SI100B: Introduction to Information Science and Technology

Lecture 10

By Ana Bell of MIT

DICTIONARIES

`list` vs `dict`

`list`	`dict`
Ordered sequence of elements	**Matches** “keys” to “values”
Look up elements by an integer index	Look up one item by another item
Index is an **integer**	Key can be any **immutable** type
Indices have an **order**	**No order** is guaranteed
Value can be any type	Value can be any type

EXAMPLE: FIND MOST COMMON WORDS IN A SONG'S LYRICS

1) Create a **frequency dictionary** mapping `str:int`
2) Find **word that occurs most often** and how many times
   - Use a list, in case more than one word with same number
   - Return a tuple `(list,int)` for (words_list, highest_freq)
3) Find the **words that occur greater than X times**
   - Let user choose X, so allow as parameter
   - Return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency
   - IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY

```
song = "RAH RAH AH AH AH ROM MAH RO MAH MAH"

def generate_word_dict(song):
    song_words = song.lower()
    words_list = song_words.split()
    word_dict = {}
    for w in words_list:
        if w in word_dict:
            word_dict[w] += 1
        else:
            word_dict[w] = 1
    return word_dict
```

Convert all chars to lower case

Convert string to list of words; divides based on spaces

Can iterate over list of words in song

If word in dict (as a key), increase # times you've seen it, update entry

If word not in dict, first time seeing word, create entry

Return is a dict mapping str:int

```
{'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

USING THE DICTIONARY

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def find_frequent_word(word_dict):
    words = []
    highest = max(word_dict.values())
    for k,v in word_dict.items():
        if v == highest:
            words.append(k)
    return (words, highest)
```

Highest frequency in dict's values

Loop to see which word has the highest freq

Append to list of all words that have that highest freq

Return is a tuple of (['ah', 'mah'], 3)

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Repeat the next few steps as long as the highest frequency is greater than x

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with the highest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- Save them in the result

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with the highest frequency

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {                    'rom':1,            'ro':1}
```

- Add them to the result so far

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with the highest frequency
- The highest frequency is now not greater than x=1, so stop

```
word_dict = {                    'rom':1,          'ro':1}
```

- The final result

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

LEVERAGING DICT PROPERTIES

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def occurs_often(word_dict, x):
    freq_list = []
    word_freq_tuple = find_frequent_word(word_dict)

    while word_freq_tuple[1] > x:
        freq_list.append(word_freq_tuple)
        for word in word_freq_tuple[0]:
            del(word_dict[word])
        word_freq_tuple = find_frequent_word(word_dict)
    return freq_list
```

Gives us a word tuple like (['ah', 'mah'], 3)

Stay in loop while we still have frequencies higher than x

Add those words to result

Mutate dict to remove ALL those words; on next loop, will find next most common words

SUMMARY

- Dictionaries have entries that **map a key to a value**
- **Keys are immutable/hashable and unique** objects
- **Values** can be **any object**
- Dictionaries can make code efficient
  - Implementation-wise
  - Runtime-wise

RECURSION

ITERATIVE ALGORITHMS SO FAR

- Looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- Can capture computation in a set of **state variables** that update, based on a set of rules, on each iteration through loop
  - What is **changing each time** through loop, and how?
  - When can I **stop**?
  - Where is the **result** when I stop?

MULTIPLICATION

- The * operator does this for us
- Make a function

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

MULTIPLICATION
THINK in TERMS of ITERATION

- Can you make this iterative? Assuming integer b>0
- Define `a*b` as `a+a+a+a... b` times
- Write a function

```
def mult(a, b):
    total = 0
    for n in range(b):
        total += a
    return total
```

MULTIPLICATION
THINK in TERMS of RECURSION

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers

Original problem

- $\boxed{5*4}$
- $\boxed{= 5+(\quad 5*3 \quad)}$
- = 5+(5+(5*2))
- = 5+(5+(5+(5*1)))

A multiplication with 5 is
5+5*one_less

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - 5*4
  - = 5+($\boxed{5*3}$)
  - = 5+$\boxed{(5+(5*2))}$
  - = 5+(5+(5+(5*1))

Similar problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION
FIND SMALLER VERSIONS of the PROBLEM

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - 5*4
  - = 5+(5*3)
  - = 5+(5+($\boxed{5*2}$))
  - = 5+(5+$\boxed{(5+(5*1))}$)

Similar problem

A multiplication with 5 is 5+5*one_less

MULTIPLICATION REACHED the END

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - 5*4
  - = 5+(5*3)
  - = 5+(5+(5*2))
  - = 5+(5+(5+($\boxed{5*1}$)))

Basic fact: a number multiplied with itself is the same number.

MULTIPLICATION
BUILD the RESULT BACK UP

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - 5*4
  - = 5+(5*3)
  - = 5+(5+($\boxed{5*2}$))
  - = 5+(5+$\boxed{(5+(5)}$))

Similar problem

10

MULTIPLICATION
BUILD the RESULT BACK UP

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers
  - 5*4
  - $= 5+(\boxed{5*3})$ Similar problem
  - $= 5+\boxed{(5+(10))}$ 15
  - = 5+(5+(5+(5))

MULTIPLICATION
BUILD the RESULT BACK UP

- If **a = 5** and **b = 4**
  - 5*4 is 5+5+5+5
- **Decompose** the original problem into
  - **Something you know** and
  - the **same problem** again
- Original problem is using * between two numbers

Original problem

  - $\boxed{5*4}$
  - = $\boxed{5+(\quad 15 \quad)}$ 20
  - = 5+(5+(10))
  - = 5+(5+(5+(5))

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations

$$\texttt{a*b} = \underbrace{\texttt{a + a + a + a + … + a}}_{\text{b times}}$$

$$= \texttt{a} + \underbrace{\texttt{a + a + a + … + a}}_{\text{b-1 times}}$$

$$= \texttt{a + a * (b-1)}$$

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations

$$\boxed{a*b} = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

$$= a + \underbrace{a + a + a + \ldots + a}_{\text{b-1 times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

MULTIPLICATION – RECURSIVE and BASE STEPS

- **Recursive step**
  - Decide how to reduce problem to a **simpler/smaller version** of same problem, plus simple operations
- **Base case**
  - Keep reducing problem until reach a simple case that can be **solved directly**
  - When `b=1, a*b=a`

$$\boxed{a*b} = \underbrace{a + a + a + a + \ldots + a}_{b \text{ times}}$$

$$= a + \underbrace{a + a + a + \ldots + a}_{b-1 \text{ times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

MULTIPLICATION – RECURSIVE CODE

- **Recursive step**
  - `If b != 1, a*b = a + a*(b-1)`
- **Base case**
  - `If b = 1, a*b = a`

```
def mult_recur(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recur(a, b-1)
```

base case

recursive step

WHAT IS RECURSION?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - Reduce a problem to simpler versions of the same problem or to problem that can be solved directly
- Semantically: a programming technique where a **function calls itself**
  - In programming, goal is to NOT have infinite recursion
  - Must have **1 or more base cases** that are easy to solve directly
  - Must solve the same problem on **some other input** with the goal of simplifying the larger input problem, ending at base case

YOU TRY IT!

