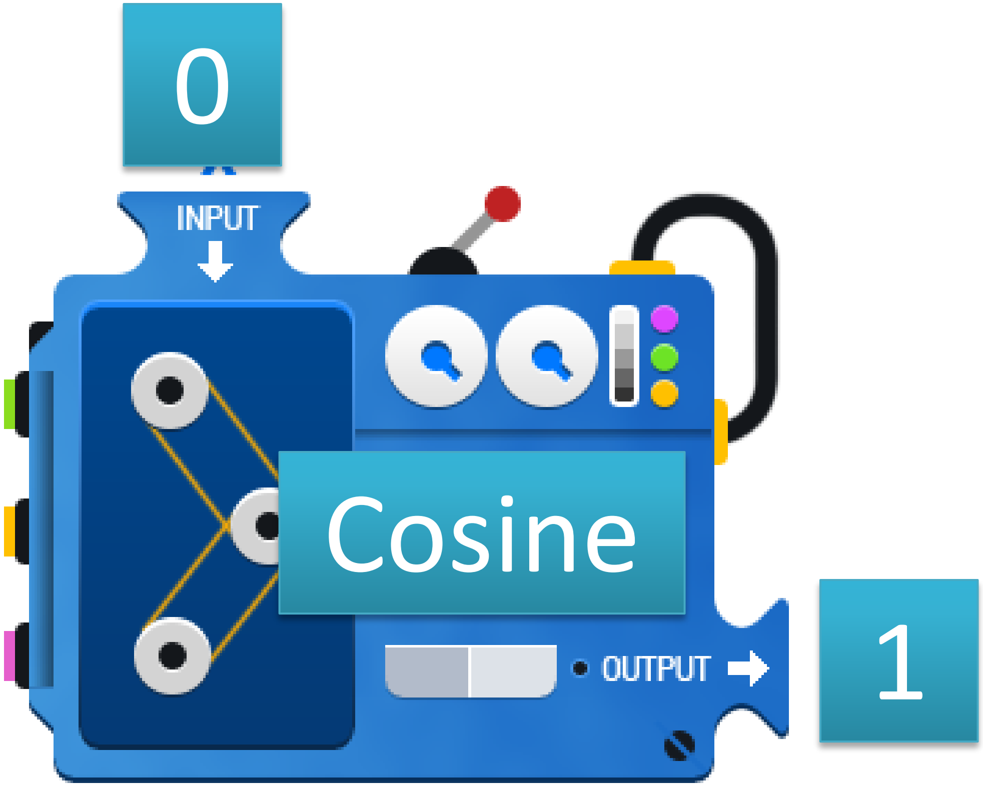
```
>>> foo_print3()
3
>>> foo_return3()
3
>>>
```

By the print function

IDLE’s echo

Function

- “Cosine” is a function
  - Input 0
  - Output 1
  - x = cos(0)
  - x = 1

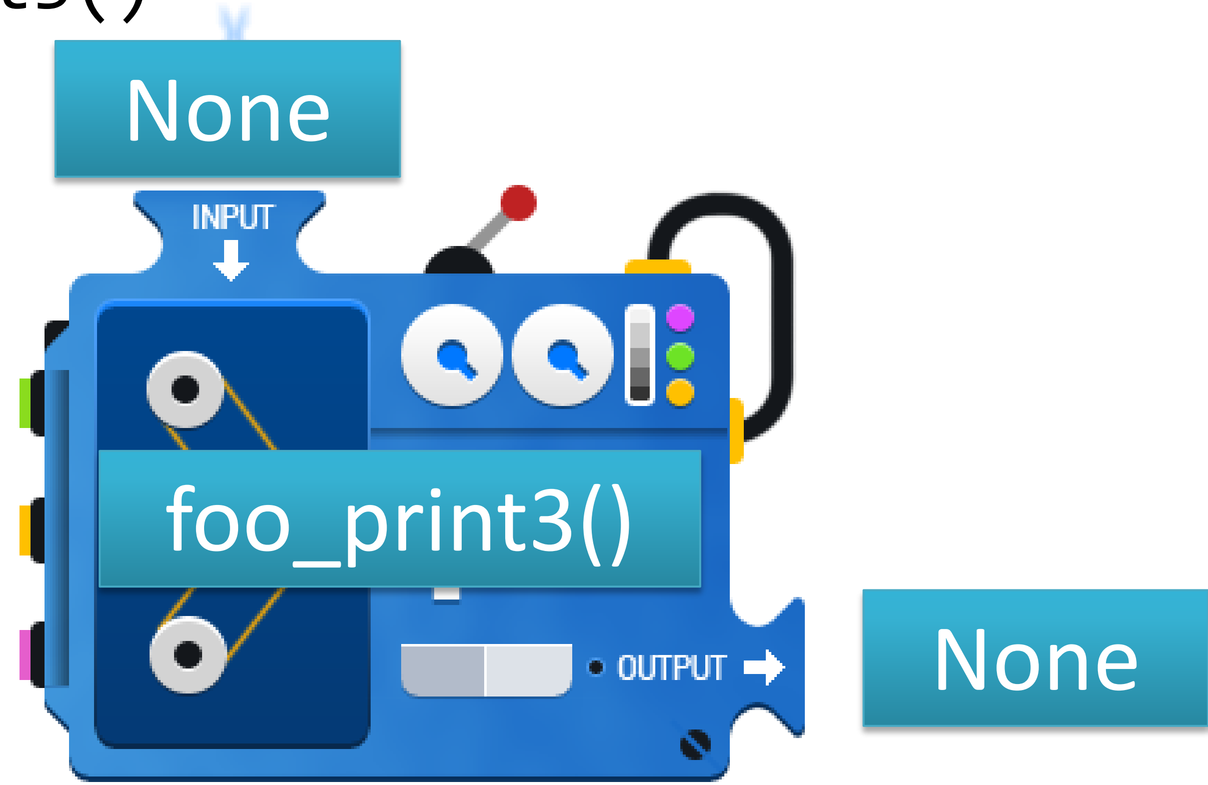


Function

- “foo_print3()” is a function
  - Input 0
  - No output

```
y = foo_print3()
```

