

FUNCTIONS,  
LAMBDA  
FUNCTIONS, OOP  
ELEMENTS

# FUNCTIONS

- Reusable pieces of code, called **functions** or **procedures**
- Capture steps of a computation so that we can use with any input
- A function is just some **code written in a special, reusable way**

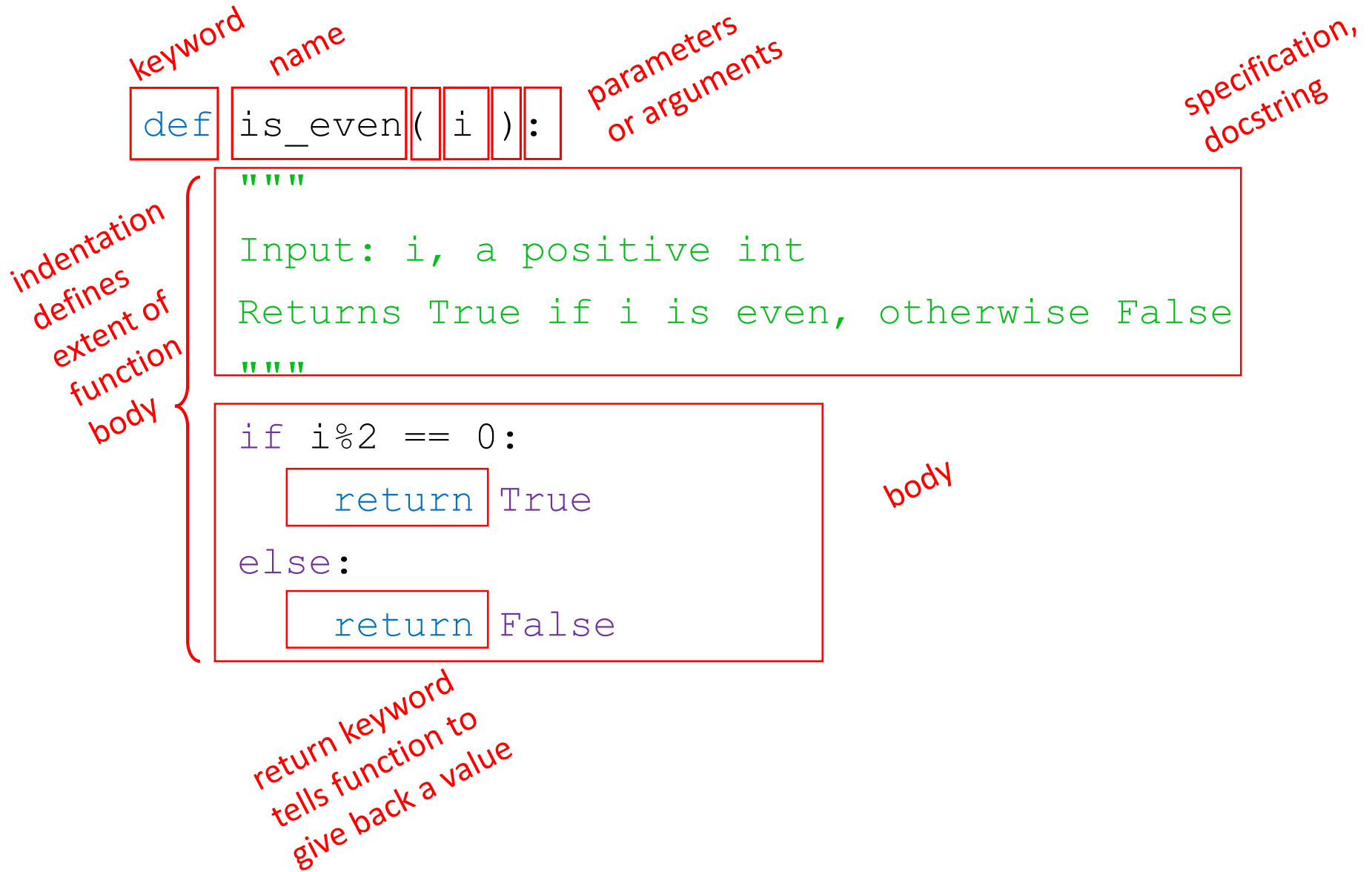
# FUNCTIONS

- **Defining a function** tells Python some code now exists in memory
- Functions are only useful when they are **run** (“**called**” or “**invoked**”)
- You write a function once but can run it many times!
- Compare to code in a file
  - It doesn't run when you load the file
  - It runs when you hit the run button

# FUNCTION CHARACTERISTICS

- Has a **name**
  - (think: variable bound to a function object)
- Has (formal) **parameters** (0 or more)
  - The inputs
- Has a **docstring** (optional but recommended)
  - A comment delineated by `"""` (triple quotes) that provides a **specification** for the function – contract relating output to input
- Has a **body**, a set of instructions to execute when function is called
- **Returns** something
  - Keyword `return`

# HOW to WRITE a FUNCTION



# HOW TO THINK ABOUT WRITING A FUNCTION

- **What** is the problem?
  - Given an int, call it `i`, want to know if it is even
  - Use this to write the function name and specs

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
```

# HOW TO THINK ABOUT WRITING A FUNCTION

- How to **solve** the problem?
  - Can check that remainder when divided by 2 is 0
  - Think about what value you need to give back

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

# HOW TO THINK ABOUT WRITING A FUNCTION

- Can you make the code **cleaner**?
  - `i%2` is a Boolean that evaluates to True/False already

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    return i%2 == 0
```

# HOW TO CALL (INVOKE) A FUNCTION

*Name of the function*

*Values for parameters  
of the function*

`is_even(3)`

`is_even(8)`

- That's all!

# HOW TO CALL (INVOKE) A FUNCTION

```
is_even(3)
```

```
is_even(8)
```

*Replaced by the return!*

- That's all!

# ALL TOGETHER IN A FILE

- This code might be in one file

```
def is_even( i ):  
    return i%2 == 0
```

Function definition

```
is_even(3)
```

Function call

# WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:  
**formal parameters** in function def with **values from function call**  
**i** replaced with **3**

```
def is_even( i ) :  
    return i%2 == 0
```

*i mapped to 3*

```
is_even(3)
```

# WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:  
**formal parameters** in function def with **values from function call**  
**i** replaced with **3**
- Python **executes expressions in the body** of the function
  - `return 3%2 == 0`

```
def is_even( i ):
```

```
    return i%2 == 0
```

keyword

expression to evaluate  
and return to invoker  
`3%2 == 0` is False

```
is_even(3)
```

Replaced by False

# WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:  
**formal parameters** in function def with **values from function call**  
**i** replaced with **3**

```
def is_even( i ):  
    return i%2 == 0
```

```
is_even(3)  
print(is_even(3))
```

Replaced by False

# BIG IDEA

A function's code  
only runs when you  
call (aka invoke) the function

# YOU TRY IT!

- Write code that satisfies the following specs

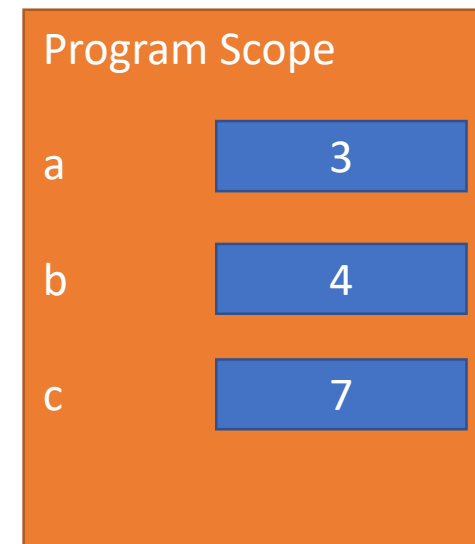
```
def div_by(n, d):  
    """ n and d are ints > 0  
        Returns True if d divides n evenly and False otherwise """
```

Test your code with:

- $n = 10$  and  $d = 3$
- $n = 195$  and  $d = 13$

# ZOOMING OUT (no functions)

```
a = 3  
b = 4  
c = a+b
```



# ZOOMING OUT

This is my “black box”

```
def is_even( i ):  
    print("inside is_even")  
    return i%2 == 0
```

```
a = is_even(3)  
b = is_even(10)  
c = is_even(123456)
```

This is me telling my black box to do something

Program Scope

is\_even

function  
object

# ZOOMING OUT

This is my "black box"

```
def is_even( i ):  
    print("inside is_even")  
    return i%2 == 0
```

```
a = is_even(3)  
b = is_even(10)  
c = is_even(123456)
```

One function call

Program Scope

is\_even

function  
object

a

False

# ZOOMING OUT

This is my "black box"

```
def is_even( i ):  
    print("inside is_even")  
    return i%2 == 0
```

```
a = is_even(3)  
b = is_even(10)  
c = is_even(123456)
```

One function call

Program Scope

is\_even

function  
object

a

False

b

True

# ZOOMING OUT

This is my "black box"

```
def is_even( i ):  
    print("inside is_even")  
    return i%2 == 0
```

```
a = is_even(3)  
b = is_even(10)  
c = is_even(123456)
```

One function call

Program Scope	
is_even	function object
a	False
b	True
c	True

# INSERTING FUNCTIONS IN CODE

- Remember how expressions are replaced with the value?
- The **function call** is **replaced** with the **return value**!

```
print("Numbers between 1 and 10: even or odd")
```

```
for i in range(1,10):  
    if is_even(i):  
        print(i, "even")  
    else:  
        print(i, "odd")
```

# ANOTHER EXAMPLE

- Suppose we want to add all the odd integers between (and including)  $a$  and  $b$
- What is the **input**?
  - Values for  $a$  and  $b$
- What is the **output**?
  - The `sum_of_odds`

```
def sum_odd(a, b):  
    # your code here  
    return sum_of_odds
```

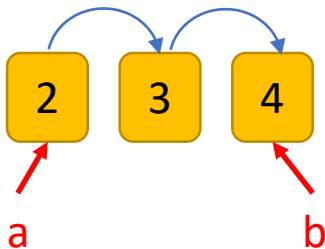
# PAPER FIRST

- Suppose we want to add all the odd integers between (and including)  $a$  and  $b$
- Start with a **simple example on paper**
- Systematically solve the example

```
def sum_odd(a, b):  
    # your code here  
    return sum_of_odds
```

# SIMPLE TEST CASE

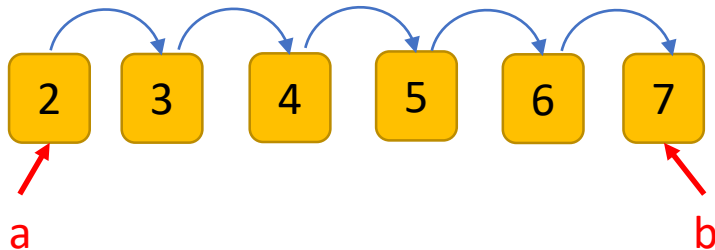
- Suppose we want to add all the odd integers between (and including)  $a$  and  $b$
- Start with a simple example on paper
- $a = 2$  and  $b = 4$ 
  - `sum_of_odds` should be 3



```
def sum_odd(a, b):  
    # your code here  
    return sum_of_odds
```

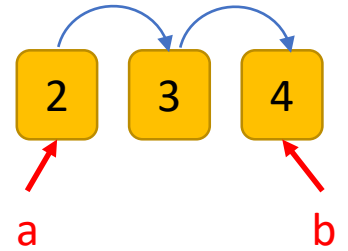
# MORE COMPLEX TEST CASE

- Suppose we want to add all the odd integers between (and including)  $a$  and  $b$
- Start with a simple example on paper
- $a = 2$  and  $b = 7$ 
  - `sum_of_odds` should be 15



```
def sum_odd(a, b):  
    # your code here  
    return sum_of_odds
```

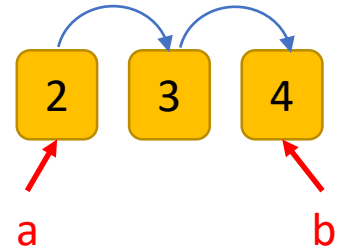
# SOLVE SIMILAR PROBLEM



- Start by looking at each number between (and including) a and b
- A similar problem that is easier that you know how to do?
  - Add **ALL** numbers between (and including) a and b
  - Start with this

```
def sum_odd(a, b):  
    # your code here  
    return sum_of_odds
```

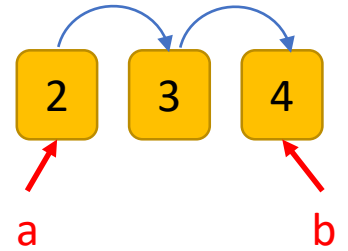
# CHOOSE BIG-PICTURE STRUCTURE



- Add **ALL** numbers between (and including) a and b
  - It's a loop
- while or for?
  - Your choice

```
def sum_odd(a, b):  
    # your code here  
    return sum_of_odds
```

# WRITE the LOOP (for adding all numbers)



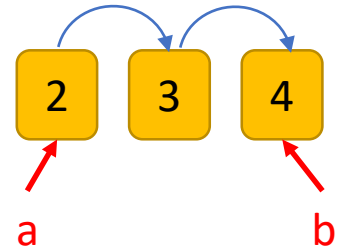
## for LOOP

```
def sum_odd(a, b):  
    for i in range(a, b):  
        # do something  
    return sum_of_odds
```

## while LOOP

```
def sum_odd(a, b):  
    i = a  
    while i <= b:  
        # do something  
        i += 1  
    return sum_of_odds
```

# DO the SUMMING (for adding all numbers)



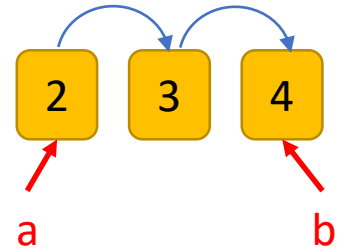
## for LOOP

```
def sum_odd(a, b):  
    for i in range(a, b):  
        sum_of_odds += i  
    return sum_of_odds
```

## while LOOP

```
def sum_odd(a, b):  
    i = a  
    while i <= b:  
        sum_of_odds += i  
        i += 1  
    return sum_of_odds
```

INITIALIZE the SUM  
(for adding all numbers)



for LOOP

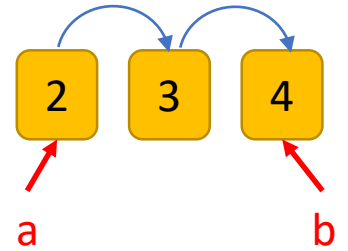
```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b):  
        sum_of_odds += i  
    return sum_of_odds
```

while LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        sum_of_odds += i  
        i += 1  
    return sum_of_odds
```

# TEST!

(for adding all numbers)



## for LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b):  
        sum_of_odds += i  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

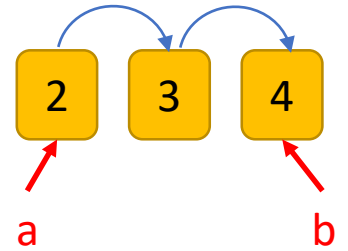
## while LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        sum_of_odds += i  
        i += 1  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

# WEIRD RESULTS...

(for adding all numbers)



## for LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b):  
        sum_of_odds += i  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

5

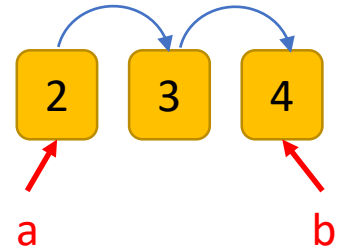
## while LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        sum_of_odds += i  
        i += 1  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

9

# DEBUG! aka ADD PRINT STATEMENTS (for adding all numbers)



## for LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b):  
        sum_of_odds += i  
        print(i, sum_of_odds)  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

5

```
2 2  
3 5
```

## while LOOP

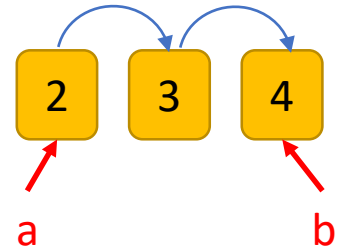
```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        sum_of_odds += i  
        print(i, sum_of_odds)  
        i += 1  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

```
2 2  
3 5  
4 9
```