- Complete the function that calculates n^p for integer variables n and p>=0

```
def power_recur(n, p):
    if _______:
        return ______
    else:
        return _________________
```

FACTORIAL

```
n! = n*(n-1)*(n-2)*(n-3)* … * 1
```

- For what n do we know the factorial?

  n = 1 →

  ```
  if n == 1:
      return 1
  ```

  base case

- How to reduce problem? Rewrite in terms of something simpler to reach base case

  n*(n-1)! →

  ```
  else:
      return n*fact(n-1)
  ```

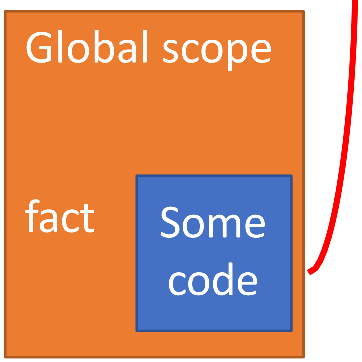
  recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```

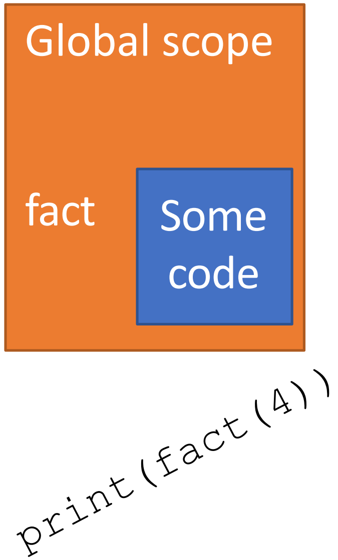
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



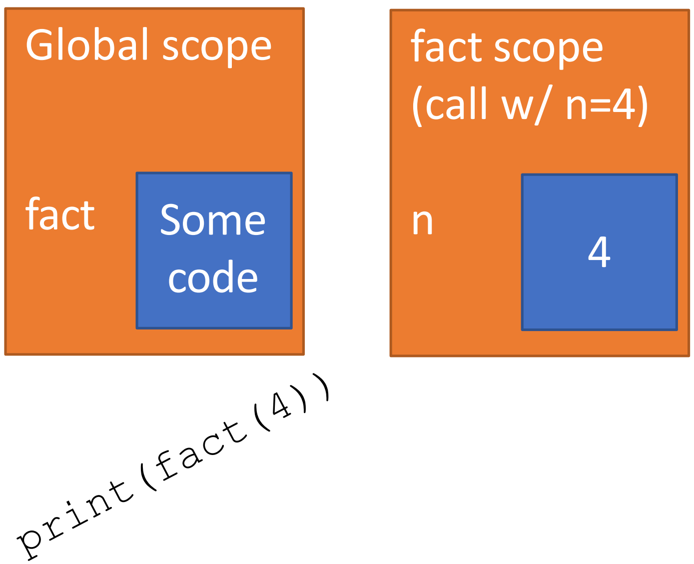
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



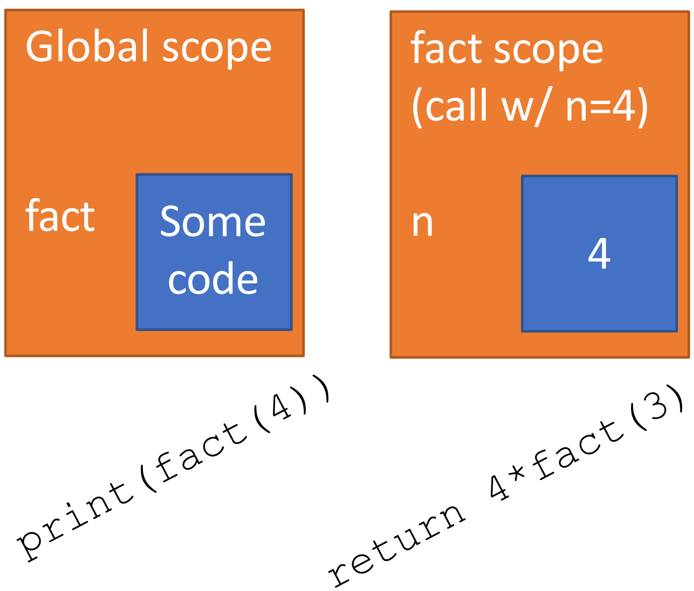
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



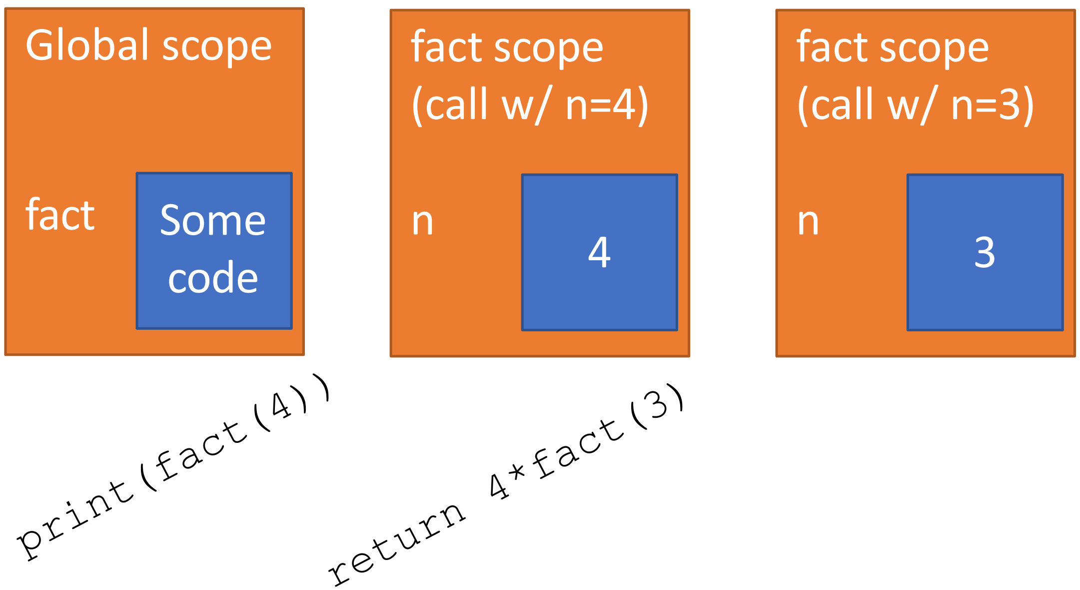
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



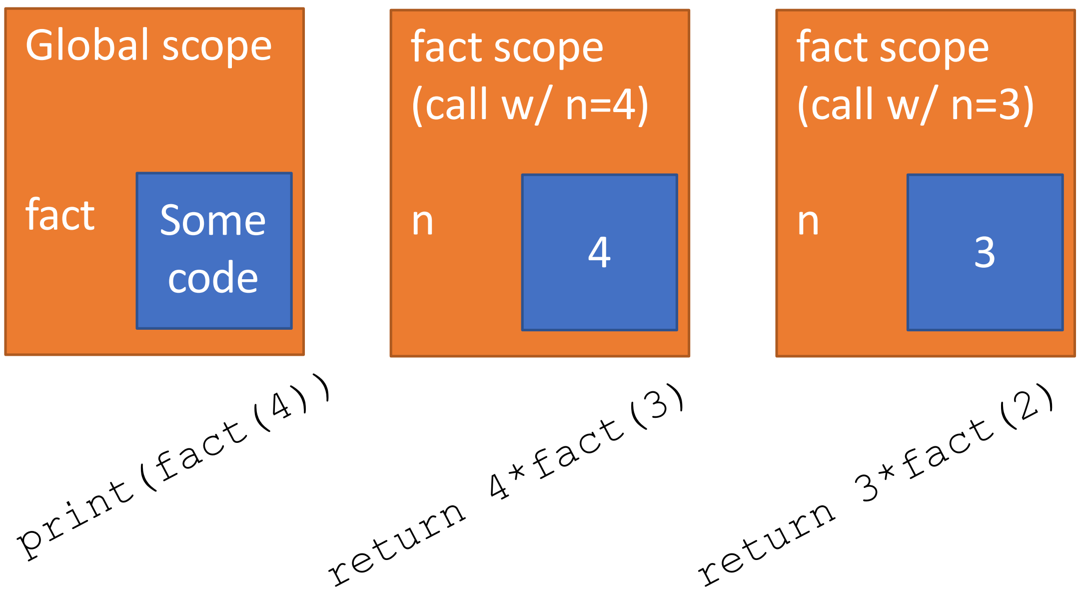
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



