TUPLES, LISTS, MUTABILITY, RECURSION

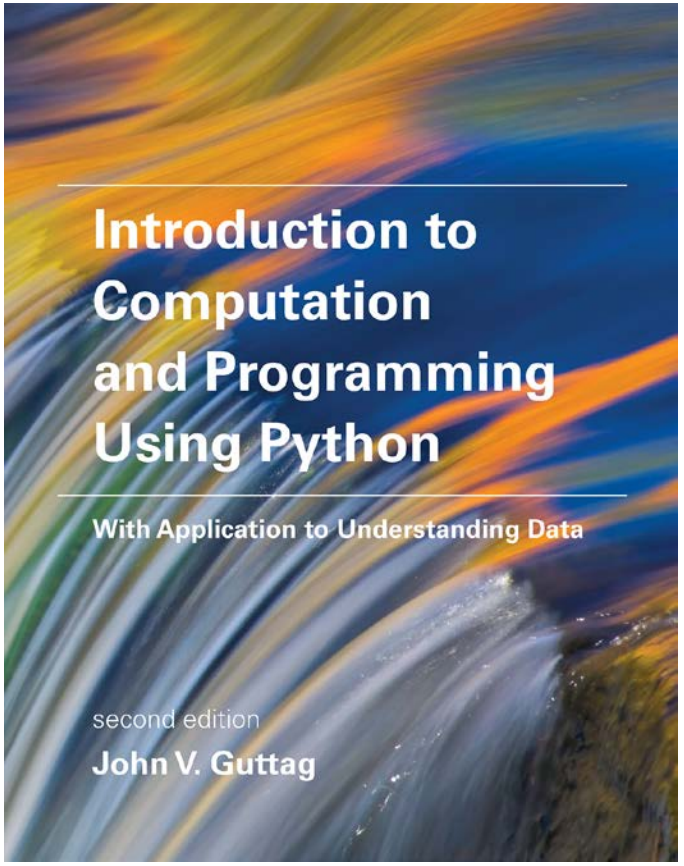
(download slides and .py files from Stellar to follow along)

6.0001 LECTURE 5

Ana Bell

ASSIGNED READING

- Sections 5.1 – 5.5
- Sections 4.3 – 4.6



https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

TODAY

- Have seen variable types: `int`, `float`, `bool`, `string`
- Introduce new **compound data types**
  - tuples
  - lists
- Ideas of
  - Mutability
  - Aliasing
  - Cloning
- Recursion

TUPLES

- **Indexable ordered sequence** of objects, can mix object types
- Cannot change element values, **immutable**

```
te = ()
ts = (2,)
t = (2, "mit", 3)
```

Empty tuple

Extra comma means tuple with one element

remember strings?

`t[0]` → evaluates to 2

`(2,"mit",3) + (5,6)` → evaluates to `(2,"mit",3,5,6)`

`t[1:2]` → slice tuple, evaluates to `("mit",)`

`t[1:3]` → slice tuple, evaluates to `("mit",3)`

`len(t)` → evaluates to 3

`max((3,5,0))` → evaluates 5

`t[1] = 4` → gives error, can't modify object

INDICES AND SLICING

```
seq = (2,'a',4,(1,2))
```

index: 0 1 2 3

```
print(len(seq))      → 4
print(seq[2]+1)      → 5
print(seq[3])        → (1,2)
print(seq[-1])       → (1,2)
print(seq[3][0])     → 1
print(seq[4])        → error
```

An element of a sequence is at an **index**, indices start at 0

```
print(seq[1])        → a
print(seq[:-1])      → (2,'a',4)
print(seq[1:3])      → 'a',4
```

Slices extract subsequences

```
for e in seq:        → 2
    print(e)           'a'
                       4
                       (1,2)
```

Iterating over sequences

TUPLES

- Conveniently used to **swap** variable values

```
x = y
y = x
```

```
temp = x
x = y
y = temp
```

```
(x, y) = (y, x)
```

- Used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(4,5)
```

integer division

LISTS

- **Indexable ordered sequence** of objects
  - Usually homogeneous (i.e., all integers, all strings, all lists)
  - Can contain mixed types (not common)
- Denoted by **square brackets**, [ ]
- **Mutable**, this means you can change element values

INDICES AND ORDERING

```
a_list = []
```

empty list

```
L = [2, 'a', 4, [1,2]]
```

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

