Branching, Iterations, Strings, Functions, Lists, Dictionaries, Debugging, Testing, Exceptions

AF3214 Recitation 2

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **concatenate** strings

```
name = "trump"

greet = hi + name #put strings together, no space

greeting = hi + " " + name
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3 #star operator, repeat
```

INPUT/OUTPUT: `print`

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
x_str = str(x) #casting 1 to a string
print("my fav num is", x, ".", "x =", x)#auto add space
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: `input("")`

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value (string object) to a variable

```
text = input("Type anything... ")#propmt user with sth
print(5*text)
```

- `input` **gives you a string** so must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

COMPARISON OPERATORS ON `int, float, string`

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

`str(i) > str(j)`, `lexicographically (alphabetical order)`

LOGIC OPERATORS ON bools

- `a` and `b` are variable names (with Boolean values)

`not a` → `True` if `a` is `False`
`False` if `a` is `True`

`a and b` → `True` if both are `True`

`a or b` → `True` if either or both are `True`

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

CONTROL FLOW - BRANCHING

CONTROL FLOW - BRANCHING

```
if <condition>:
    <expression>
    <expression>
    ...
```

code block denoted by indentation

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

Three ways to add control flow

Enter the very first one that's true

Never enter more than one code block

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`

INDENTATION

- matters in Python: indent with code block
- how you denote blocks of code

```
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!")
```

= vs ==

```
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!")
```

What if x = y here?

get a SyntaxError

CONTROL FLOW: `while` LOOPS

`while <condition>:` true or false

```
    <expression>
    <expression>
    ...
```

code block denoted by indentation

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

CONTROL FLOW: `while` and `for` LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1
```

not sure about # of iterations

```
# shortcut with for loop
for n in range(5):
    print(n)
```

sure about # of iterations

CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the previous value + 1
- etc.

range(start,stop,step)

- default values are `start = 0` and `step = 1` and optional
- loop until value is `stop - 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

open-close parentheses with comma in between

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

(5, 11, -2)

(11, 5, -2)

If the step is negative, the range decreases from start down to stop. Numbers reaching or beyond the stop are omitted. For example, 5 is beyond 11 when step is decreasing. Thus, the for loop is invalid. `mysum` takes the original value and gets printed 0.
But the green range works and prints 27.

`break` STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

`break` STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1
print(mysum)
```

- what happens in this program?

`for` VS `while` LOOPS

`for` loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

`while` loops

- **unbounded** number of iterations
- can **end early** via `break`
- 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 (i.e., user input)

STRINGS

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

`s = "abc"` count how many characters in the string

`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

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

the way indexing to a string is with []

```
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** (go halfway into) strings using `[start:stop:step]`
  go up until stop+1
- if give two numbers, `[start:stop]`, `step=1` by default
- you can also omit numbers and leave just colons

```
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]`
inverse of your string

`s[4:1:-2]` → evaluates to `"ec"`

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

STRINGS

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

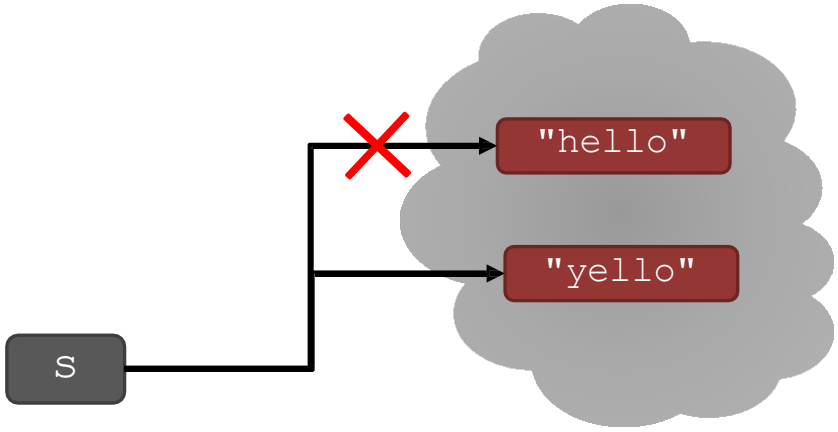
```
s = "hello"
```

```
s[0] = 'y'
```

→ gives an type 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>
```

all the way up until

→ `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 = "abcdefgh"
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:

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Repeat times (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:
```

Function

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- structure your program such that you write nice coherent code, reusable code by hiding away some of the details in your code
- to do that, we need **functions**
- mechanism to achieve **decomposition** and **abstraction**

EXAMPLE – PROJECTOR

- a projector is a black box(resistors, fan, light bulb, lens, casing, other parts)
- don't know how to build, but a fully assembled one?
- 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; feed it different inputs, does exactly the same thing behind the scenes, but produce different output for each one of these inputs

CREATE STRUCTURE with DECOMPOSITION

- in projector example, separate devices
- in programming, divide code into **modules**
  - are **self-contained** (mini programs-feed inputs, do tasks, return sth back)
  - used to **break up** code
  - intended to be **reusable** (module to reuse many times with different inputs)
  - keep code **organized**
  - **keep code coherent**

benefits

- achieve decomposition with **functions**
- so decomposition is creating structure in your code

IDEA for 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** (what are inputs, what it does, what are outputs)

a string used to document a Python module, class, function or method

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, abstraction)
  - 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 #T or F
```

body

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

```
is_even(3)
```

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
```

module/mini program

run some commands

keyword

expression to evaluate and return

VARIABLE SCOPE(environment)