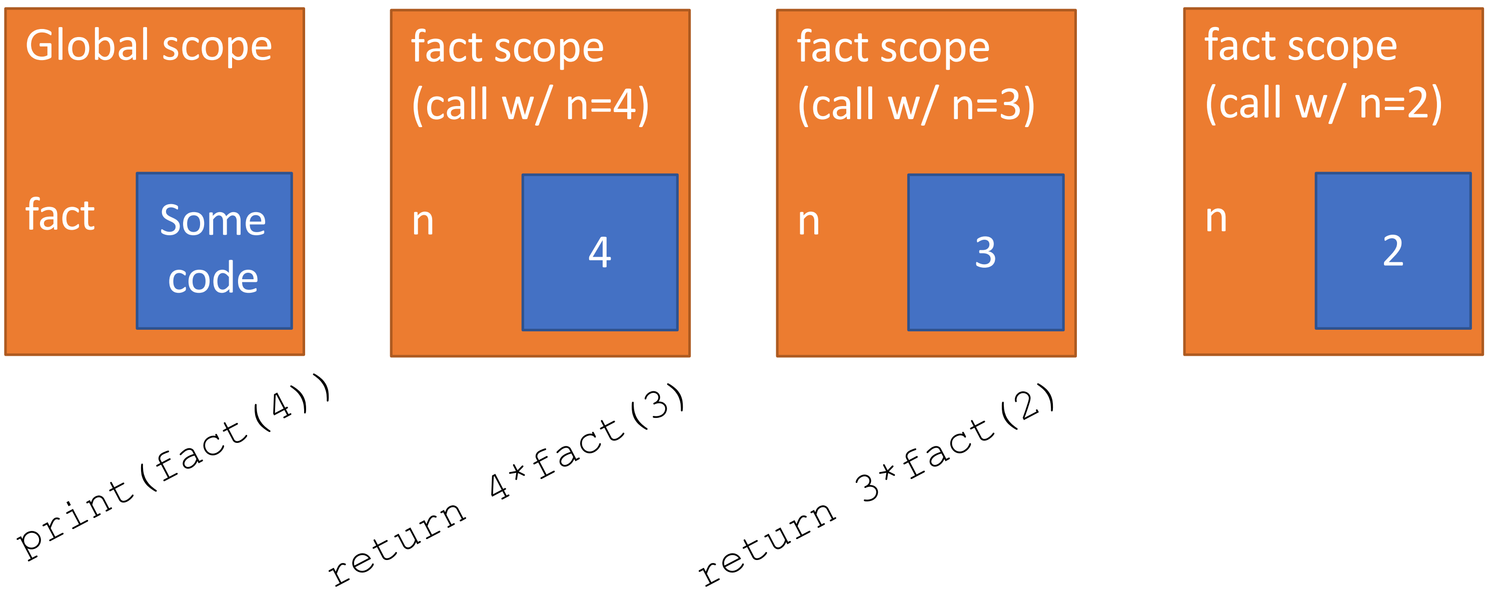
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



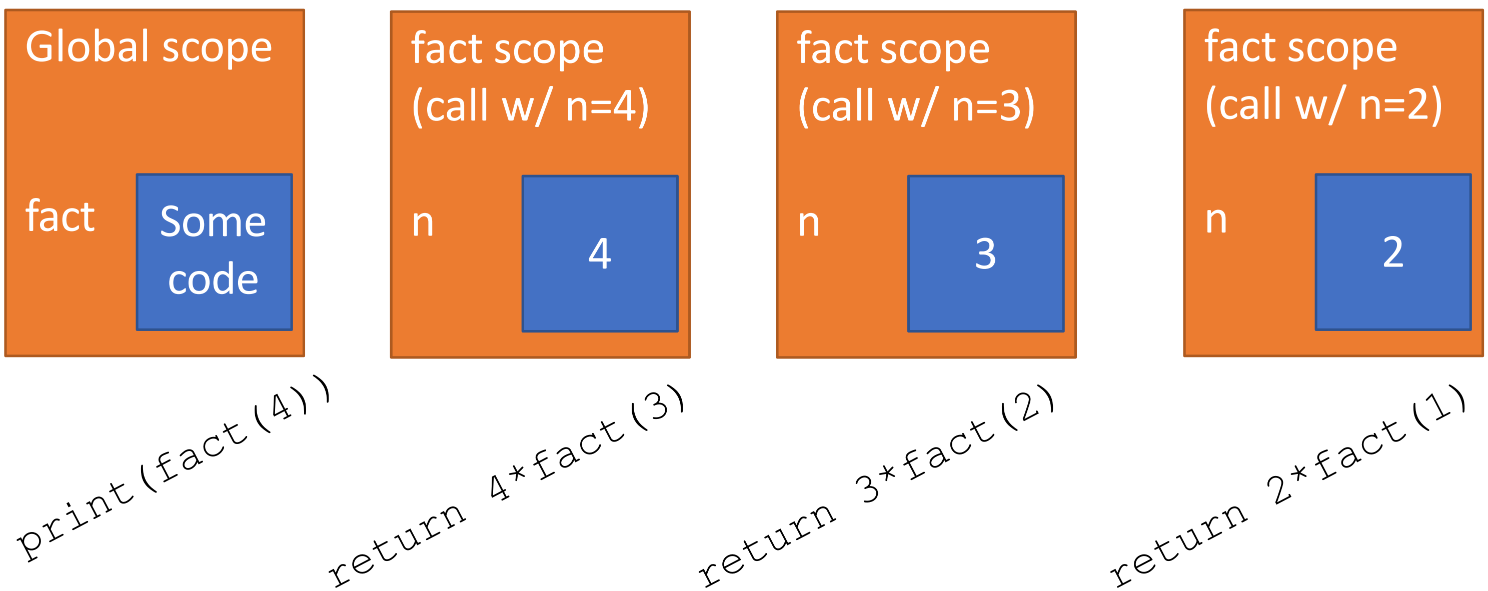
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



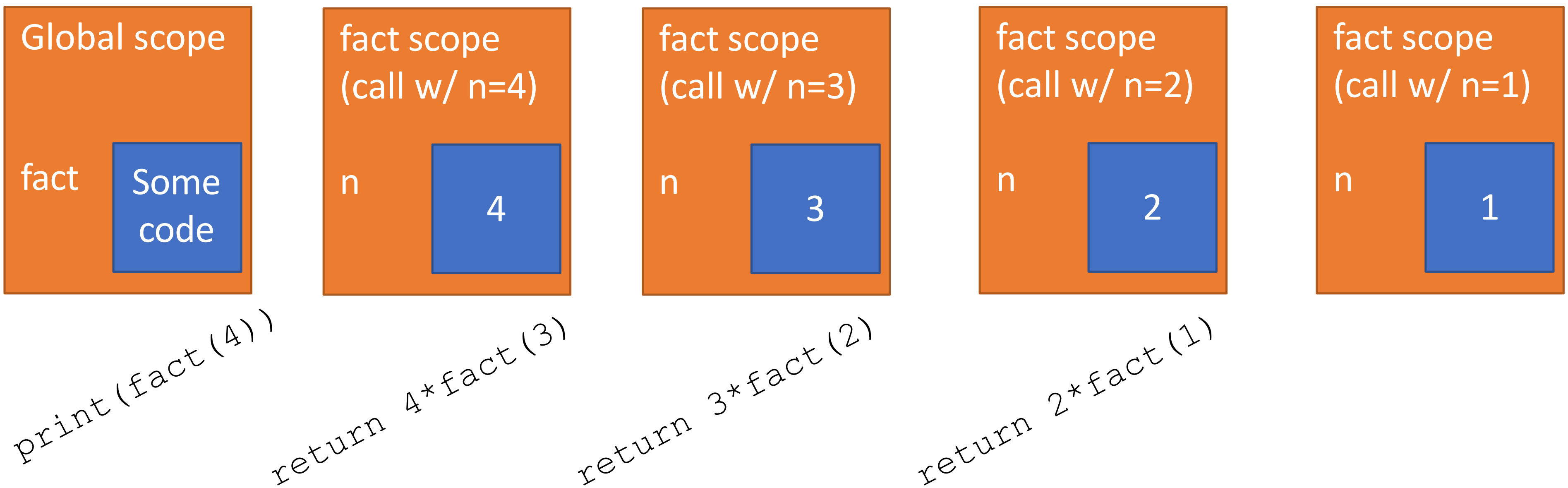
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