None



None

foo_print3()

None

In general, we called all these “functions”

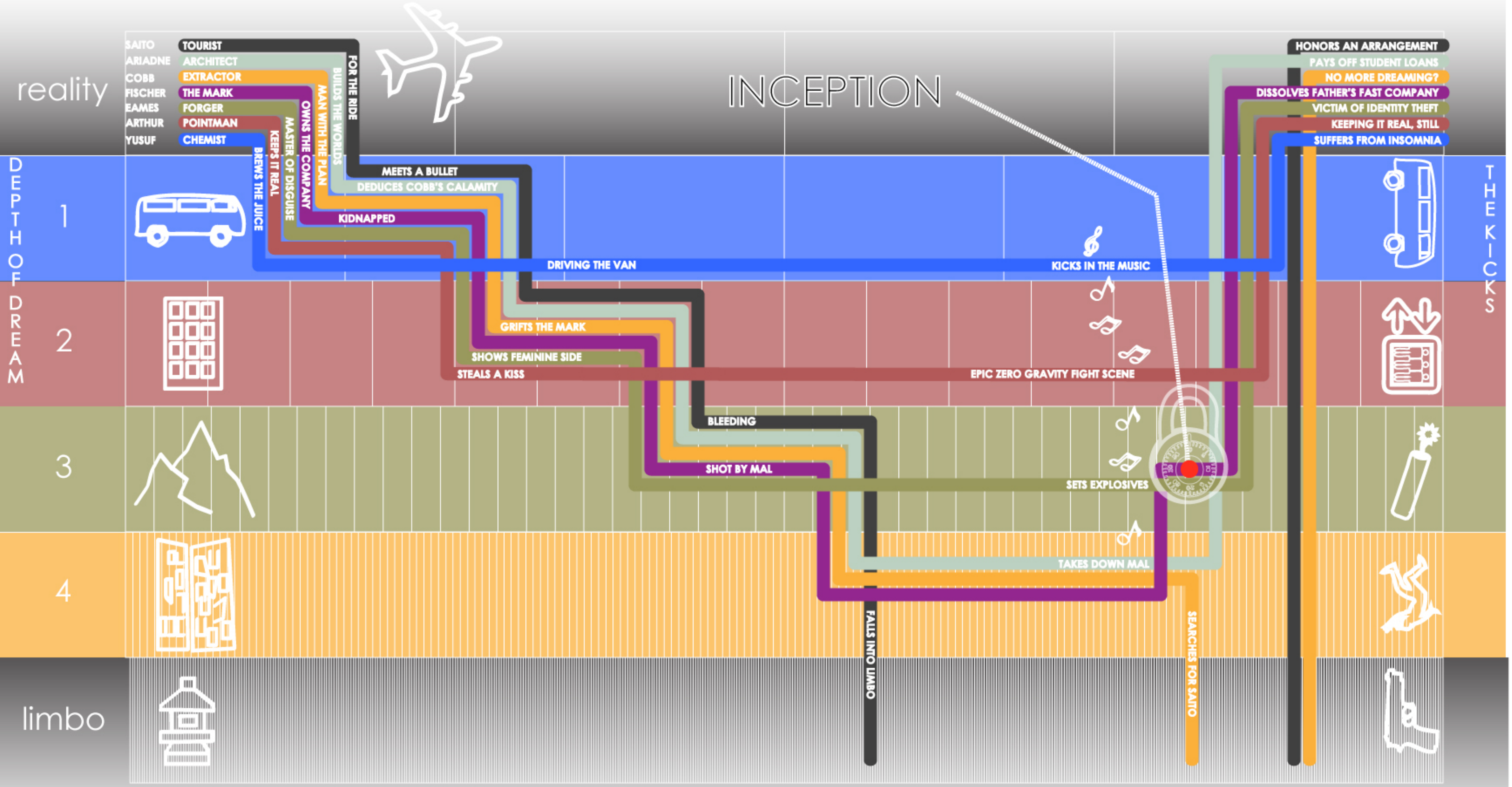
But for a function that “returns” nothing. Sometime we call it a “procedure”

Return Values

- All functions returns “something”
- `foo_return3()` return the integer 3
- `foo_print3()`
  - Do not have any return statement
  - So it returns “None”

Question: Can we assume that a function always return something of the same TYPE?

The Call Stack



INCEPTION

reality

SAITO TOURIST

ARIADNE ARCHITECT

COBB EXTRACTOR

FISCHER THE MARK

EAMES FORGER

ARTHUR POINTMAN

YUSUF CHEMIST

FOR THE RIDE

BUILDS THE WORLDS

MAN WITH THE PLAN

OWNS THE COMPANY

MASTER OF DISGUISE

KEEPS IT REAL

BREWS THE JUICE

DEPTH OF DREAM

1

MEETS A BULLET

DEDUCES COBB'S CALAMITY

KIDNAPPED

DRIVING THE VAN

KICKS IN THE MUSIC

2

GRIFTS THE MARK

SHOWS FEMININE SIDE

STEALS A KISS

EPIC ZERO GRAVITY FIGHT SCENE

3

BLEEDING

SHOT BY MAL

SETS EXPLOSIVES

4

TAKES DOWN MAL

FALLS INTO LIMBO

SEARCHES FOR SAITO

limbo

HONORS AN ARRANGEMENT

PAYS OFF STUDENT LOANS

NO MORE DREAMING?