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

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

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

formal parameter

actual parameter

Function definition

Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

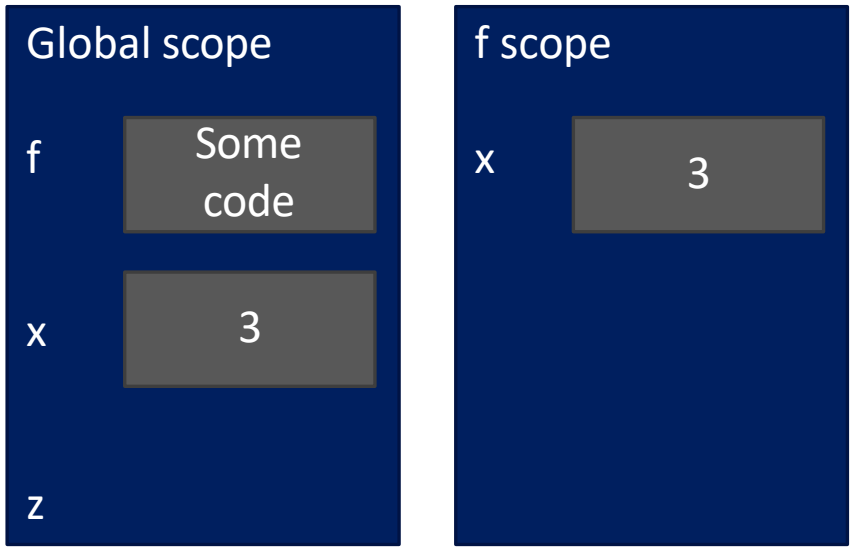
VARIABLE SCOPE

function definition

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


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

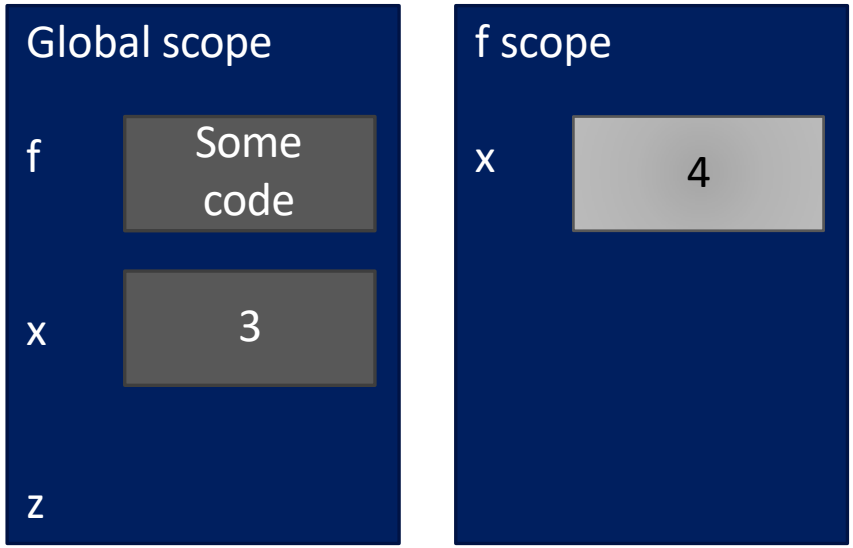
create a new scope/environment

as soon as hit a function call



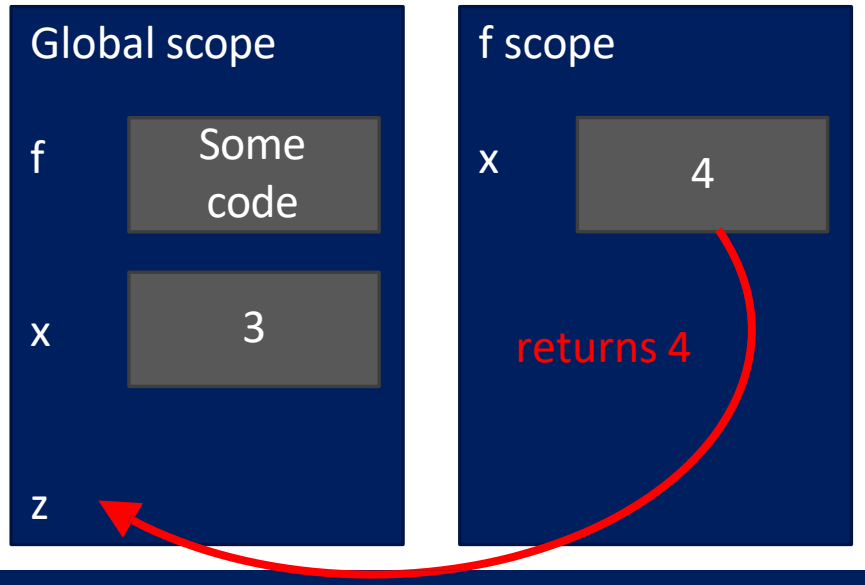
VARIABLE SCOPE

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

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



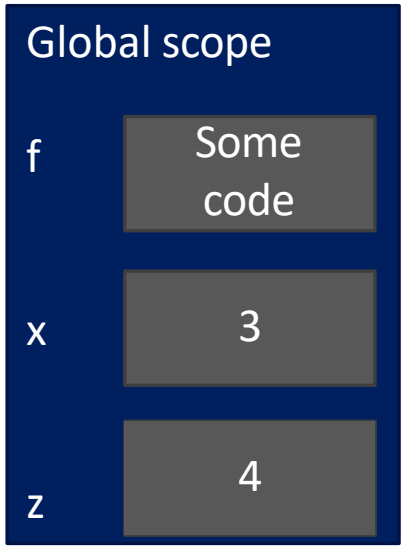
VARIABLE SCOPE

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


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



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))
```

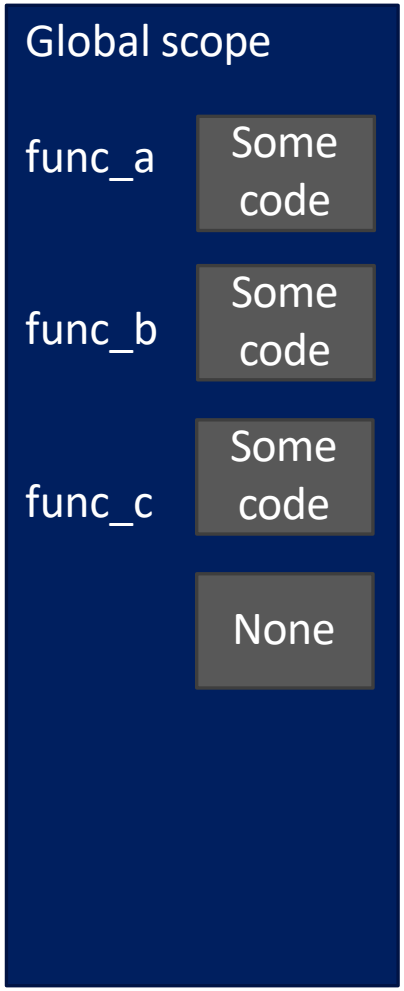
call `func_a`, takes no parameters

call `func_b`, takes one parameter

call `func_c`, takes one parameter, another function

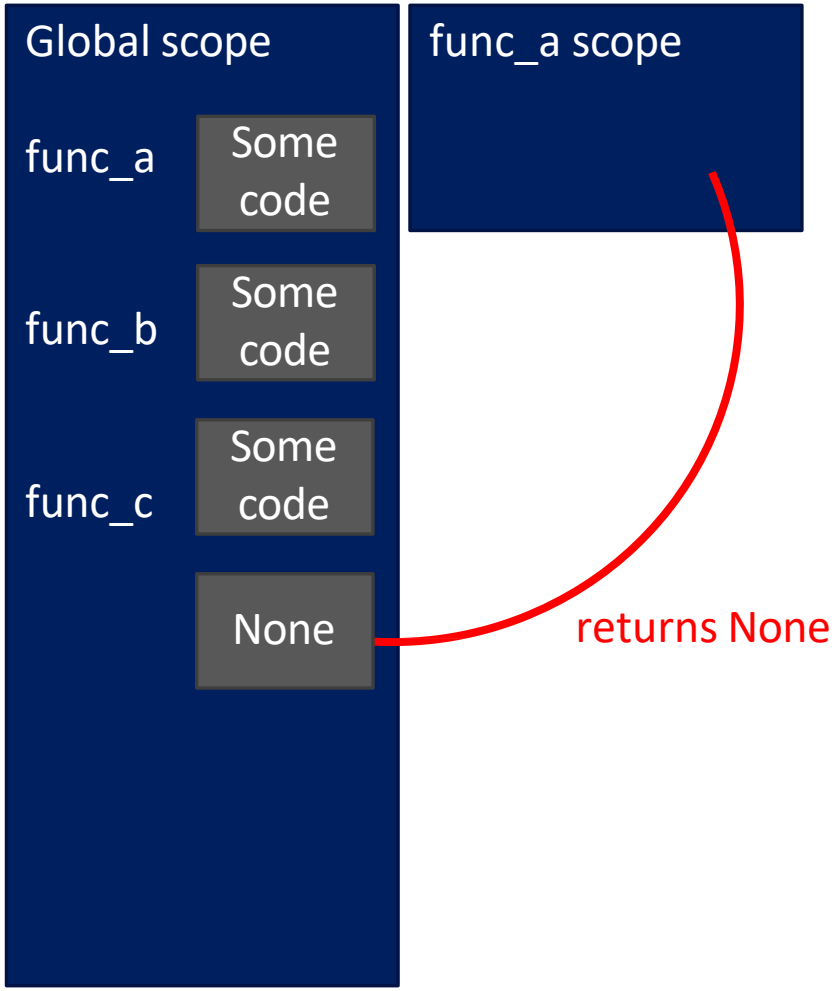
FUNCTIONS AS ARGUMENTS