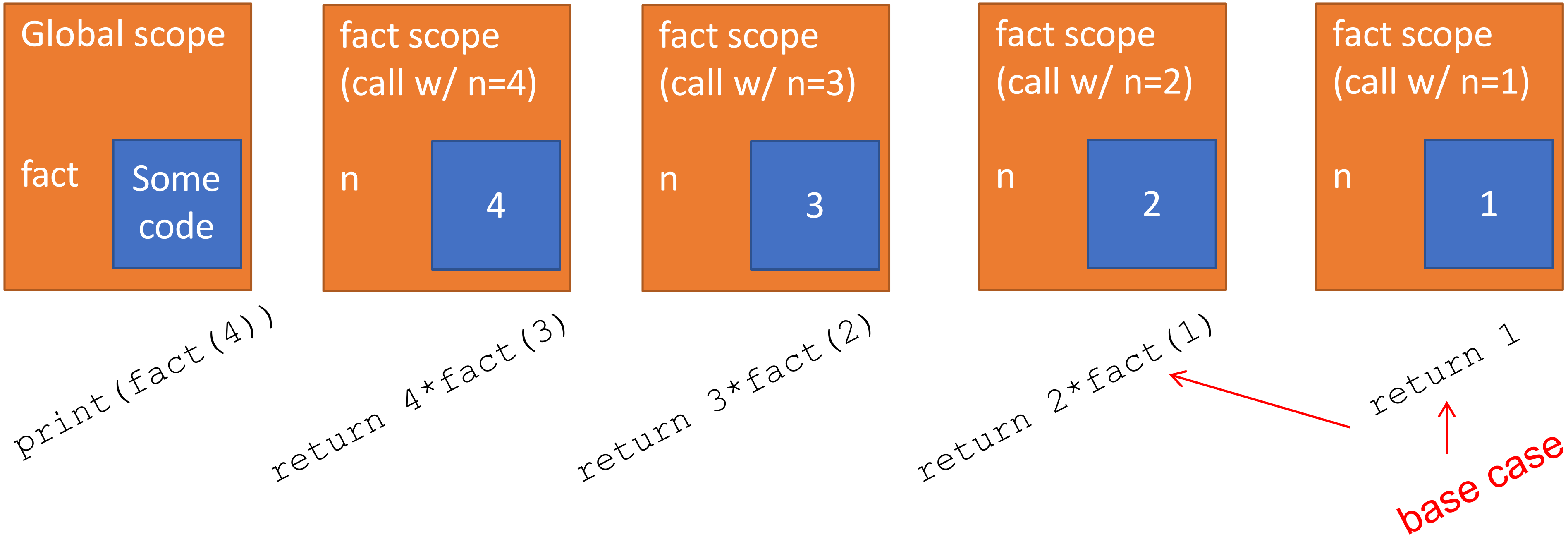
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



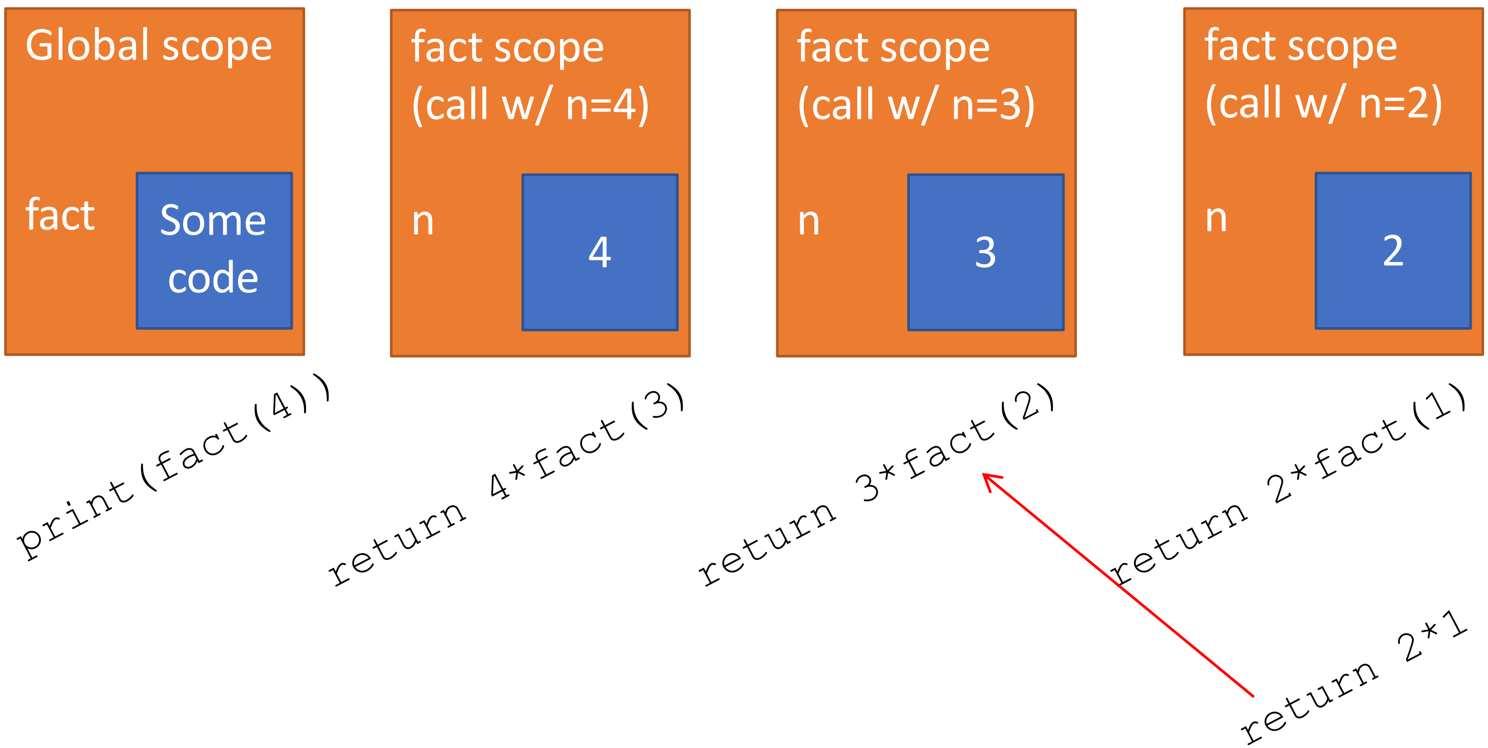
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



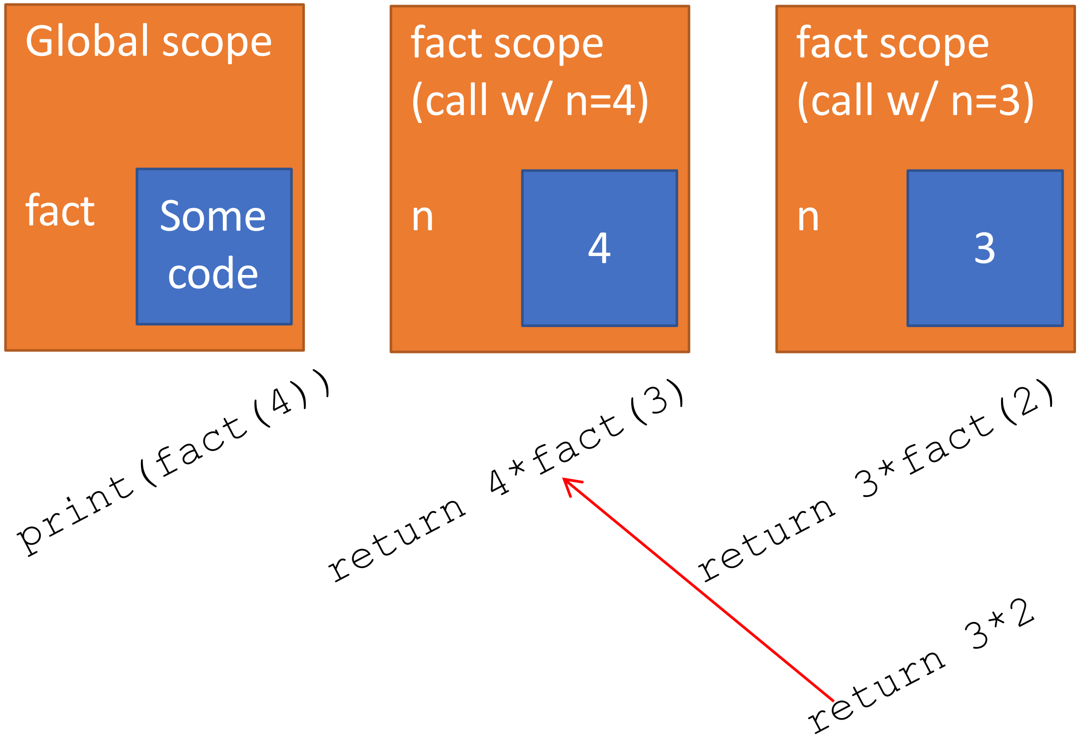
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