```
i = 2
```

`L[i-1]` → evaluates to `'a'` since `L[1]='a'`

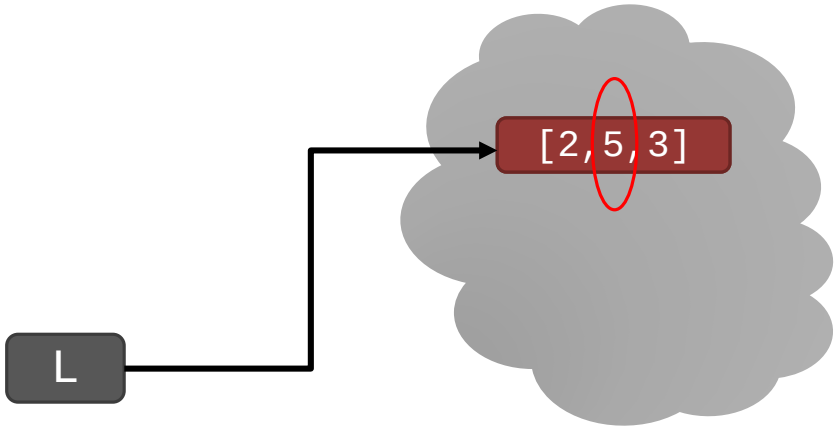
`max([3,5,0])` → evaluates 5

MUTABILITY

- Lists are **mutable**!
- Assigning to an element at an index **changes** the value

```
L = [2, 1, 3]
L[1] = 5
```

- L is now `[2, 5, 3]`, note this is the **same object** L



different from strings and tuples!

ITERATING OVER A LIST

- Compute the **sum of elements** of a list
- Common pattern

```
total = 0
for i in range(len(L)):
    total += L[i]
print(total)
```

```
total = 0
for i in L:
    total += i
print(total)
```

Like strings, can iterate over list elements directly

This version is more "pythonic"!

- Notice
  - List elements are indexed `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

OPERATION ON LISTS: append

- **Add** elements to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```

→ L is now `[2,1,3,5]`

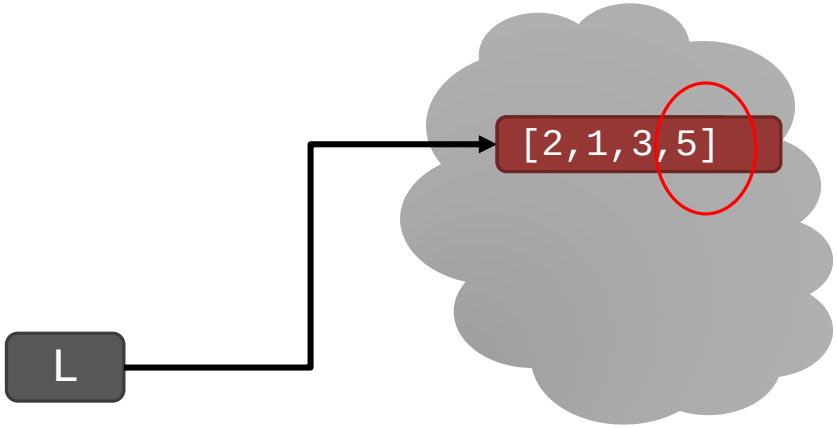
- What is the dot?
  - Lists are Python objects, everything in Python is an object
  - Objects have data
  - Objects have methods and functions
  - Access this information by `object_name.do_something()`
  - Will learn more about these later

OPERATION ON LISTS – append

- **Add** element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)
```

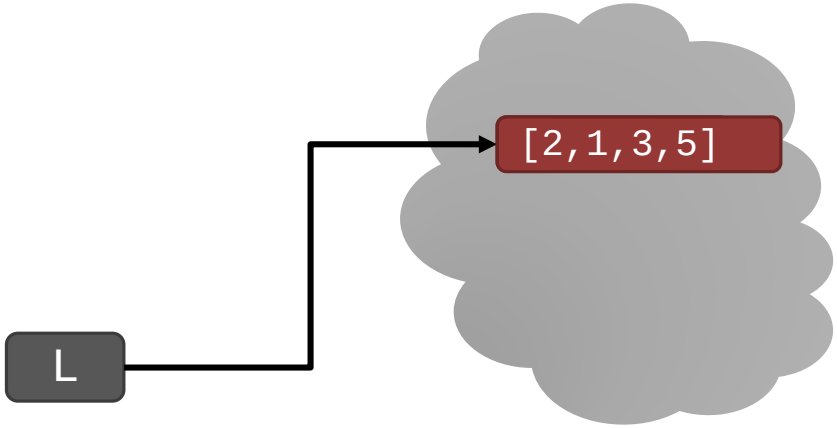
→ L is now `[2,1,3,5]`



OPERATION ON LISTS – append

- **Add** element to end of list with `L.append(element)`
- **Mutates** the list!

