INTRODUCTION TO PYTHON

LECTURE 2: Loops and Strings

Asem Elshimi

asem.elshimi@austin.utexas.edu

Watch out for keywords!

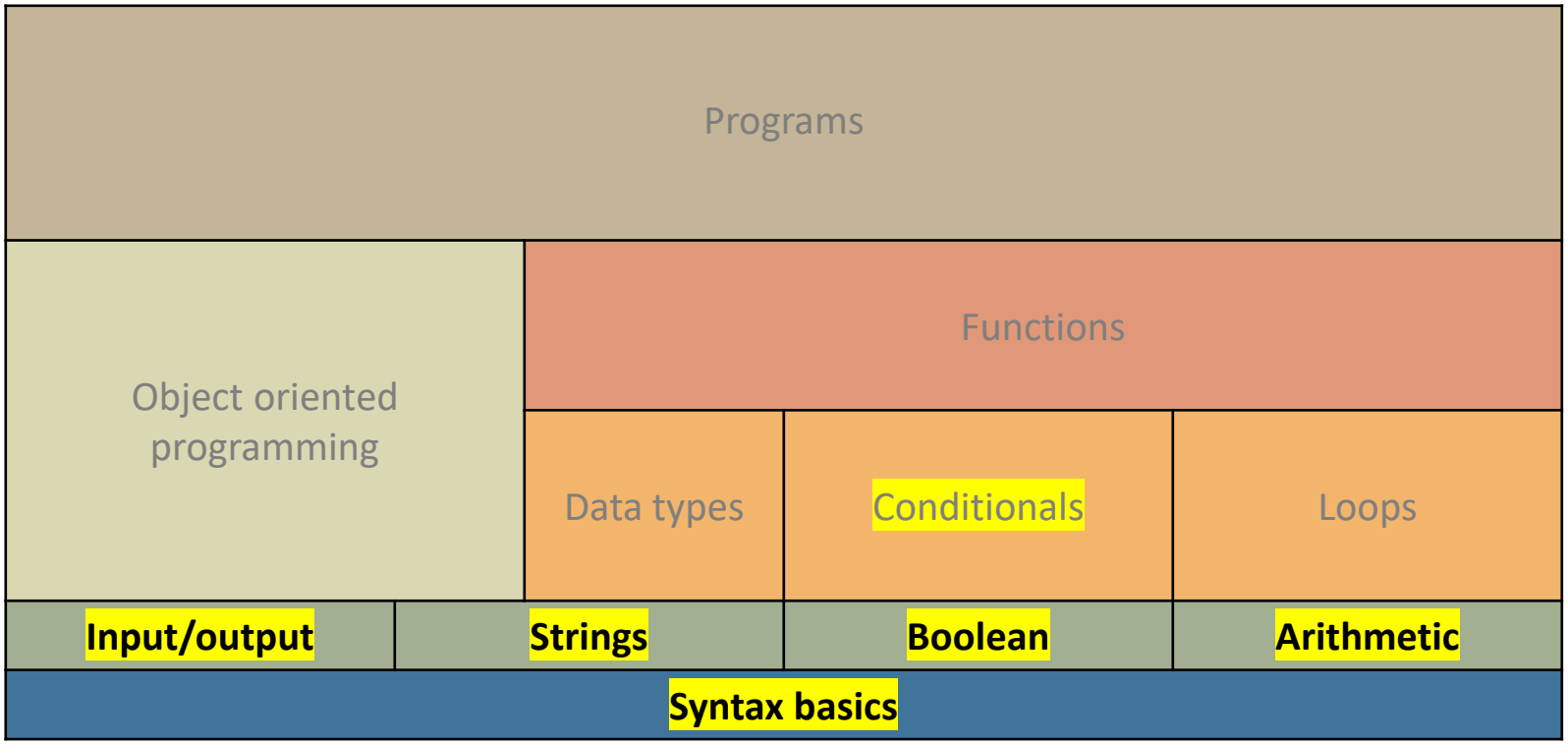
Keywords in Python programming language

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

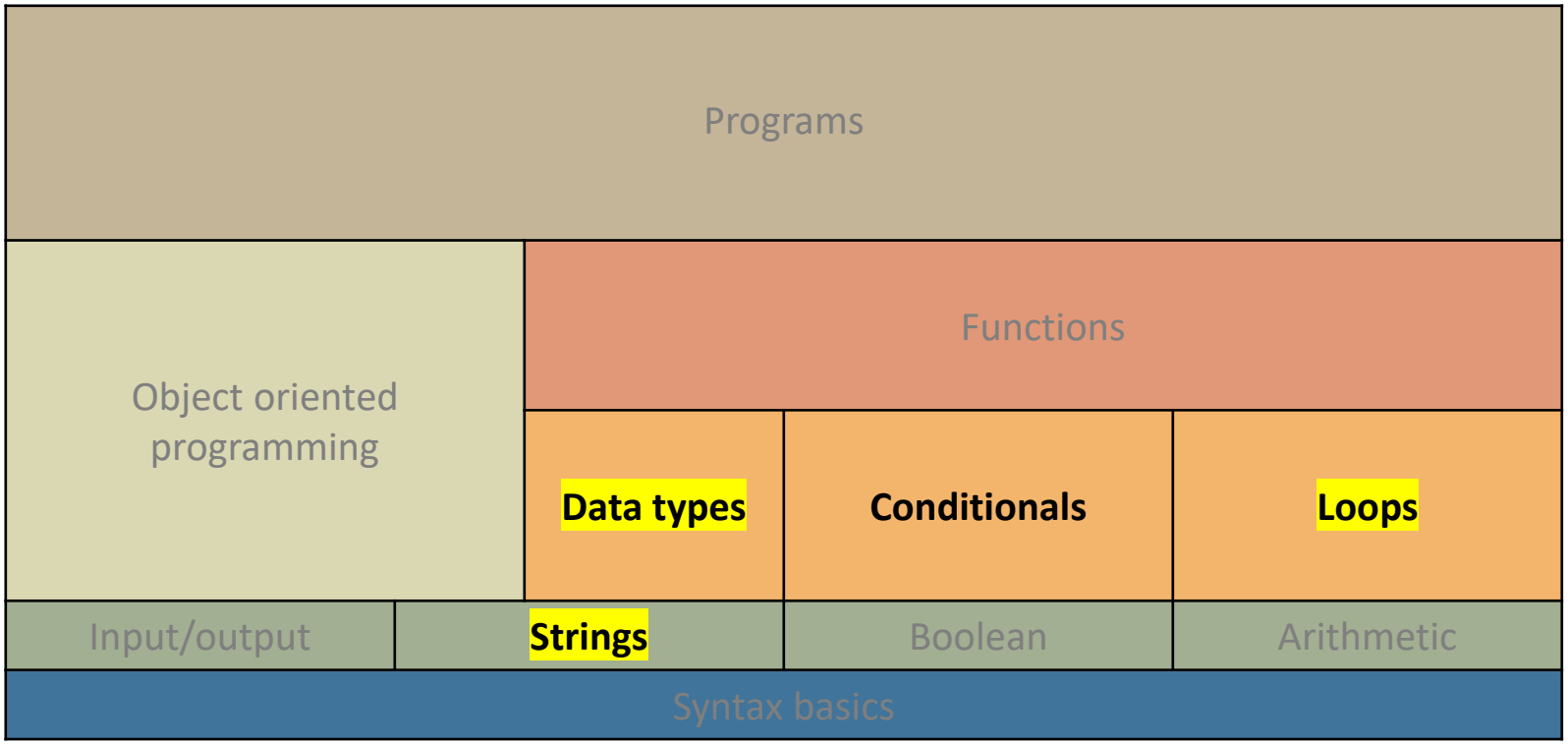
Solution: Use numerated variable names

