Lecture 3
Strings and Functions

Dr. Muhammad Jawad Khan

Robotics and Intelligent Machine Engineering Department,
School of Mechanical and Manufacturing,
National University of Sciences and Technology,
Islamabad, Pakistan.

\

STRINGS

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc.
- `len()` is a function used to retrieve the **length** of the string in the parentheses

```
s = "abc"
```

`len(s)` → evaluates to 3

STRINGS

- square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0

index: -3 -2 -1 ← last element always at index -1

`s[0]` → evaluates to "a"

`s[1]` → evaluates to "b"

`s[2]` → evaluates to "c"

`s[3]` → trying to index out of bounds, error

`s[-1]` → evaluates to "c"

`s[-2]` → evaluates to "b"

`s[-3]` → evaluates to "a"

STRINGS

- can **slice** strings using `[start:stop:step]`
- if give two numbers, `[start:stop]`, `step=1` by default
- you can also omit numbers and leave just colons

If unsure what some command does, try it out in your console!

`s = "abcdefgh"`

`s[3:6]` → evaluates to `"def"`, same as `s[3:6:1]`

`s[3:6:2]` → evaluates to `"df"`

`s[::]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

`s[::-1]` → evaluates to `"hgfedbca"`, same as `s[-1:-(len(s)+1):-1]`

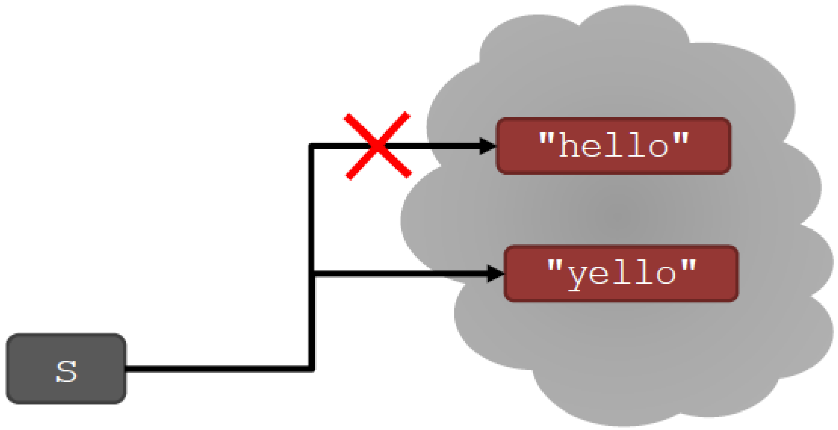
`s[4:1:-2]` → evaluates to `"ec"`

STRINGS

- strings are "**immutable**" – cannot be modified

```
s = "hello"
```

```
s[0] = 'y'                → gives an error
s = 'y'+s[1:len(s)]       → is allowed,
                            s bound to new object
```



for LOOPS RECAP

- `for` loops have a **loop variable** that iterates over a set of values

```
for var in range(4):
    <expressions>
```

→ `var` iterates over values 0,1,2,3

→ expressions inside loop executed with each value for `var`

```
for var in range(4,6):
    <expressions>
```

→ `var` iterates over values 4,5

- `range` is a way to iterate over numbers, but a for loop variable can **iterate over any set of values**, not just numbers!

STRINGS AND LOOPS

- these two code snippets do the same thing
- bottom one is more "pythonic"

```
s = "demo loops"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")

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

CODE EXAMPLE:
ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"
word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

i = 0
while i < len(word):
    char = word[i]
    if char in an_letters:
        print("Give me an " + char + "! " + char)
    else:
        print("Give me a  " + char + "! " + char)
    i += 1
print("What does that spell?")
for i in range(times):
    print(word, "!!!")
```

CODE EXAMPLE:
ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

i = 0
while i < len(word):
    char = word[i]
    if char in an_letters:
        print("Give me an " + char + "! " + char)
    else:
        print("Give me a  " + char + "! " + char)
    i += 1
print("What does that spell?")
for i in range(times):
    print(word, "!!!")
```

```
for char in word:
```

EXERCISE

```
s1 = "mit u rock"
s2 = "i rule mit"
if len(s1) == len(s2):
    for char1 in s1:
        for char2 in s2:
            if char1 == char2:
                print("common letter")
                break
```

GUESS-AND-CHECK

- the process below also called **exhaustive enumeration**

- given a problem...
- you are able to **guess a value** for solution
- you are able to **check if the solution is correct**
- keep guessing until find solution or guessed all values

GUESS-AND-CHECK – cube root

```
cube = 27
#cube = 8120601
for guess in range(cube+1):
    if guess**3 == cube:
        print("Cube root of", cube, "is", guess)
```

GUESS-AND-CHECK – cube root