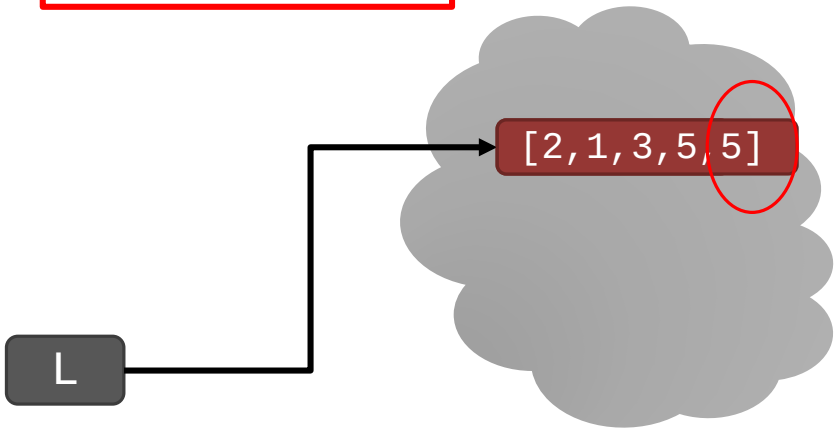
```
L = [2,1,3]
L.append(5)      → L is now [2,1,3,5]
L = L.append(5)
```



OPERATION ON LISTS – append

- **Add** element to end of list with `L.append(element)`
- **Mutates** the list!

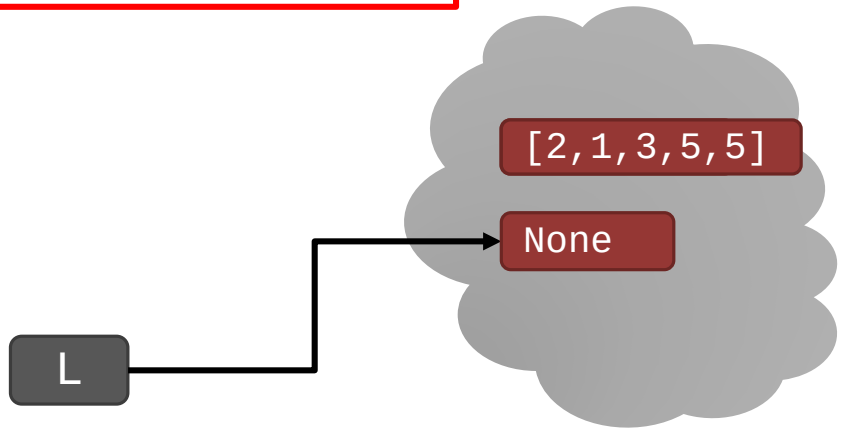
```
L = [2,1,3]
L.append(5)        → L is now [2,1,3,5]
L = L.append(5)
```



OPERATION ON LISTS – append

- **Add** element to end of list with `L.append(element)`
- **Mutates** the list!

```
L = [2,1,3]
L.append(5)        → L is now [2,1,3,5]
L = L.append(5)
```



TRICKY EXAMPLE 1: append

- **Range returns something that behaves like a tuple** (but isn't)
- Generates the first element, and provides an iteration method by which subsequent elements can be generated

`range(5)` → evaluates to tuple `(0,1,2,3,4)`
`range(2,6)` → evaluates to tuple `(2,3,4,5)`

```
L = [1,2,3,4]
for i in range(len(L)):
    L.append(i)
print(L)
```

Iteration sequence is pre-determined at the beginning

1st time: L is [1, 2, 3, 4, 0]
2nd time: L is [1, 2, 3, 4, 0, 1]
3rd time: L is [1, 2, 3, 4, 0, 1, 2]
4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

TRICKY EXAMPLE 2: append

```
L = [1,2,3,4]
i = 0
for e in L:
    L.append(i)
    i += 1
print(L)
```

Originally [1,2,3,4]

L is mutated each iteration

1st time: L is [1, 2, 3, 4, 0]

2nd time: L is [1, 2, 3, 4, 0, 1]

3rd time: L is [1, 2, 3, 4, 0, 1, 2]

4th time: L is [1, 2, 3, 4, 0, 1, 2, 3]

5th time: L is [1, 2, 3, 4, 0, 1, 2, 3, 4]