- Var1, int1, str1, str2, num3

Last week


Programs
Object oriented programming
Functions
Data types
Conditionals
Loops
Input/output
Strings
Boolean
Arithmetic
Syntax basics

Today

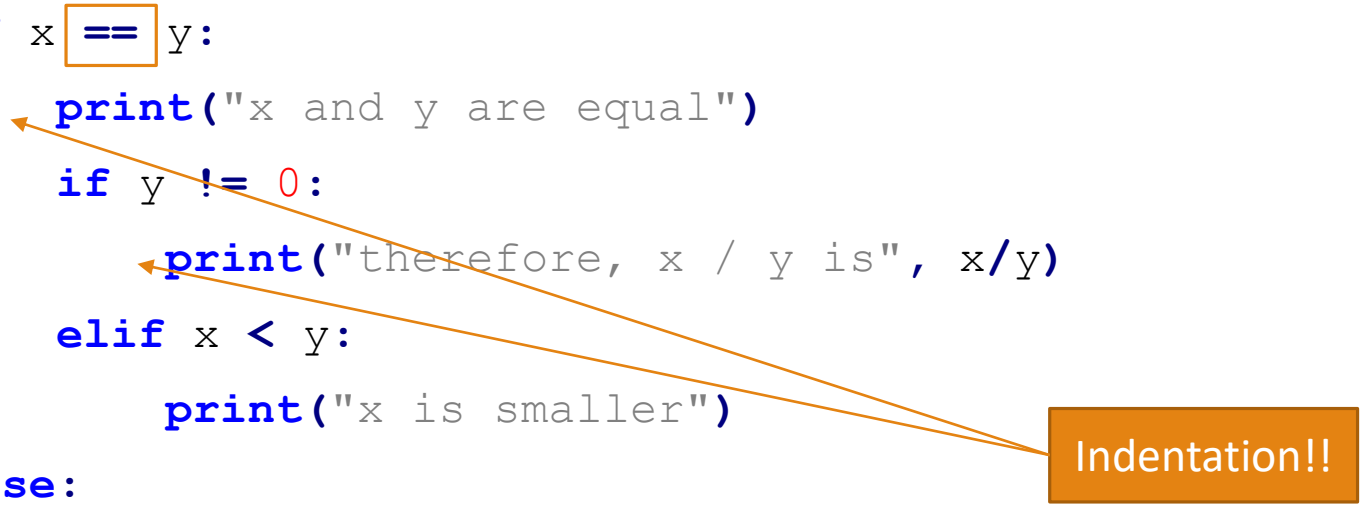

Programs
Object oriented programming
Functions
Data types
Conditionals
Loops
Input/output
Strings
Boolean
Arithmetic
Syntax basics