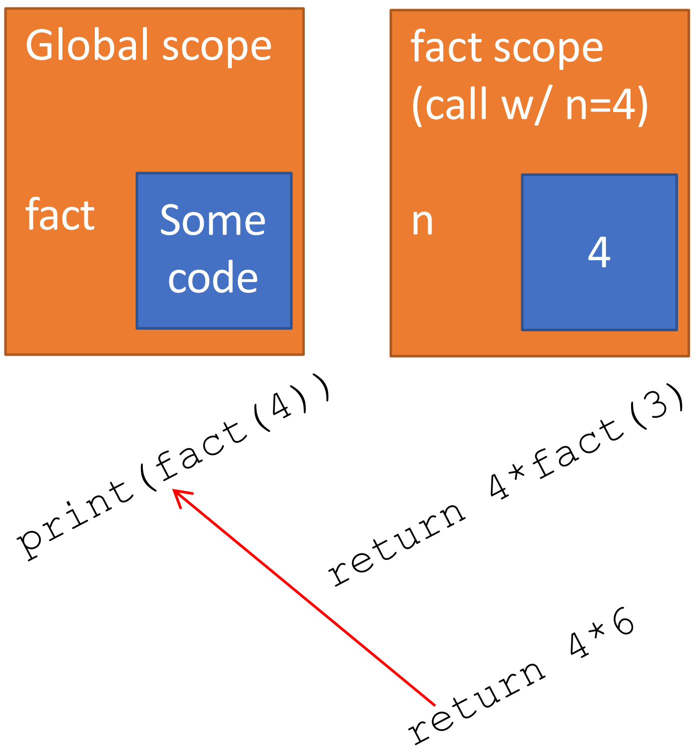
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



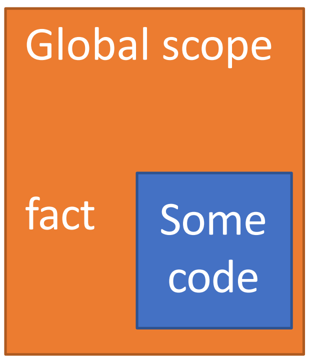
RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



`print(fact(4))`

`print(24)`

BIG IDEA

In recursion, each function call is completely separate.

Separate scope/environments.
Fully I-N-D-E-P-E-N-D-E-N-T

SOME OBSERVATIONS

- Each recursive call to a function creates its own **scope/environment**
- **Bindings of variables** in a scope are not changed by recursive call to same function
- Values of variable binding **shadow bindings** in other frames
- Flow of control passes back to **previous scope** once function call returns value

Using the same variable names but they are different objects in separate scopes

BIG IDEA

"Earlier" function calls are waiting on results before completing.

ITERATION vs. RECURSION

```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

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

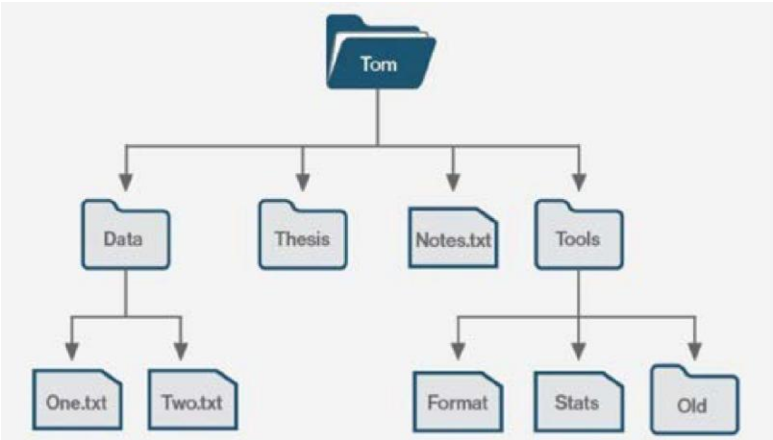
- Recursion may be efficient from programmer POV
- Recursion may not be efficient from computer POV

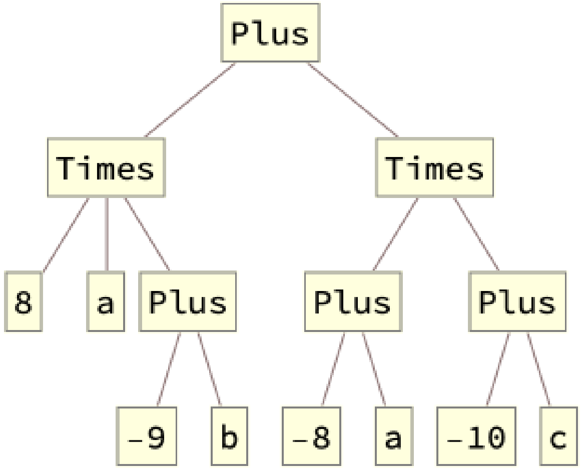
WHEN to USE RECURSION?
SO FAR WE SAW VERY SIMPLE CODE

- Multiplication of two numbers did not need a recursive function, did not even need an iterative function!
- Factorial was a little more intuitive to implement with recursion
  - But it can also be easily implemented with an iterative function
- MOST problems do not need recursion to solve them
  - If iteration is more intuitive for you, then solve them using loops!

WHEN to USE RECURSION

- SOME problems yield far simpler code using recursion
  - Searching a file system for a specific file
  - Evaluating mathematical expressions that use parentheses for order of operations

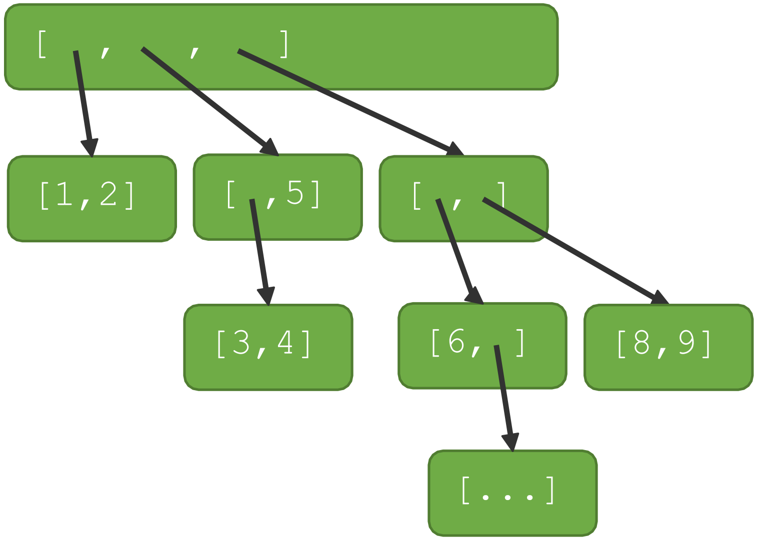




8*a*(-9+b)+(-8+a)*(-10+c)

WHEN to USE RECURSION

- In the list examples so far, we knew how many levels we needed to iterate.
  - Either look at elems directly or in one level down
- But lists can have elements that are lists, which can in turn have elements that are lists, which can in turn have elements that are lists, etc.



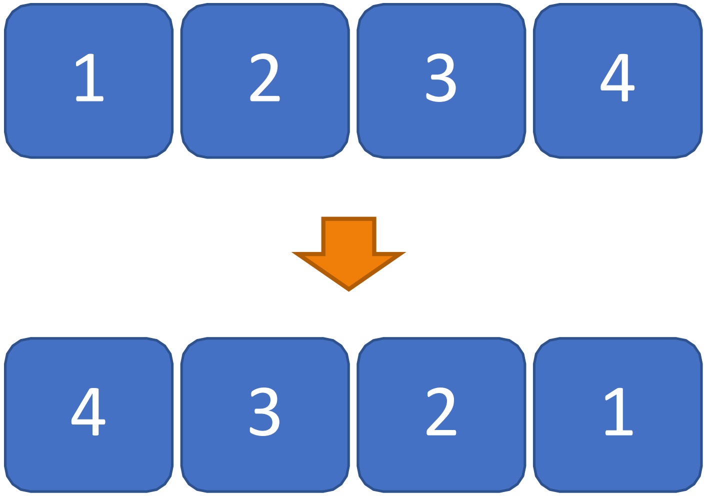
WHEN to USE RECURSION

- In the list examples so far, we knew how many levels we needed to iterate.
  - Either look at elems directly or in one level down
- But lists can have elements that are lists, which can in turn have elements that are lists, which can in turn have elements that are lists, etc.
- How can we use iteration to do these checks? It's hard.