9

# FIX for LOOP END INDEX (for adding all numbers)



## for LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b+1):  
        sum_of_odds += i  
        print(i, sum_of_odds)  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

9

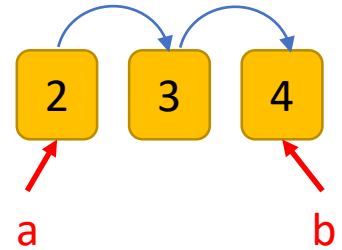
## while LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        sum_of_odds += i  
        print(i, sum_of_odds)  
        i += 1  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

9

# ADD IN THE ODD PART!



## for LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b+1):  
        if i%2 == 1:  
            sum_of_odds += i  
            print(i, sum_of_odds)  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

3

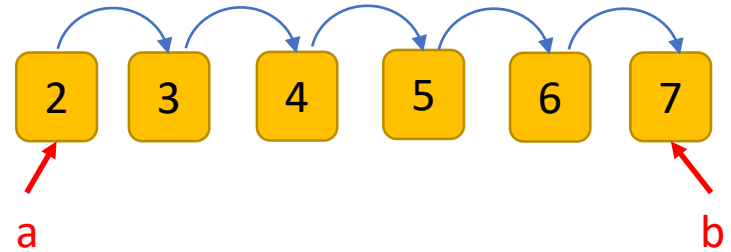
## while LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        if i%2 == 1:  
            sum_of_odds += i  
            print(i, sum_of_odds)  
        i += 1  
    return sum_of_odds
```

```
print(sum_odd(2,4))
```

3

# TRY IT ON ANOTHER EXAMPLE



## for LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b+1):  
        if i%2 == 1:  
            sum_of_odds += i  
    return sum_of_odds
```

```
print(sum_odd(2,7))
```

15

## while LOOP

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    i = a  
    while i <= b:  
        if i%2 == 1:  
            sum_of_odds += i  
        i += 1  
    return sum_of_odds
```

```
print(sum_odd(2,7))
```

15

# PYTHON TUTOR

- Also a decent debugging tool

# YOU TRY IT!

- Write code that satisfies the following specs

```
def is_palindrome(s):  
    """ s is a string  
    Returns True if s is a palindrome and False otherwise  
    """
```

For example:

- If `s = "222"` returns `True`
- If `s = "2222"` returns `True`
- If `s = "abc"` returns `False`

# SUMMARY

- Functions allow us to **suppress detail** from a user
- Functions **capture computation** within a black box
- A programmer writes functions with
  - 0 or more **inputs**
  - Something to **return**
- A function only **runs when it is called**
- The entire function call is **replaced with the return value**
  - Think expressions! And how you replace an entire expression with the value it evaluates to.

# WHAT IF THERE IS NO `return` KEYWORD

```
def is_even( i ):  
    """  
    Input: i, a positive int  
    Does not return anything  
    """
```

`i%2 == 0`

*without a return  
statement*

- Python returns the value **None**, if no **return** given
- Represents the absence of a value
  - If invoked in shell, nothing is printed
- No static semantic error generated

```
def is_even( i ):  
    """  
    Input: i, a positive int  
    Does not return anything  
    """  
    i%2 == 0  
    return None
```

A line Python adds  
implicitly  
(do not add it yourself)

None is a value of type NoneType  
(not a string, not a number, etc)

# YOU TRY IT!

- What is printed if you run this code as a file?

```
def add(x,y):  
    return x+y  
def mult(x,y):  
    print(x*y)
```

```
add(1,2)  
print(add(2,3))  
mult(3,4)  
print(mult(4,5))
```

## return vs. print

- return only has meaning **inside** a function
  - only **one** return executed inside a function
  - code inside function, but after return statement, not executed
  - has a value associated with it, **given to function caller**
- print can be used **outside** functions
  - can execute **many** print statements inside a function
  - code inside function can be executed after a print statement
  - has a value associated with it, **outputted** to the console
  - print expression itself returns `None` value

# YOU TRY IT!

- Fix the code that tries to write this function

```
def is_triangular(n):  
    """ n is an int > 0  
    Returns True if n is triangular, i.e. equals a continued  
    summation of natural numbers (1+2+3+...+k), False otherwise """  
    total = 0  
    for i in range(n):  
        total += i  
        if total == n:  
            print(True)  
    print(False)
```

# FUNCTIONS SUPPORT MODULARITY

- Here is our bisection square root method as a function

```
def bisection_root(x):
```

```
    epsilon = 0.01
```

```
    low = 0
```

```
    high = x
```

```
    ans = (high + low)/2.0
```

Initialize variables

```
    while abs(ans**2 - x) >= epsilon:
```

guess not close enough

```
        if ans**2 < x:
```

```
            low = ans
```

```
        else:
```

```
            high = ans
```

update low or high,  
depends on guess too  
small or too large

```
        ans = (high + low)/2.0
```

new value for guess

```
    # print(ans, 'is close to the root of', x)
```

```
    return ans
```

return result

iterate

# FUNCTIONS SUPPORT MODULARITY

- Call it with different values

```
print(bisection_root(4))  
print(bisection_root(123))
```

- Write a function that calls this one!

# YOU TRY IT!

- Write a function that satisfies the following specs

```
def count_nums_with_sqrt_close_to (n, epsilon):  
    """ n is an int > 2  
        epsilon is a positive number < 1  
        Returns how many integers have a square root within epsilon of n """
```

Use `bisection_root` we already wrote to get an approximation for the sqrt of an integer.

For example: `print(count_nums_with_sqrt_close_to(10, 0.1))`  
prints 4 because all these integers have a sqrt within 0.1

- sqrt of 99 is 9.949699401855469
- sqrt of 100 is 9.999847412109375
- sqrt of 101 is 10.049758911132812
- sqrt of 102 is 10.099456787109375

# ZOOMING OUT

This is my “black box”

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b+1):  
        if i%2 == 1:  
            sum_of_odds += i  
    return sum_of_odds
```

```
low = 2  
high = 7  
my_sum = sum_odd(low, high)
```

One function call

## Program Scope

sum\_odd

function  
object

low

2

high

7

my\_sum

# ZOOMING OUT

*a=2 b=7*

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b+1):  
        if i%2 == 1:  
            sum_of_odds += i  
    return sum_of_odds
```

```
low = 2  
high = 7  
my_sum = sum_odd(low, high)
```

## Program Scope

sum\_odd

function  
object

low

2

high

7

my\_sum

# ZOOMING OUT

This is my "black box"

```
def sum_odd(a, b):  
    sum_of_odds = 0  
    for i in range(a, b+1):  
        if i%2 == 1:  
            sum_of_odds += i  
    return sum_of_odds
```

```
low = 2  
high = 7  
my_sum = sum_odd(low, high)
```

15

## Program Scope

sum_odd	function object
low	2
high	7
my_sum	15

FUNCTION SCOPE

# UNDERSTANDING FUNCTION CALLS

- How does Python execute a function call?
- How does Python know what value is associated with a variable name?
- It **creates a new environment** with every function call!
  - Like a **mini program** that it needs to complete
  - The mini program runs with **assigning its parameters** to some inputs
  - It does the work (aka the **body** of the function)
  - It **returns** a value
  - The **environment disappears** after it returns the value

# ENVIRONMENTS

- **Global** environment
  - Where user interacts with Python interpreter
  - Where the program starts out
- Invoking a function creates a **new environment** (frame/scope)

# VARIABLE SCOPE

- **Formal parameters** get bound to the value of **input parameters**
- **Scope** is a mapping of names to objects
  - Defines context in which body is evaluated
  - Values of variables given by bindings of names
- Expressions in body of function evaluated wrt this new scope

```
def f( x ) :  
    x = x + 1  
    print( 'in f(x): x =', x )  
    return x
```

*formal  
parameter*

*Function  
definition*

```
y = 3
```

```
z = f( y )
```

*actual  
parameter*

*Main program code*  
\* initializes a variable x  
\* makes a function call f(x)  
\* assigns return of function to variable z  
  
*Can be any legal value*

# VARIABLE SCOPE

after evaluating def

This is my “black box”

```
def f( x ) :  
    x = x + 1  
    print( 'in f(x): x =', x )  
    return x
```

→  
x = 3  
z = f( x )

Global scope

f

function  
object

# VARIABLE SCOPE

after exec 1<sup>st</sup> assignment

This is my “black box”

```
def f( x ) :  
    x = x + 1  
    print( 'in f(x): x =', x )  
    return x
```

→ `x = 3`  
`z = f( x )`

Global scope

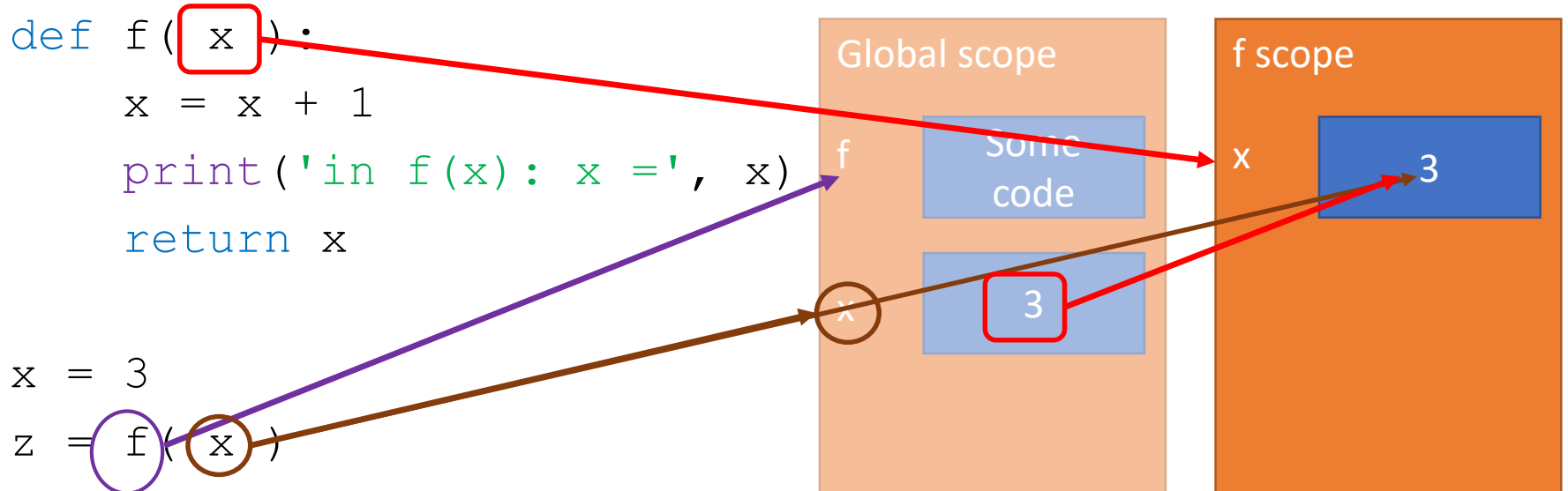
f

Some  
code

x

3

# VARIABLE SCOPE after f invoked



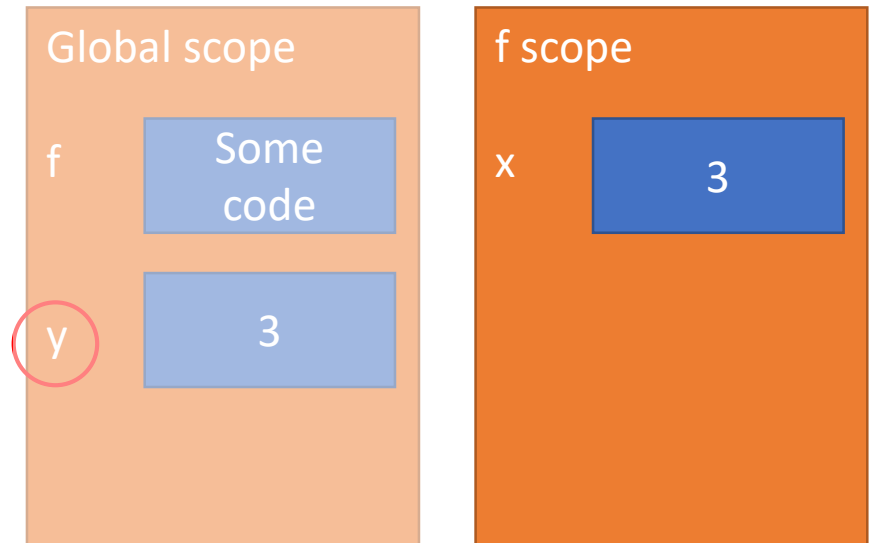
# VARIABLE SCOPE

## after f invoked

Name of variable irrelevant, only value important. You can also pass in the value directly.

```
def f( x ) :  
    x = x + 1  
    print( 'in f(x): x =', x )  
    return x
```

$y = 3$   
→  $z = f(y)$



# VARIABLE SCOPE

eval body of f in f's scope

in f(x): x = 4 printed out

```
def f( x ):
```

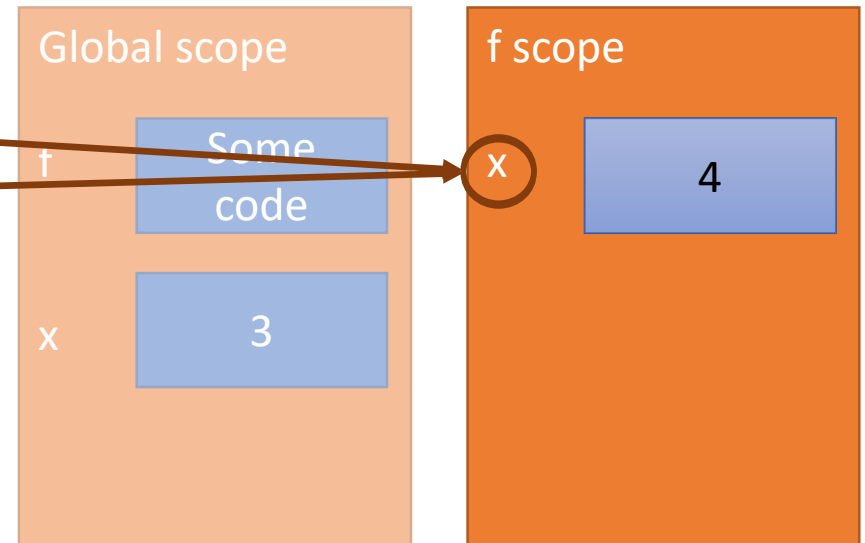
```
    x = x + 1
```

```
    print('in f(x): x =', x)
```

```
    return x
```

```
x = 3
```

```
z = f( x )
```



# VARIABLE SCOPE

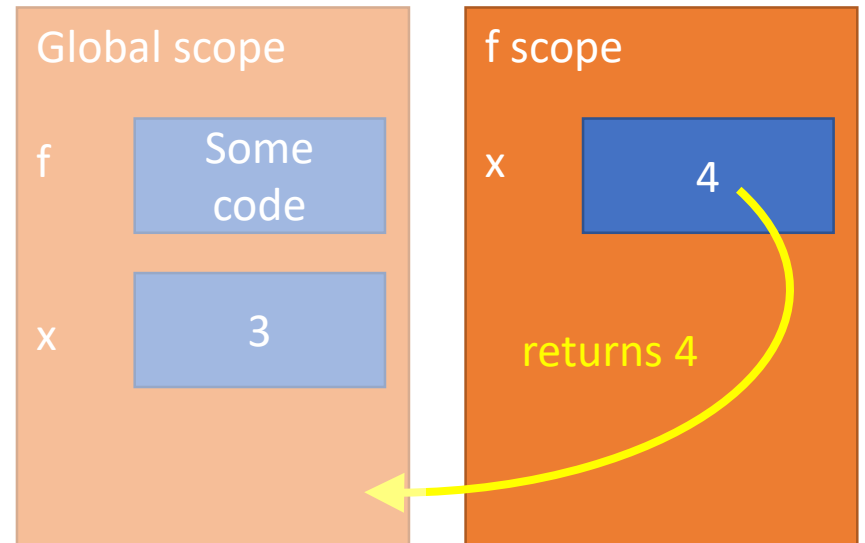
## during return

```
def f( x ):  
    x = x + 1  
    print('in f(x): x =', x)  
    return x
```

```
x = 3
```

```
z = f( x )
```

Function call  
replaced with  
return value



# VARIABLE SCOPE

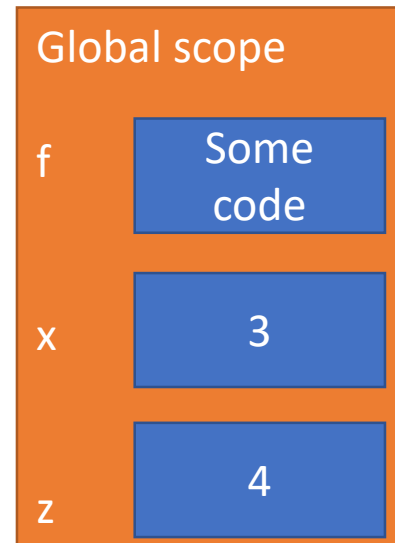
after exec 2<sup>nd</sup> assignment

*If I now ask for value of x in  
Python interpreter, it will print 3*

```
def f( x ) :  
    x = x + 1  
    print( 'in f(x): x =', x )  
    return x
```

```
x = 3
```

```
z = f( x )
```



# ANOTHER SCOPE EXAMPLE

- Inside a function, **can access** a variable defined outside
- Inside a function, **cannot modify** a variable defined outside (can by using **global variables**, but frowned upon)
- Use the Python Tutor to step through these!