DISSOLVES FATHER'S FAST COMPANY

VICTIM OF IDENTITY THEFT

KEEPING IT REAL, STILL

SUFFERS FROM INSOMNIA

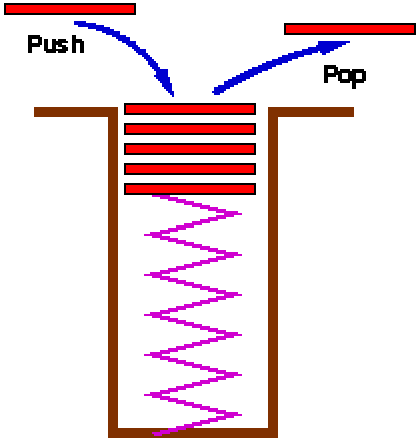
THE KICKS

PERCEPTION OF TIME BECOMES SLOWER AS DREAM DEPTH INCREASES

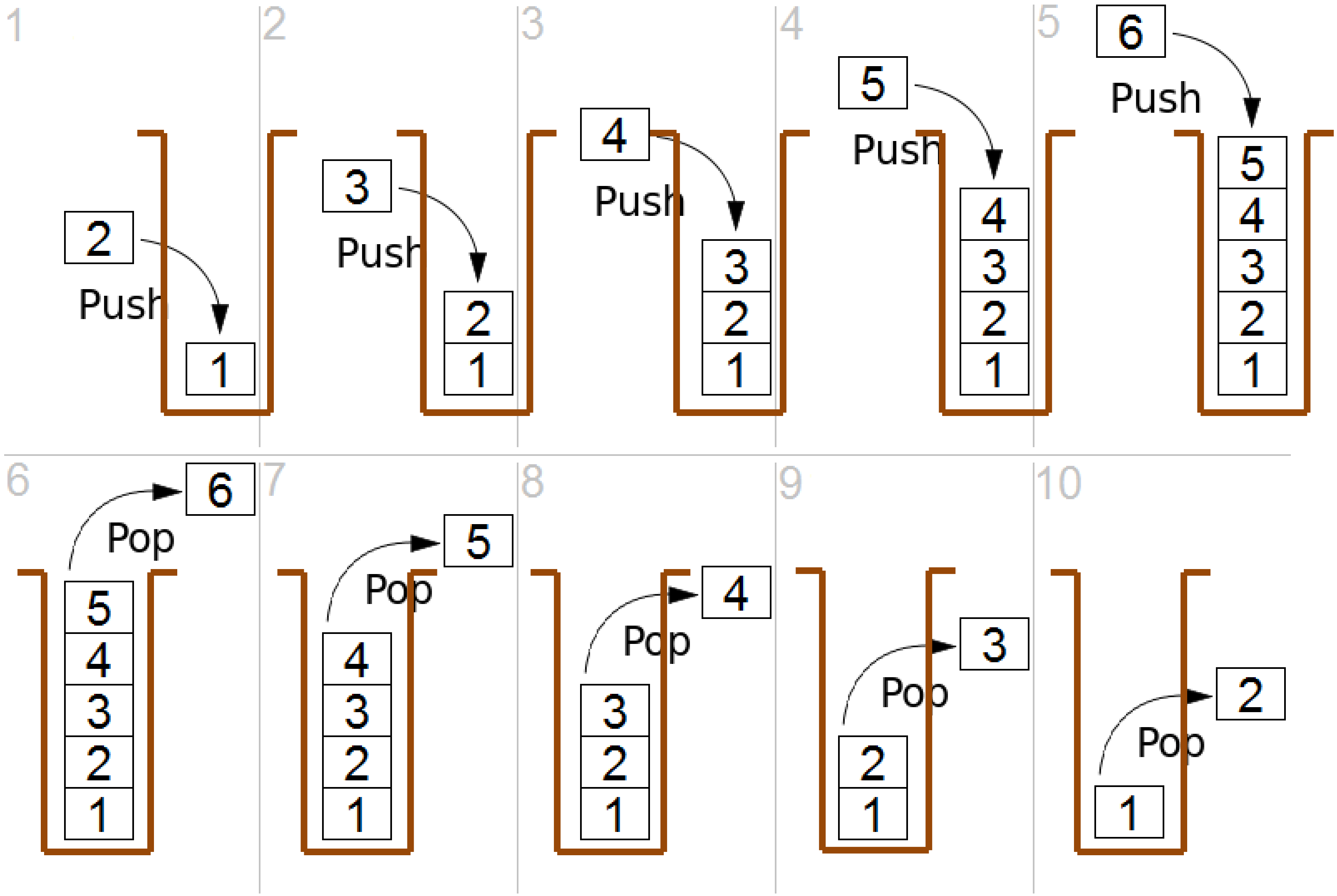
2010 INFOGRAPHIC
BY DANIEL WANG

Stack

- First in last out order



First in Last Out



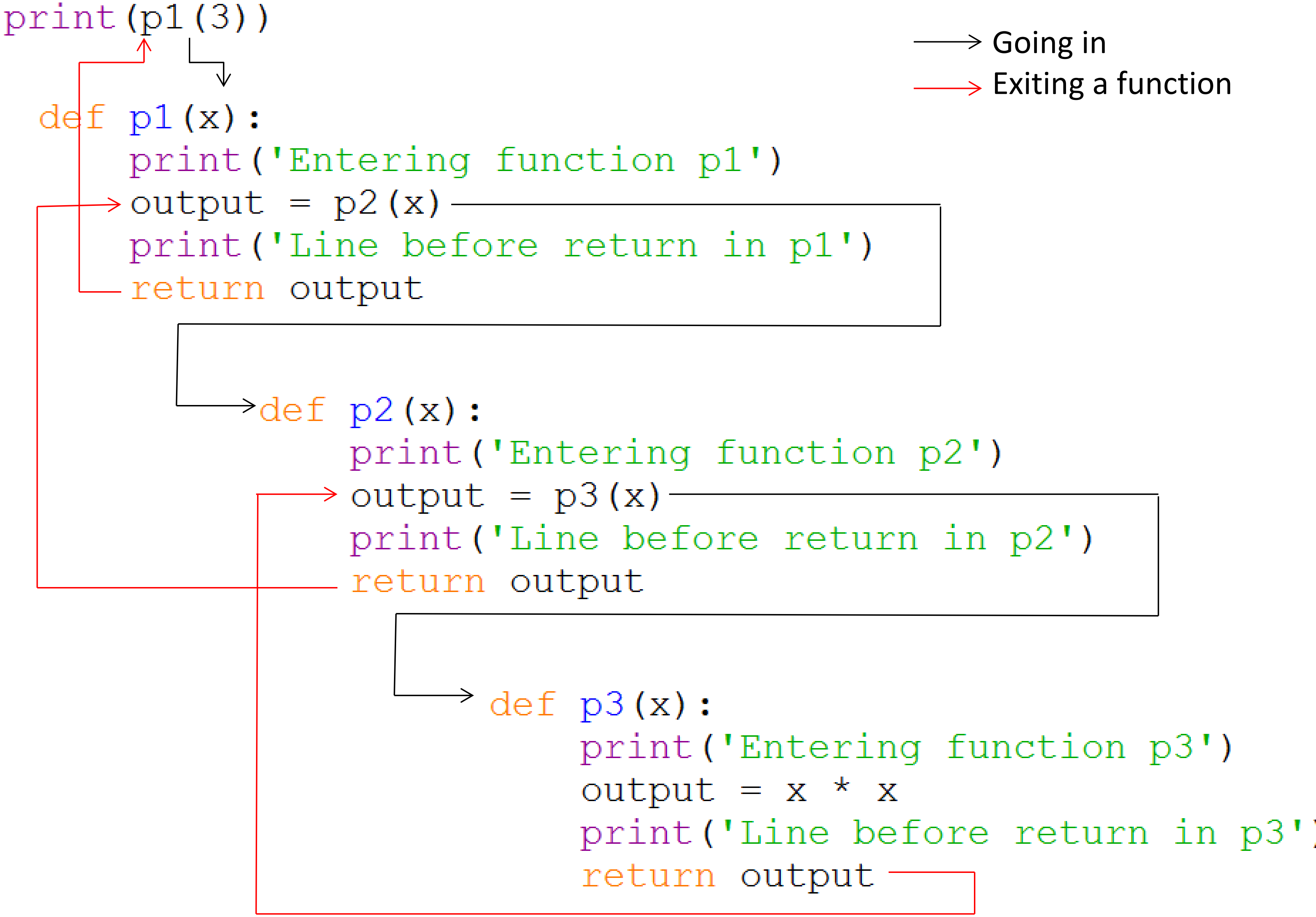
The Stack (or the Call Stack)

```
def p1(x):
    print('Entering function p1')
    output = p2(x)
    print('Line before return in p1')
    return output


def p2(x):
    print('Entering function p2')
    output = p3(x)
    print('Line before return in p2')
    return output

def p3(x):
    print('Entering function p3')
    output = x * x
    print('Line before return in p3')
    return output


print(p1(3))
```