```
cube = 27
#cube = 8120601
for guess in range(abs(cube)+1):
    # passed all potential cube roots
    if guess**3 >= abs(cube):
        # no need to keep searching
        break
if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')
else:
    if cube < 0:
        guess = -guess
    print('Cube root of ' + str(cube) + ' is ' + str(guess))
```

APPROXIMATE SOLUTIONS

- **good enough** solution
- start with a guess and increment by some **small value**
- keep guessing if `|guess`^{3}`-cube| >= epsilon` for some **small epsilon**

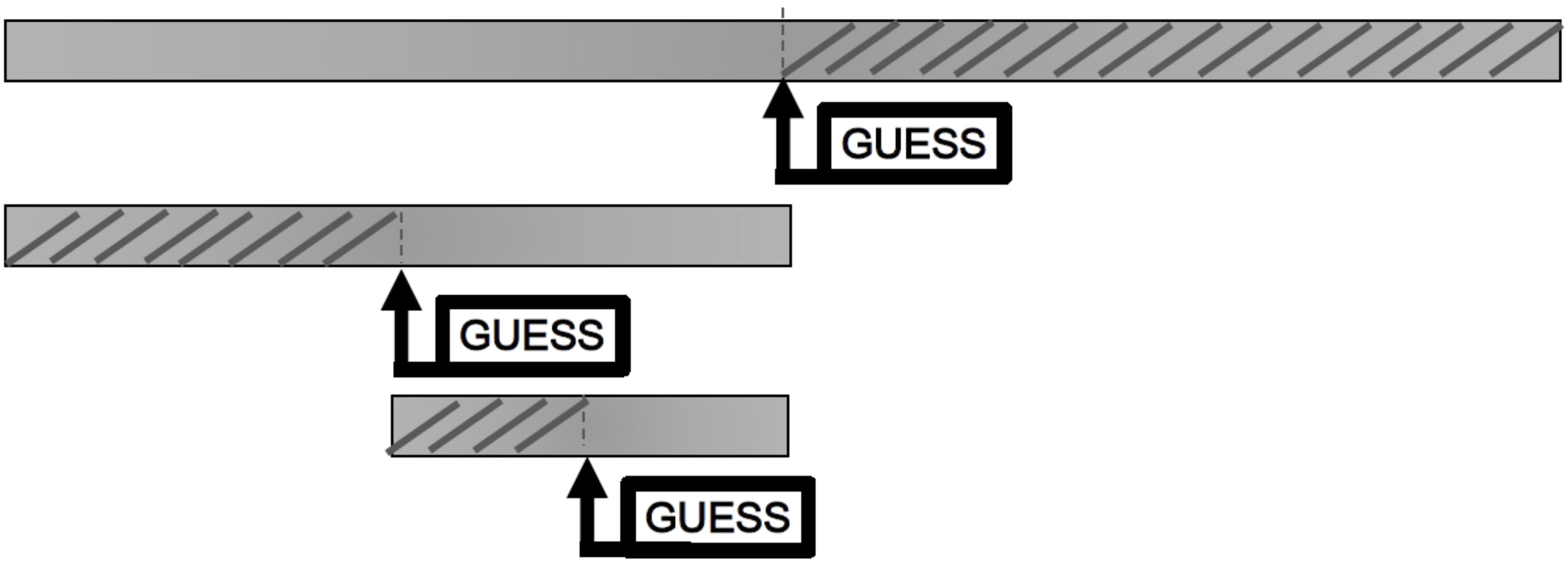
- decreasing increment size → slower program
- increasing epsilon → less accurate answer

APPROXIMATE SOLUTION – cube root

```
cube = 27
#cube = 8120601
#cube = 10000
epsilon = 0.1
guess = 0.0
increment = 0.01
num_guesses = 0
# look for close enough answer and make sure
# didn't accidentally skip the close enough bound
while abs(guess**3 - cube) >= epsilon and guess <= cube:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube, "with these parameters.")
else:
    print(guess, 'is close to the cube root of', cube)
```

BISECTION SEARCH

- half interval each iteration
- new guess is halfway in between
- to illustrate, let's play a game!



```
cube = 27
#cube = 8120601
# won't work with x < 1 because initial upper bound is less than ans
#cube = 0.25
epsilon = 0.01
num_guesses = 0
low = 0
high = cube
guess = (high + low)/2.0
while abs(guess**3 - cube) >= epsilon:
    if guess**3 < cube:
        # look only in upper half search space
        low = guess
    else:
        # look only in lower half search space
        high = guess
    # next guess is halfway in search space
    guess = (high + low)/2.0
    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to the cube root of', cube)
```

BISECTION SEARCH CONVERGENCE

- search space
  - first guess: N/2
  - second guess: N/4
  - kth guess: $N/2^k$
- guess converges on the order of $\log_2 N$ steps
- bisection search works when value of function varies monotonically with input
- code as shown only works for positive cubes > 1 – why?
- challenges → modify to work with negative cubes!
  → modify to work with x < 1!

Assignment 2

$x < 1$

- if $x < 1$, search space is 0 to x but cube root is greater than x and less than 1
- modify the code to choose the search space depending on value of x

HOW DO WE WRITE CODE?

- so far...
  - covered language mechanisms
  - know how to write different files for each computation
  - each file is some piece of code
  - each code is a sequence of instructions
- problems with this approach
  - easy for small-scale problems
  - messy for larger problems
  - hard to keep track of details
  - how do you know the right info is supplied to the right part of code

GOOD PROGRAMMING

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- introduce **functions**
- mechanism to achieve **decomposition** and **abstraction**

EXAMPLE – PROJECTOR

- a projector is a black box
- don't know how it works
- know the interface: input/output
- connect any electronic to it that can communicate with that input
- black box somehow converts image from input source to a wall, magnifying it
- **ABSTRACTION IDEA**: do not need to know how projector works to use it

EXAMPLE – PROJECTOR

- projecting large image for Olympics decomposed into separate tasks for separate projectors
- each projector takes input and produces separate output
- all projectors work together to produce larger image
- **DECOMPOSITION IDEA**: different devices work together to achieve an end goal

APPLY THESE CONCEPTS

TO PROGRAMMING!

CREATE STRUCTURE with DECOMPOSITION

- in projector example, separate devices
- in programming, divide code into **modules**
  - are **self-contained**
  - used to **break up** code
  - intended to be **reusable**
  - keep code **organized**
  - **keep code coherent**
- this lecture, achieve decomposition with **functions**
- in a few weeks, achieve decomposition with **classes**

SUPRESS DETAILS with ABSTRACTION

- in projector example, instructions for how to use it are sufficient, no need to know how to build one
- in programming, think of a piece of code as a **black box**
  - cannot see details
  - do not need to see details
  - do not want to see details
  - hide tedious coding details
- achieve abstraction with **function specifications** or **docstrings**

FUNCTIONS

- write reusable pieces/chunks of code, called **functions**
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

HOW TO WRITE and CALL/INVOKE A FUNCTION

keyword

name

parameters or arguments

specification docstring

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

body