```
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))
```



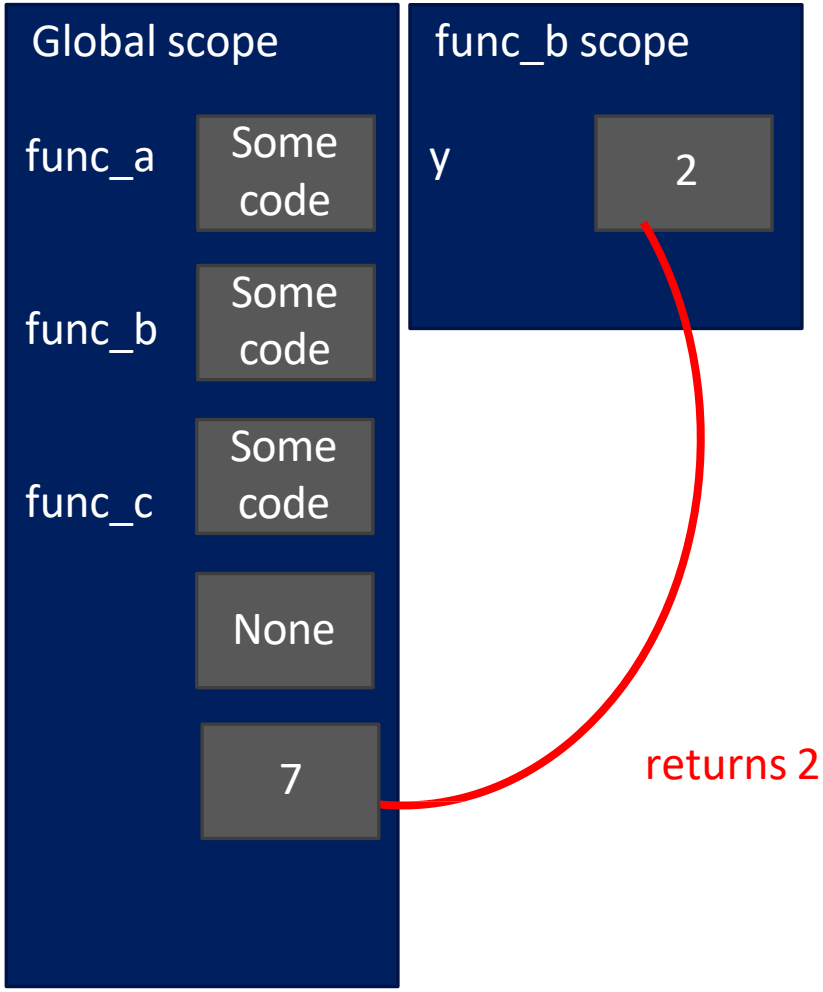
FUNCTIONS AS ARGUMENTS

```
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))
```



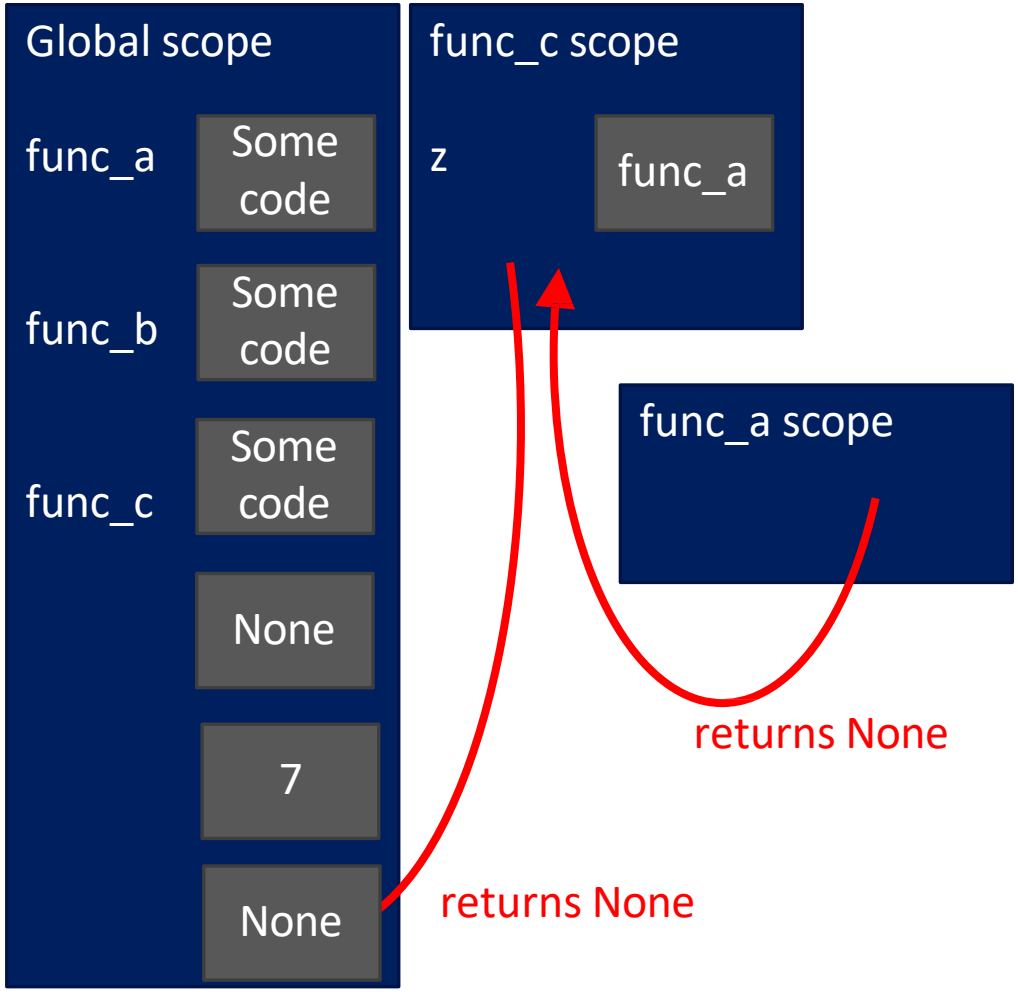
FUNCTIONS AS ARGUMENTS