The Stack (or the Call Stack)

```
>>> p1(3)
Entering function p1
Entering function p2
Entering function p3
Line before return in p3
Line before return in p2
Line before return in p1
9
```

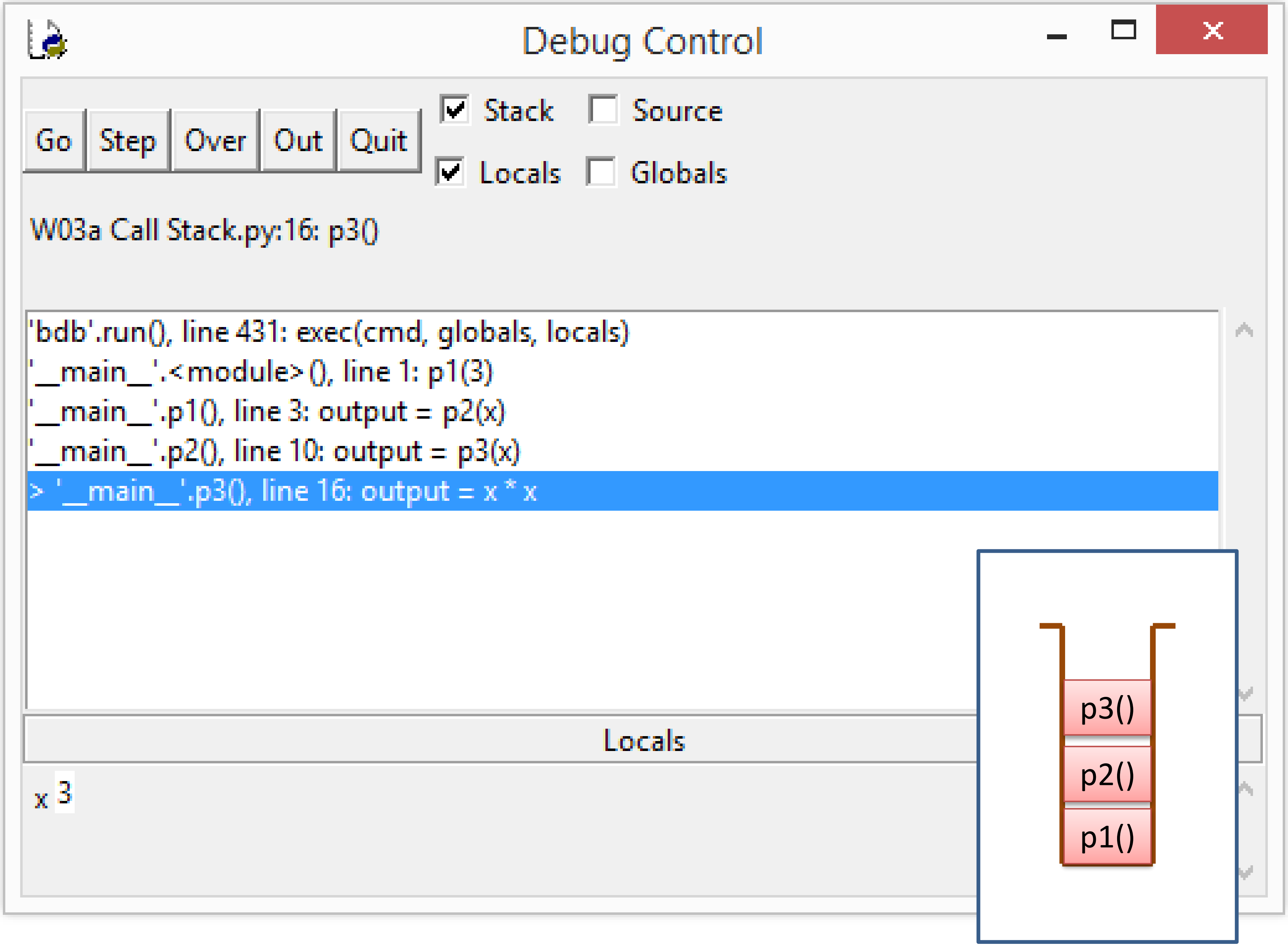
FILO!



```
print(p1(3))

  def p1(x):
      print('Entering function p1')
      output = p2(x)
      print('Line before return in p1')
      return output

            def p2(x):
                print('Entering function p2')
                output = p3(x)
                print('Line before return in p2')
                return output

                      def p3(x):
                          print('Entering function p3')
                          output = x * x
                          print('Line before return in p3'
                          return output
```