```
for i in L:
    if type(i) == list:
        for j in i:
            if type(j) == list:
                for k in j:
                    if type(k) == list:
                        # and so on and on
                    else:
                        # do what you need to do
            else:
                # do what you need to do
    else:
        # do what you need to do
# done with the loop over L and all its elements
```

You don't know how deep this goes

Example: reverse a list's elements

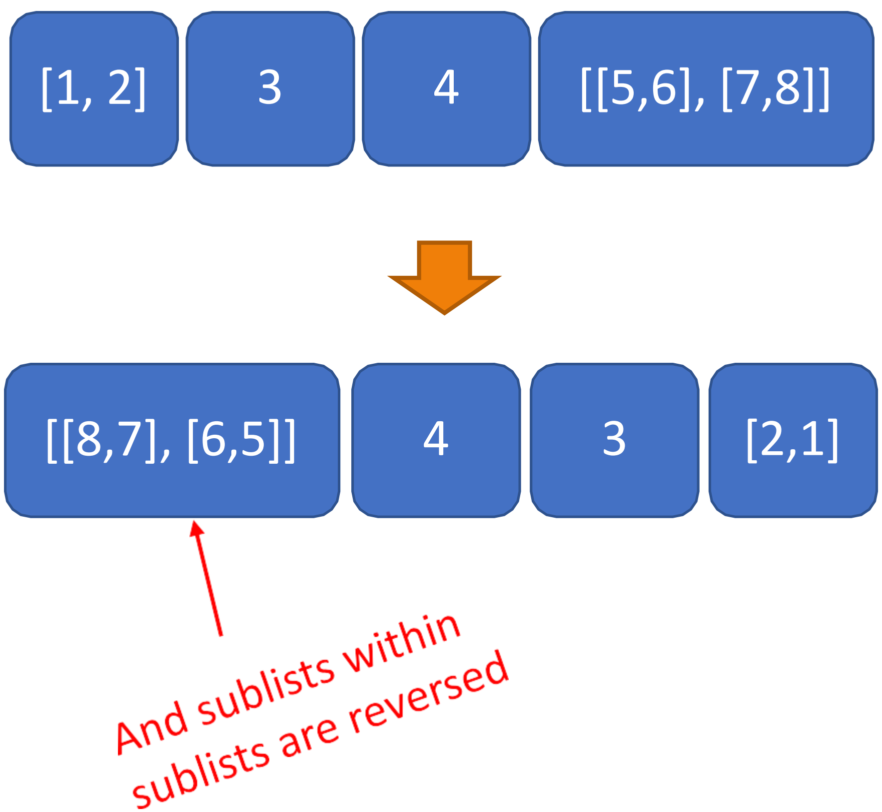



```
def rev(L):
    L2 = []
    for e in L[::-1]:
        L2.append(e)
    return L2
```

Or even simpler:

```
def rev(L):
    return L[::-1]
```

Example: reverse all elements in all sublists



Now need to know whether we are appending an element or a list

- lists must be reversed recursively

```
def deep_rev(L):
    L2 = []
    for e in L[::-1]:
        if type(e) == list:
            L2.append(deep_rev(e))
        else:
            L2.append(e)
    return L2
```

YOU TRY IT!

- Count the number of int in a nested list

```
def count_elem(L):
    """
    Return the total number of integers in a nested list of integers.
    """
    # Your code here