```
is_even(3)
```

later in the code, you call the function using its name and values for parameters

IN THE FUNCTION BODY

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

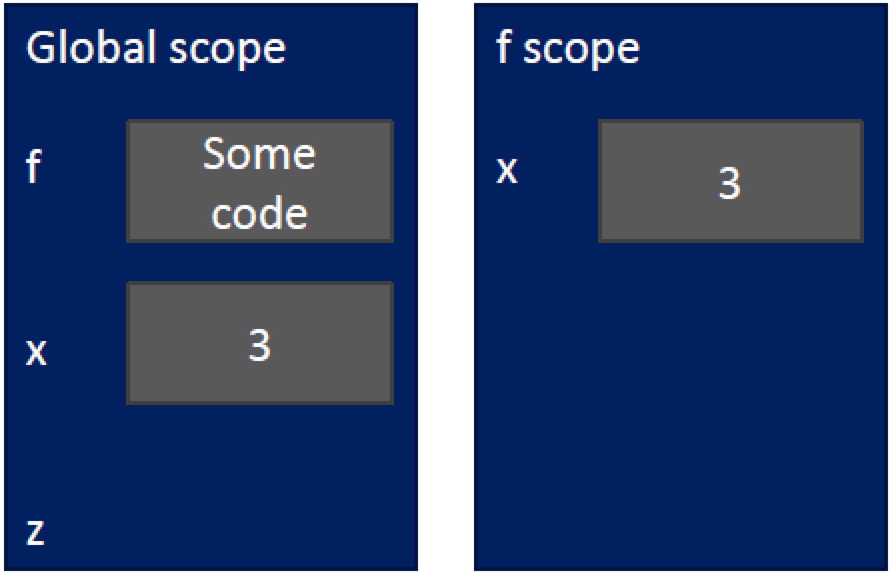
run some commands

keyword

expression to evaluate and return

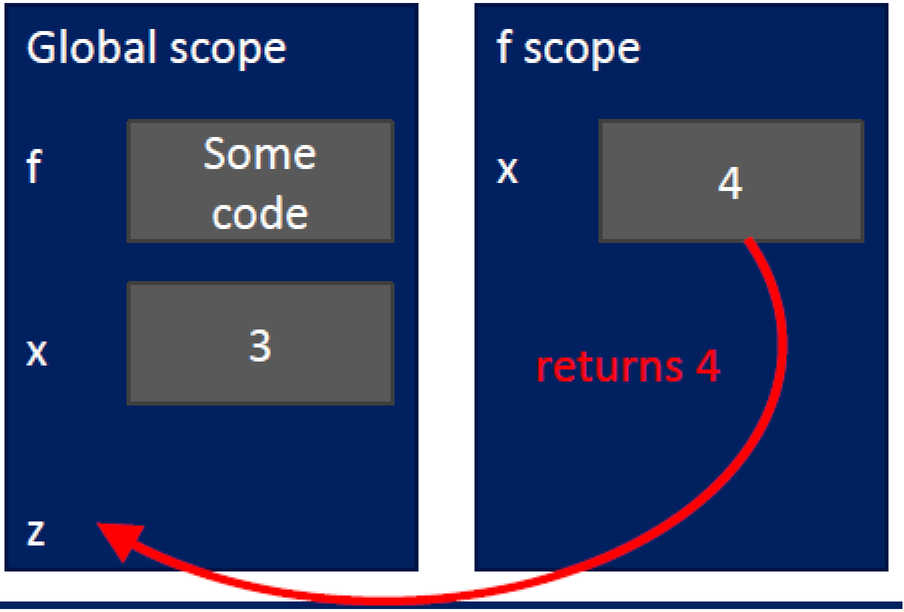
VARIABLE SCOPE

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

x = 3
z = f( x )
```



VARIABLE SCOPE

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

x = 3
z = f( x )
```



ONE WARNING IF NO `return` STATEMENT

```
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

`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

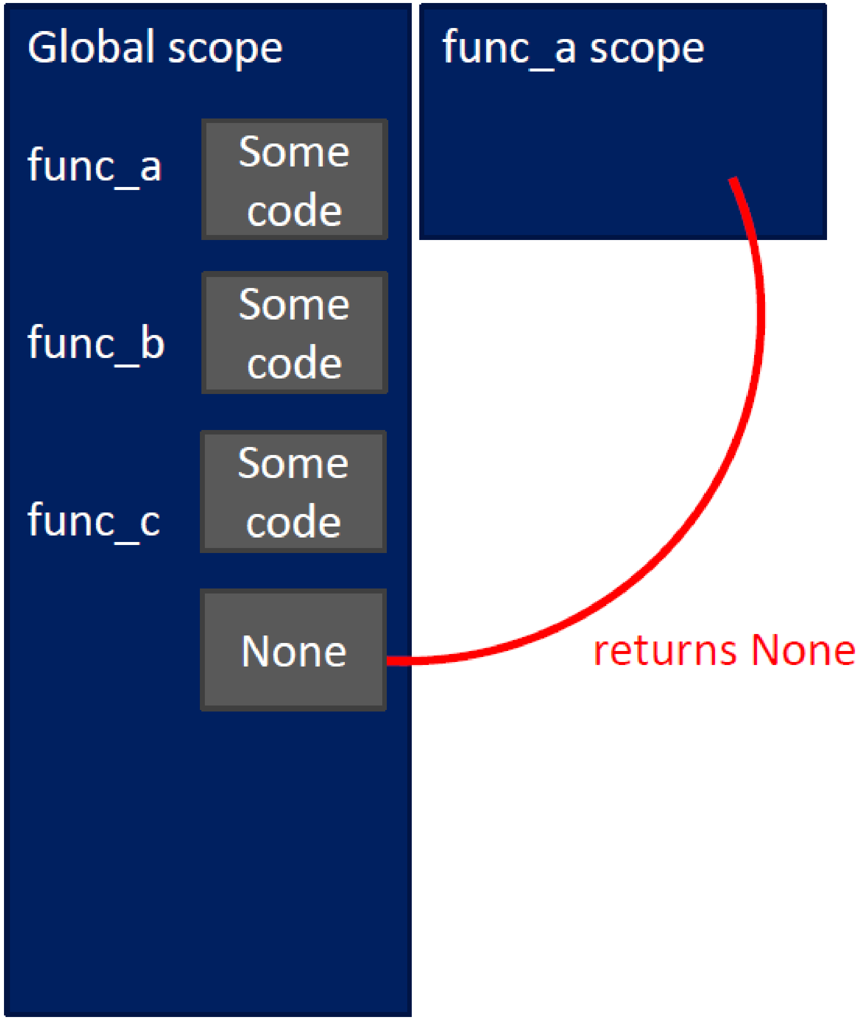
FUNCTIONS AS ARGUMENTS

- arguments can take on any type, even functions

```
def func_a():
    print('inside func_a')

def func_b(y):
    print('inside func_b')
    return y

def func_c(z):
    print('inside func_c')
    return z()

print(func_a())
print(5+func_b(2))
print(func_c(func_a))
```