```
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))
```



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

HARDER SCOPE EXAMPLE

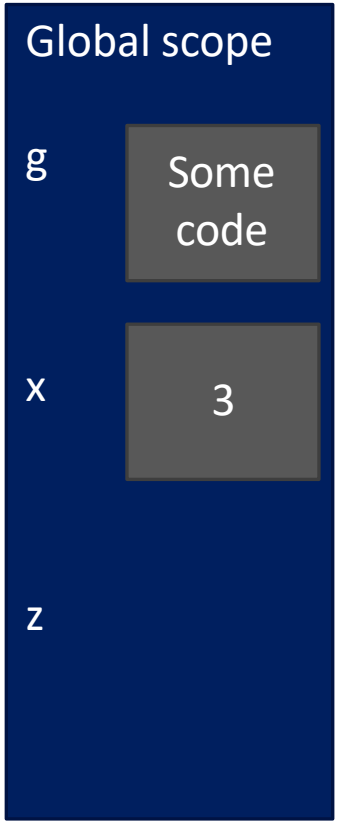
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

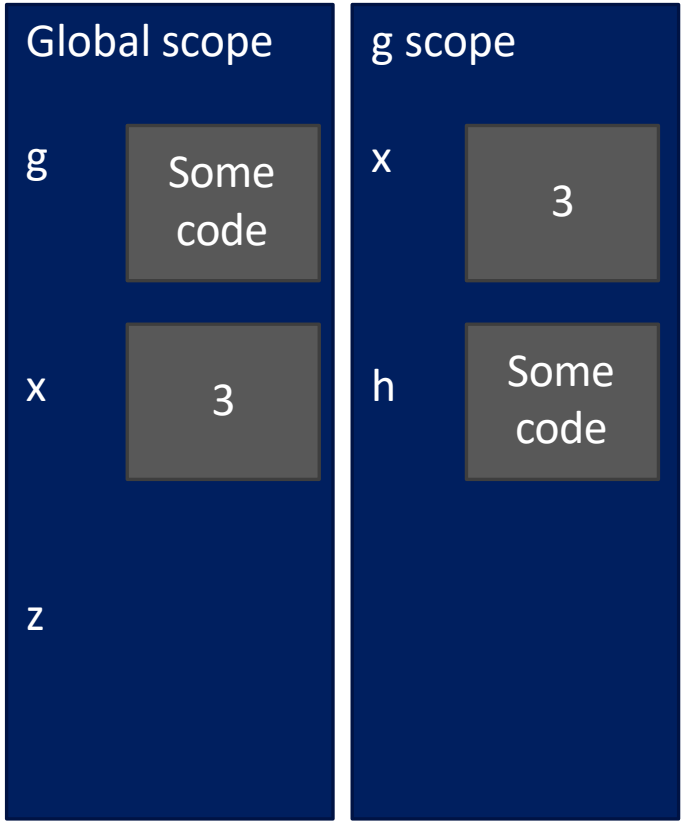
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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

Some code



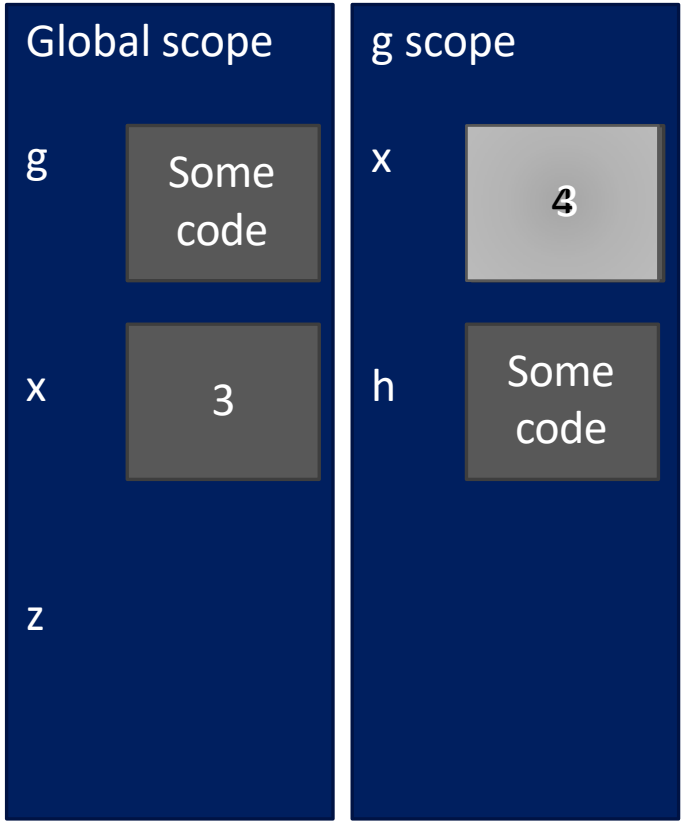
SCOPE DETAILS

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


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



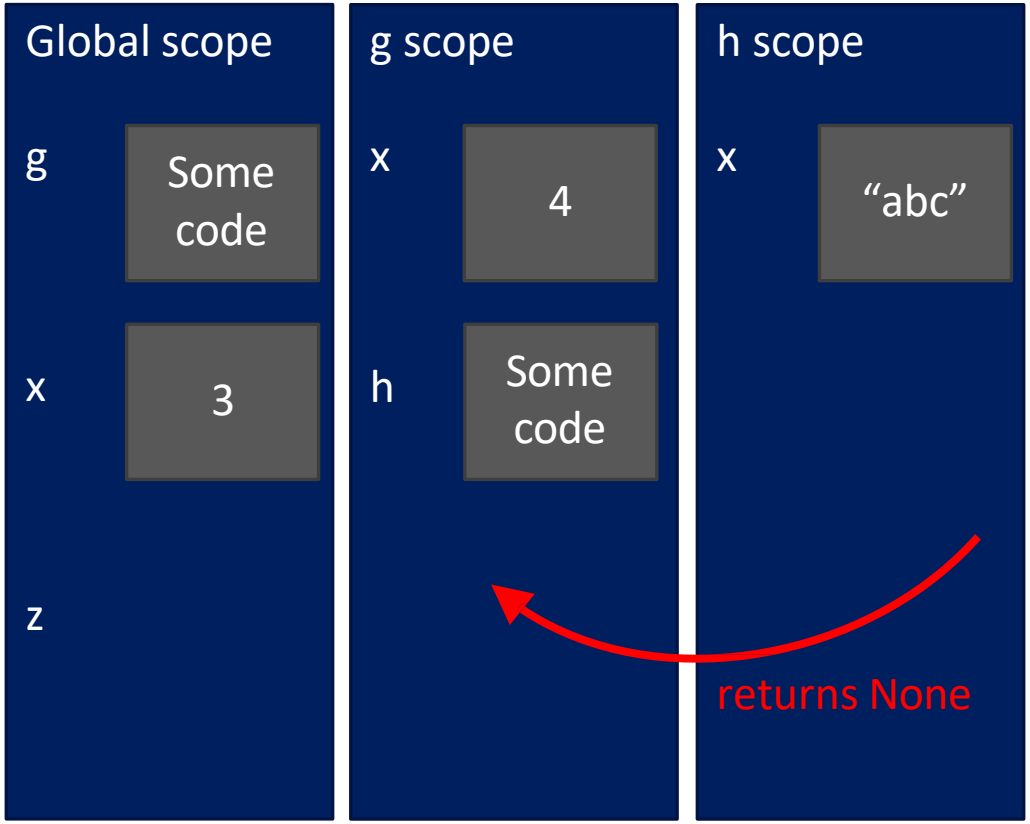
SCOPE DETAILS

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


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



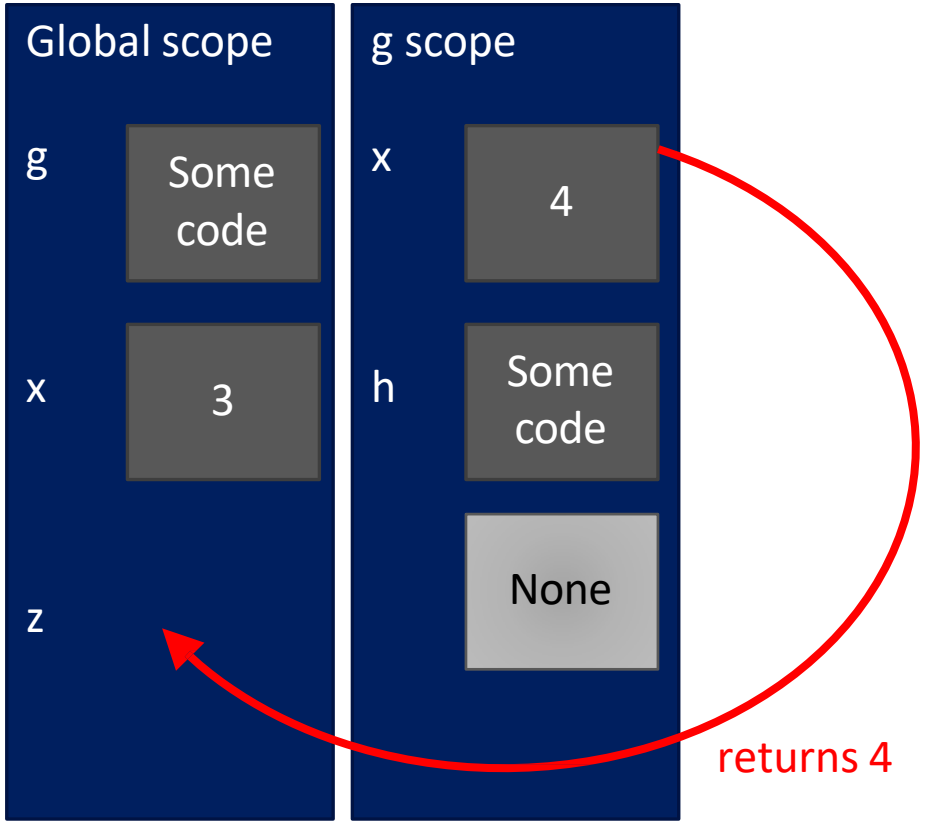
SCOPE DETAILS

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

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



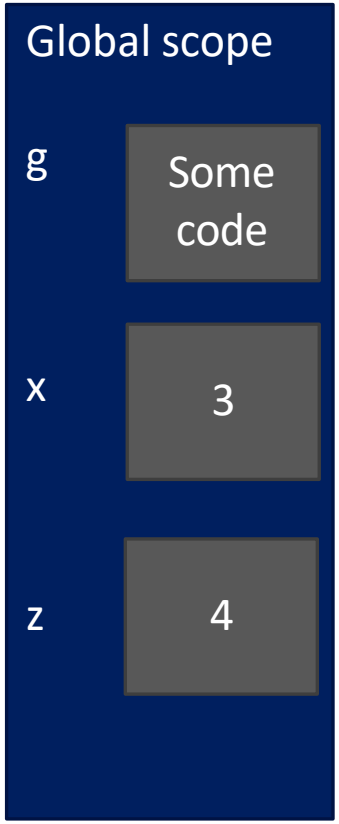
SCOPE DETAILS