print(count_elem([1, 2, [[3, 4], 5]])) #5
```

Summary

- Most problems are solved more intuitively with iteration
  - We show recursion on these to:
    - Show you a different way of thinking about the same problem (algorithm)
    - Show you how to write a recursive function (programming)
- Some problems have nicer solutions with recursion
  - If you recognize solving the same problem repeatedly, use recursion
- Tips
  - Every case in your recursive function must return the same type of thing
    - i.e. don't have a base case `return []`
    - and a recursive step `return len(L[0])+recur(L[1:])`
  - It's ok to:
    - have more than one base case
    - have more than one recursive cases, as long as you are making progress towards a base case recursively

Object Oriented Programming: Classes

Objects

- Python supports many different kinds of data
  - `1234`
  - `3.14159`
  - `“Hello”`
  - `[1, 38, 4, 1, 35, 4]`
  - `{“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`

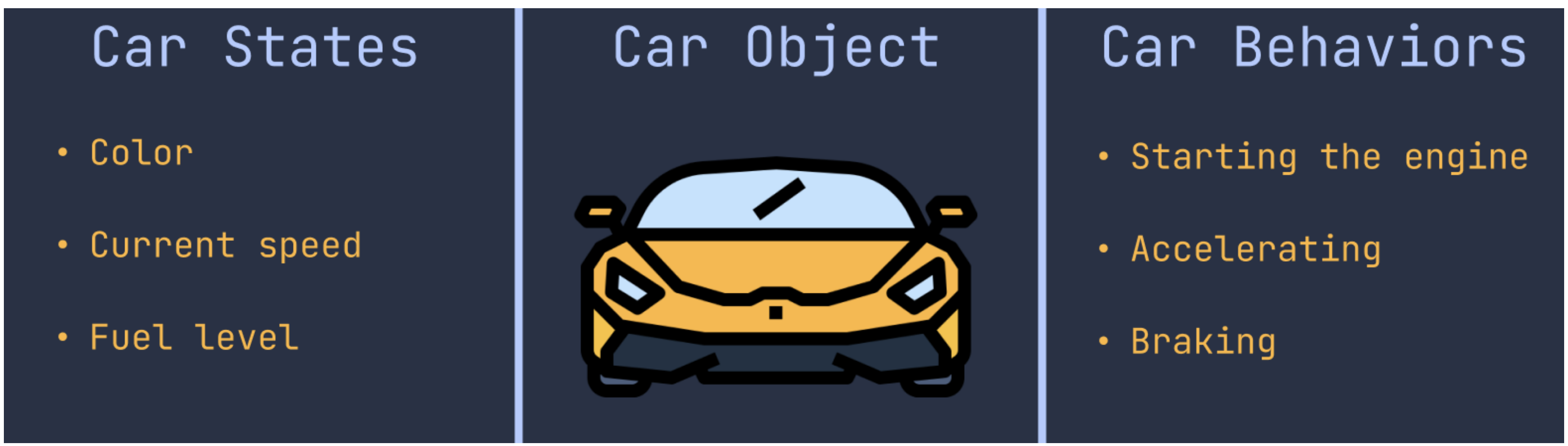
OBJECTS & TYPES

- **EVERYTHING IN PYTHON IS AN OBJECT**
  - 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"

- **EVERY OBJECT HAS A TYPE**
  - This lecture: create new types with class

OBJECTS & TYPES

- Objects of a specific type have…
  - An internal representation
    - Through data attributes
  - An interface for interacting with object
    - Through methods (i.e., procedural attributes)
    - Defines behaviors but hides implementation



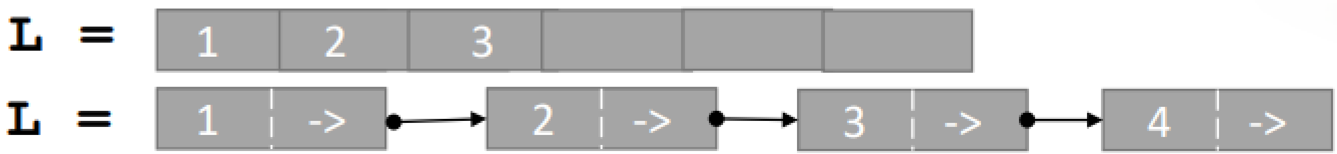
REAL-LIFE EXAMPLES

- 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

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

- How are lists represented internally?
  - Does not matter for so much for us as users (private representation)

follow pointer to the next index

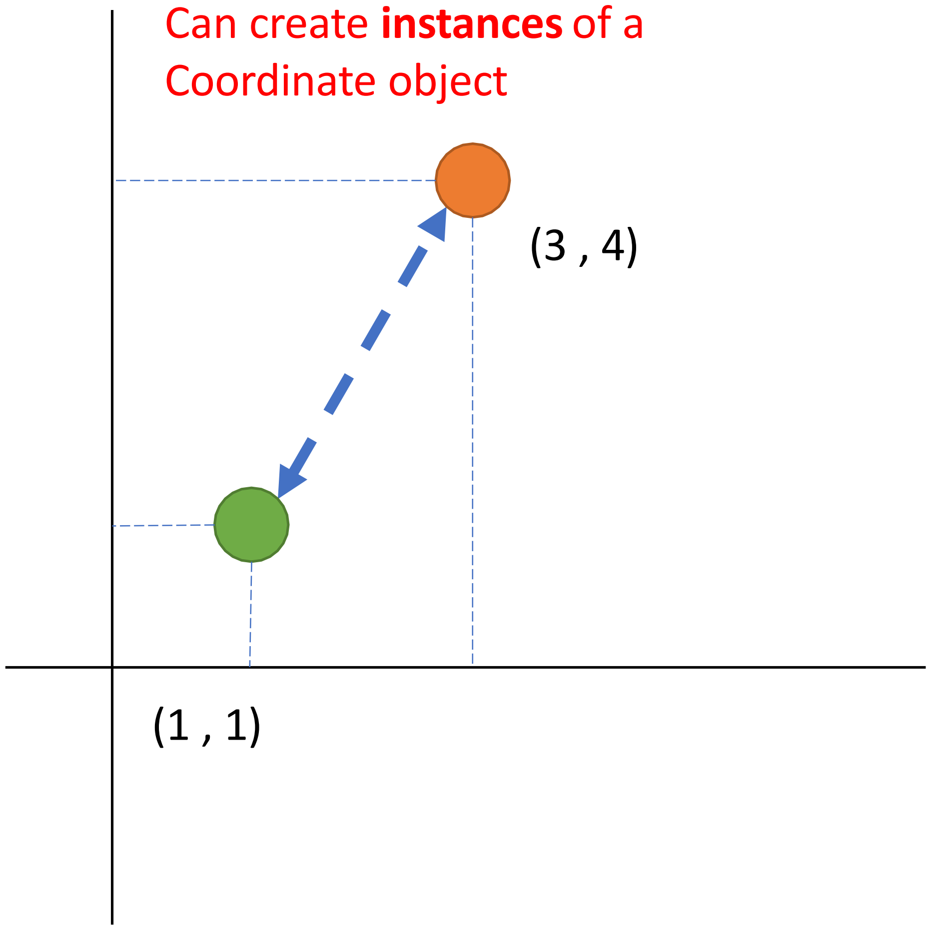


- How to interface with, and manipulate, lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(), L.extend(), L.count(), L.index(), L.insert(), L.pop(), L.remove(), L.reverse(), L.sort()`
- Internal representation should be private
- Correct behavior may be compromised if you manipulate internal representation directly

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- Creating the class involves
  - Defining the class name
  - Defining class attributes
    - Data attributes: representation
    - Procedural attributes: interface
  - *for example, 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)`

COORDINATE TYPE DESIGN DECISIONS



- 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

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)
  - Can be omitted

ATTRIBUTES

- Data attributes
  - Think of data as other objects that represent the object
  - *for example, a coordinate is made up of two numbers*
- Methods (i.e., 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*

Initialize data attributes

- Use a special method called __init__ to initialize some data attributes or perform initialization operations when creating an instance of class

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

what data initializes a Coordinate object

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

special method to create an instance is double __ underscore

two data attributes make up your type

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

ACTUALLY CREATING AN INSTANCE OF A CLASS

Recall the __init__ method in the class def:

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

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

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

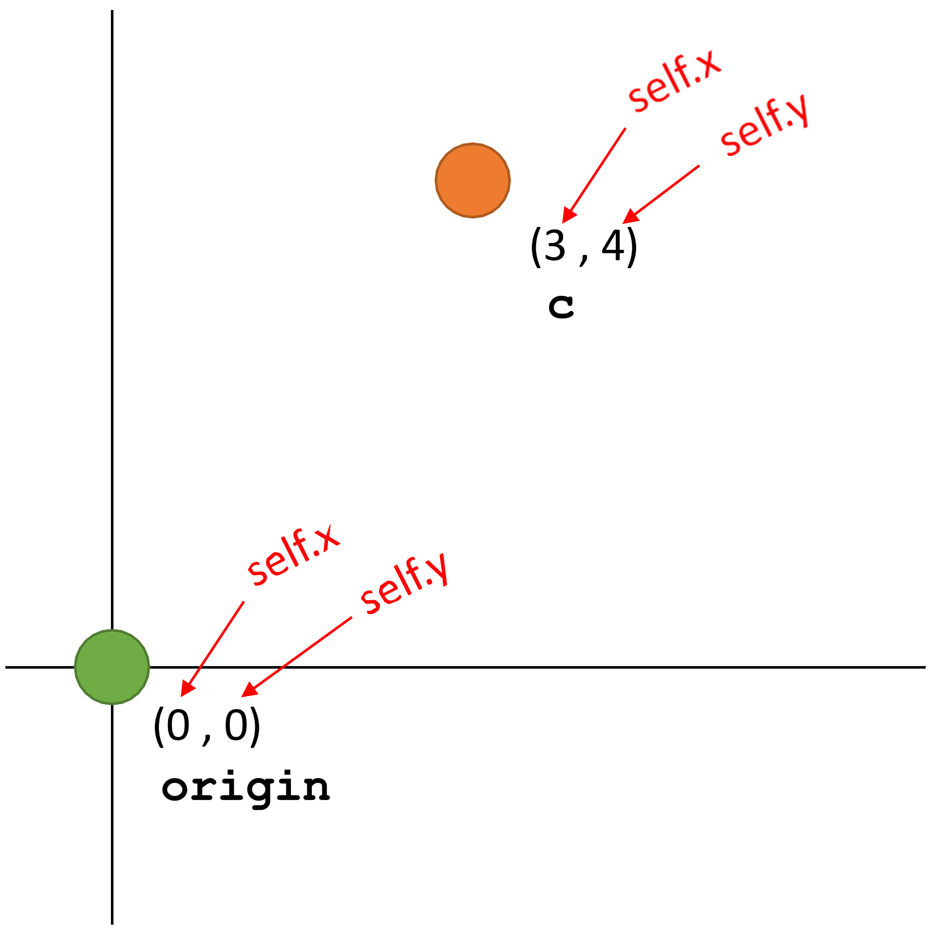
create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

- Data attributes of an instance are called instance variables
  - Data attributes 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: 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)
```

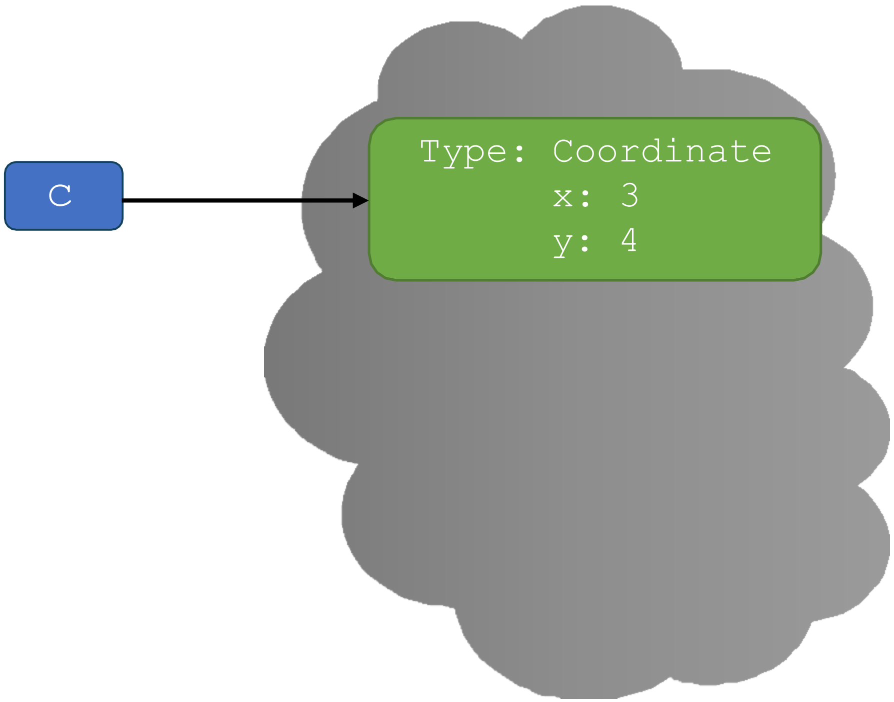
Code to make actual tangible Coordinate objects (aka instances)