Recap:

```
pi = 3.14159
# prompt user to enter radius
radius = float(input("radius: "))
# area of circle equation
area = pi*(radius**2)
print("area is equal to: " + str(area))
```

Recap:

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
    elif x < y:
        print("x is smaller")
else:
    print("y is smaller")
    print("thanks!")
```



`while` loops

```
n = 0
while n < 5:
    print(n)
    n = n+1
```

```
0
1
2
3
4
```

Question:

How do you create a running prompt for the user?

```
keeprunning=True
winningNumber=5
while(keeprunning):
    guess=input("Guess a number: ")
    if float(guess)==winningNumber:
        keeprunning=False
        print("Congratulations! Correct guess!")
```

break

```
 i=1
print("entering loop")
while i<20:
    i=i+1
    if i==5:
        print("breaking loop")
        break
    print(i)
print("outside loop")
```

```
entering loop
2
3
4
breaking loop
outside loop
```

Q: What happens with nested loops?

Back to running prompt question:

How do you create a running prompt for the user?

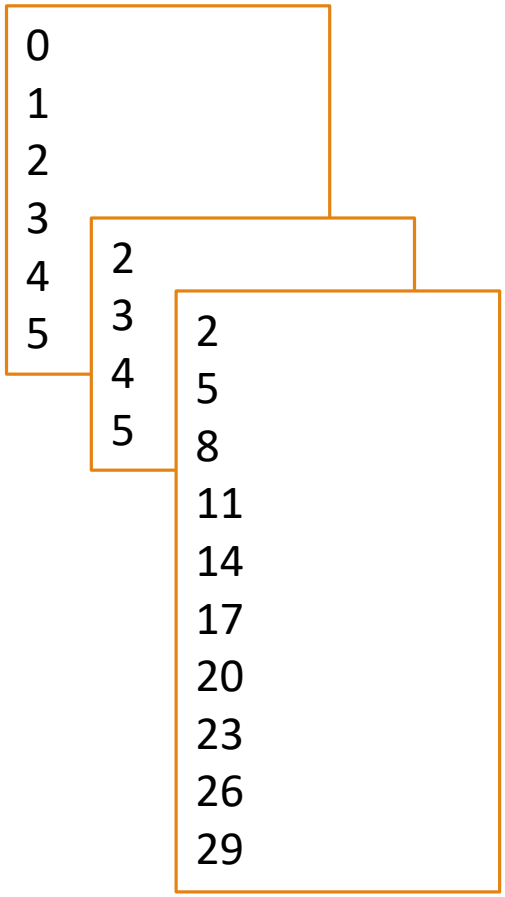
```
keeprunning=True
winningNumber=5
while(True):nning):
    guess=input("Guess a number: ")
    if float(guess)==winningNumber:
        print("Congratulations! Correct guess!")
        break("Congratulations! Correct guess!")
```

`for` loops

```
for x in range(6):
  print(x)
```

```
for x in range(2, 6):
  print(x)
```

```
for x in range(2, 30, 3):
  print(x)
```



```
range(start, stop+1, step)
```

```
range(start=0, stop+1, step=1)
```

range() is an iterable object…

`for` loops

- **know** number of iterations

- **can rewrite** a `for` loop using a `while` loop

`while` loops

- **unbounded** number of iterations

- can use a **counter but must initialize** before loop and increment it inside loop

- **may not be able to rewrite** a `while` loop using a `for` loop

Strings

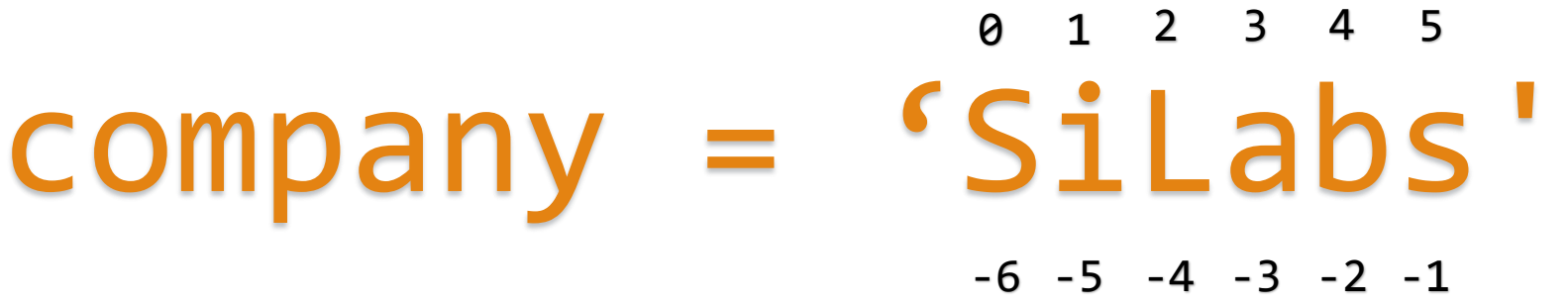
Strings