→ Going in

→ Exiting a function



☑ Stack □ Source

☑ Locals □ Globals

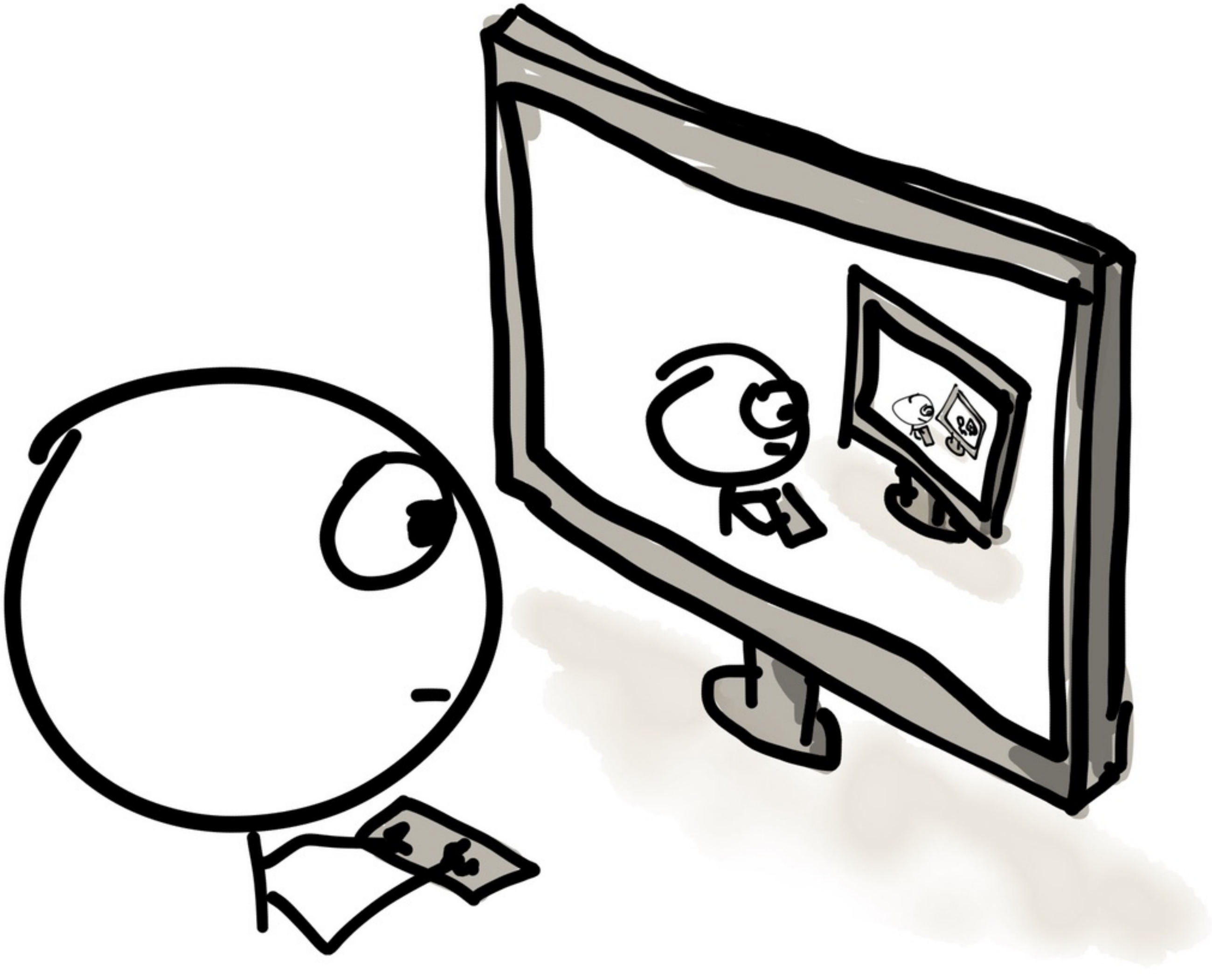
W03a Call Stack.py:16: p3()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'__main__'.<module>(), line 1: p1(3)
'__main__'.p1(), line 3: output = p2(x)
'__main__'.p2(), line 10: output = p3(x)
> '__main__'.p3(), line 16: output = x * x
```

Locals

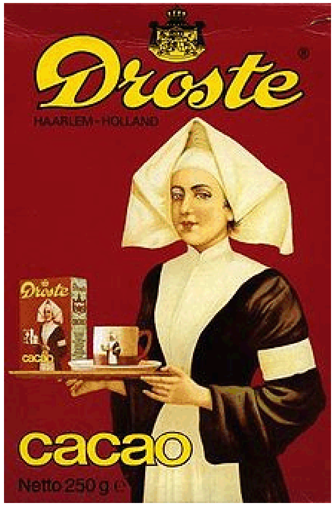
x 3

Recursion

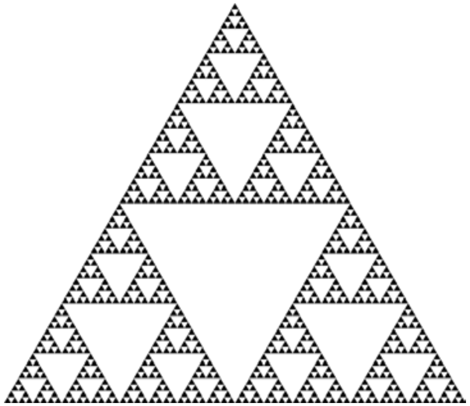


A Central Idea of CS

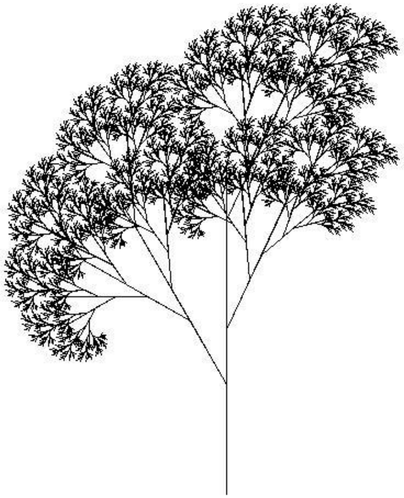
Some examples of recursion (inside and outside CS):



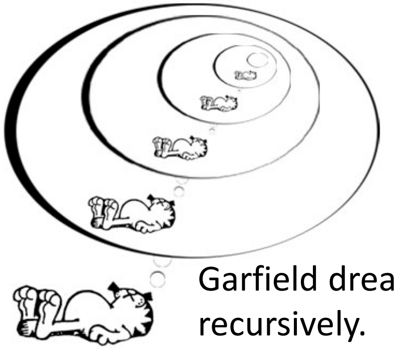
Droste effect



Sierpinksi triangle



Recursive tree



Garfield dreaming recursively.

Mandelbrot Fractal Endless Zoom

Recursion

- A function that calls itself
- And extremely powerful technique
- Solve a big problem by solving a smaller version of itself
  - Mini-me

Factorial

- The factorial n! is defined by

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

- Write a function for factorial?