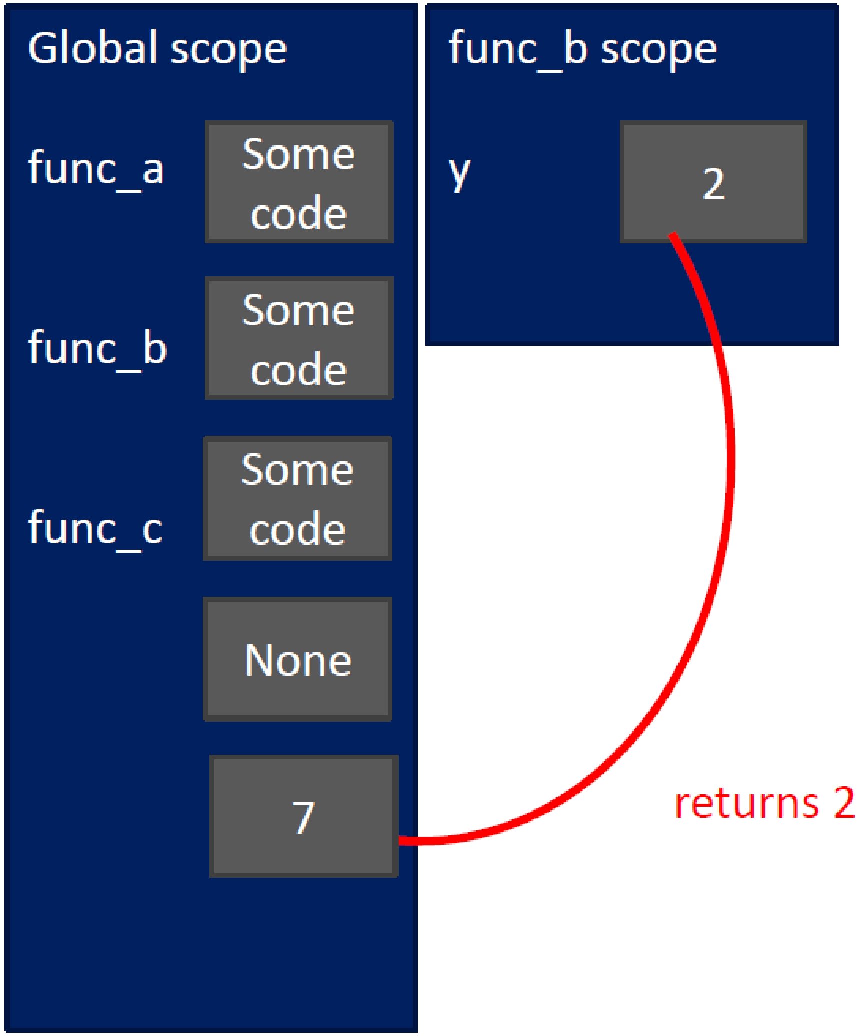
FUNCTIONS AS ARGUMENTS

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



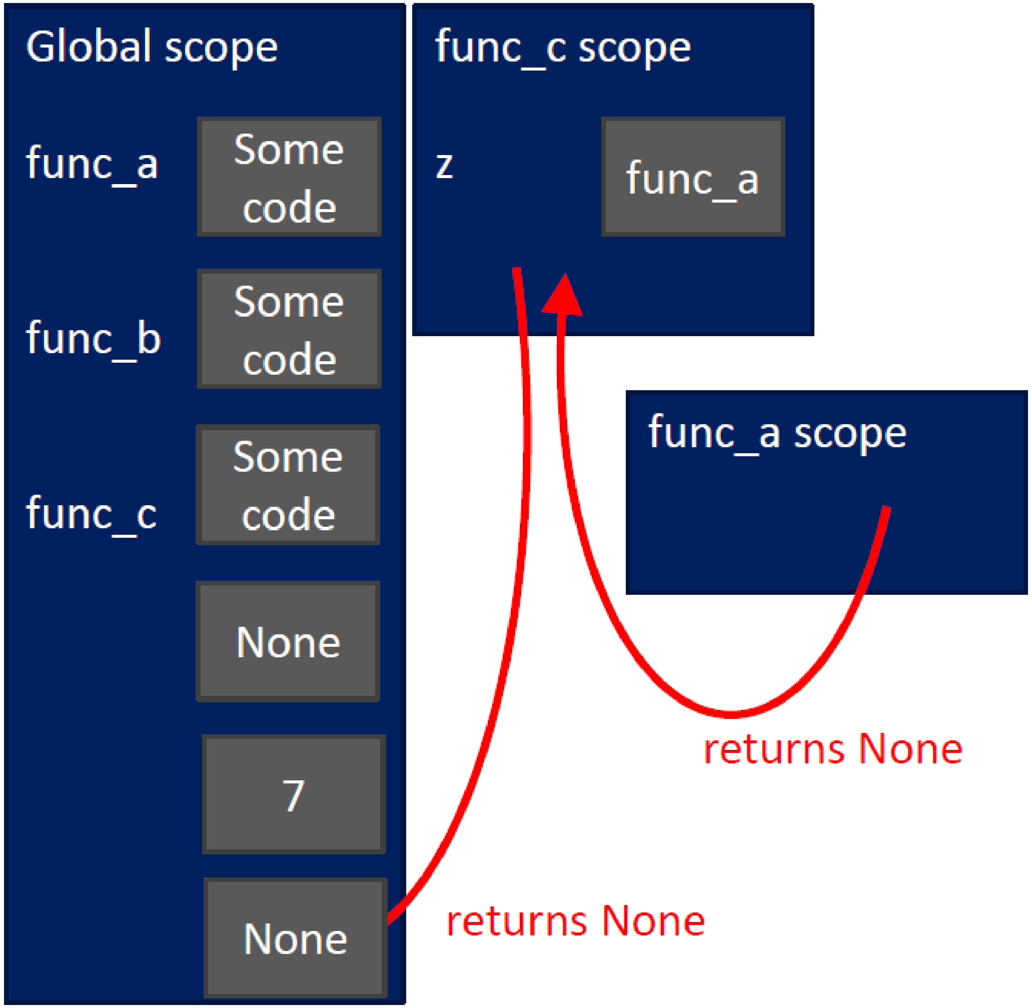
FUNCTIONS AS ARGUMENTS

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



FUNCTIONS AS ARGUMENTS

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



SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can using **global variables**, but frowned upon

```
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

```
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

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

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can using **global variables**, but frowned upon

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

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

```
def h(y):
    pass
    #x += 1 #leads to an error without
# line `global x` inside h
x = 5
h(x)
print(x)
```

HARDER SCOPE EXAMPLE

IMPORTANT
and
TRICKY!