```
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!

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, `[ ]`
- a list contains **elements**
  - usually homogeneous (i.e., 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
```

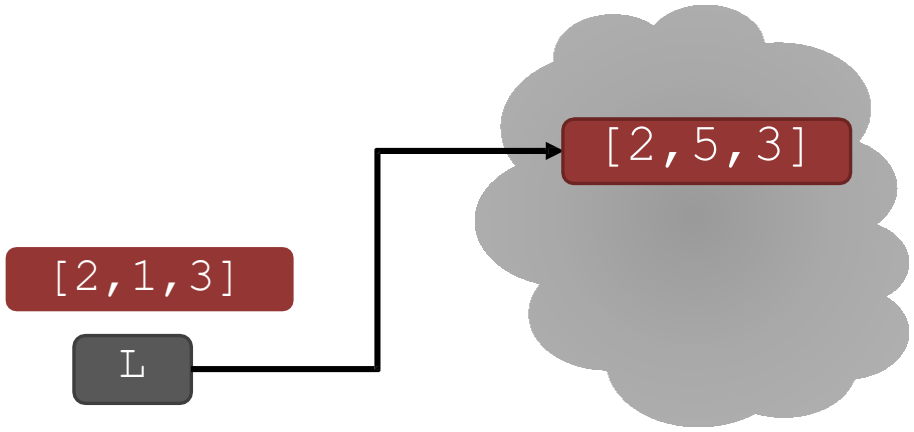
`L[i-1]` → evaluates to ‘a’ since `L[1]='a'`

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 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`

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()`

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]
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
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]
```

`sorted(L)` → returns sorted list, does **not mutate** `L`

`L.sort()` → **mutates** `L=[0,3,6,9]`

`L.reverse()` → **mutates** `L=[9,6,3,0]`

LISTS IN MEMORY

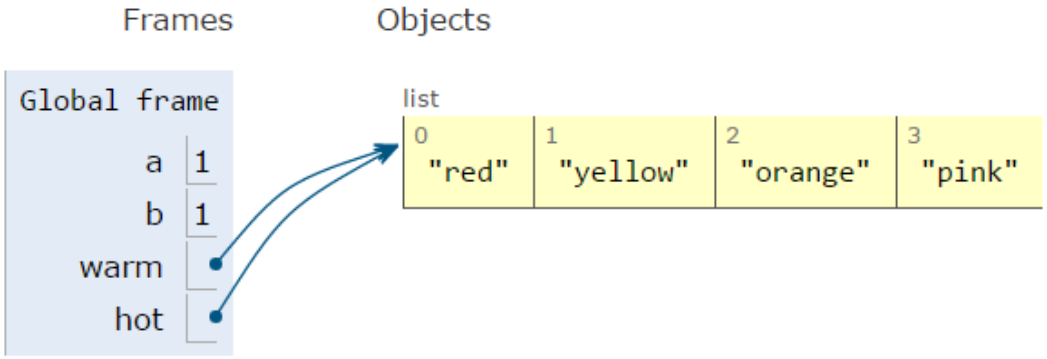
- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- **key phrase to keep in mind when working with lists is side effects**

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']
```

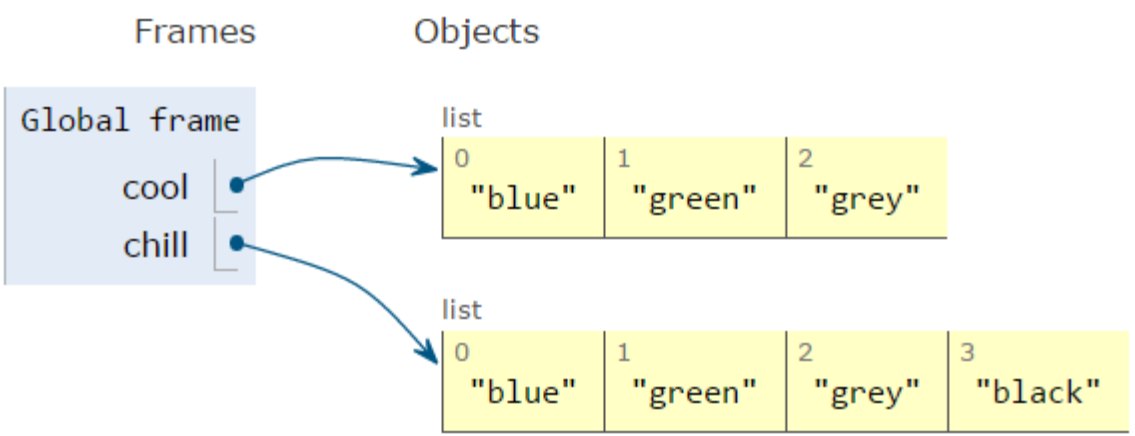


CLONING A LIST

- 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']
```