```
def factorial(n):
    ans = 1
    i = 1
    while i <= n:
          ans = ans * i
          i = i + 1
    print(ans)
```

```
>>> factorial(3)
6
>>> factorial(6)
720
>>>
```

Factorial

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{otherwise} \end{cases}$$

Factorial

```
def factorial(n):
        ans = 1
        i = 1
        while i <= n:
                ans = ans * i
                i = i + 1
        print(ans)
```

```
def factorialR(n):
    if n == 1:
        return 1
    else:
        return n * factorialR(n-1)
```

Recursion

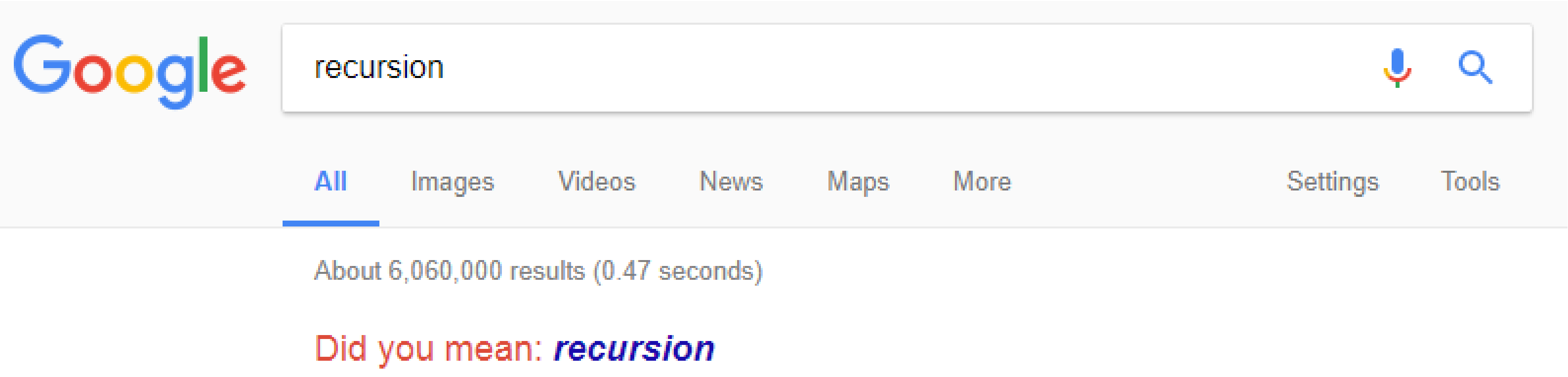
- Rules of recursion

Must have a **terminal** condition

```
def factorialR(n):
    if n == 1:
        return 1
    else:
        return n * factorialR(n-1)
```

Must **reduce** the **size** of the problem for every layer

Google about Recursion



About 6,060,000 results (0.47 seconds)

Did you mean: ***recursion***

- Try to search these in Google:
  - Do a barrel roll
  - Askew
  - Anagram
  - Google in 1998
  - Zerg rush
- More in [Google Easter Eggs]

Variable Scope

- What is the difference between the area you receive your **cellular** data signal and your **home wifi** signal?

Global Variable