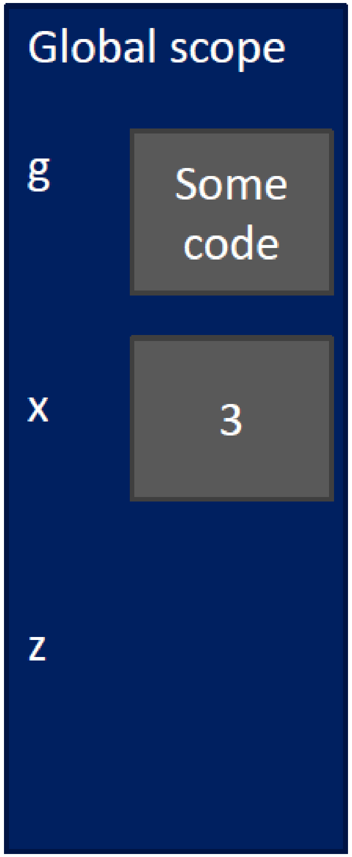
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

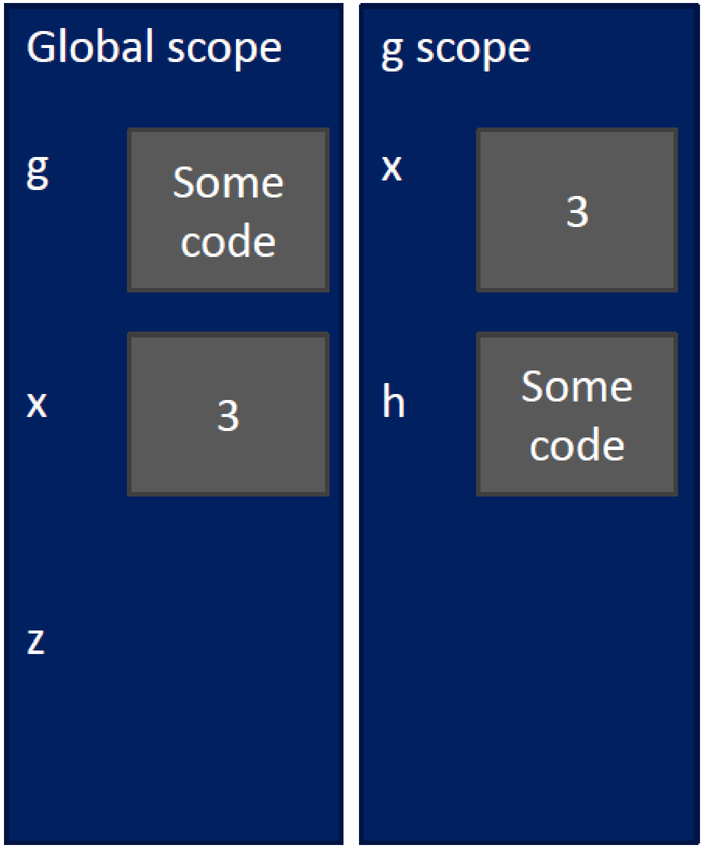
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```

Some code



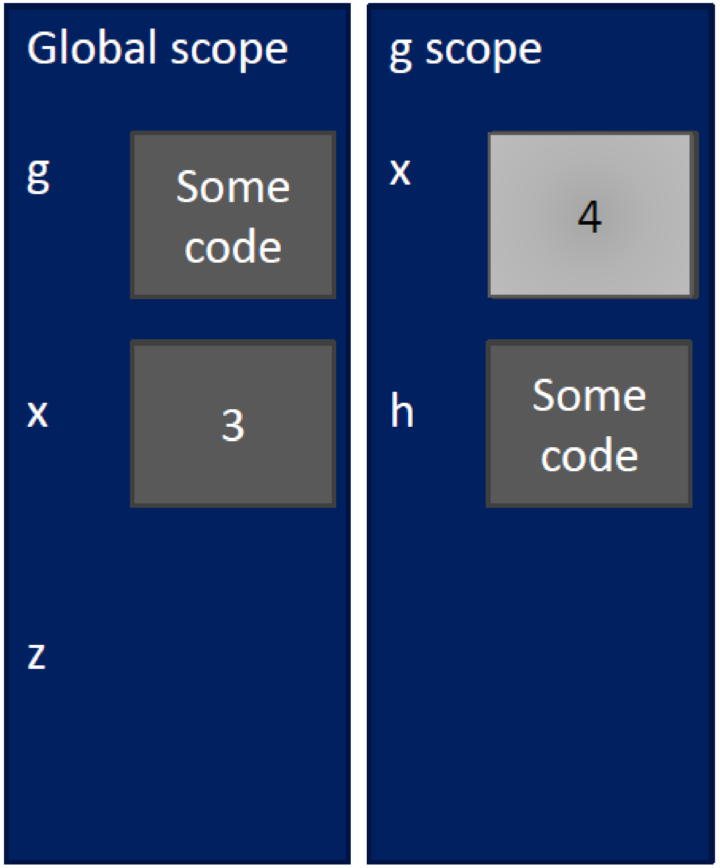
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