```
str1 = "Hello"          # => "Hello"
str2 = 'Hello'          # => "Hello"

str1 + ', world!'       # => "Hello, world!"
```

string is an iterable object…

Indexing



Slicing

```
str1 = 'abcdefg'
str2= str1 [0:2]                 # => 'ab'
str2= str1[0:-1]                # => 'abcdef'
str2= str1[0:-1:2]              # => 'ace'

str2= str1[:4]                  # => 'abcd'
str2= str1[:4:2]                # => 'ac'

str2= str1[4:1:-1]              # => 'edc'
str2= str1[-1::-1]              # => 'gfedcba'
str2= str1[::-1]                # => 'gfedcba'
```

String operations

```
s = 'Hello, world! '
var1= len(s)                                  # => 14
str2= s.lower()                               # => 'hello, world! '
str2= s.upper()                               # => 'HELLO, WORLD! '
str2= s.strip()                               # => 'Hello, world!'
var1= s.find('world')                         # => 7 (-1 if not found)
bool1= 'world' in s                           # => True

str2= s.replace('world', 'Texas')             # => 'Hello, Texas! '
str2= s.replace('o', 'u')                     # => 'Hellu, wurld! '
```

Method notation
aka dot notation

Strings and loops

```
s1="abcdiefgh"
for index in range(len(s1)):
    if s1[index]=='i' or s1[index]=='u':
        print("there is an i or u")
```

```
for char in s1:
    if char=='i' or char=='u':
        print("there is an i or u")
```


```
if 'i' in s1 or 'u' in s1:
    print("there is an i or u")
```

Data structures

5 MIN BREAK

Standard library data structures

String
S="Hello"

List
L=[obj1, obj2, obj3,..]

Tuple
T=(obj1, obj2, obj3,..)

Set
St={obj1, obj2, obj3,..}

Dictionary
D={'key1':obj1, 'key2':obj2, 'key3':obj3,..}

Everything in Python is an object. Functions and data structures are objects.

Memory management

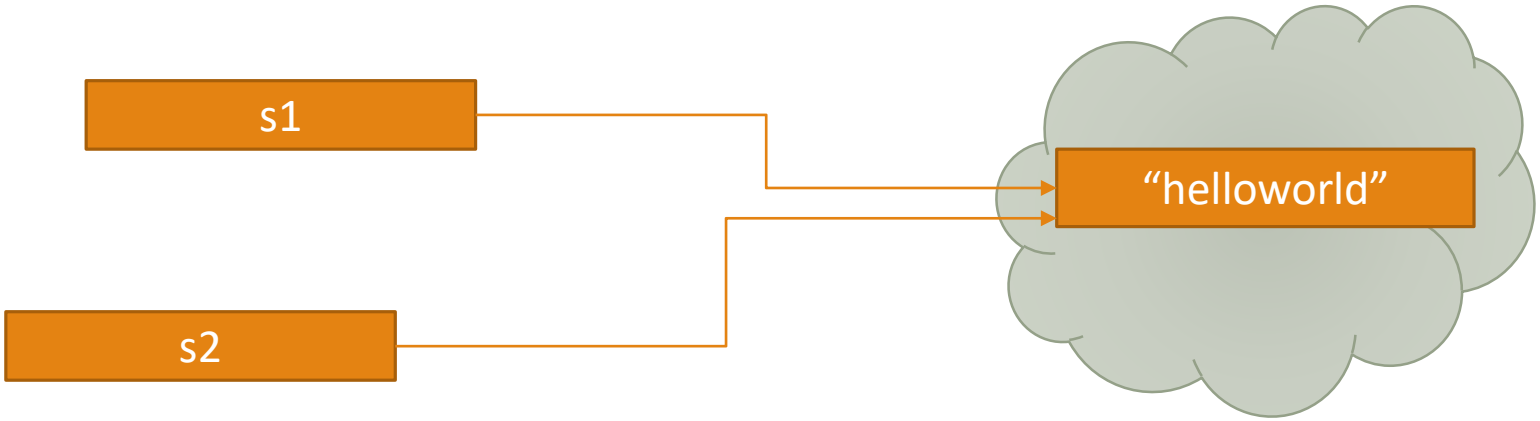
Everything in Python is an object.

Functions and data structures are objects.

Python manages memory for you, yay!

Variables are pointers to memory

When you "copy" a variable, it's just adding another reference to the data



Standard library data structures

Strings	Tuples	Lists	Sets	Dicts
“”	()			
Sequence of characters	Sequence of objects			
Immutable	Immutable			

Immutable will make more sense once we work with **mutable**

Tuples