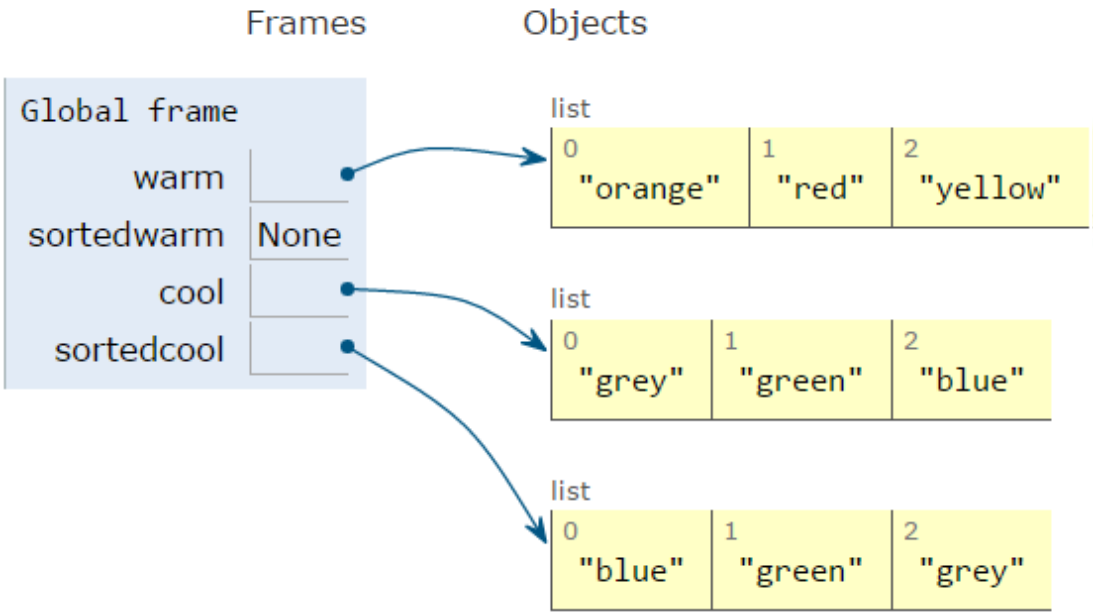


SORTING LISTS

- 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)
```

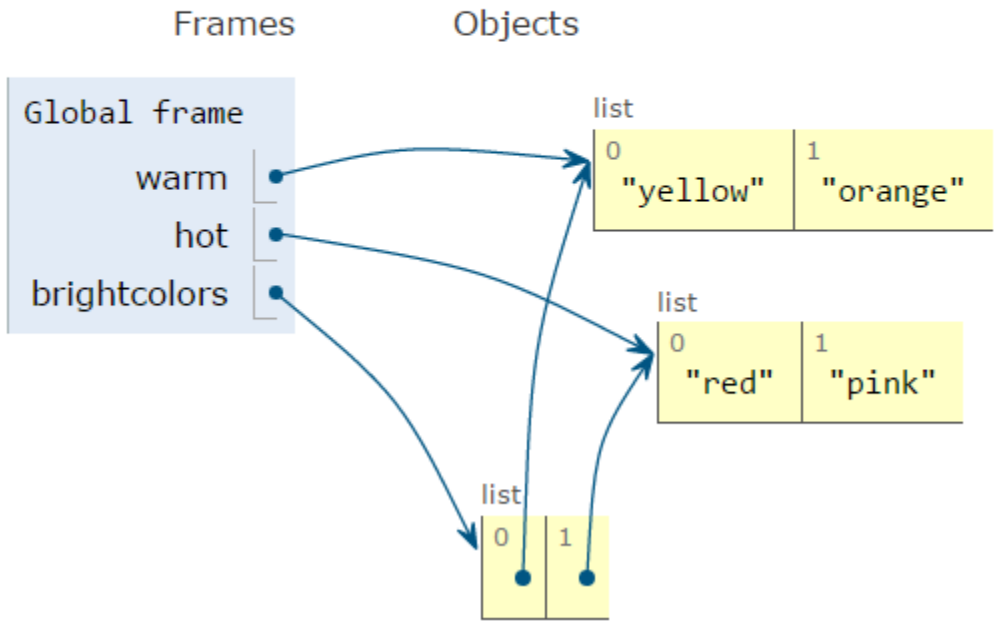


LISTS OF LISTS OF LISTS OF….

- 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)
```



DICTIONARIES HOW TO STORE STUDENT INFO

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [3210, 3212, 3214, 3216]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

A BETTER AND CLEANER WAY – A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

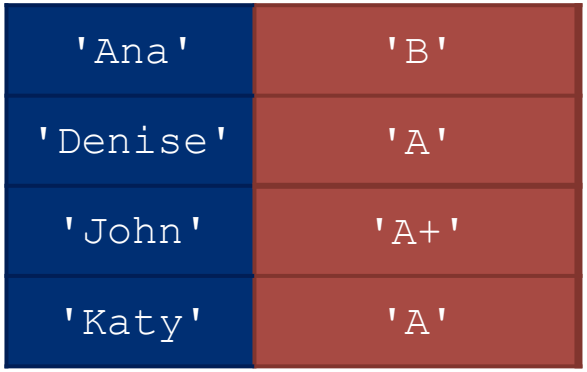
A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element

A PYTHON DICTIONARY

- store pairs of data
  - key
  - value



empty dictionary

```
my_dict = {}
```

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

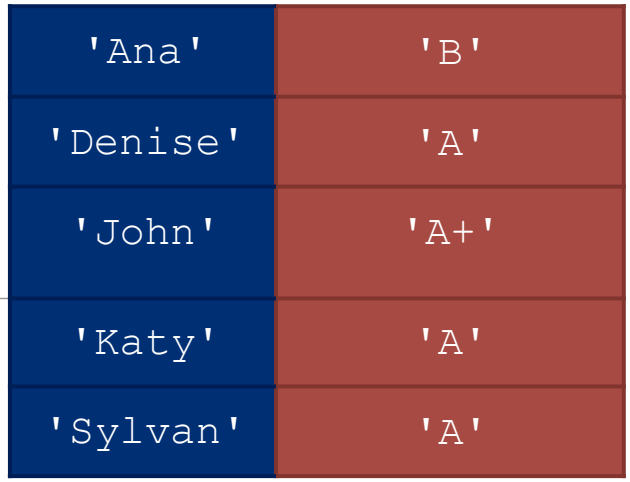
- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
grades['John']
```
➔ evaluates to `'A+'`

```
grades['Sylvan']
```
➔ gives a `KeyError`

DICTIONARY OPERATIONS




```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

```
'John' in grades
'Daniel' in grades
```

➔ returns `True`
➔ returns `False`

- **delete** entry

```
del(grades['Ana'])
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys**

  no guaranteed order

  `grades.keys()` ➔ returns `['Denise','Katy','John','Ana']`

- get an **iterable that acts like a tuple of all values**

  `grades.values()` ➔ returns `['A', 'A', 'A+', 'B']`

  no guaranteed order

DICTIONARY KEYS and VALUES

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - careful with `float` type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

`list` vs `dict`

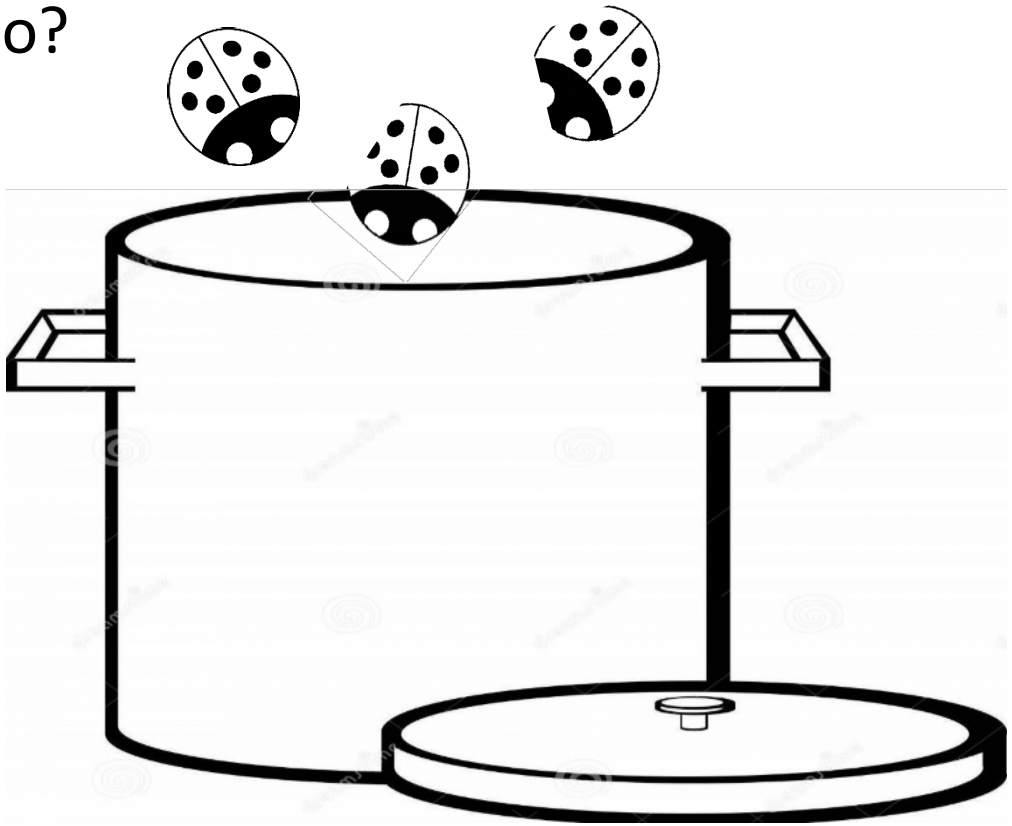
- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

WE AIM FOR HIGH QUALITY – AN ANALOGY WITH SOUP

You are making soup but bugs keep falling in from the ceiling. What do you do?

- check soup for bugs
  - testing
- keep lid closed
  - defensive programming
- clean kitchen
  - eliminate source of bugs



DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)

TESTING/VALIDATION

- **Compare** input/output pairs to specification
- "It's not working!"
- "How can I break my program?"

DEBUGGING

- **Study events** leading up to an error
- "Why is it not working?"
- "How can I fix my program?"

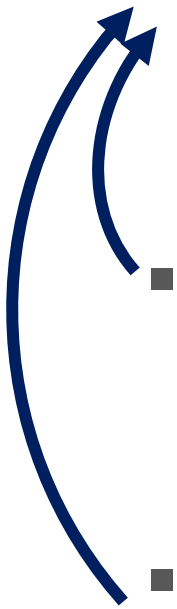
SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

- from the **start**, design code to ease this part
- break program up into **modules** that can be tested and debugged individually
- **document constraints** on modules
  - what do you expect the input to be?
  - what do you expect the output to be?
- **document assumptions** behind code design

WHEN ARE YOU READY TO TEST?

- ensure **code runs**
  - remove syntax errors
  - remove static semantic errors
  - Python interpreter can usually find these for you
- have a **set of expected results**
  - an input set
  - for each input, the expected output

CLASSES OF TESTS



- **Unit testing**
  - validate each piece of program
  - **testing each function** separately
- **Regression testing**
  - add test for bugs as you find them
  - **catch reintroduced** errors that were previously fixed
- **Integration testing**
  - does **overall program** work?
  - tend to rush to do this

TESTING APPROACHES

- **intuition** about natural boundaries to the problem