```
def f(y):  
    x = 1  
    x += 1  
    print(x)  
  
x = 5  
f(x)  
print(x)
```

*x is re-defined in scope of f*

*different x objects*

2  
5

```
def g(y):  
    print(x)  
    print(x + 1)  
  
x = 5  
g(x)  
print(x)
```

*x from outside g*

*x inside g is picked up from scope that called function g*

5  
6  
5

```
def h(y):  
    x += 1  
  
x = 5  
h(x)  
print(x)
```

*UnboundLocalError: local variable 'x' referenced before assignment*

Error

# FUNCTIONS as ARGUMENTS

# HIGHER ORDER PROCEDURES

- Objects in Python have a type
  - int, float, str, Boolean, NoneType, function
- Objects can appear in RHS of assignment statement
  - Bind a name to an object
- Objects
  - Can be **used as an argument** to a procedure
  - Can be **returned as a value** from a procedure
- Functions are also **first class objects!**
- Treat functions just like the other types
  - Functions can be arguments to another function
  - Functions can be returned by another function

# OBJECTS IN A PROGRAM

```
def is_even(i):  
    return i%2 == 0
```

```
r = 2
```

```
pi = 22/7
```

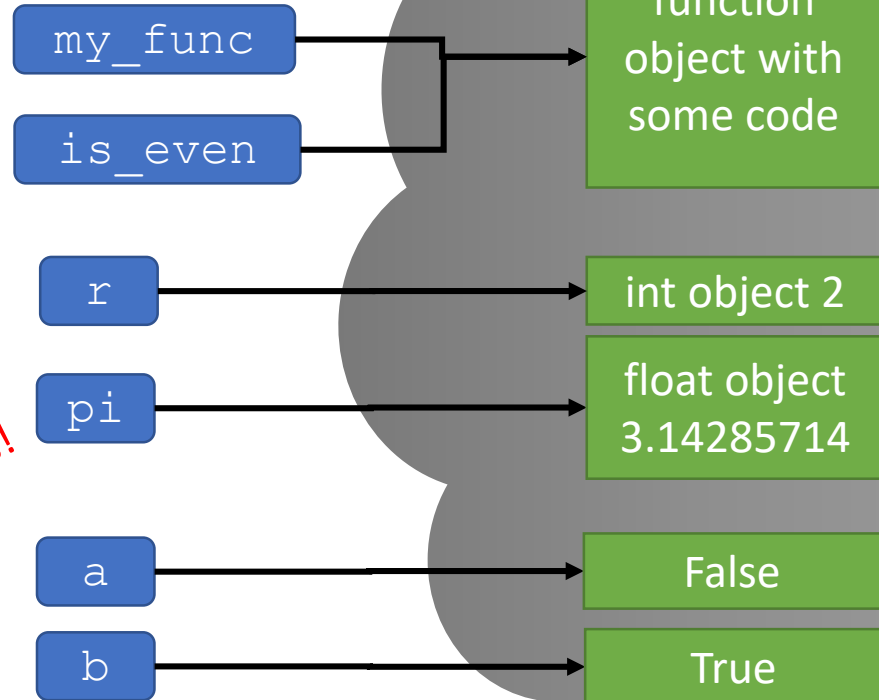
```
my_func = is_even
```

```
a = is_even(3)
```

```
b = my_func(4)
```

*NOT a function  
call, just names!*

*Two function calls*



# FUNCTION AS A PARAMETER

```
def calc(op, x, y):  
    return op(x, y)
```

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

```
def div(a, b):  
    if b != 0:  
        return a / b  
    print("Denominator was 0.")
```

```
print(calc(add, 2, 3))
```

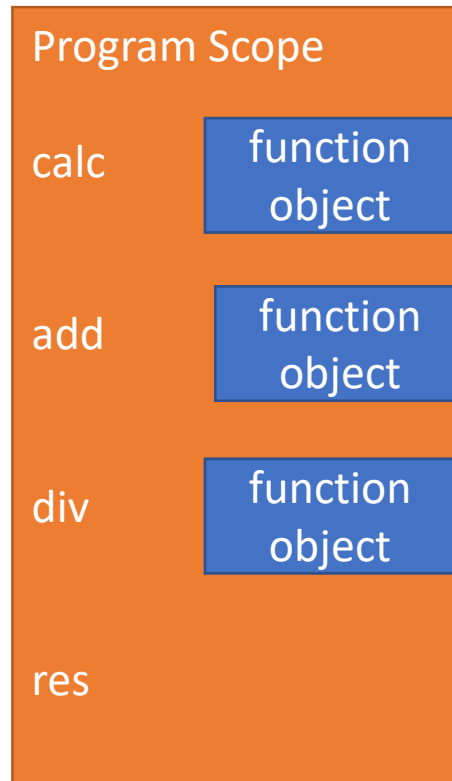
# STEP THROUGH THE CODE

```
def calc(op, x, y):  
    return op(x, y)
```

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

```
def div(a, b):  
    if b != 0:  
        return a / b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



# CREATE calc SCOPE

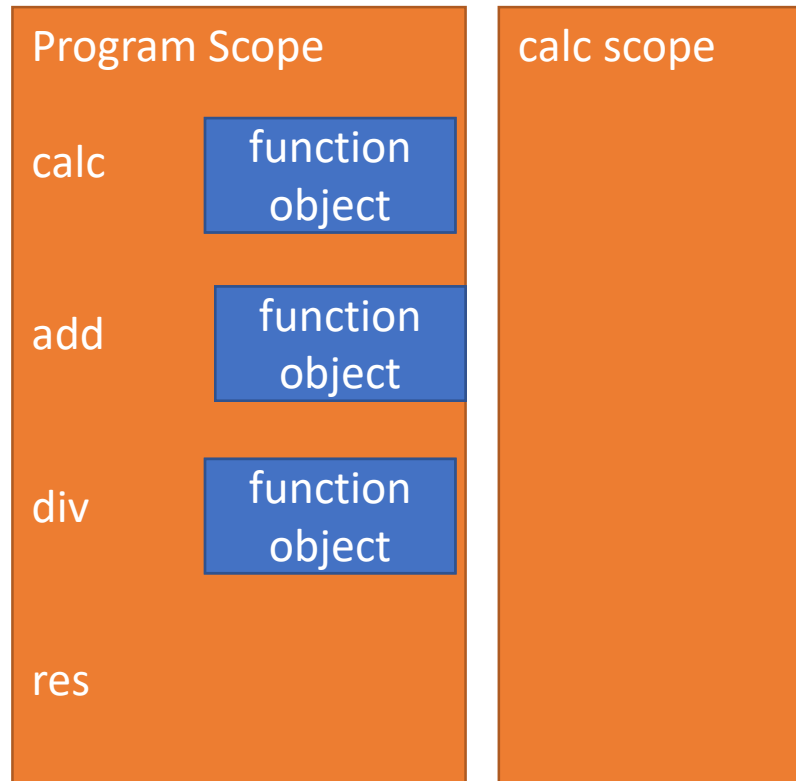
```
def calc(op, x, y):  
    return op(x,y)
```

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

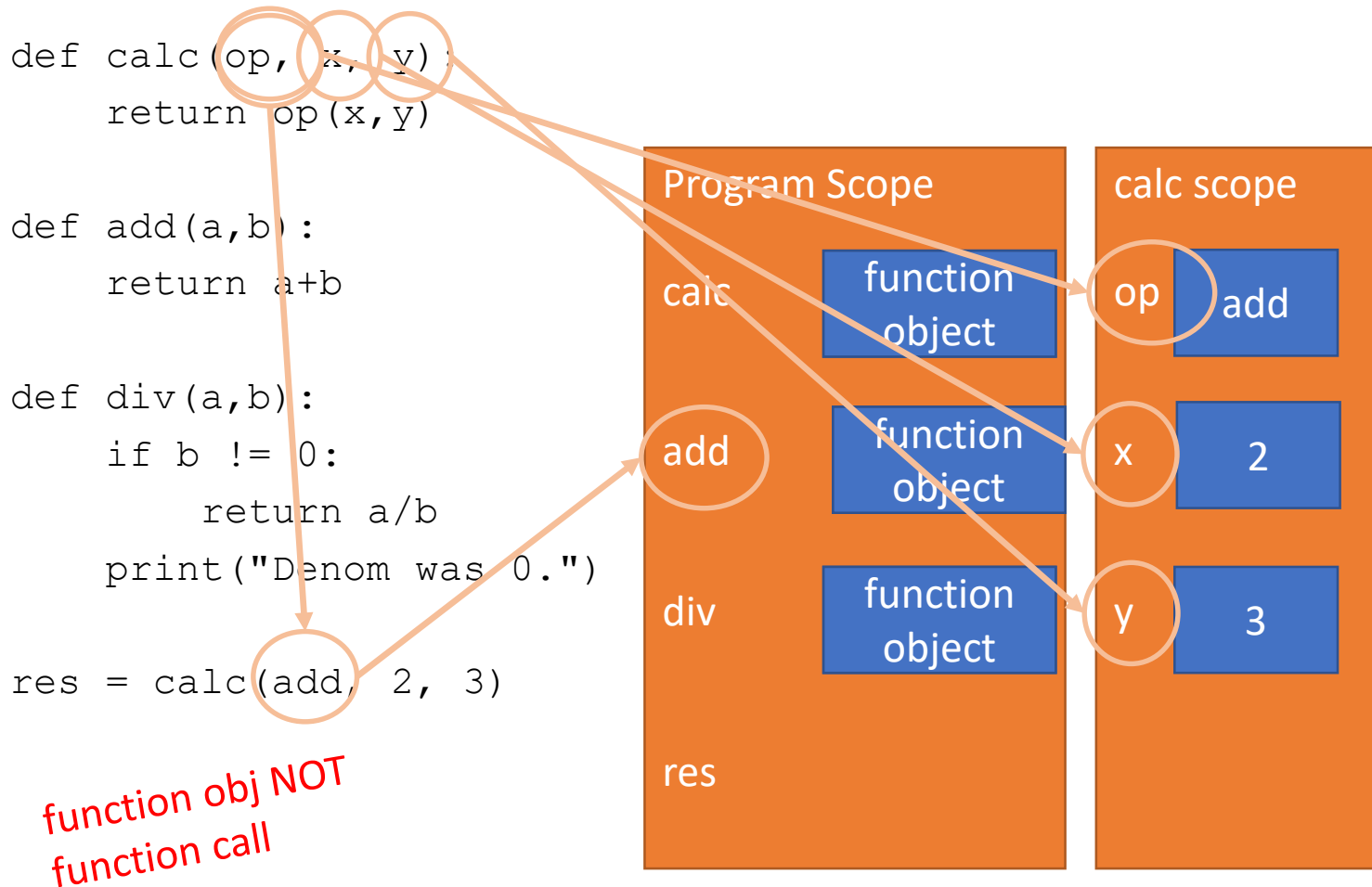
```
def div(a,b):  
    if b != 0:  
        return a/b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```

Function call



# MATCH FORMAL PARAMS in calc



# FIRST (and only) LINE IN calc

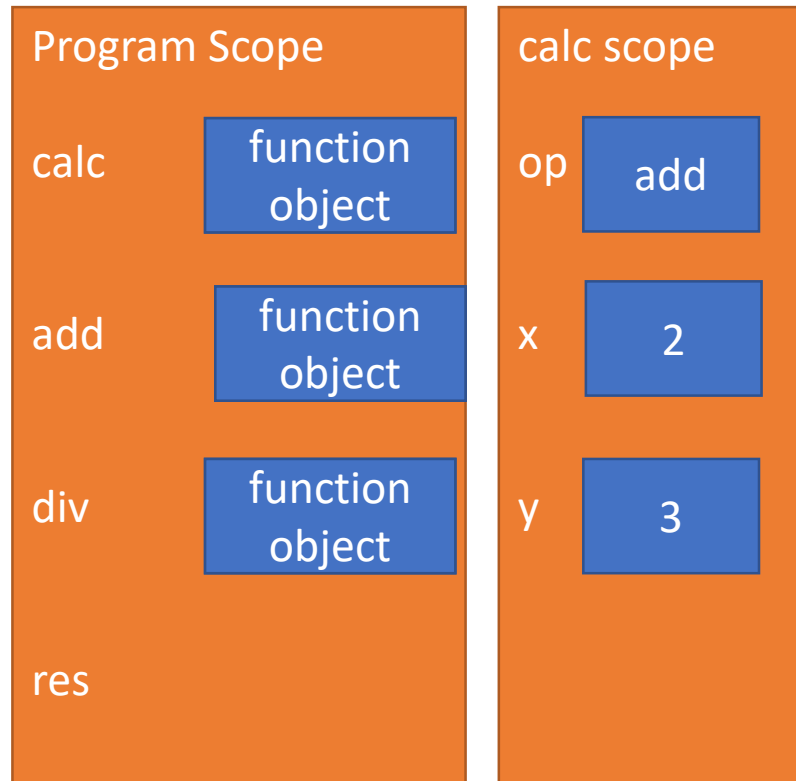
```
def calc(op, x, y):  
    return op(x, y)
```

*return add(2,3)  
Just replace each param with its value*

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

```
def div(a,b):  
    if b != 0:  
        return a/b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



# CREATE SCOPE OF add

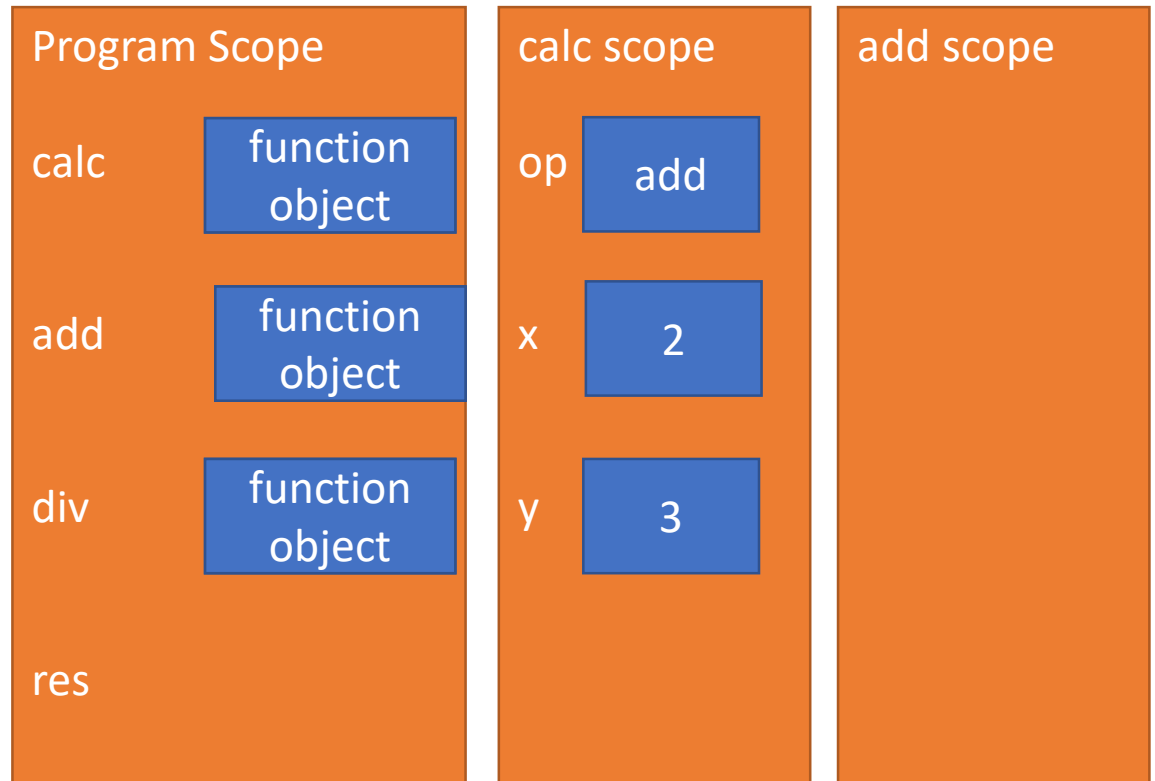
```
def calc(op, x, y):  
    return op(x, y)
```