```
te = ()
t = (2,"mit",3)
t2= t[0]                          #=> 2
t3=(2,"mit",3) + (5,6)            #=> (2,"mit",3,5,6)
t4= t[1:3]                        #=> ("mit",3)
t5= t[1:2]                        #=> ("mit",)
var1=len(t)                       #=> 3
t[1] = 4                          #=> Error!
```

Empty tuple

Slicing.

,) means single element tuple

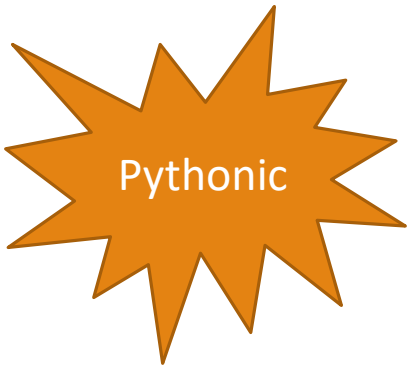
Because a tuple is immutable

Variable swapping

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

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

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



Standard library data structures

Strings	Tuples	Lists	Sets	Dicts
“”	()	[]		
Sequence of characters	Sequence of objects	Sequence of (preferably) same type objects		
Immutable	Immutable	Mutable		

Lists

```
a_list = []          #empty list
L = [2, 'a', 4, [1,2]]
len(L)               #=> 4
var1=L[0]            #=> 2
var2=L[2]+1          #=> 5
var3=L[3]            #=> [1,2]
Var4=L[4]            #=> gives an error
```

Another list

From strings to lists, and back

```
s = "I<3 cs"                    #=> s is a string
L1=list(s)
                        #=> ['I','<','3',' ','c','s']
L2= s.split('<')                #=> ['I', '3 cs']
```

Useful for csv

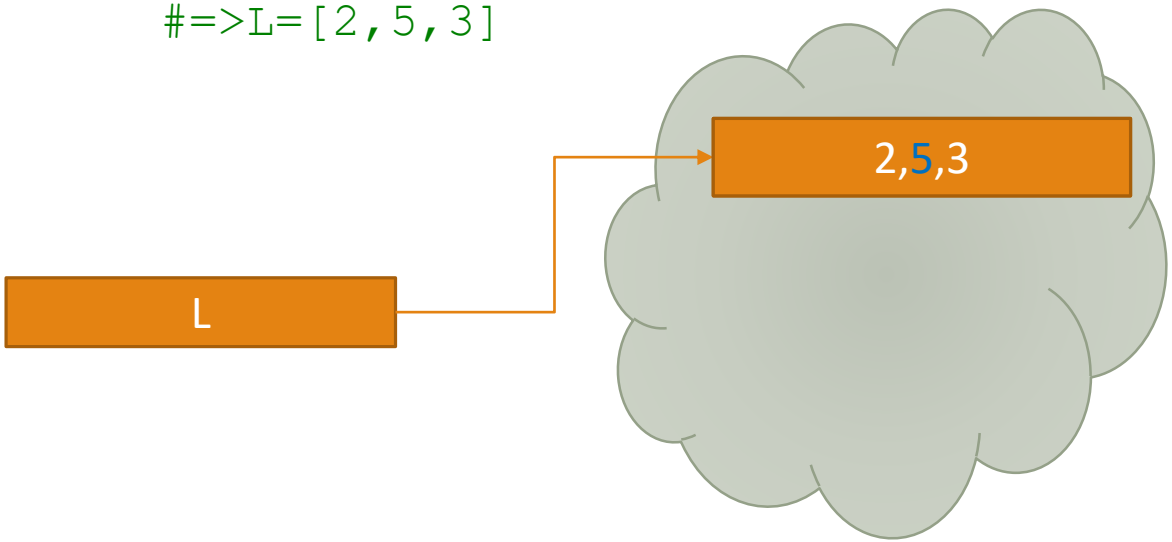
```
L = ['a','b','c']               #=> L is a list
s1 = ''.join(L)                 #=> "abc"
s2 = '_'.join(L)                      #=> "a_b_c"
```

Changing elements

Lists are mutable

- Can change contents of variable without cloning or re-referencing

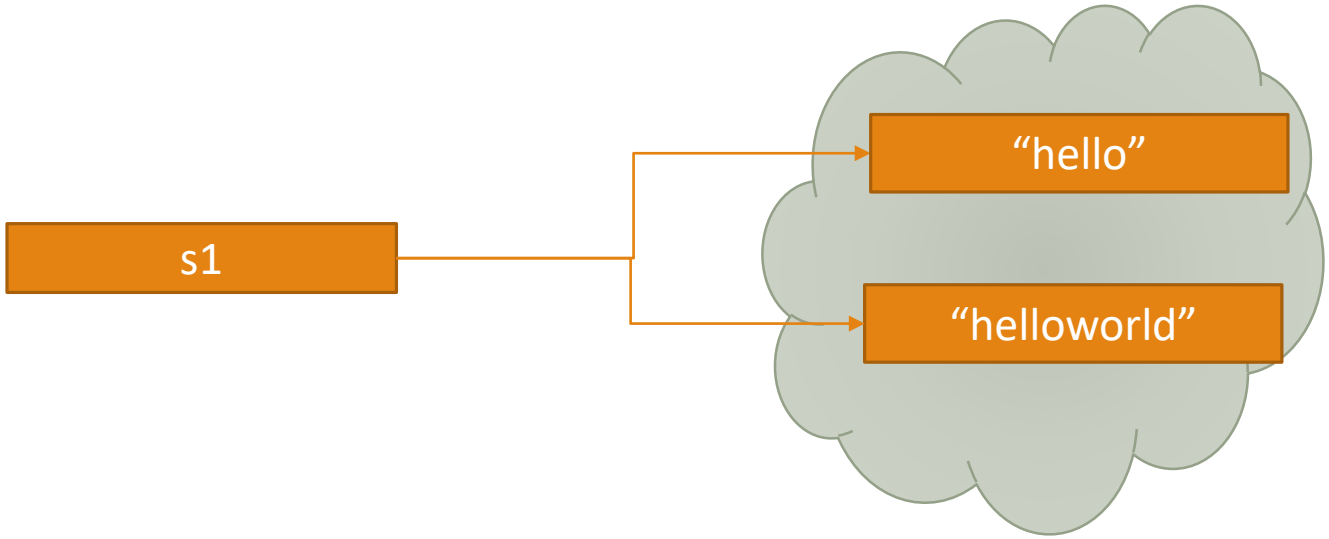
```
L=[2,1,3]
L[1]=5          #=>L=[2,5,3]
```



Q: Can you think of when you might need mutation?

What was this?