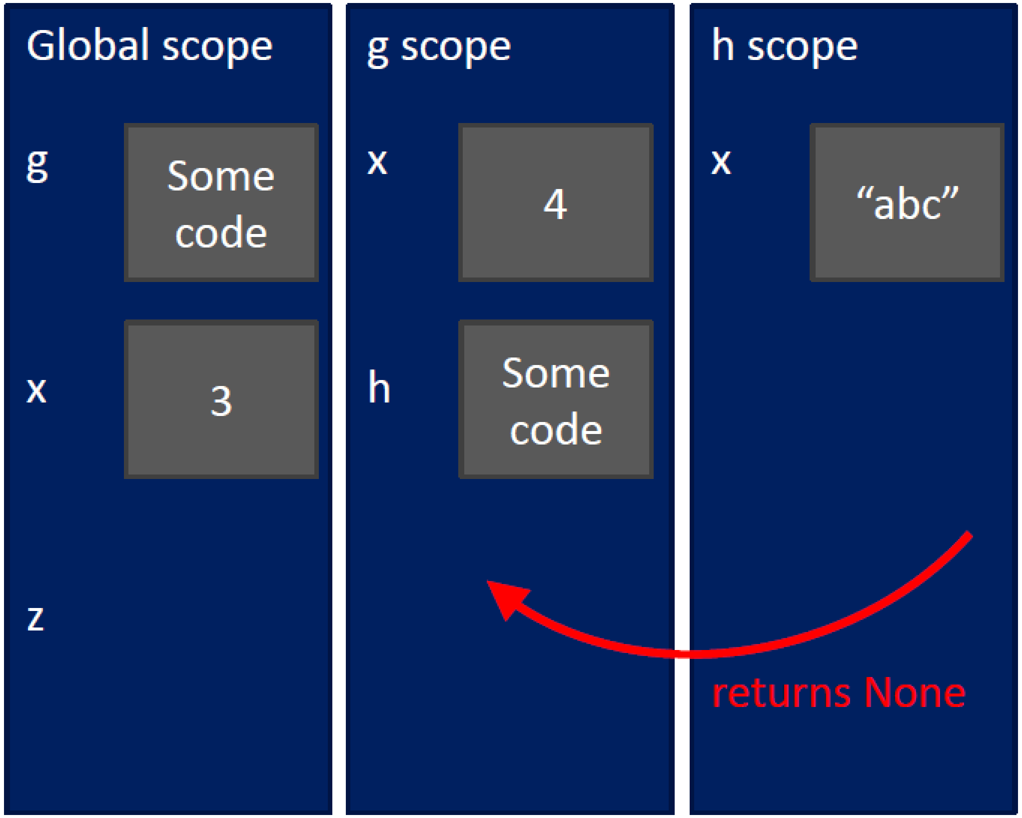
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



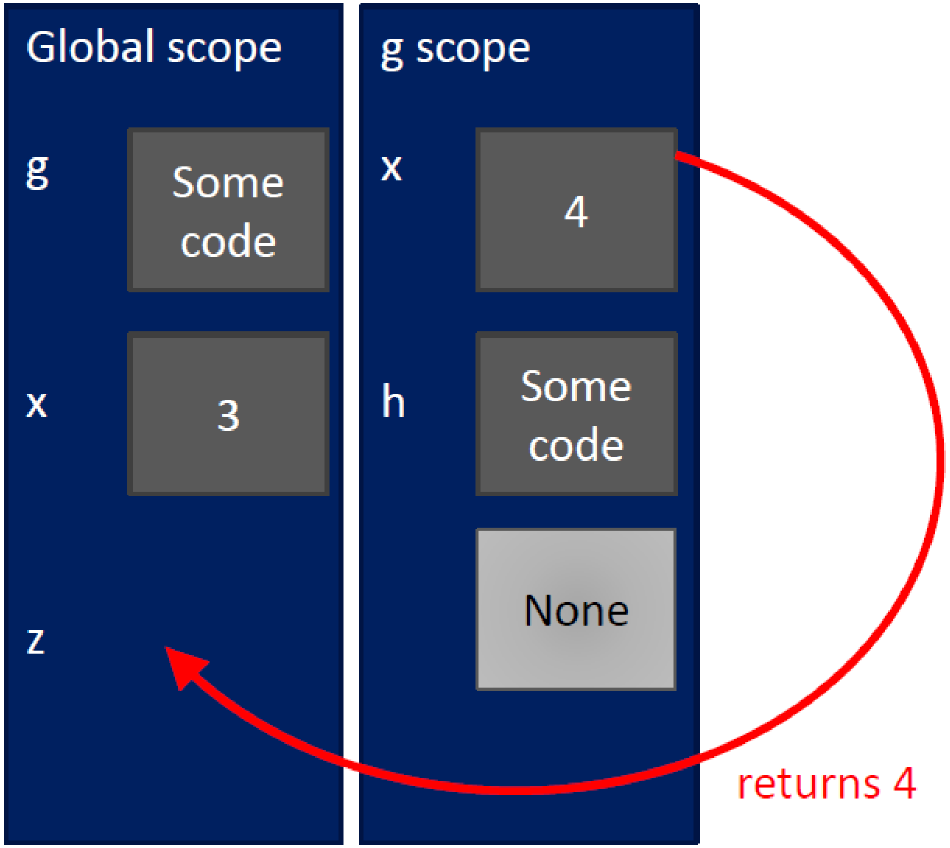
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



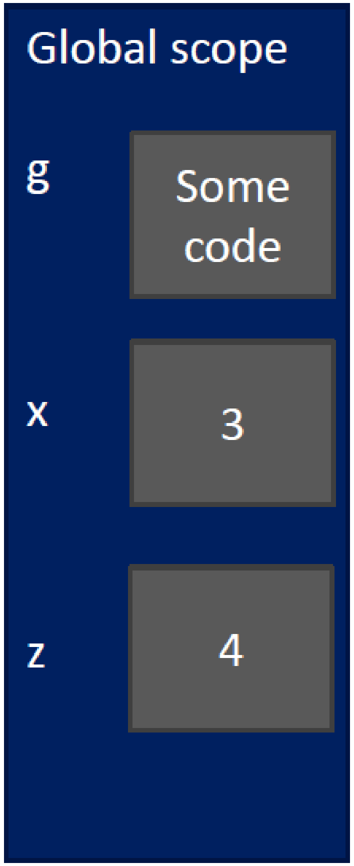
SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



SCOPE DETAILS

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



DECOMPOSITION & ABSTRACTION

- powerful together
- code can be used many times but only has to be debugged once!