Function call in calc scope: add(2,3)

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

```
def div(a,b):  
    if b != 0:  
        return a/b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



# MATCH FORMAL PARAMS IN add

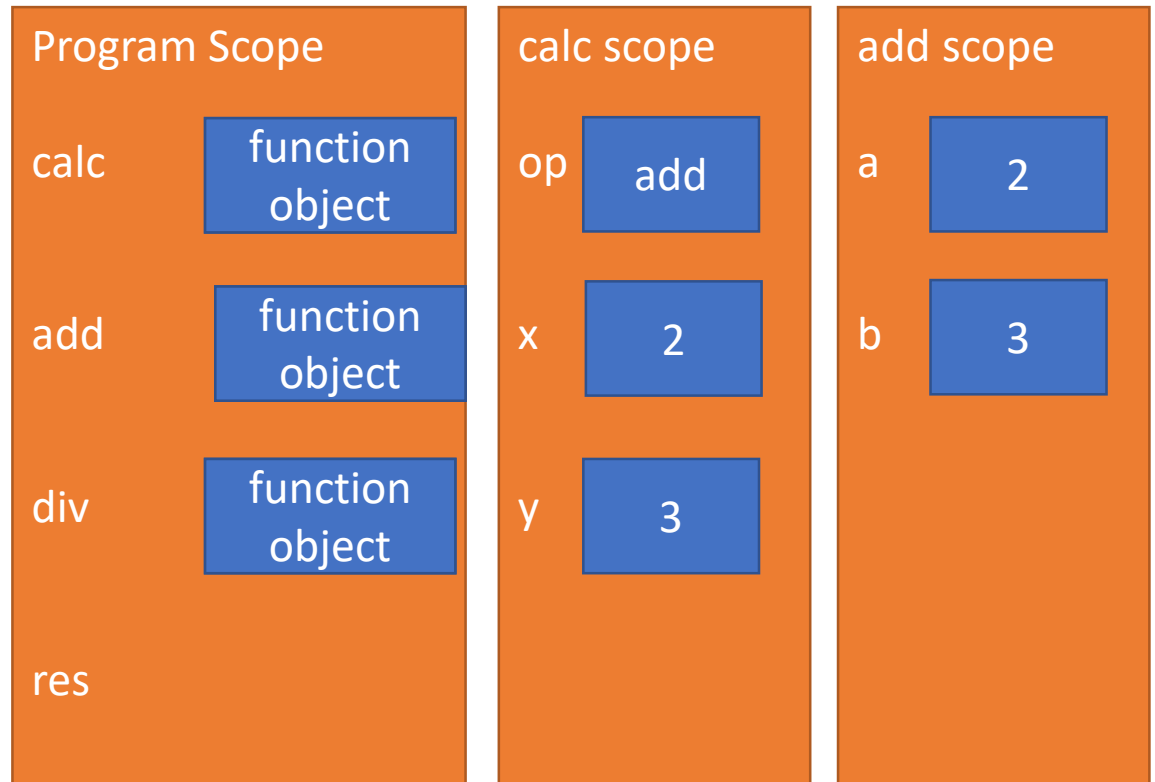
```
def calc(op, x, y):  
    return op(x, y)
```

Function call in calc scope: add with formal params a=2 and b=3

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

```
def div(a, b):  
    if b != 0:  
        return a/b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



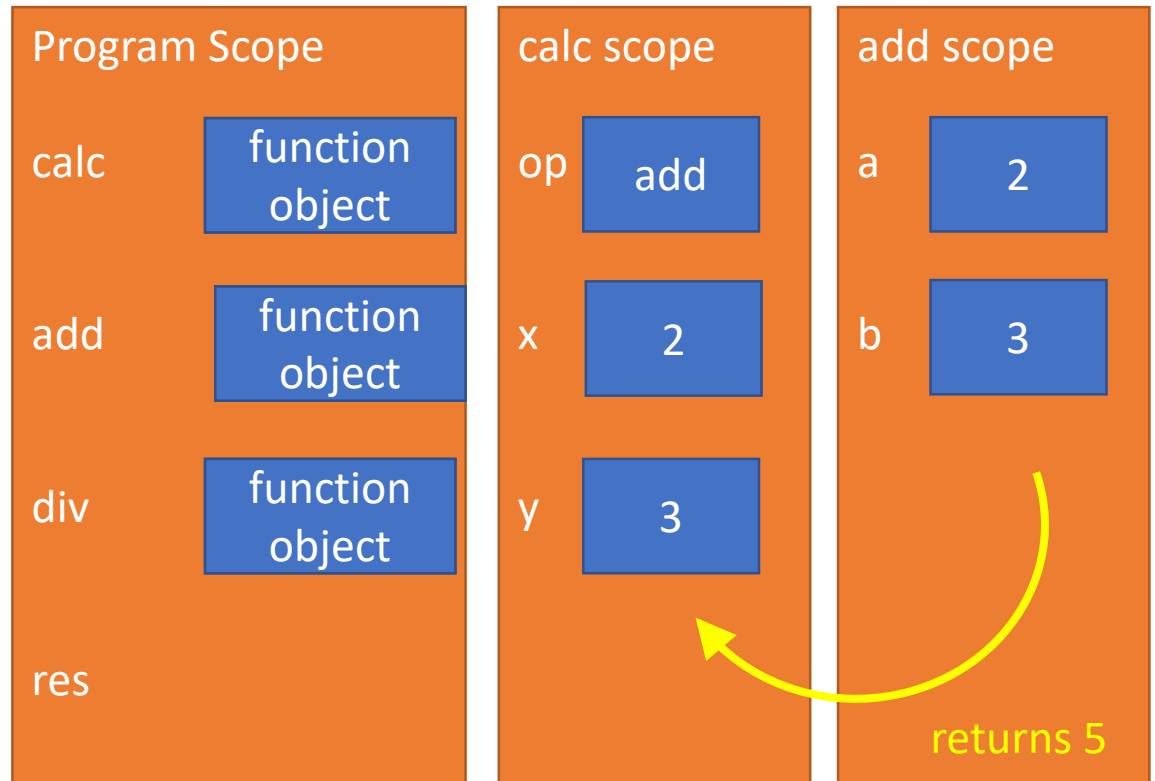
# EXECUTE LINE OF add

```
def calc(op, x, y):  
    return op(x,y)
```

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

```
def div(a,b):  
    if b != 0:  
        return a/b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



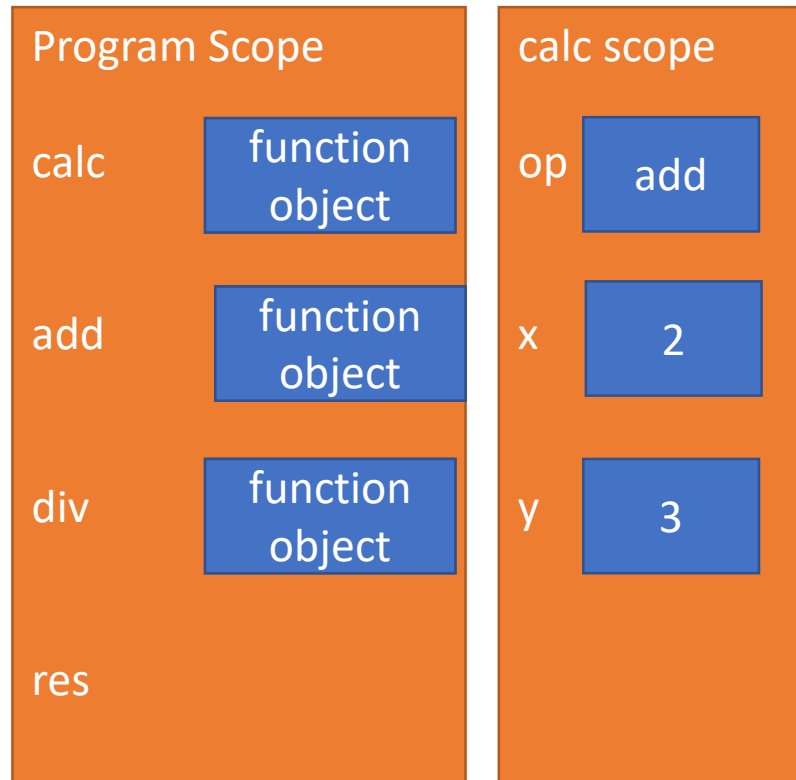
# REPLACE FUNC CALL WITH RETURN

```
def calc(op, x, y):  
    return op(x, y) 5
```

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

```
def div(a, b):  
    if b != 0:  
        return a / b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



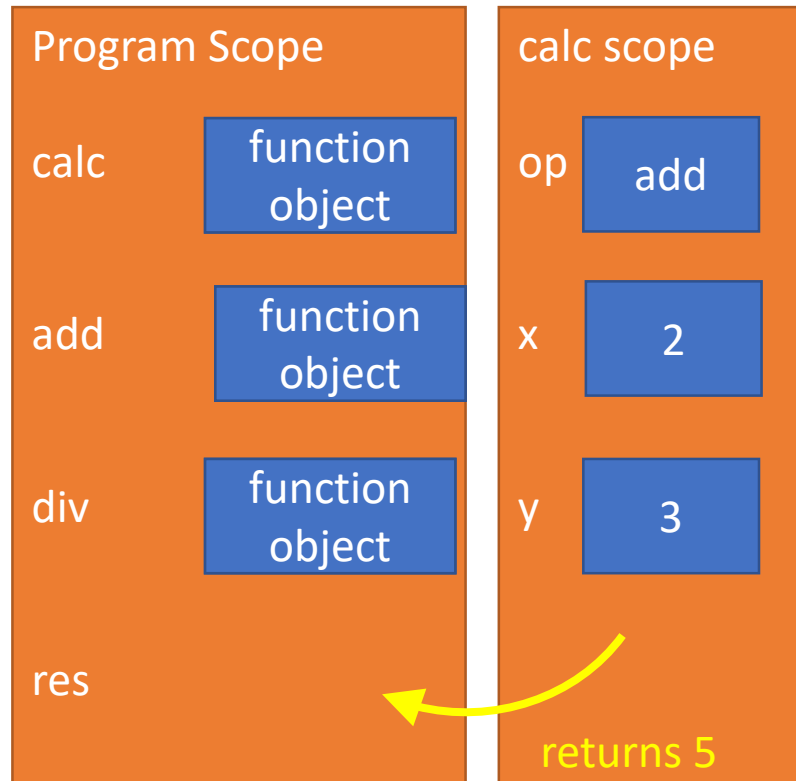
# EXECUTE LINE OF calc

```
def calc(op, x, y):  
    return op(x, y)
```

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

```
def div(a, b):  
    if b != 0:  
        return a / b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```



# REPLACE FUNC CALL WITH RETURN

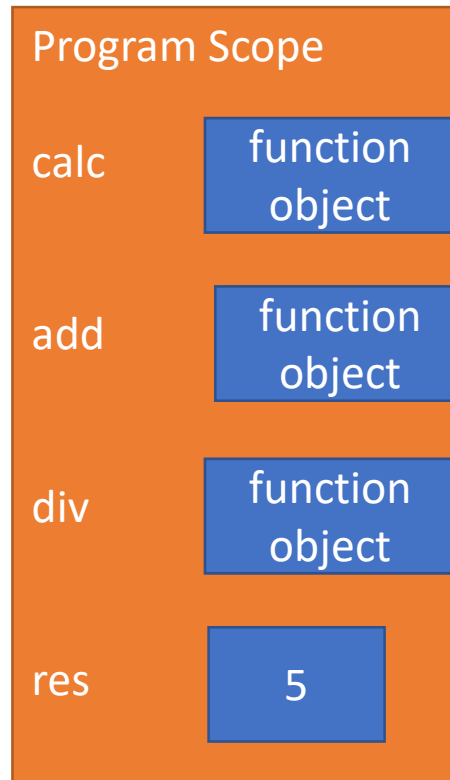
```
def calc(op, x, y):  
    return op(x, y)
```

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

```
def div(a, b):  
    if b != 0:  
        return a / b  
    print("Denom was 0.")
```

```
res = calc(add, 2, 3)
```

5



# YOU TRY IT!

- Do a similar trace with the function call

```
def calc(op, x, y):  
    return op(x,y)  
  
def div(a,b):  
    if b != 0:  
        return a/b  
    print("Denom was 0.")  
  
res = calc(div,2,0)
```

What is the value of res and what gets printed?

# ANOTHER EXAMPLE: FUNCTIONS AS PARAMS

```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```

call func\_a, takes no parameters  
call func\_b, takes one parameter, an int  
call func\_c, takes two parameters,  
another function and an int

# FUNCTIONS AS PARAMETERS

No bindings (no parameters)

```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

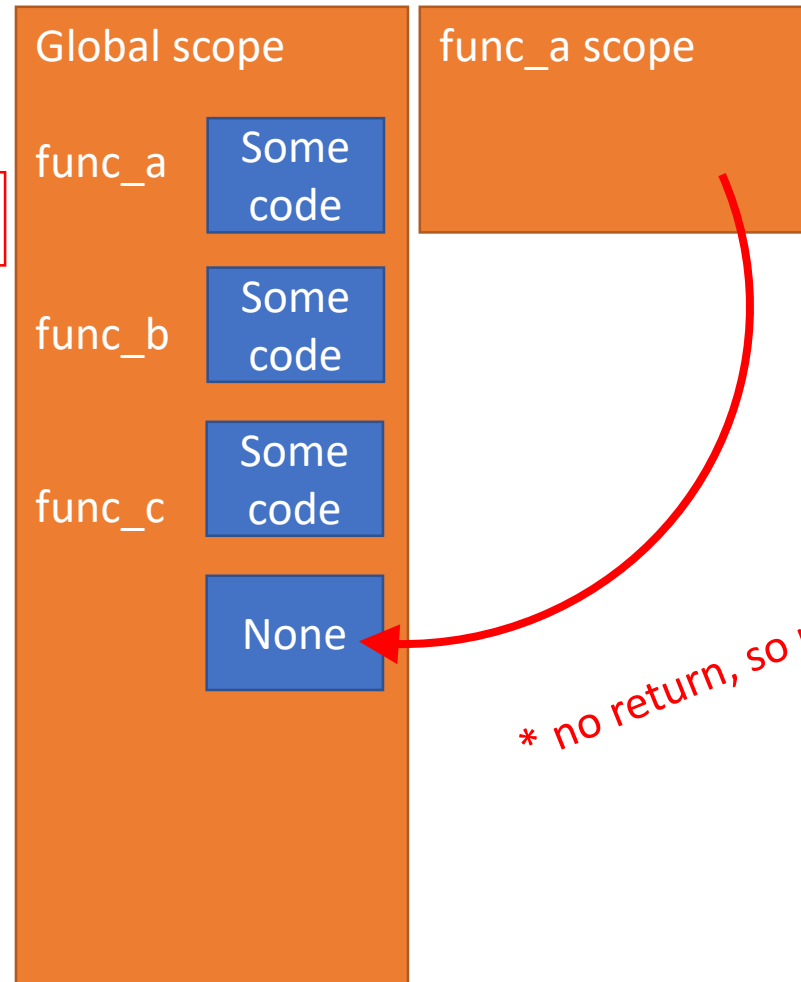
func\_c

Some  
code

func\_a scope

# FUNCTIONS AS PARAMETERS