```
x = 0

def foo_printx():
    print(x)

foo_printx()
print(x)
```

Refers to

- This code will print

```
0
0
```

Global vs Local Variables

```
x = 0

def foo_printx():
    x = 999
    print(x)

foo_printx()
print(x)
```

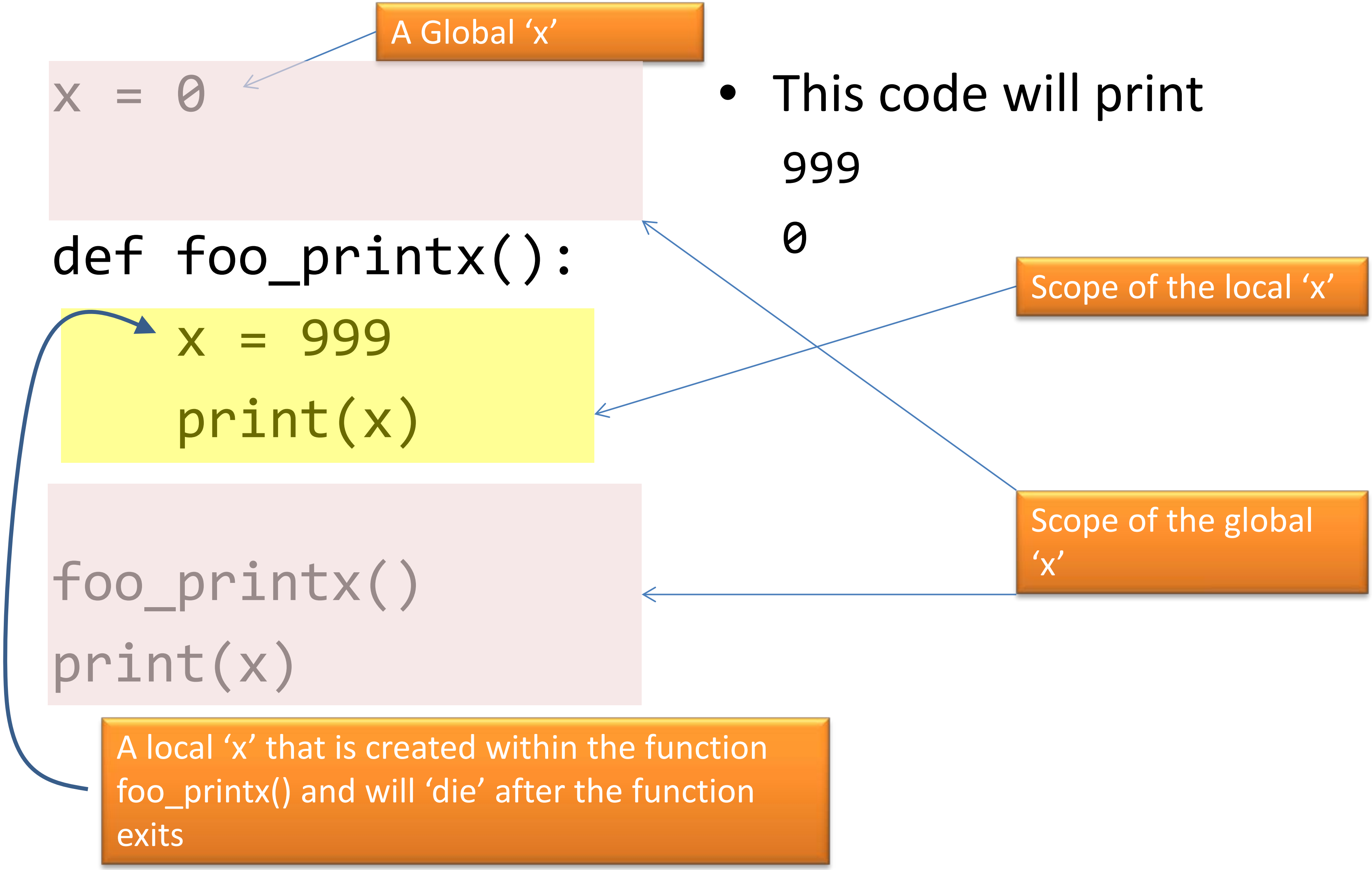
Because, a new ‘x’ is born here!

- This code will print

```
999
0
```

- The first ‘999’ makes sense
- But why the second one is ‘0’?

Global vs Local Variables



A Global ‘x’

```
x = 0

def foo_printx():
    x = 999
    print(x)


foo_printx()
print(x)
```

- This code will print

```
999
0
```

Scope of the local ‘x’

Scope of the global ‘x’

A local ‘x’ that is created within the function foo_printx() and will ‘die’ after the function exits

Global vs Local Variables

- A variable which is defined in the main body of a file is called a ***global*** variable. It will be **visible throughout the file**, and also inside any file which imports that file. EXCEPT…
- A variable which is defined inside a function is ***local*** to that function. It is accessible **from the point at which it is defined until the end of the function**, and exists for as long as the function is executing.
- The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it.

Crossing Boundary

- What if we really want to modify a global variable from inside a function?
- Use the “global” keyword
- (No local variable x is created)

```
x = 0

def foo_printx():
    global x
    x = 999
    print(x)