```
def is_bigger(x, y):
    """ Assumes x and y are ints
    Returns True if y is less than x, else False """
```

  - can you come up with some natural partitions?
- if no natural partitions, might do **random testing**
  - probability that code is correct increases with more tests
  - the more random testing, the greater likelihood program is correct
- **black box testing**
  - explore paths through specification(docstring, come up with test cases based on it)
- **glass box testing**
  - explore paths through code(come up with test cases that hit upon all possible paths through the code)

two rigorous ways of doing testing

BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

- designed **without looking** at the code
- great thing is that whoever implements this function can implement it in whatever way they wish
- testing can be **reused** if implementation changes
- **paths** through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

BLACK BOX TESTING

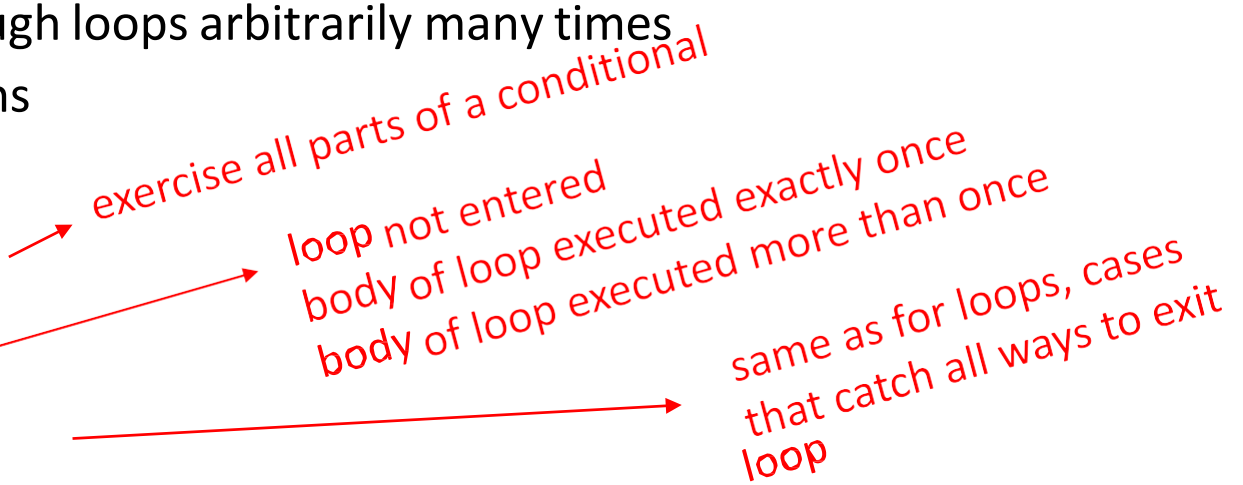
```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

- the important thing about black box testing is that you are creating the test cases based on the specifications only.

GLASS BOX TESTING

- **use code itself** directly to guide design of your test cases
- called **path-complete** if every potential path through code is tested at least once
- what are some **drawbacks** of this type of testing?
  - can go through loops arbitrarily many times
  - missing paths
- Guidelines
  - branches
  - for loops
  - while loops



GLASS BOX TESTING

```
def abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
    if x < -1:
        return -x else:
        return x
```

specifications

implementations

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test boundary cases(hit upon any boundary condition)

DEBUGGING - history

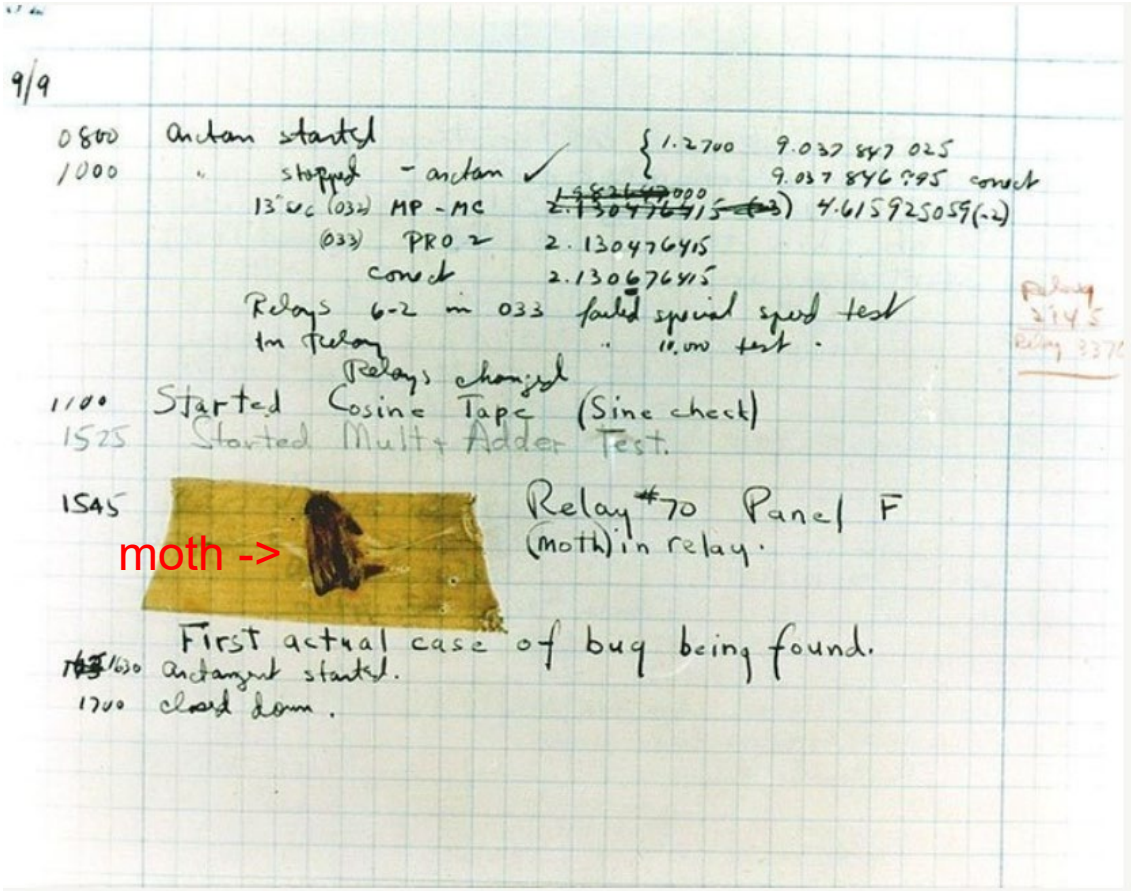
Mark II Aiken Relay Computer, September, 9, 1947

addition 0.1 sec
multiplication 0.7 sec
log of sth, 5 sec

DEBUGGING - history

Group of engineers tried to find the trigonometric function.
Grace Hoper, one of the famous female scientists.

went through all of the panels and all of the relays in the computer



Credit: US Navy

moth impeding the calculation.