```
def func_a():  
    print('inside func_a')  
def func_b(y):  
    print('inside func_b')  
    return y  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
→ print(5 + func_b(2))  
print(func_c(func_b, 3))
```



body prints 'inside func\_a' on console

\* no return, so returns None

# FUNCTIONS AS PARAMETERS

```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```

Global scope

func\_a

Some  
code

func\_b

Some  
code

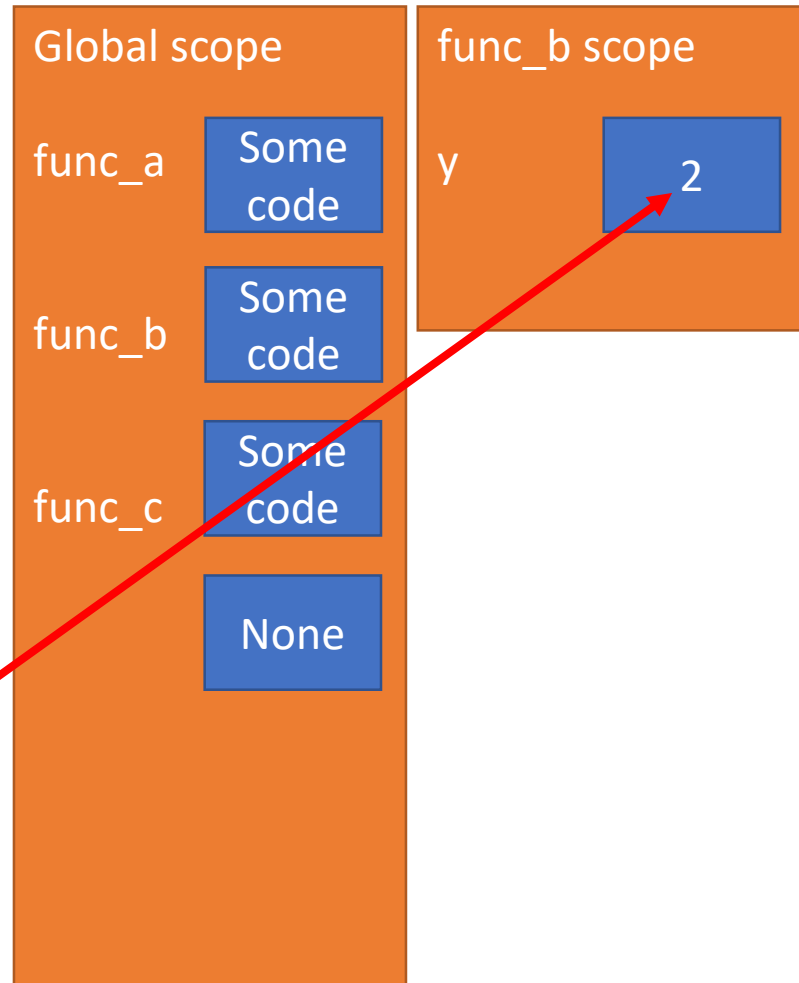
func\_c

Some  
code

*print displays None on console*

# FUNCTIONS AS PARAMETERS

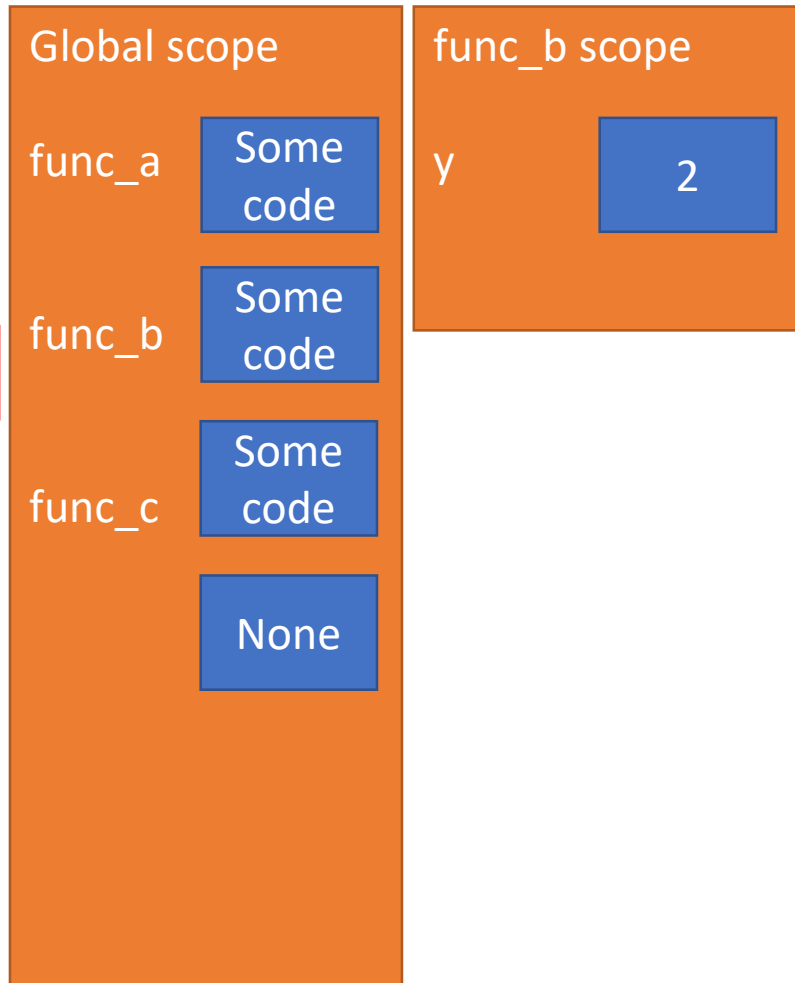
```
def func_a():  
    print('inside func_a')  
def func_b(y):  
    print('inside func_b')  
    return y  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```



# FUNCTIONS AS PARAMETERS

body prints 'inside func\_b'  
on console

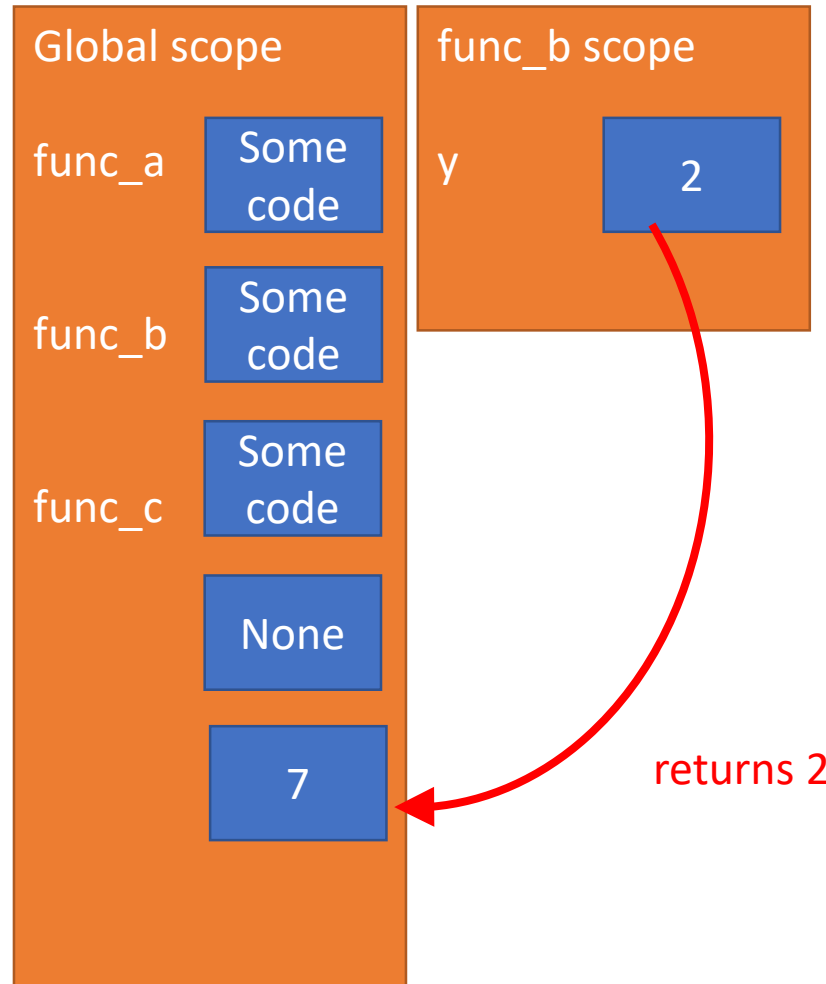
```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```



# FUNCTIONS AS PARAMETERS

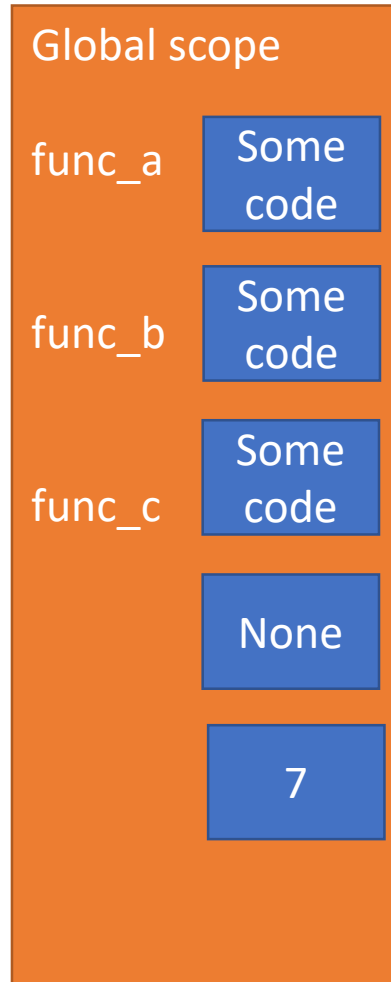
value of 2 is returned and added to 5

```
def func_a():  
    print('inside func_a')  
def func_b(y):  
    print('inside func_b')  
    return y  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```



# FUNCTIONS AS PARAMETERS