LAST TIME

- functions
- decomposition – create structure
- abstraction – suppress details
- from now on will be using functions a lot

TODAY

- have seen variable types: `int`, `float`, `bool`, `string`
- introduce new **compound data types**
  - tuples
  - lists
- idea of aliasing
- idea of mutability
- idea of cloning

TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

remember strings?

```
te = ()
```

empty tuple

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

`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

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

extra comma means a tuple with one element

TUPLES

- conveniently used to **swap** variable values

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

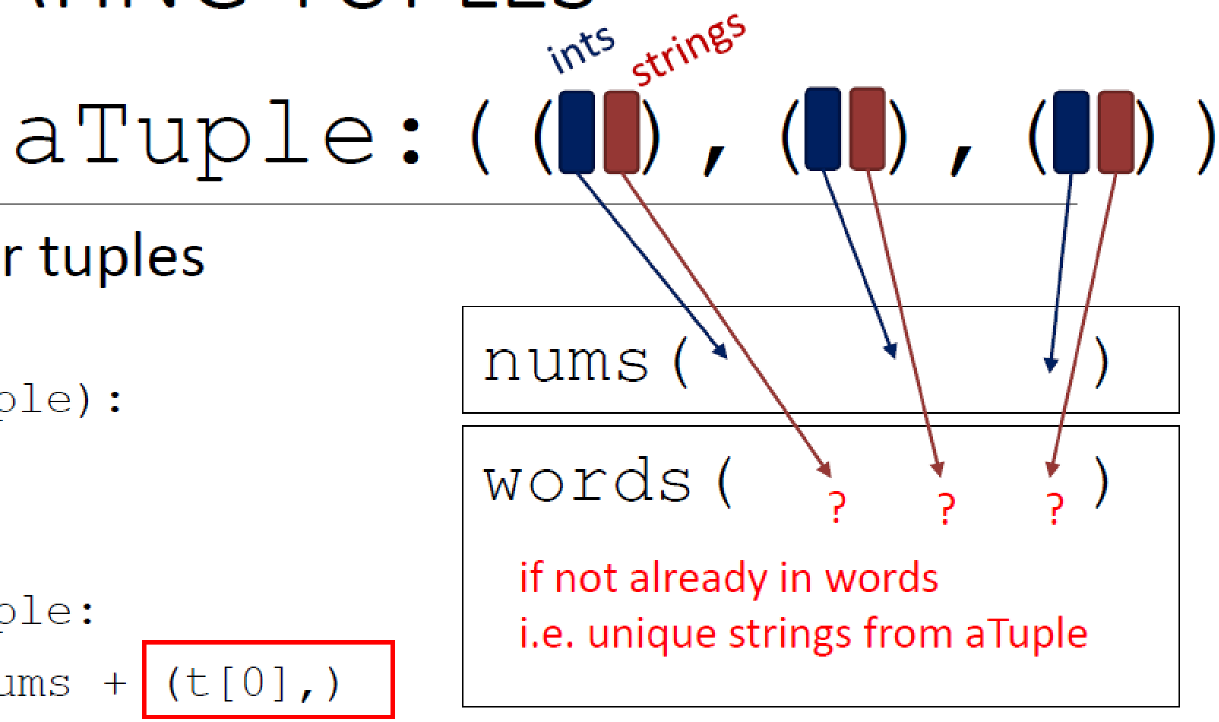
```
temp = x
x = y
y = temp
```

```
(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(5,3)
print(quot)
print(rem)
```

MANIPULATING TUPLES



- can **iterate** over tuples

```
def get_data(aTuple):
    nums = ()
    words = ()
    for t in aTuple:
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words + (t[1],)
    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

empty tuple

singleton tuple

```
def get_data(aTuple):
nums = ()    # empty tuple
    words = ()
    for t in aTuple:
        # concatenating with a singleton tuple
        nums = nums + (t[0],)
        # only add words haven't added before
        if t[1] not in words:
            words = words + (t[1],)
    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

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
```

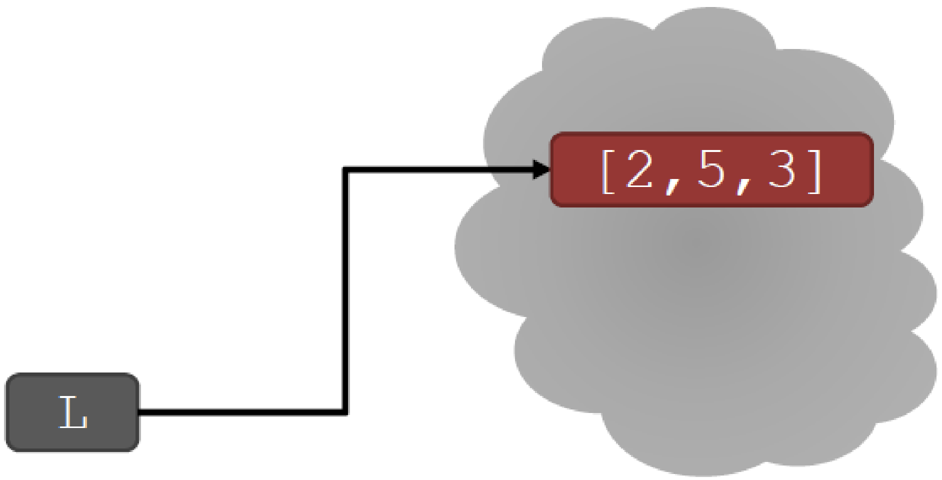
`L[i-1]` → evaluates to 'a' since `L[1]='a'` above

CHANGING ELEMENTS

- 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`



ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

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

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

like strings, can iterate over list elements directly

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

```python
def sum_elem_method1(L):
  total = 0
  for i in range(len(L)):
      total += L[i]
  return total
```

```python
def sum_elem_method2(L):
    total = 0
    for i in L:
        total += i
    return total
```

```python
print(sum_elem_method1([1,2,3,4]))
print(sum_elem_method2([1,2,3,4]))
```

OPERATIONS ON LISTS - ADD

- **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

OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2         →  L3 is [2,1,3,4,5,6]
                        L1, L2 unchanged

L1.extend([0,6])     →  mutated L1 to [2,1,3,0,6]
```