…

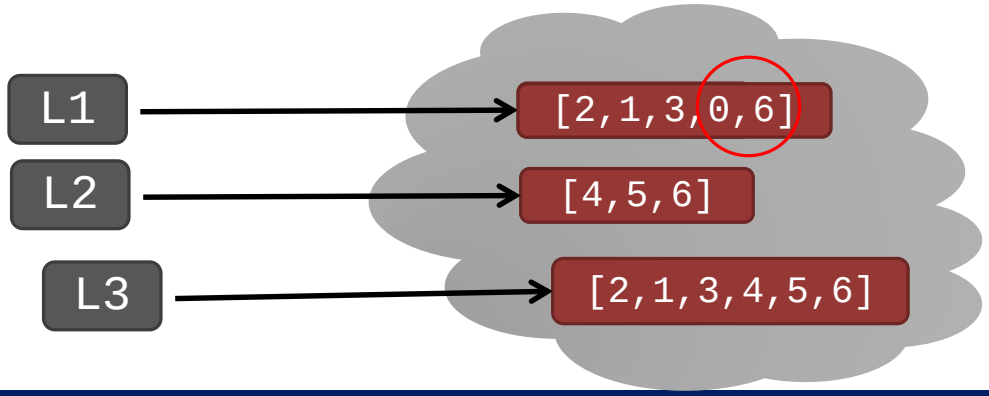
COMBINING LISTS

- **Concatenation**, + operator, creates a **new** list
- **Mutate** list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
L1.extend([0,6])
```

→ L3 is `[2,1,3,4,5,6]`

→ mutated L1 to `[2,1,3,0,6]`



TRICKY EXAMPLE 3: combining

```
L = [1,2,3,4]
i = 0
for e in L:
    L = L + L
    i += 1
print(L)
```

Originally [1,2,3,4]

L is a new object each iteration
e in L iterates only 4 times,
over the original L=[1,2,3,4]

1st time: new L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: new L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

3rd time: new L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

4th time: new L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

OPERATION ON LISTS: REMOVE

- Delete element at a **specific index** with `del(L[index])`
- Remove element at **end of list** with `L.pop()`, returns the removed element
- Remove a **specific element** with `L.remove(element)`
  - Looks for the element and removes it
  - If element occurs multiple times, removes first occurrence
  - If element not in list, gives an error

all these operations mutate the list

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2) → mutates L = [1,3,6,3,7,0]
L.remove(3) → mutates L = [1,6,3,7,0]
del(L[1])   → mutates L = [1,3,7,0]
L.pop()     → returns 0 and mutates L = [1,3,7]
```

MUTATION AND ITERATION
Try this in Python Tutor!

- **Avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

Clone list first
Note that `L1_copy = L1` does NOT clone

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - Mutating changes the list length but Python doesn't update the counter
  - Loop never sees element 2

CONVERT LISTS TO STRINGS AND BACK

- Convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- Can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- Use `''.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

```
s = "I<3 cs"          → s is a string
list(s)               → returns ['I','<','3',' ','c','s']
s.split('<')          → returns ['I', '3 cs']
L = ['a','b','c']     → L is a list
''.join(L)            → returns "abc"
'_'.join(L)           → returns "a_b_c"
```

OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`
- and many more!
  https://docs.python.org/3/tutorial/datastructures.html

```
L=[9,6,0,3]
a = sorted(L)
L.sort()
L.reverse()
```

→ returns sorted list, does **not mutate** L

→ **mutates** L=[0,3,6,9]

→ **mutates** L=[9,6,3,0]

MUTATION, ALIASING, CLONING

Again, Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

LISTS IN MEMORY

- Lists are **mutable**
- Behave differently than immutable types
- Is an object in memory
- Variable name points to object
- Using equal sign between mutable objects creates aliases
- Any variable pointing to that object is affected
- Key phrase to keep in mind when working with lists is **side effects**

ALIASING

Boston
The Hub
Beantown

- City may be known by many names
- Attributes of a city
  - small, tech-savvy
- All nicknames point to the **same city**
  - add new attribute to **one nickname** …

Boston	small	tech-savvy	windy

… all the **aliases** refer to the old attribute and all the new ones

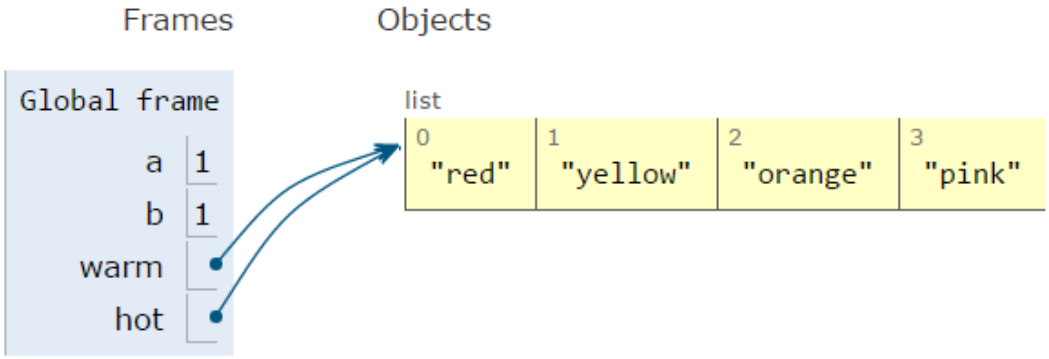
The Hub	small	tech-savvy	windy
Beantown	small	tech-savvy	windy