```
s1="hello"
s1=s1+"world"      #=>s1="helloworld
```



Re-referencing is different from mutating…

More mutation

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

Method notation aka dot notation

```
L1=[2,1,3]
L2=[4,5,6]
L3=L1+L2                       #=>L3=[2,1,3,5,6]
```

L1 and L2 remain the same

```
L1=[2,1,3]
L2=[4,5,6]
L1=L1+L2                       #=>L1=[2,1,3,5,6]
```

Was this a mutation?

Confused?

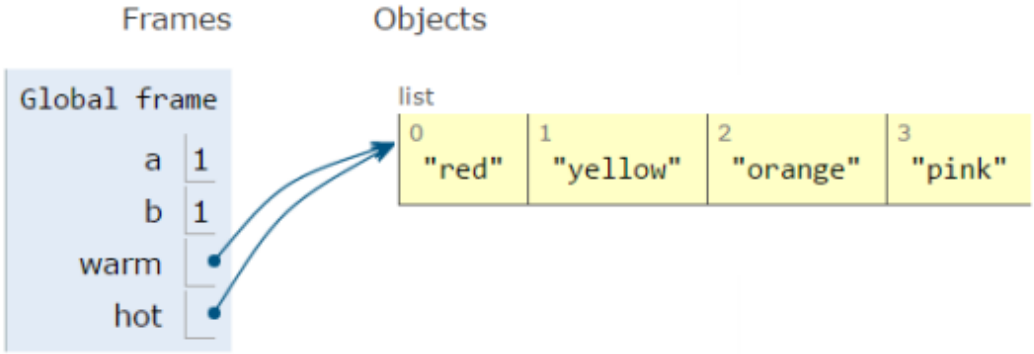
PYTHON TUTOR: HTTPS://GOO.GL/QTD7TQ

Mutation side effects

Aliases

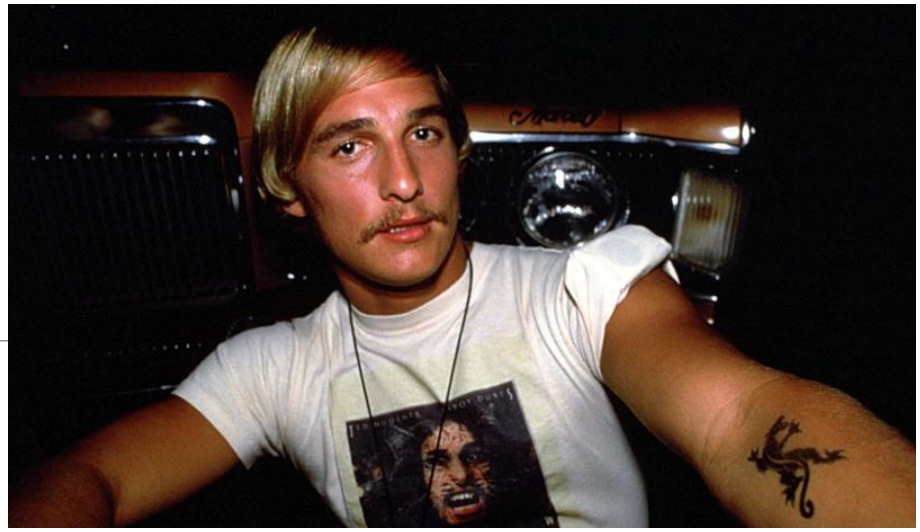
```
5
6  warm = ['red', 'yellow', 'orange']
7  hot = warm
8  hot.append('pink')
9  print(hot)
10 print(warm)
```

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



Hot is an alias of warm- Changing one changes the other

Analogy:



Attributes (=values) :

- Actor, Austinite

Many names
(=variables pointing to attributes)

Mathew.M	Actor	Austinite	rich

Adding a new attribute affects all other names

Muh-kAA-nuh-hee	Actor	Austinite	rich
M.McConaughey	Actor	Austinite	rich

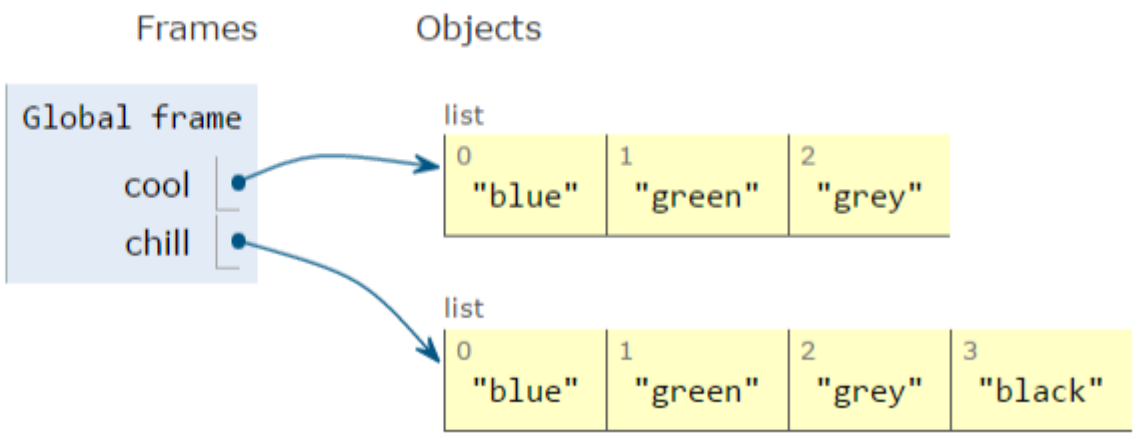
Cloning

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']
```