OPERATIONS 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]
print(L.pop())           → returns 0 and mutates L = [1,3,7]
```

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

IMPORTANT
and
TRICKY!

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
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

AN ANALOGY

- attributes of a person
  - singer, rich
- he is known by many names
- all nicknames point to the **same person**
  - add new attribute to **one nickname** ...

Justin Bieber	singer	rich	troublemaker

  - ... **all his nicknames** refer to old attributes AND all new ones

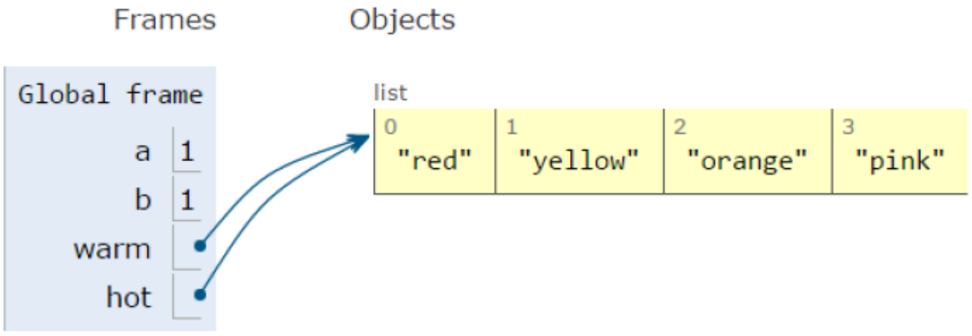
The Bieb	singer	rich	troublemaker
JBeebs	singer	rich	troublemaker

ALIASES

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

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
hot.append('pink')
print(hot)
print(warm)
```

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

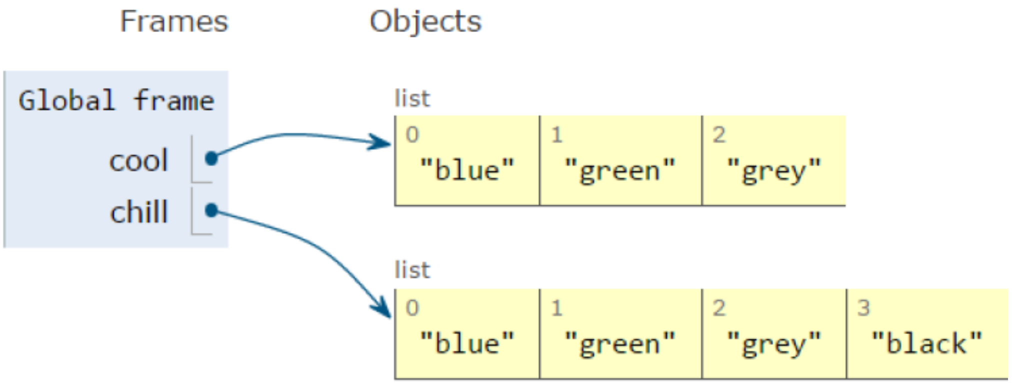


CLONING A LIST

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

```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

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

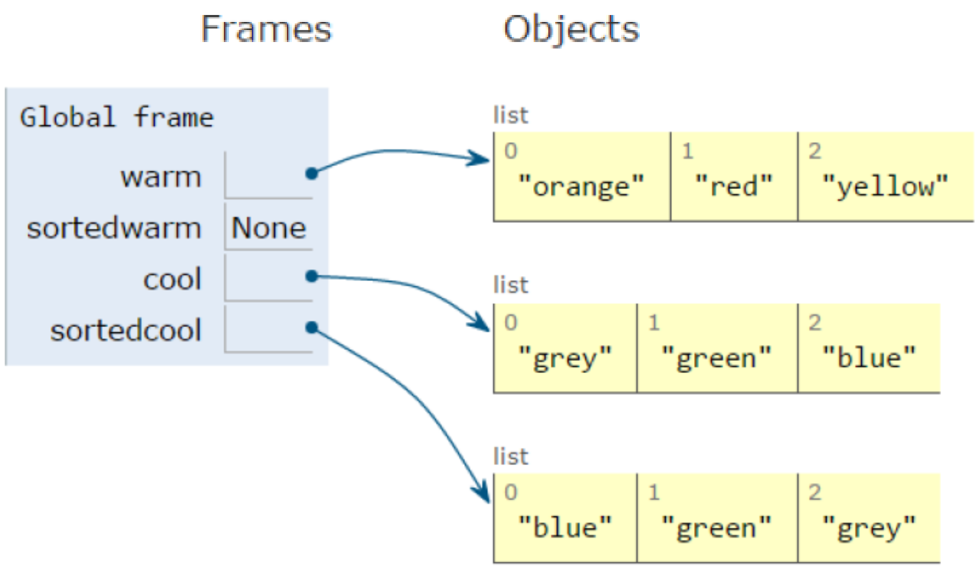


SORTING LISTS

- 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']
```

```python
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

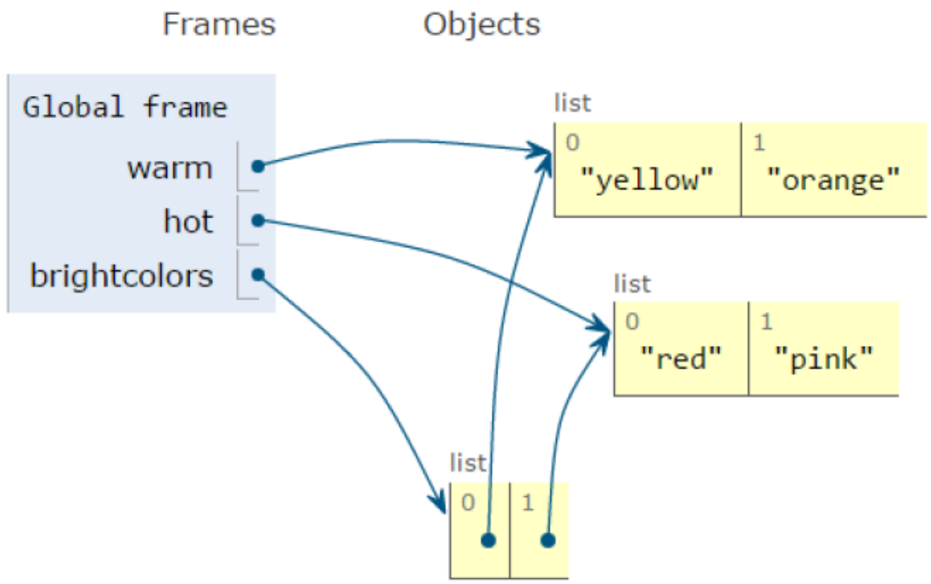


LISTS OF LISTS OF LISTS OF....

- can have **nested** lists
- side effects still possible after mutation

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

```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



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)
```

clone list first, note that `L1_copy = L1` does NOT clone

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

- `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

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



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

```python
def remove_dups_new(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_new(L1, L2)
print(L1, L2)
```