foo_printx()
print(x)
```

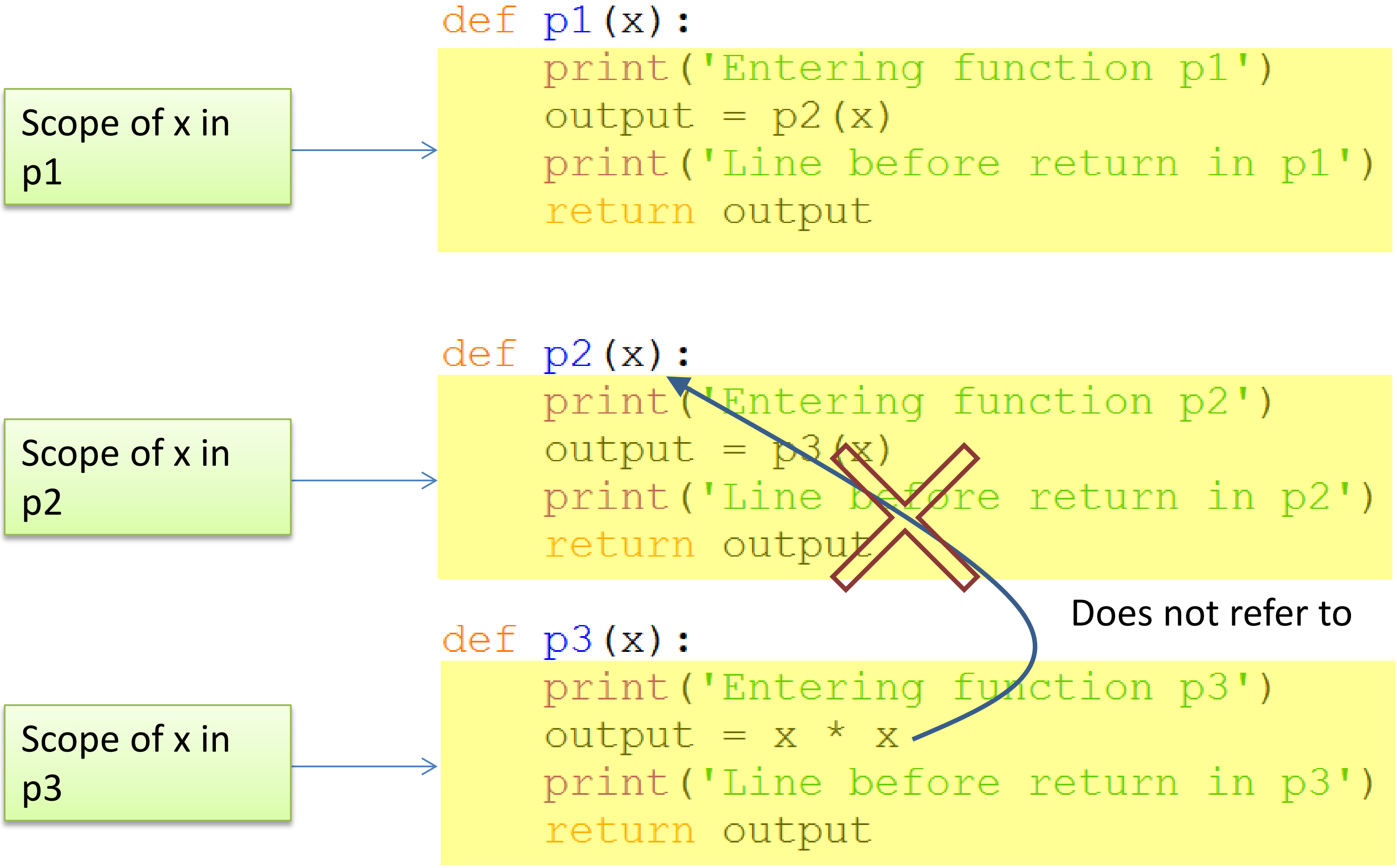
```
Output:
999
999
```

How about… this?

```
x = 0

def foo_printx():
    print(x)
    x = 999
    print(x)

foo_printx()
```

- Local or global?
- Error!
- Because the line "x=999" creates a local version of 'x'
- Then the first print(x) will reference a **local** x that is not assigned with a value
- The line that causes an error

Parameters are LOCAL variables



Scope of x in p1

```
def p1(x):
    print('Entering function p1')
    output = p2(x)
    print('Line before return in p1')
    return output
```

Scope of x in p2

```
def p2(x):
    print('Entering function p2')
    output = p3(x)
    print('Line before return in p2')
    return output
```

Does not refer to

Scope of x in p3

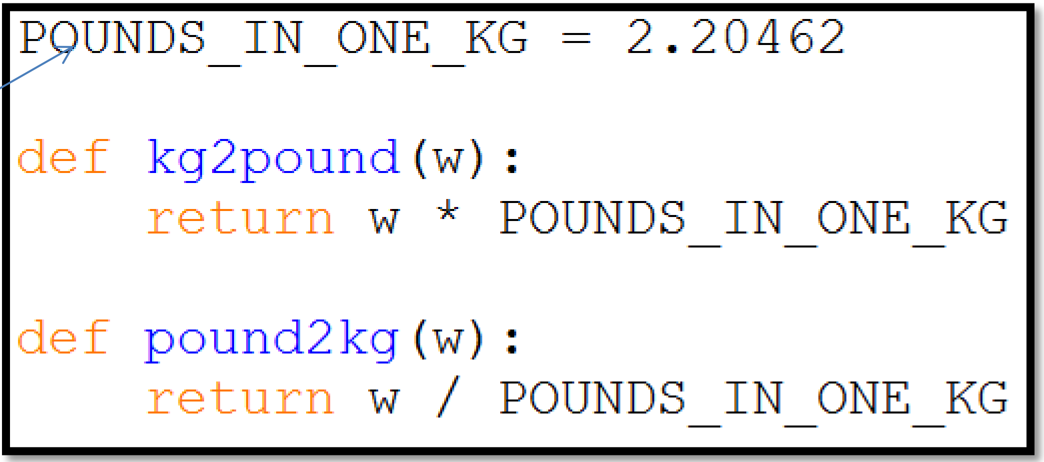
```
def p3(x):
    print('Entering function p3')
    output = x * x
    print('Line before return in p3')
    return output
```

```
print(p1(3))
```

Practices (Convention)

- Global variables are VERY **bad** practices, especially if modification is allowed
- 99% of time, global variables are used as CONSTANTS
  - Variables that every function could access
  - But not expected to be modified

Convention:
Usually in all CAPs



```
POUNDS_IN_ONE_KG = 2.20462

def kg2pound(w):
    return w * POUNDS_IN_ONE_KG

def pound2kg(w):
    return w / POUNDS_IN_ONE_KG
```

Today

- More about functions
  - Parameters
  - Return values
  - Call stacks
  - Recursion
  - Variable Scope

Admin

- This Friday is a holiday, no Tlab
- Remember to submit your Part A within the day of the Tlab
- Remember to submit your Part B before
  - 3 Sept Sunday 11:59pm