ALIASES

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append( )` has a side effect

```
1  a = 1
2  b = a
3  print(a)
4  print(b)
5
6  warm = ['red', 'yellow', 'orange']
7  hot = warm
8  hot.append('pink')
9  print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

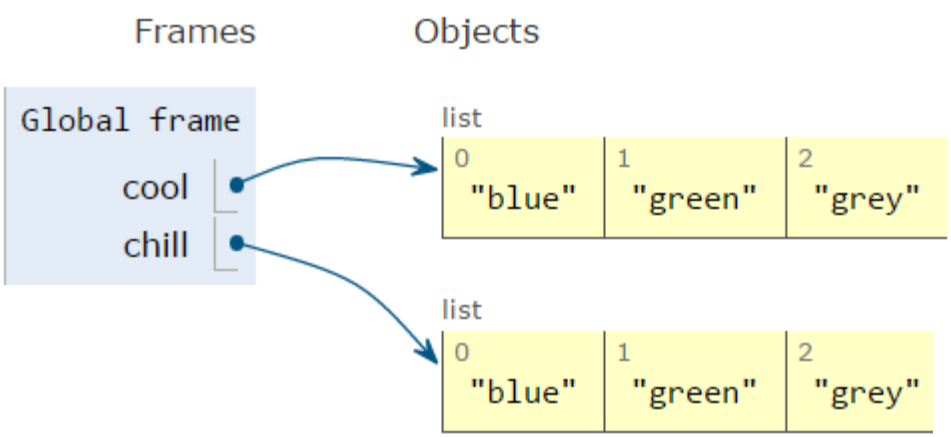


CLONING A LIST

- Create a new list and **copy every element** using `chill = cool[:]`

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

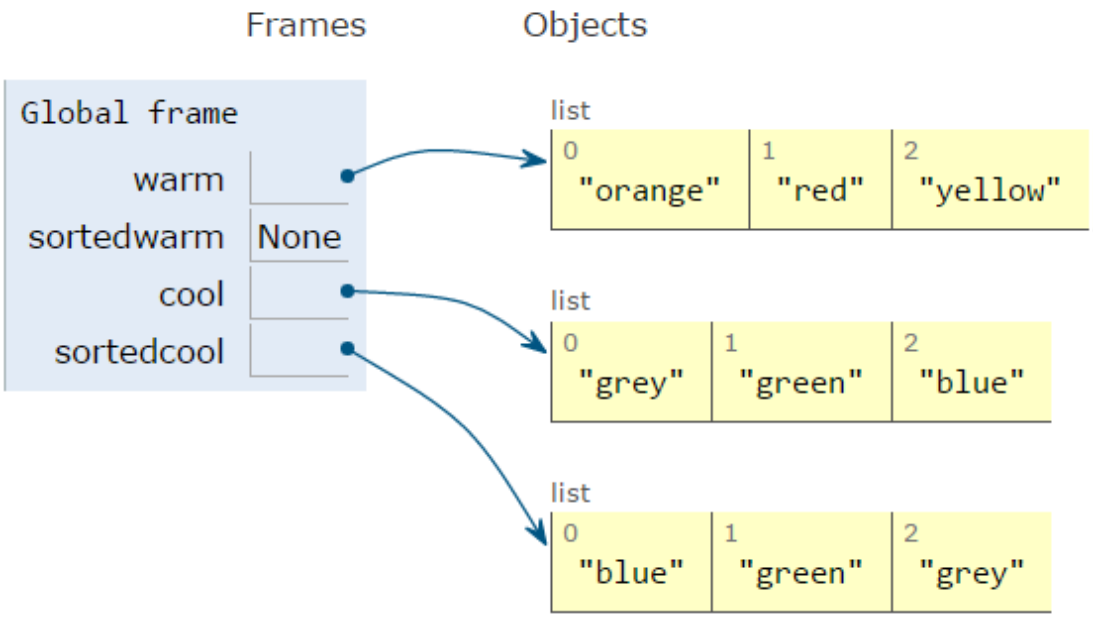


SORTING LISTS

- Calling `sort()` **mutates** the list, returns nothing
- Calling `sorted()` **does not mutate** list, must assign result to a variable

```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```

```
1  warm = ['red', 'yellow', 'orange']
2  sortedwarm = warm.sort()
3  print(warm)
4  print(sortedwarm)
5
6  cool = ['grey', 'green', 'blue']
7  sortedcool = sorted(cool)
8  print(cool)
9  print(sortedcool)
```