```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```

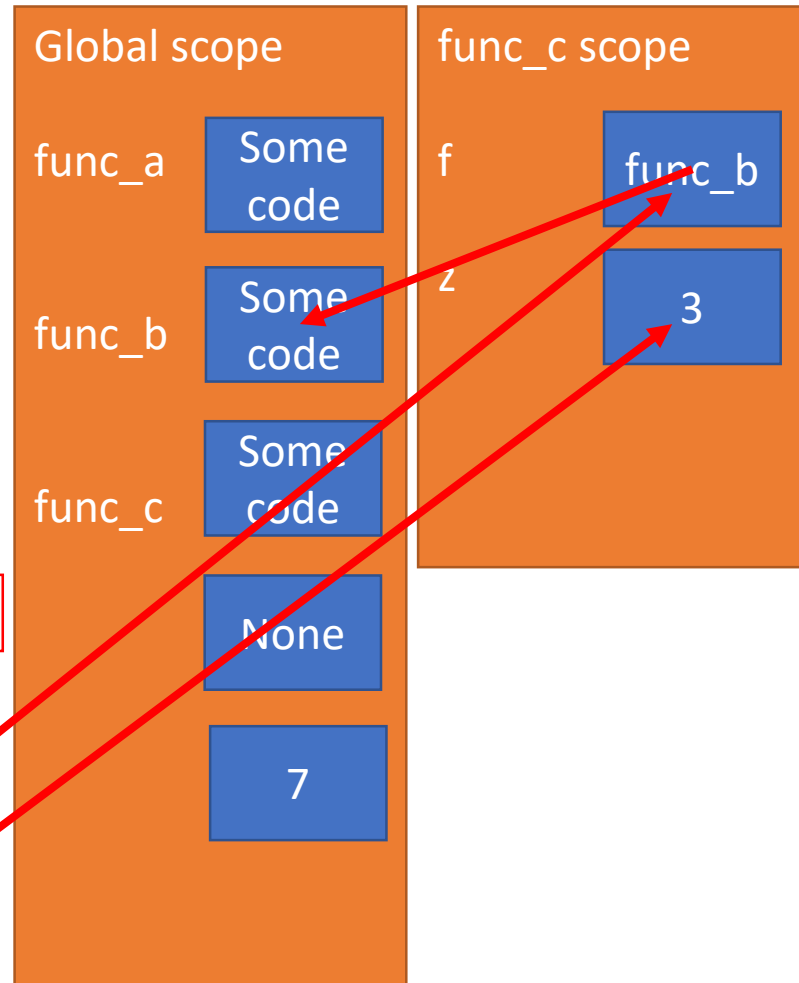


*print displays 7 on console*

# FUNCTIONS AS PARAMETERS

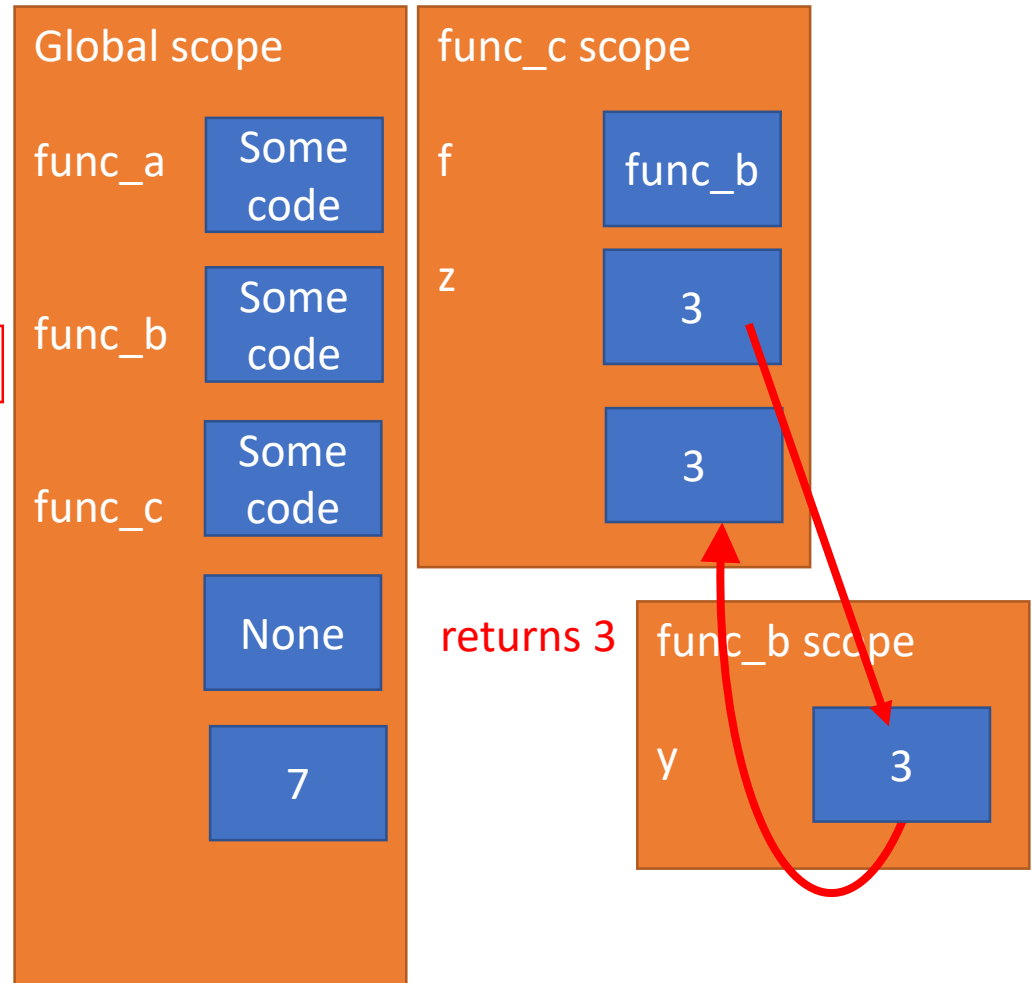

*body of func\_c prints  
'inside func\_c' on console*

```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```



# FUNCTIONS AS PARAMETERS

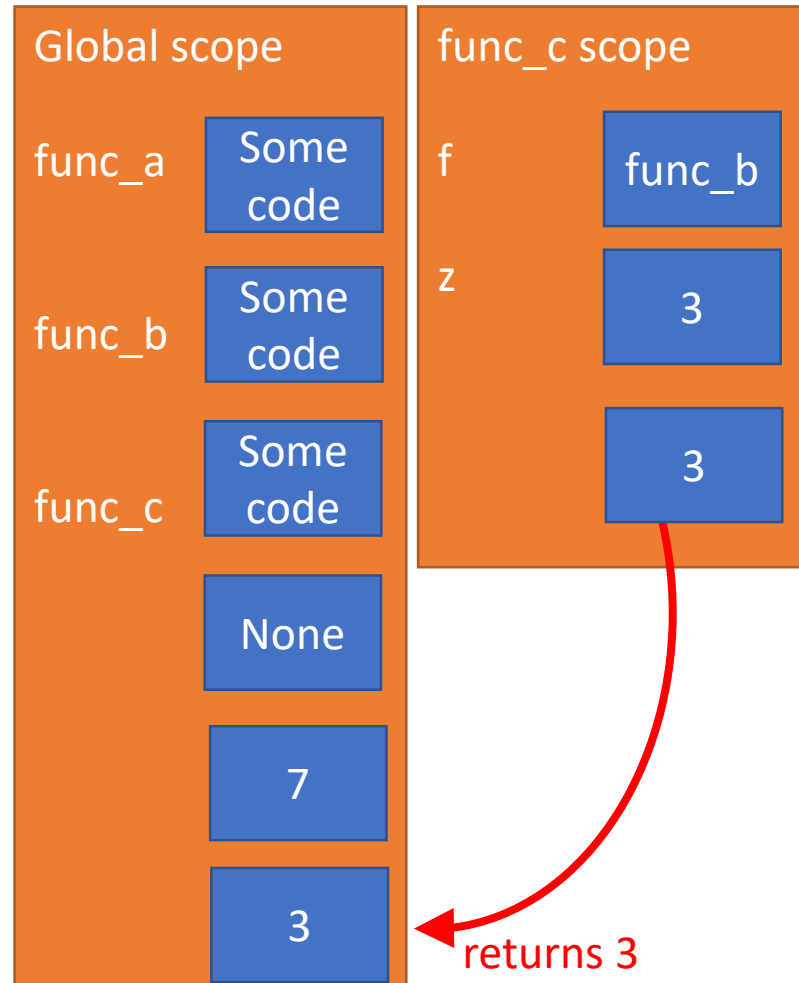
```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z) 3  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```



# FUNCTIONS AS PARAMETERS

*print displays 3 on console*

```
def func_a():  
    print('inside func_a')  
  
def func_b(y):  
    print('inside func_b')  
    return y  
  
def func_c(f, z):  
    print('inside func_c')  
    return f(z)  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_b, 3))
```



# YOU TRY IT!

- Write a function that meets these specs.

```
def apply(criteria,n):  
    """  
    * criteria is a func that takes in a number and returns a bool  
    * n is an int  
    Returns how many ints from 0 to n (inclusive) match  
    the criteria (i.e. return True when run with criteria)  
    """
```

# SUMMARY

- Functions are first class objects
  - They have a **type**
  - They can be **assigned as a value** bound to a name
  - They can be used as an **argument** to another procedure
  - They can be **returned** as a value from another procedure
- Have to be careful about environments
  - Main program runs in the global environment
  - Function calls each get a new temporary environment
- This enables the creation of concise, easily read code

# ANONYMOUS FUNCTIONS

- Sometimes don't want to name functions, especially simple ones. This function is a good example:

```
def is_even(x):  
    return x%2==0
```

- Can use an **anonymous** procedure by using `lambda`

```
lambda x: x%2 == 0
```

parameter

Body of lambda

Note no return keyword

- `lambda` creates a procedure/function object, but simply does not bind a name to it

# ANONYMOUS FUNCTIONS

- Function call with a named function:

```
apply( is_even , 10 )
```

- Function call with an anonymous function as parameter:

```
apply( lambda x: x%2 == 0 , 10 )
```

- `lambda` function is **one-time use**. It can't be reused because it has no name!

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))  
  
print(do_twice(3, lambda x: x**2))
```

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))
```

```
print(do_twice(3, lambda x: x**2))
```

Global environment

do\_twice

function object

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))  
  
print(do_twice(3, lambda x: x**2))
```

## Global environment

do\_twice      function object

## do\_twice environment

n      3  
fn      lambda x: x\*\*2

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))  
  
print(do_twice(3, lambda x: x**2))
```

Global environment

do\_twice      function object

do\_twice environment

n      3  
fn    lambda x: x\*\*2

lambda x: x\*\*2  
environment

x      ???

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))
```

```
print(do_twice(3, lambda x: x**2))
```

Global environment

do\_twice      function object

do\_twice environment

n      3  
fn      lambda x: x\*\*2

lambda x: x\*\*2  
environment

x      ???

lambda x: x\*\*2  
environment

x      3

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))
```

9

```
print(do_twice(3, lambda x: x**2))
```

Global environment

do\_twice      function object

do\_twice environment

n      3  
fn      lambda x: x\*\*2

lambda x: x\*\*2  
environment

x

9

lambda x: x\*\*2  
environment

x

3

Returns 9

# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))
```

81

```
print(do_twice(3, lambda x: x**2))
```

Global environment

do\_twice      function object

do\_twice environment

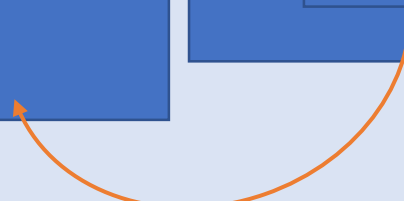
n    3  
fn   lambda x: x\*\*2

lambda x: x\*\*2  
environment

x

9

Returns 81



# YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):  
    return fn(fn(n))
```

81

```
print(do_twice(3, lambda x: x**2))
```

Global environment

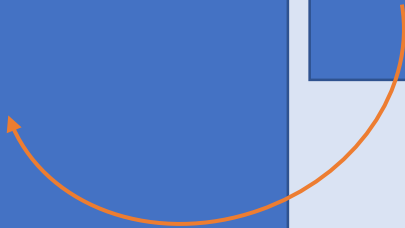
do\_twice      function object

PRINTS 81

do\_twice environment

n      3  
fn      lambda x: x\*\*2

Returns 81



# OBJECTS

- Python supports many different kinds of data

```
1234          3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- Each is an **object**, and every object has:
  - An internal **data representation** (primitive or composite)
  - A set of procedures for **interaction** with the object
- An object is an **instance** of a **type**
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a `str`

# OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- Can **create new objects** of some type
- Can **manipulate objects**
- Can **destroy objects**
  - Explicitly using `del` or just “forget” about them
  - Python system will reclaim destroyed or inaccessible objects – called “garbage collection”

# WHAT ARE OBJECTS?

- Objects are **a data abstraction** that captures...

## (1) An **internal representation**

- Through data attributes

## (2) An **interface** for interacting with object

- Through methods  
(aka procedures/functions)
- Defines behaviors but  
hides implementation

# EXAMPLE:

[1,2,3,4] has type list

- (1) How are lists **represented internally**?

Does not matter for so much for us as users (private representation)



or



*follow pointer to  
the next index*

- (2) How to **interface with, and manipulate**, lists?

- `L[i]`, `L[i:j]`, `+`
- `len()`, `min()`, `max()`, `del(L[i])`
- `L.append()`, `L.extend()`, `L.count()`, `L.index()`,  
`L.insert()`, `L.pop()`, `L.remove()`, `L.reverse()`,  
`L.sort()`

- Internal representation should be private

- Correct behavior may be compromised if you manipulate internal representation directly

# REAL-LIFE EXAMPLES

- **Elevator**: a box that can change floors
  - Represent using length, width, height, max\_capacity, current\_floor
  - Move its location to a different floor, add people, remove people
- **Employee**: a person who works for a company
  - Represent using name, birth\_date, salary
  - Can change name or salary
- **Queue at a store**: first customer to arrive is the first one helped
  - Represent customers as a list of str names
  - Append names to the end and remove names from the beginning
- **Stack of pancakes**: first pancake made is the last one eaten
  - Represent stack as a list of str
  - Append pancake to the end and remove from the end

# ADVANTAGES OF OOP

- **Bundle data into packages** together with procedures that work on them through well-defined interfaces
- **Divide-and-conquer** development
  - Implement and test behavior of each class separately
  - Increased modularity reduces complexity
- **Classes** make it easy to **reuse** code
  - Many Python modules define new classes
  - Each class has a separate environment (no collision on function names)
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

# CREATING AND USING YOUR OWN TYPES WITH CLASSES

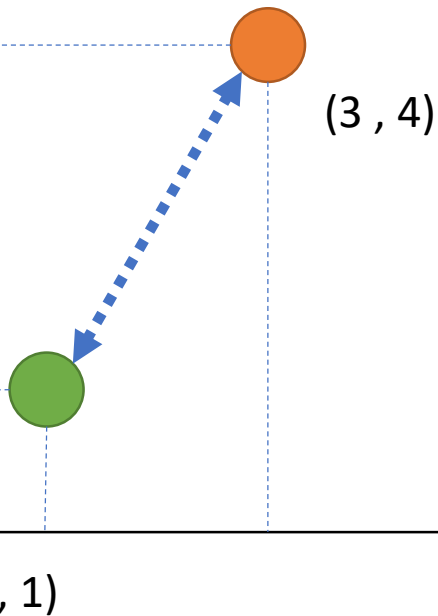
- Make a distinction between **creating a class** and **using an instance** of the class
- **Creating** the class involves
  - Defining the class name
  - Defining class attributes
  - *for example, someone wrote code to implement a list class*
- **Using** the class involves
  - Creating new **instances** of the class
  - Doing operations on the instances
  - *for example, `L=[1, 2]` and `len(L)`*

# A PARALLEL with FUNCTIONS

- **Defining a class** is like defining a function
  - With functions, we tell Python this procedure exists
  - With classes, we tell Python about a **blueprint for this new data type**
    - Its data attributes
    - Its procedural attributes
- **Creating instances of objects** is like calling the function
  - With functions we make calls with different actual parameters
  - With classes, we **create new object tinstances in memory of this type**
    - L1 = [1,2,3]  
L2 = [5,6,7]

# COORDINATE TYPE DESIGN DECISIONS

Can create **instances** of a  
Coordinate object



- Decide what **data** elements constitute an object
  - In a 2D plane
  - A coordinate is defined by an **x and y value**
- Decide **what to do** with coordinates
  - Tell us how far away the coordinate is on the x or y axes
  - Measure the **distance** between two coordinates, Pythagoras

# DEFINE YOUR OWN TYPES

- Use the `class` keyword to define a new type

*class definition*  
*name/type*  
*class parent*

```
class Coordinate(object):  
    #define attributes here
```

- Similar to `def`, indent code to indicate which statements are part of the **class definition**
- The word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (will see in future lects)

# WHAT ARE ATTRIBUTES?

- Data and procedures that “**belong**” to the class
- **Data attributes**
  - Think of data as other objects/variables that make up the class
  - *for example, a coordinate is made up of two numbers*
- **Methods** (procedural attributes)
  - Think of methods as functions that only work with this class
  - How to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

# DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- First have to define **how to create an instance** of class
- Use a **special method called `__init__`** to initialize some data attributes or perform initialization operations

```
class Coordinate(object):
```

```
    def __init__(self, xval, yval):  
        self.x = xval  
        self.y = yval
```

special method to  
create an instance  
— is double  
underscore

two data attributes  
make up your type

parameter to  
refer to an  
instance of the  
class without  
having created  
one yet

what data initializes a  
Coordinate object

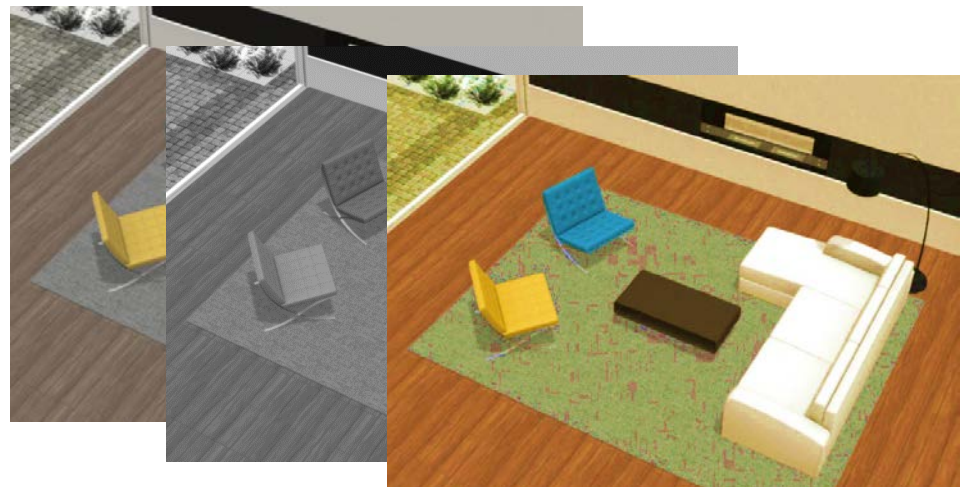
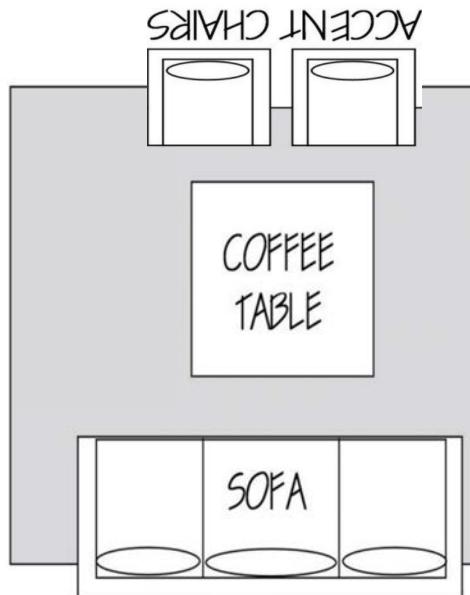
- `self` allows you to create **variables that belong to this object**
- Without `self`, you are just creating regular variables!

# WHAT is self?

## ROOM EXAMPLE

*self is the blueprint's way  
of accessing attributes  
(data and methods)*

- Think of the class definition as a **blueprint** with placeholders for actual items
  - self has a chair
  - self has a coffee table
  - self has a sofa
- Now when you create **ONE** instance (name it living\_room), self becomes this actual object
  - living\_room has a blue chair
  - living\_room has a black table
  - living\_room has a white sofa
- Can make **many instances** using the same blueprint



Recall the `__init__` method in the class def:

```
def __init__(self, xval, yval):  
    self.x = xval  
    self.y = yval
```

## ACTUALLY CREATING AN INSTANCE OF A CLASS

- Don't provide argument for `self`, Python does this automatically

```
c = Coordinate(3, 4)
```

```
origin = Coordinate(0, 0)
```

create a new object  
of type  
Coordinate and  
pass in 3 and 4 to  
the `__init__`

- Data attributes of an instance are called **instance variables**
  - Data attributes were defined with `self.XXX` and they are accessible with dot notation for the lifetime of the object
  - All instances have these data attributes, but with different values!