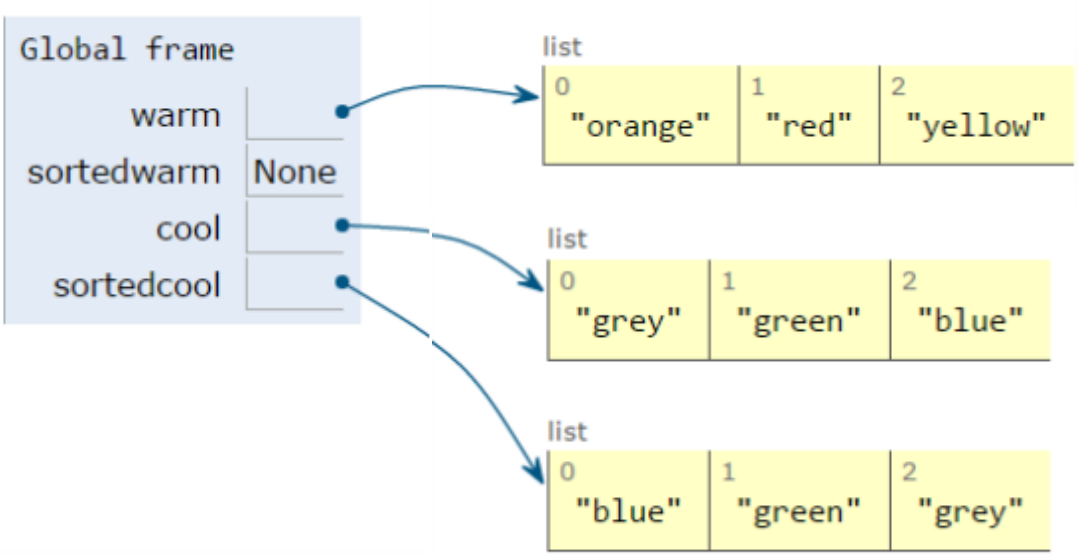


Sorting a list

L.sort() → mutates your list

L2=L.sorted() → creates a sorted clone of the list
(No mutation)

```
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)
```