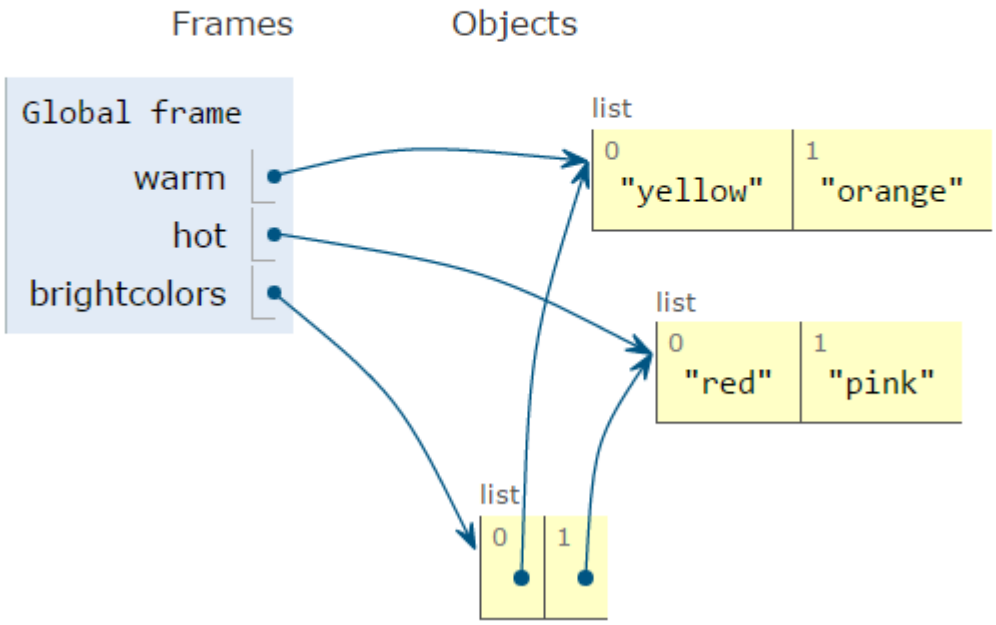


LISTS OF LISTS OF LISTS OF….

- Can have **nested** lists
- Side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```

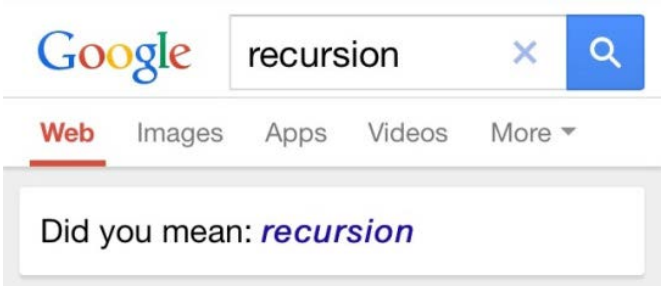
```
1 warm = ['yellow', 'orange']
2 hot = ['red']
3 brightcolors = [warm]
4 brightcolors.append(hot)
5 print(brightcolors)
6 hot.append('pink')
7 print(hot)
8 print(brightcolors)
```



Five Minute Break

Debugging your pset

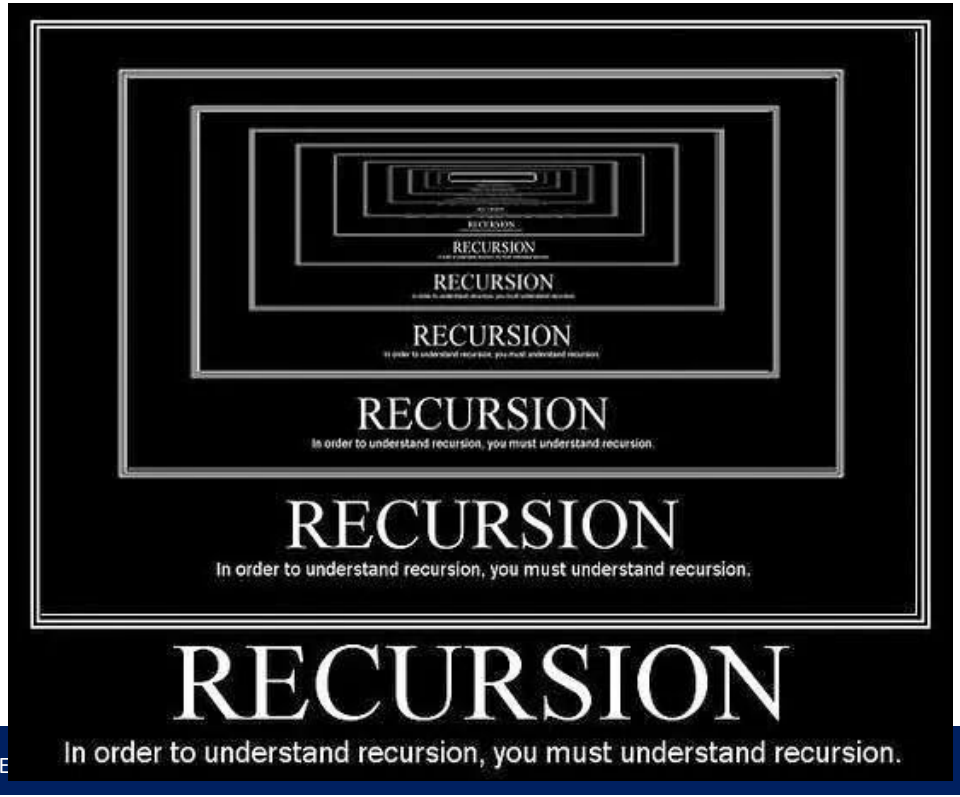
RECURSION



A recursive light bulb joke

Q: How many twists does it take to screw in a light bulb?