```
print(c.x)
```

```
print(origin.x)
```

use the dot  
notation to access  
an attribute of  
instance `c`

# VISUALIZING INSTANCES

- Suppose we create an instance of a coordinate

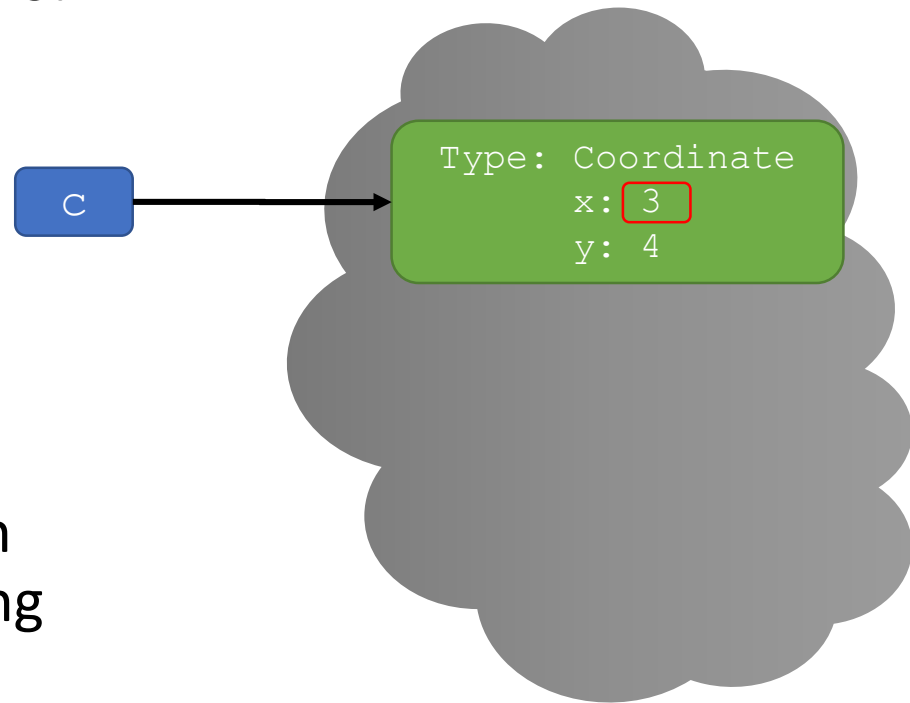
```
c = Coordinate(3, 4)
```

- Think of this as creating a structure in memory

- Then evaluating

```
c.x
```

looks up the structure to which `c` points, then finds the binding for `x` in that structure



# VISUALIZING INSTANCES: in memory

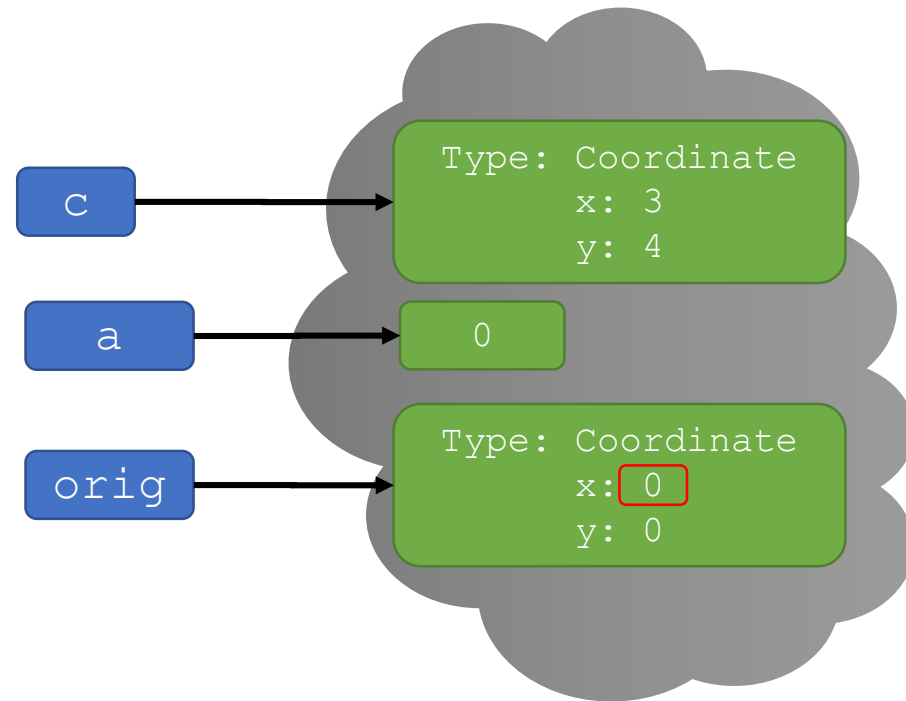
- Make another instance using a variable

```
a = 0
```

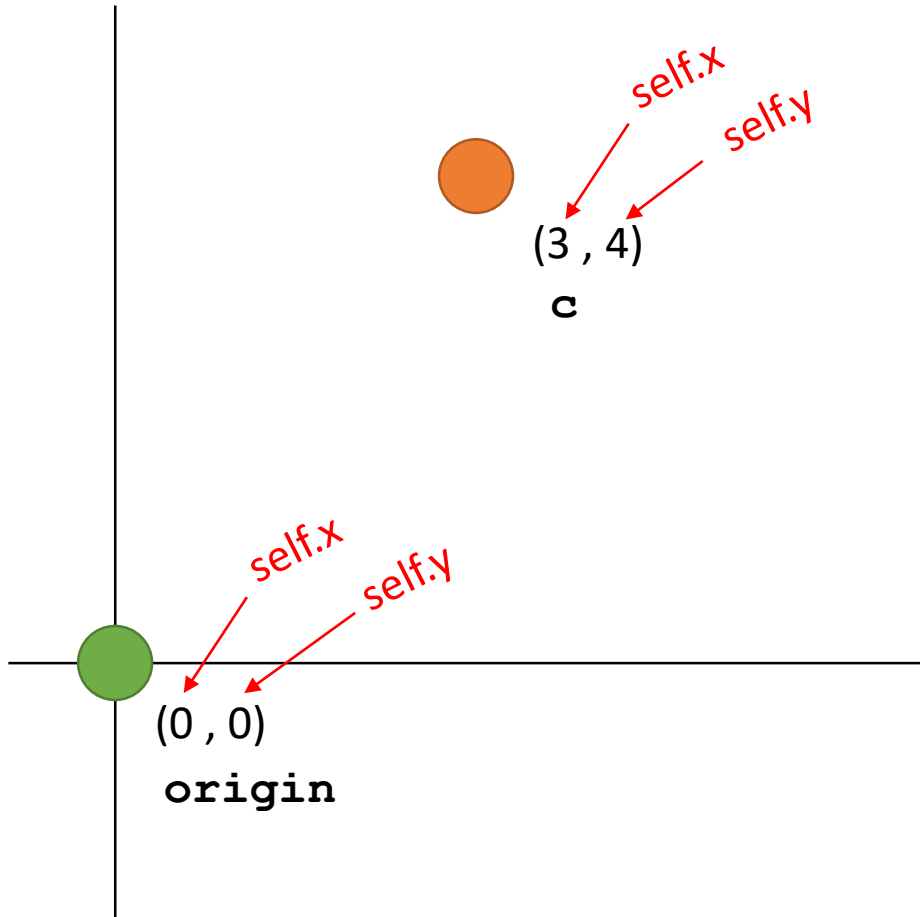
```
orig = Coordinate(a, a)
```

```
orig.x
```

- All these are just objects in memory!
- We just access attributes of these objects



# VISUALIZING INSTANCES: draw it



The template for a  
Coordinate type

```
class Coordinate(object):  
    def __init__(self, xval, yval):  
        self.x = xval  
        self.y = yval
```

```
c = Coordinate(3,4)  
origin = Coordinate(0,0)  
print(c.x)  
print(origin.x)
```

Code to make actual  
tangible Coordinate  
objects (aka instances)

# WHAT IS A METHOD?

- Procedural attribute
  - Think of it like a **function that works only with this class**
- Python always passes the object as the first argument
  - Convention is to use **self** as the name of the first argument of all methods

# DEFINE A METHOD FOR THE Coordinate CLASS

```
class Coordinate(object):
```

```
    def __init__(self, xval, yval):
```

```
        self.x = xval
```

```
        self.y = yval
```

```
    def distance(self, other):
```

```
        x_diff_sq = (self.x - other.x) ** 2
```

```
        y_diff_sq = (self.y - other.y) ** 2
```

```
        return (x_diff_sq + y_diff_sq) ** 0.5
```

use it to refer  
to the obj I call  
this method on

another parameter to method

dot notation to access x of self

dot notation to access x of other

- Other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

# HOW TO CALL A METHOD?

- The “.” **operator** is used to access any attribute
  - A data attribute of an object (we saw `c.x`)
  - A method of an object

- Dot notation

`<object_variable>.<method>(<parameters>)`

*Object to call  
method on, becomes  
self in the class def*

*Name of  
method*

*Not including self.  
self is the obj  
before the dot!*

- Familiar?

```
my_list.append(4)
```

```
my_list.sort()
```

Recall the definition of distance method:

```
def distance(self, other):  
    x_diff_sq = (self.x-other.x)**2  
    y_diff_sq = (self.y-other.y)**2  
    return (x_diff_sq + y_diff_sq)**0.5
```

## HOW TO USE A METHOD

Using the class:

```
c = Coordinate(3,4)  
orig = Coordinate(0,0)  
print(c.distance(orig))
```

object to call  
method on

name of  
method

parameters not including self  
(self is implied to be c)

- Notice that `self` becomes the object you call the method on (the thing before the dot!)

# VISUALIZING INVOCATION

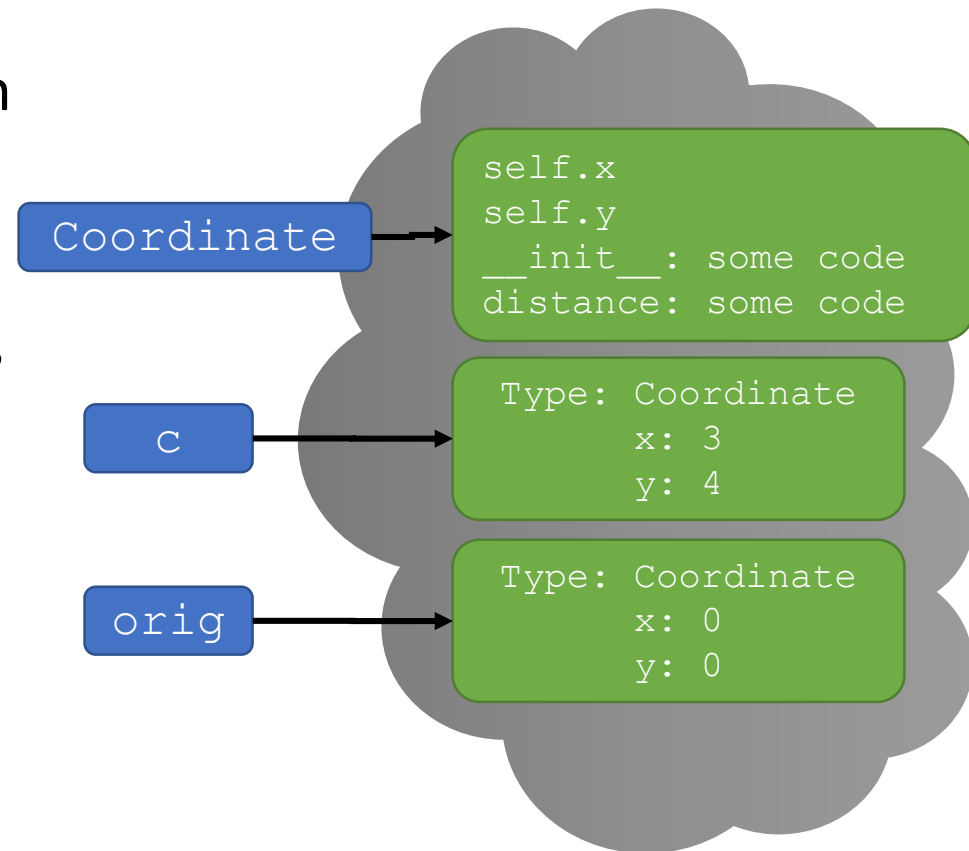
- Coordinate class is an object in memory

- From the class definition

- Create two Coordinate objects

```
c = Coordinate(3,4)
```

```
orig = Coordinate(0,0)
```

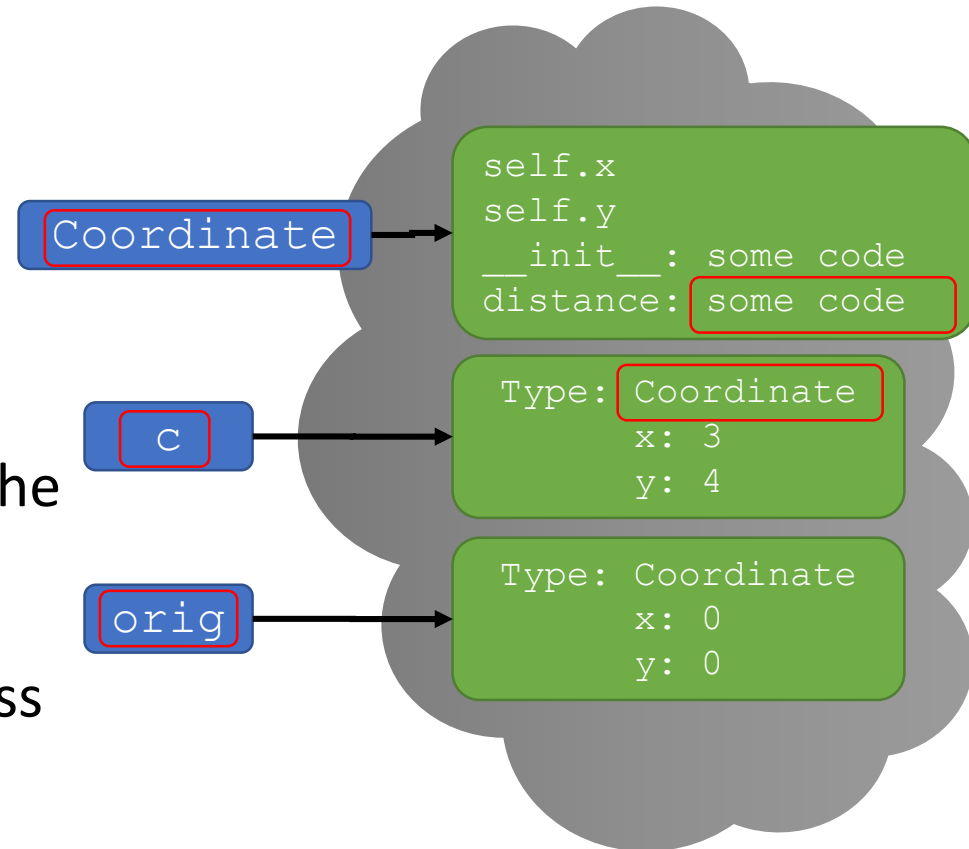


# VISUALIZING INVOCATION

- Evaluate the method call

`c.distance(orig)`

- 1) The object is before the dot
- 2) Looks up the type of `c`
- 3) The method to call is after the dot.
- 4) Finds the binding for `distance` in that object class
- 5) Invokes that method with  
    `c` as `self` and  
    `orig` as `other`



# HOW TO USE A METHOD

## ■ Conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
c.distance(zero)
```

object to  
call  
method  
on, this is  
self in the  
class def

name of  
method

parameters not  
including self  
(self is  
implied to be c)

## ■ Equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
Coordinate.distance(c, zero)
```

name of  
class (NOT  
an object of  
type  
Coordinate)

name of  
method

parameters, including an  
object to call the method  
on, representing self

# THE POWER OF OOP

- **Bundle together objects** that share
  - Common attributes and
  - Procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

# IMPLEMENTING THE CLASS

vs

# USING THE CLASS

- Write code from two different perspectives

**Implementing** a new object type with a class

- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

Class abstractly captures **common** properties and behaviors

**Using** the new object type in code

- Create **instances** of the object type
- Do **operations** with them

Instances have **specific values** for attributes

# RECALL THE COORDINATE CLASS

- Class **definition** tells Python the **blueprint** for a type Coordinate