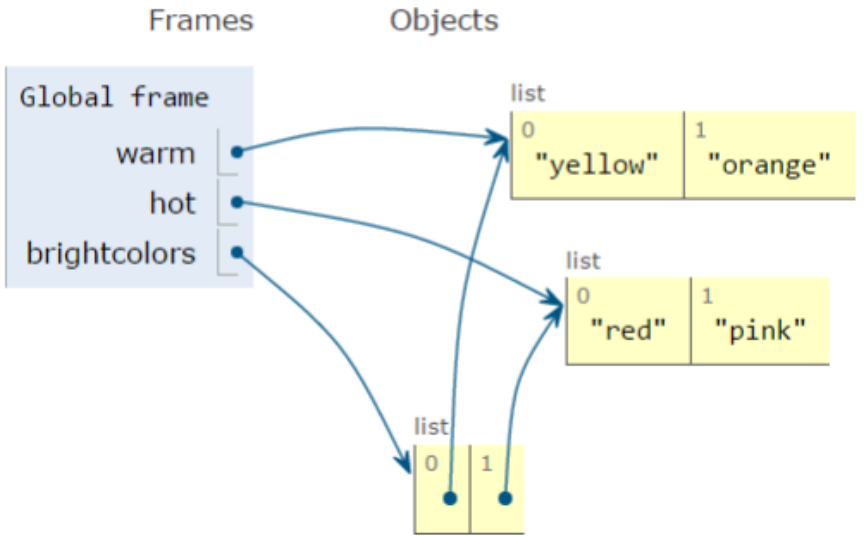


Lists of lists of lists of…

- 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)
```



Mutation and iteration

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
for e in L1:
    if e in L2:
        L1.remo e
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
L1_copy=L1[:]
for e in L1_copy:
    if e in L2:
        L1.remove(e)
```

L1 is [2,3,4] not [3,4] Why?

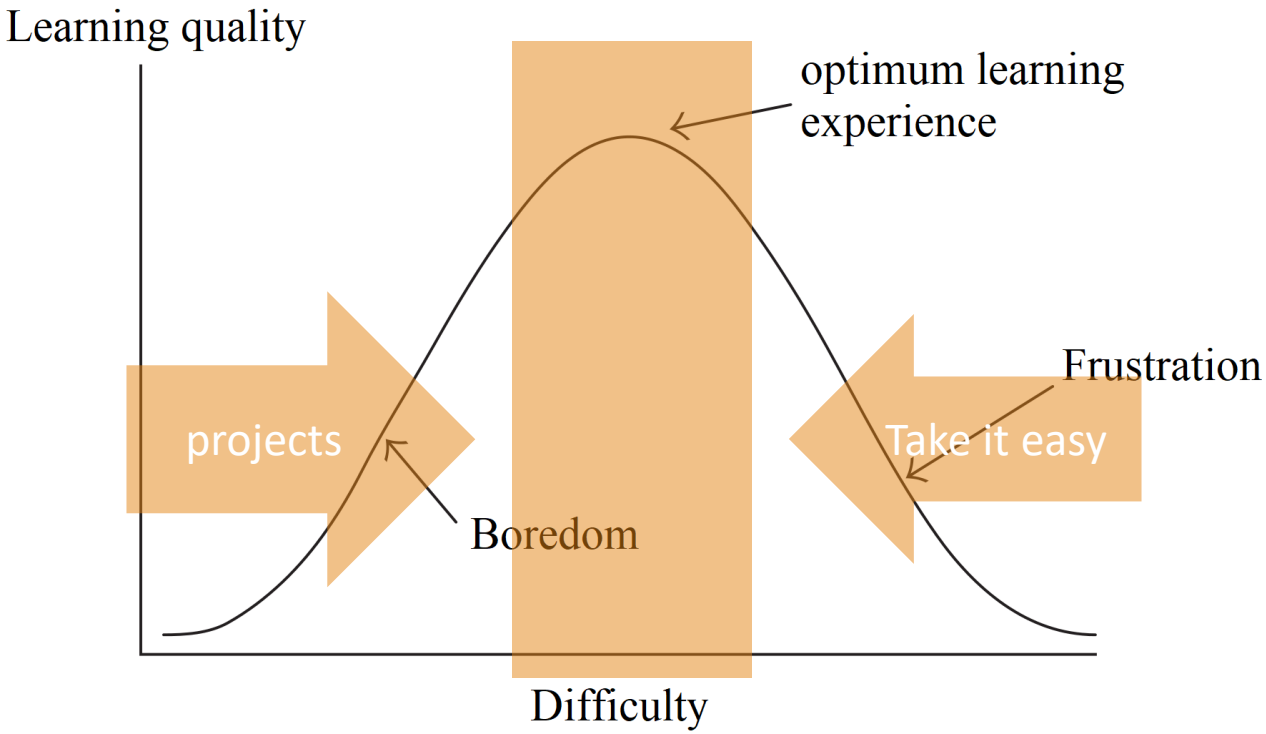
Python uses an internal counter to keep track of index:

- While it is in the loop
- mutating changes the length of the list,
- but Python doesn't update the counter
- loop never sees element 2

Standard library data structures

Strings	Tuples	Lists	Sets	Dicts
“”	()	[]		
Sequence of characters	Sequence of objects	Sequence of (preferably) same type objects		
Immutable	Immutable	Mutable		

Challenging enough?



(Relationship between challenge and learning.)