A: Is it already screwed in? Then zero. If not, then twist it once, ask me again, and add 1 to my answer.



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 on each iteration through loop

WHAT IS RECURSION

- Most joked about CS topic
- A way to design solutions to problems by **divide-and-conquer or decrease-and-conquer**
- 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
  - Must solve the same problem on **some other input** with the goal of simplifying the larger problem input

MULTIPLICATION – ITERATIVE SOLUTION

- "multiply a * b" is equivalent to "add a to itself b times"
- Capture **state** by
  - An **iteration** number (`i`) starts at b
    `i ← i-1` and stop when 0
  - A current **value of computation** (`result`)
    `result ← result + a`

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

iteration

current value of computation, a running sum

MULTIPLICATION – RECURSIVE SOLUTION

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

```
a*b = a + a + a + a + … + a
```
b times

```
    = a + a + a + a + … + a
```
b-1 times

```
    = a + a * (b-1)
```

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

base case

recursive step

FACTORIAL

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

- What n do we know the factorial of?

  n = 1 → `if n == 1:`
  `return 1` base case

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

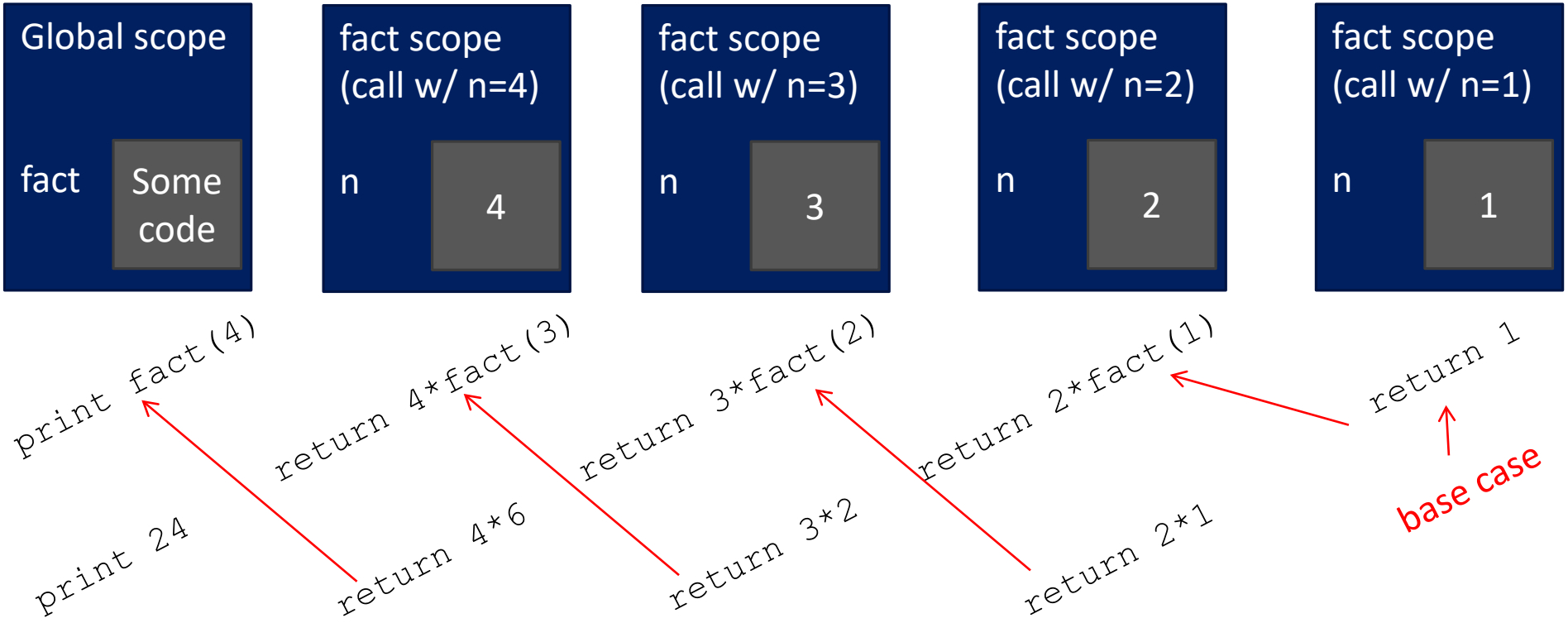
  n*(n-1)! → `else:`
  `return n*factorial(n-1)`

recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

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

print(fact(4))
```



SOME OBSERVATIONS

- Each recursive call to a function creates its **own scope/environment**
- **Bindings of variables** in a scope is not changed by recursive call
- 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

ITERATION vs. RECURSION

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

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

- Recursion may be simpler, more intuitive
- Recursion may be efficient from programmer POV
- Recursion may not be efficient from computer POV

MULTIPLY ELEMENTS OF LIST