```
class Coordinate(object):  
    """ A coordinate made up of an x and y value """  
    def __init__(self, x, y):  
        """ Sets the x and y values """  
        self.x = x  
        self.y = y  
    def distance(self, other):  
        """ Returns euclidean dist between two Coord obj """  
        x_diff_sq = (self.x-other.x)**2  
        y_diff_sq = (self.y-other.y)**2  
        return (x_diff_sq + y_diff_sq)**0.5
```

# ADDING METHODS TO THE COORDINATE CLASS

- Methods are functions that **only work with objects of this type**

```
class Coordinate(object):
    """ A coordinate made up of an x and y value """
    def __init__(self, x, y):
        """ Sets the x and y values """
        self.x = x
        self.y = y
    def distance(self, other):
        """ Returns euclidean dist between two Coord obj """
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def to_origin(self):
        """ always sets self.x and self.y to 0,0 """
        self.x = 0
        self.y = 0
```

# MAKING COORDINATE INSTANCES

- Creating **instances** makes actual Coordinate **objects in memory**
- The objects can be **manipulated**
  - Use **dot notation** to call methods and access data attributes

```
c = Coordinate(3, 4)
origin = Coordinate(0, 0)
```

x data attr has a value of 3  
y data attr has a value of 4

```
print(f"c's x is {c.x} and origin's x is {origin.x}")
print(c.distance(origin))
```

```
c.to_origin()
print(c.x, c.y)
```

Method didn't return anything,  
just set c's x and y to 0.

# CLASS DEFINITION OF AN OBJECT TYPE

vs

# INSTANCE OF A CLASS

- Class name is the **type**  

```
class Coordinate(object)
```
- Class is defined generically
  - Use `self` to refer to some instance while defining the class  

```
(self.x - self.y)**2
```
  - `self` is a parameter to methods in class definition
- Class defines data and methods **common across all instances**

- Instance is **one specific object**  

```
coord = Coordinate(1,2)
```
- Data attribute values vary between instances  

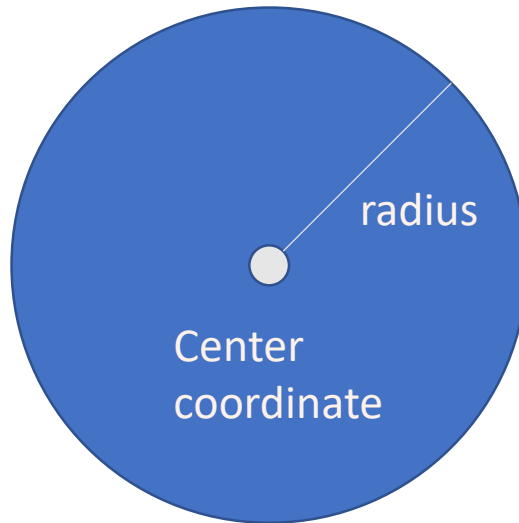
```
c1 = Coordinate(1,2)
```

```
c2 = Coordinate(3,4)
```

  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects
- Instance has the **structure of the class**

# USING CLASSES TO BUILD OTHER CLASSES

- Example: use Coordinates to build Circles
- Our implementation will use **2 data attributes**
  - Coordinate object representing the center
  - int object representing the radius



# CIRCLE CLASS: DEFINITION and INSTANCES

```
class Circle(object):  
    def __init__(self, center, radius):  
        self.center = center  
        self.radius = radius
```

Data  
attributes,  
do not need  
to have the  
same names  
as params

Will be a Coordinate object  
Will be an int

```
center = Coordinate(2, 2)  
my_circle = Circle(center, 2)
```

# YOU TRY IT!

- Add code to the init method to check that the type of center is a Coordinate obj and the type of radius is an int. If either are not these types, raise a ValueError.

```
def __init__(self, center, radius):  
    self.center = center  
    self.radius = radius
```

# CIRCLE CLASS: DEFINITION and INSTANCES

```
class Circle(object):  
    def __init__(self, center, radius):  
        self.center = center  
        self.radius = radius  
    def is_inside(self, point):  
        """ Returns True if point is in self, False otherwise """  
        return point.distance(self.center) < self.radius
```

*self is a Circle object*

*point is a Coordinate object*

*Coordinate object*

*Method called on a Coordinate obj*

*Coordinate object*

```
center = Coordinate(2, 2)  
my_circle = Circle(center, 2)  
p = Coordinate(1,1)  
print(my_circle.is_inside(p))
```

*Method that only works with obj of type Circle*

*Circle object*

*Coordinate object*

# YOU TRY IT!

- Are these two methods in the Circle class functionally equivalent?

```
class Circle(object):  
    def __init__(self, center, radius):  
        self.center = center  
        self.radius = radius  
  
    def is_inside1(self, point):  
        return point.distance(self.center) < self.radius  
  
    def is_inside2(self, point):  
        return self.center.distance(point) < self.radius
```

# EXAMPLE:

## FRACTIONS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
  - Numerator
  - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - Add, subtract
  - Invert the fraction
- Let's write it together!

# NEED TO CREATE INSTANCES

```
class SimpleFraction(object):  
    def __init__(self, n, d):  
        self.num = n  
        self.denom = d
```

# MULTIPLY FRACTIONS

```
class SimpleFraction(object):
```

```
    def __init__(self, n, d):
```

```
        self.num = n
```

```
        self.denom = d
```

```
    def times(self, oth):
```

```
        top = self.num*oth.num
```

```
        bottom = self.denom*oth.denom
```

```
    return top/bottom
```





SimpleFraction objects so they each have  
\* num  
\* denom

Access num or denom to do the math

# ADD FRACTIONS

```
class SimpleFraction(object):  
    def __init__(self, n, d):  
        self.num = n  
        self.denom = d  
  
    .....  
  
    def plus(self, oth):  
        top = self.num*oth.denom + self.denom*oth.num  
        bottom = self.denom*oth.denom  
        return top/bottom
```

# LET'S TRY IT OUT

```
f1 = SimpleFraction(3, 4)
f2 = SimpleFraction(1, 4)
print(f1.num)            3
print(f1.denom)          4
print(f1.plus(f2))       1.0
print(f1.times(f2))      0.1875
```

# YOU TRY IT!

- Add two methods to invert fraction object according to the specs below:

```
class SimpleFraction(object):
    """ A number represented as a fraction """
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom
    def get_inverse(self):
        """ Returns a float representing 1/self """
        pass
    def invert(self):
        """ Sets self's num to denom and vice versa.
            Returns None. """
        pass

# Example:
f1 = SimpleFraction(3,4)
print(f1.get_inverse())    # prints 1.33333333 (note this one returns value)
f1.invert()                # acts on data attributes internally, no return
print(f1.num, f1.denom)    # prints 4 3
```

# LET'S TRY IT OUT WITH MORE THINGS

```
f1 = SimpleFraction(3, 4)
f2 = SimpleFraction(1, 4)
print(f1.num)           → 3
print(f1.denom)         → 4
print(f1.plus(f2))      → 1.0
print(f1.times(f2))     → 0.1875
```

```
print(f1)
```

```
print(f1 * f2)
```

**<\_\_main\_\_.SimpleFraction object at 0x00000234A8C41DF0>  
Error!**

*What if we want to keep as a fraction?*

*And what if we want to have print and \* work as we would expect?*

# SPECIAL OPERATORS IMPLEMENTED WITH DUNDER METHODS

- +, -, ==, <, >, len(), print, and many others are shorthand notations

- Behind the scenes, these **get replaced by a method!**

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

- Can **override** these to work with your class

# SPECIAL OPERATORS IMPLEMENTED WITH DUNDER METHODS

- Define them with **double underscores** before/after

<code>__add__(self, other)</code>	<code>→</code>	<code>self + other</code>
<code>__sub__(self, other)</code>	<code>→</code>	<code>self - other</code>
<code><b>__mul__(self, other)</b></code>	<code>→</code>	<code><b>self * other</b></code>
<code>__truediv__(self, other)</code>	<code>→</code>	<code>self / other</code>
<code>__eq__(self, other)</code>	<code>→</code>	<code>self == other</code>
<code>__lt__(self, other)</code>	<code>→</code>	<code>self &lt; other</code>
<code>__len__(self)</code>	<code>→</code>	<code>len(self)</code>
<code><b>__str__(self)</b></code>	<code>→</code>	<code><b>print(self)</b></code>
<code><b>__float__(self)</b></code>	<code>→</code>	<code><b>float(self) i.e cast</b></code>
<code>__pow__</code>	<code>→</code>	<code>self**other</code>

... and others

# PRINTING OUR OWN DATA TYPES

# PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **Uninformative** print representation by default
- Define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- You choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

# DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):  
    def __init__(self, xval, yval):  
        self.x = xval  
        self.y = yval  
    def distance(self, other):  
        x_diff_sq = (self.x-other.x)**2  
        y_diff_sq = (self.y-other.y)**2  
        return (x_diff_sq + y_diff_sq)**0.5  
    def __str__(self):  
        return "<" + str(self.x) + ", " + str(self.y) + ">"
```

name of  
special  
method

must  
return a  
string

# WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- Can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
```

```
>>> print(c)
```

```
<3,4>
```

```
>>> print(type(c))
```

```
<class __main__.Coordinate>
```

- This makes sense since

```
>>> print(Coordinate)
```

```
<class __main__.Coordinate>
```

```
>>> print(type(Coordinate))
```

```
<type 'type'>
```

- Use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
```

```
True
```

Return of the `__str__` method  
The type of object `c` is a class `Coordinate`

A `Coordinate` is a class  
A `Coordinate` class is a type of object

# EXAMPLE: FRACTIONS WITH DUNDER METHODS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
  - Numerator
  - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - Add, sub, mult, div to work with `+`, `-`, `*`, `/`
  - Print representation, convert to a float
  - Invert the fraction
- Let's write it together!

# CREATE & PRINT INSTANCES

```
class Fraction(object):  
    def __init__(self, n, d):  
        self.num = n  
        self.denom = d  
    def __str__(self):  
        return str(self.num) + "/" + str(self.denom)
```

Concatenation means that  
every piece has to be a str

# LET'S TRY IT OUT

```
f1 = Fraction(3, 4)
```

```
f2 = Fraction(1, 4)
```

```
f3 = Fraction(5, 1)
```

```
print(f1)
```

→ 3/4

```
print(f2)
```

→ 1/4

```
print(f3)
```

→ 5/1

**Ok, but looks weird!**

# YOU TRY IT!

- Modify the str method to represent the Fraction as just the numerator, when the denominator is 1. Otherwise its representation is the numerator then a / then the denominator.

```
class Fraction(object):  
    def __init__(self, num, denom):  
        self.num = num  
        self.denom = denom  
    def __str__(self):  
        return str(self.num) + "/" + str(self.denom)
```

```
# Example:  
a = Fraction(1,4)  
b = Fraction(3,1)  
print(a)      # prints 1/4  
print(b)      # prints 3
```

# IMPLEMENTING

+ - \* /

float()

# COMPARING METHOD vs. DUNDER METHOD

```
class SimpleFraction(object):  
    def __init__(self, n, d):  
        self.num = n  
        self.denom = d  
        .....  
    def times(self, oth):  
        top = self.num*oth.num  
        bottom = self.denom*oth.denom  
        return top/bottom
```

When we use this method, Python evaluates and returns this expression, which creates a float

```
class Fraction(object):  
    def __init__(self, n, d):  
        self.num = n  
        self.denom = d  
        .....  
    def __mul__(self, other):  
        top = self.num*other.num  
        bottom = self.denom*other.denom  
        return Fraction(top, bottom)
```

Note: we are creating and returning a new instance of a Fraction

# LETS TRY IT OUT

```
a = Fraction(1,4)
```

```
b = Fraction(3,4)
```

```
print(a)  1/4
```

```
c = a * b
```

```
 print(c)  3/16
```

*Calls the  
\_\_mul\_\_ method  
behind the scenes.  
This method returns  
Fraction(3,16)*

*Uses \_\_str\_\_ for a  
Fraction object*

# CLASSES CAN HIDE DETAILS

- These are all equivalent

```
print(a * b)
```

```
print(a.__mul__(b))
```

```
print(Fraction.__mul__(a, b))
```

*Shorthand (nice and clear!)  
Call to dunder method, bad  
style with dunder methods!*

*Explicit class call, passing in val  
for self, bad style in general!*

- Every operation in Python comes back to a method call
- The first instance makes clear the operation, without worrying about the internal details!

**Abstraction at work**

# CAN KEEP BOTH OPTIONS BY ADDING A METHOD TO CAST TO A float

```
class Fraction(object):  
    def __init__(self, n, d):  
        self.num = n  
        self.denom = d  
  
    .....  
    def __float__(self):  
        return self.num/self.denom
```

A float since it does  
the division directly

```
c = a * b
```

```
print(c)
```

→ 3/16

Repr for Fraction(3,16)

```
print(float(c))
```

→ 0.1875

# LETS TRY IT OUT SOME MORE

```
a = Fraction(1,4)
```

```
b = Fraction(2,3)
```

```
c = a * b
```

```
print(c)  2/12
```

- Not quite what we might expect? It's not reduced.
- Can we fix this?

# ADD A METHOD

```
class Fraction(object):
```

```
.....
```

```
def reduce(self):
```

```
    def gcd(n, d):
```

```
        while d != 0:
```

```
            (d, n) = (n%d, d)
```

```
        return n
```

```
    if self.denom == 0:
```

```
        return None
```

```
    elif self.denom == 1:
```

```
        return self.num
```

```
    else:
```

```
        greatest_common_divisor = gcd(self.num, self.denom)
```

```
        top = int(self.num/greatest_common_divisor)
```

```
        bottom = int(self.denom/greatest_common_divisor)
```

```
        return Fraction(top, bottom)
```

Function to find the  
greatest common divisor

Call it inside the method

Still want a Fraction object back

```
c = a*b
```

```
print(c)
```

```
print(c.reduce())
```

➡ 2/12

➡ 1/6

# WE HAVE SOME IMPROVEMENTS TO MAKE

```
class Fraction(object):
```

```
.....
```

```
def reduce(self):
```

```
    def gcd(n, d):
```

```
        while d != 0:
```

```
            (d, n) = (n%d, d)
```

```
        return n
```

```
    if self.denom == 0:
```

```
        return None
```

```
    elif self.denom == 1:
```

```
        return self.num
```

```
    else:
```

```
        greatest_common_divisor = gcd(self.num, self.denom)
```

```
        top = int(self.num/greatest_common_divisor)
```

```
        bottom = int(self.denom/greatest_common_divisor)
```

```
        return Fraction(top, bottom)
```

*Is this a good idea?  
It does not return a Fraction so  
can no longer add or multiply  
this by other Fractions*

# CHECK THE TYPES, THEY'RE DIFFERENT

```
a = Fraction(4,1)
```

```
b = Fraction(3,9)
```

```
ar = a.reduce() ➡ 4
```

```
br = b.reduce() ➡ 1/3
```

```
print(ar, type(ar)) ➡ 4 <class 'int'>
```

```
print(br, type(br)) ➡ 1/3 <class '__main__.Fraction'>
```

```
c = ar * br
```

Error! It's trying to multiply an **int** with a **Fraction**.  
We never defined how to do this –  
only a Fraction with another Fraction

# YOU TRY IT!

- Modify the code to return a Fraction object when denominator is 1

```
class Fraction(object):
    def reduce(self):
        def gcd(n, d):
            while d != 0:
                (d, n) = (n%d, d)
            return n
        if self.denom == 0:
            return None
        elif self.denom == 1:
            return self.num
        else:
            greatest_common_divisor = gcd(self.num, self.denom)
            top = int(self.num/greatest_common_divisor)
            bottom = int(self.denom/greatest_common_divisor)
            return Fraction(top, bottom)
```

# Example:

```
f1 = Fraction(5,1)
```

```
print(f1.reduce())    # prints 5/1 not 5
```

# WHY OOP and BUNDLING THE DATA IN THIS WAY?

- Code is **organized** and **modular**
- Code is easy to **maintain**
- It's easy to **build upon** objects to make more complex objects
- **Decomposition and abstraction** at work with Python classes
  - Bundling data and behaviors means you can use objects consistently
  - Dunder methods are abstracted by common operations, but they're just methods behind the scenes!