DEBUGGING

- steep learning curve
- goal is to have a bug-free program
- tools
  - **built in** to IDLE and Anaconda
  - **Python Tutor**
  - `print` statement
  - use your brain, be **systematic** in your hunt

PRINT STATEMENTS

- good way to **test hypothesis**
- when to print
  - enter function
  - parameters
  - function results
- use **bisection method**
  - put print halfway in code
  - decide where bug may be depending on values

DEBUGGING STEPS

- **study** program code
  - don’t ask what is wrong because that's actually part of the testing
  - ask how did I get the unexpected result
  - is it part of a family?
- **scientific method**
  - study available data
  - form hypothesis
  - repeatable experiments
  - pick simplest input to test with

ERROR MESSAGES – EASY

- trying to access beyond the limits of a list

  `test = [1,2,3]` then `test[4]` → `IndexError`

- trying to convert an inappropriate type

  `int(test)` → `TypeError`

- referencing a non-existent variable

  `a` → `NameError`

- mixing data types without appropriate coercion

  `'3'/4` → `TypeError`

- forgetting to close parenthesis, quotation, etc.

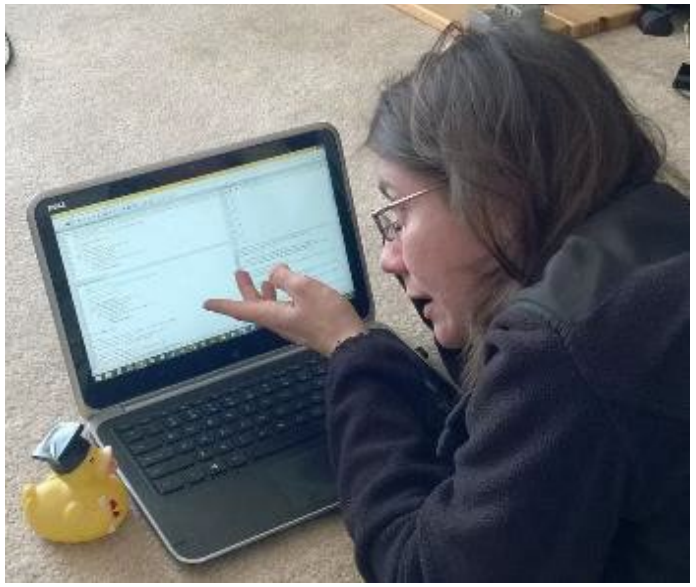
```
a = len([1,2,3]
print(a)
```

→ `SyntaxError`

LOGIC ERRORS - HARD

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
  - someone else
  - a rubber ducky debugging

https://en.wikipedia.org/wiki/Rubber_duck_debugging



DON'Ts

- Write entire program
- Test entire program
- Debug entire program

DOs

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration(unit) testing ***

- Change code
- Remember where bug was
- Test code
- Forget where bug was or what change you made
- Panic

- Backup code
- Change code
- Write down potential bug in a comment
- Test code
- Compare new version with old version

EXCEPTIONS AND ASSERTIONS

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception(error)** … to what was expected
  - trying to access beyond list limits

```
test = [1,7,4]
test[4]
```
→ `IndexError`

  - trying to convert an inappropriate type

```
int(test)
```
→ `TypeError`

  - referencing a non-existing variable

```
a
```
→ `NameError`

  - mixing data types without coercion

```
'a'/4
```
→ `TypeError`

OTHER TYPES OF EXCEPTIONS

- already seen common error types:
  - `SyntaxError`: Python can't parse program
  - `NameError`: local or global name not found
  - `AttributeError`: attribute reference fails
  - `TypeError`: operand doesn't have correct type
  - `ValueError`: operand type okay, but value is illegal
  - `IOError`: IO system reports malfunction (e.g. file not found)

DEALING WITH EXCEPTIONS

- Python code can provide **handlers** for exceptions

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

try block

except block

- exceptions **raised** by any statement in body of **`try`** are **handled** by the **`except`** statement and execution continues with the body of the `except` statement

HANDLING SPECIFIC EXCEPTIONS

- have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors

OTHER EXCEPTIONS

- else:
  - body of this is executed when execution of associated `try` block **completes with no exceptions**
- finally:
  - body of this is **always executed** after `try`, `else` and `except` clauses, even if they raised another error or executed a `break`, `continue` or `return`
  - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

WHAT TO DO WITH EXCEPTIONS?

- what to do when encounter an error?
- **fail silently**:
  - substitute default values or just continue
  - bad idea! user gets no warning
- return an **"error" value**
  - what value to choose?
  - complicates code having to check for a special value
- stop execution, **signal error** condition
  - in Python: **raise an exception**

```
raise Exception("descriptive string")
```

EXCEPTIONS AS CONTROL FLOW

- don't return special values when an error occurred and then check whether 'error value' was returned
- instead, **raise an exception** when unable to produce a result consistent with function's specification

```
raise <exceptionName>(<arguments>)
```

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

EXAMPLE: RAISING AN EXCEPTION

```
def get_ratios(L1, L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i] """
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/L2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = not a number
        except:
            raise ValueError('get_ratios called with bad arg')
    return ratios
```

manage flow of program by raising own error

EXAMPLE OF EXCEPTIONS

- assume we are **given a class list** for a subject: each entry is a list of two parts
  - a list of first and last name for a student
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
[['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

EXAMPLE CODE

```
[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```
def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
```

ERROR IF NO GRADE FOR A STUDENT

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
               [['captain', 'america'], [8.0,10.0,96.0]],
               [['deadpool'], []]]
```

- get `ZeroDivisionError: float division by zero` because try to

```
return sum(grades)/len(grades)
```

length is 0

OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- decide to **notify** that something went wrong with a msg

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```

- running on some test data gives

```
warning: no grades data
[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], None]]
```

flagged the error

because `avg` did not return anything in the except

OPTION 2: CHANGE THE POLICY

- decide that a student with no grades gets a **zero**

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return 0.0
```

- running on some test data gives

```
warning: no grades data
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

still flag the error

now `avg` returns 0

ASSERTIONS

- want to be sure that **assumptions** on state of computation are as expected
- use an **`assert` statement** to raise an `AssertionError` exception if assumptions not met
- **`assert` statement at the beginning/end** of your functions
- an example of good **defensive programming**

EXAMPLE

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

function ends immediately if assertion not met

- raises an `AssertionError` if it is given an empty list for grades
- prevent the program from propagating bad values
- as soon as a precondition isn't true, function stops

ASSERTIONS AS DEFENSIVE PROGRAMMING

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that **execution halts** whenever an expected condition is not met
- typically used to **check inputs** to functions, but can be used anywhere
- can be used to **check outputs** of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

WHERE TO USE ASSERTIONS?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a **supplement** to testing
- raise **exceptions** if users supplies **bad data input**
- use **assertions** to
  - check **types** of arguments or values
  - check that **invariants** on data structures are met
  - check **constraints** on return values
  - check for **violations** of constraints on procedure (e.g. no duplicates in a list)