```
def mult_list_recur(L):
    if len(L) == 1:
        return L[0]
    else:
        return L[0]*mult_list_recur(L[1:])
```

- Decrease and conquer
- What happens when L=[]?

error, list index out of range

FIND ELEMENT IN ORDERED LIST

```
def find_elem_recur(e, L):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return find_elem_recur(e, L[:half])
        else:
            return find_elem_recur(e, L[half:])
```

- Divide and conquer

```
[---------------------------------]
[-----------------]
          [-------]
          [---]
            [-]
```

- Look in half of original list
- Decide which half based on size of elem at half point
- What happens if list not ordered?

cannot guarantee correctness!

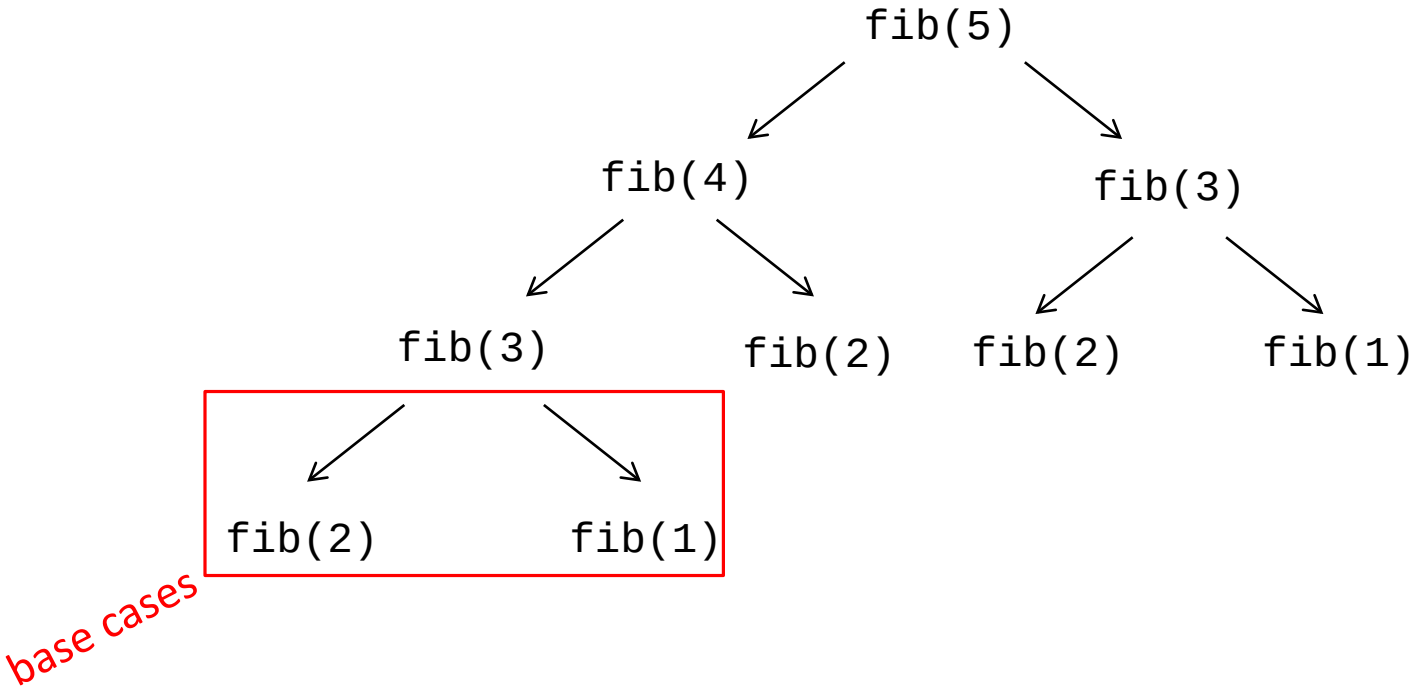
FIBONACCI RECURSIVE CODE (MULTIPLE BASE CASES)

```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1) + fib(n-2)
```

- Two base cases
- Calls itself twice
- This code is inefficient

INEFFICIENT FIBONACCI

`fib(n) = fib(n-1) + fib(n-2)`



- **Recalculating** the same values many times!

Monday

- Dictionaries
- Exceptions
- Assertions
- Debugging
- Microquiz