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

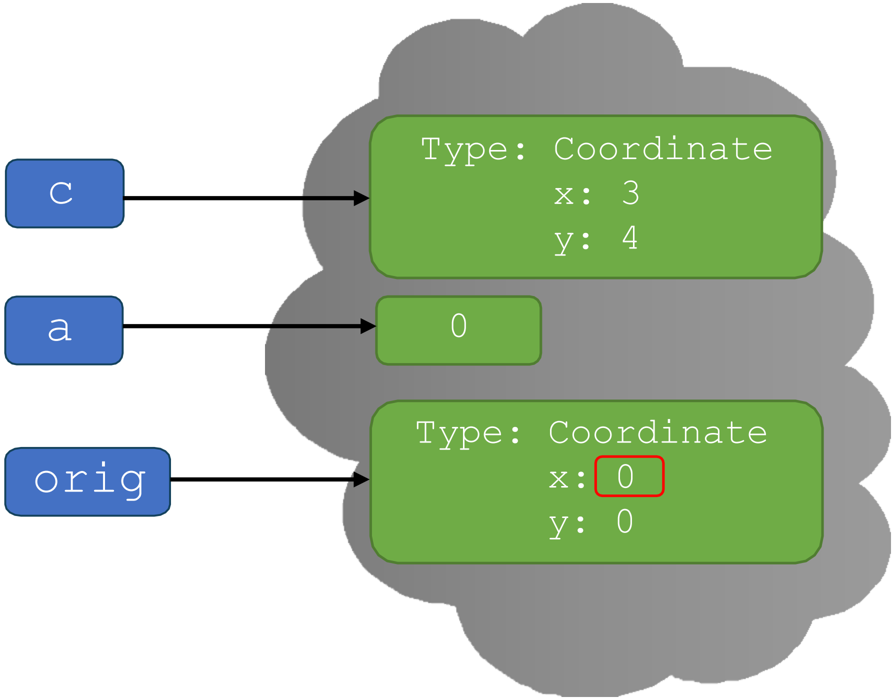


VISUALIZING INSTANCES: in memory

- 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



WHAT IS A METHOD?

- Procedural attribute
  - Think of it like a function that works only with this class

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
```

- Python always passes the object as the first argument
    - Convention is to use self as the name of the first argument of all methods
- 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()
```

HOW TO USE A METHOD

- 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
```

- Using the class:

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

object to call method on

name of method

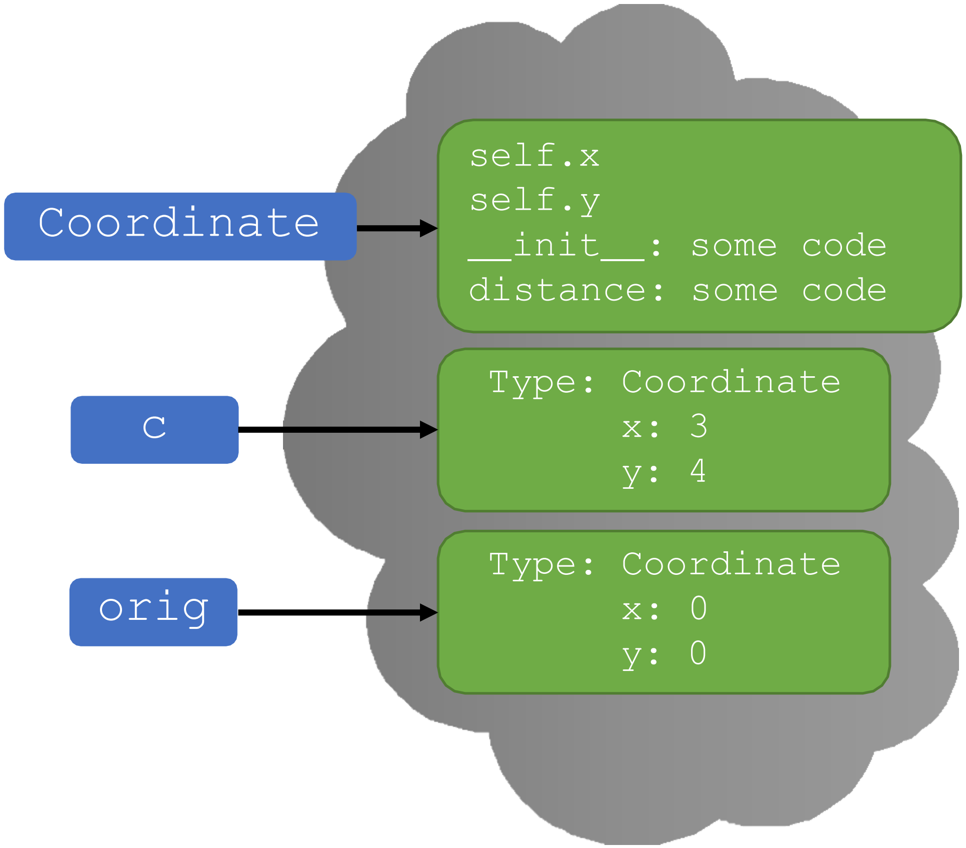
parameters not including `self` (`self` is implied to be `c`)

- 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)`

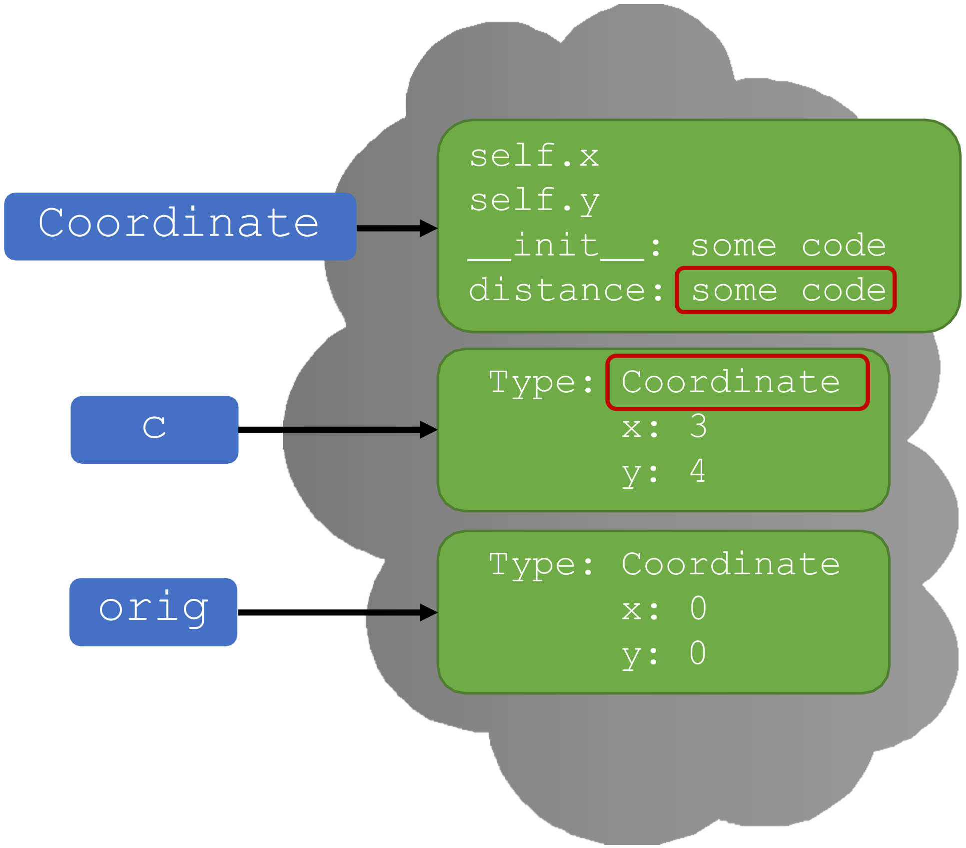
(1) The object is before the dot

(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`

BIG IDEA

The . operator accesses either data attributes or methods.

Data attributes are defined with self.something

Methods are functions defined inside the class with self as the first parameter.

Object Oriented Programming (OOP)

- Bundle related 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