CMSC201
Computer Science I for Majors

Lecture 14 – Lists (Continued)

Last Class We Covered

- The tuple data structure
  - Creation, conversion, slicing, traversal
- Casting variables
- The membership "**`in`**" operator

#TBT

Infinite **`while`** loops

Any Questions from Last Time?

Today's Objectives

- To review what we know about lists already
- To learn more about lists in Python
- To understand two-dimensional lists
  - (And more dimensions!)
- To practice passing lists to functions
- To learn about mutability and its uses

List Review

Previously Seen Operations

- Many of the operations we saw on strings are possible with lists
- Which of the following works with lists?
  - Concatenation (**+**)
  - Indexing
  - Slicing
  - **`.lower()`** and **`.upper()`**
  - **`len()`**

Concatenation

- Concatenation does work on lists!
  - But it has the same limit as string concatenation
  - You can only concatenate lists with lists

- So this works:

```
bookList + supplyList
```

- But this doesn't:

```
animalList +  "horse"
```

```
animalList + ["horse"]
```

Indexing

- Indexing does work on lists!
- In the exact same way it does for strings

- Some examples:

```
studentNames[16]
courseTitles[len(courseTitles) - 4]
songList[FAV_INDEX]
```

Slicing

- Slicing does work on lists!
- In the exact same way it does for strings

- Slicing goes “up to but not including” the end of the slice

```
>>> stuff = [17, "A", -22, True, "Hello"]
>>> print( stuff[2:4] )
[-22, True]
```

`.lower()` and `.upper()`

- These operations do not work on lists!
  - They don't make sense for a list
- In the same way, **`.append()`** and **`.remove()`** don't work on strings

- If you try, you get an error about attributes:

```
AttributeError: 'str' object has no
attribute 'remove'
```

`len()`

- Calling **`len()`** does work on lists!
- In the exact same way it does for strings

- Returns the length of the list
  - In other words, the number of elements

Two-Dimensional Lists

Two-Dimensional Lists

- Lists can hold any type (int, string, float, etc.)
  - This means they can also hold another list
- We’ve looked at lists as being one-dimensional
  - But lists can also be two- (or three- or four- or five-, etc.) dimensional!

 Image from wikimedia.org

Two-Dimensional Lists: A Grid

- It may help to think of 2D lists as a grid

```
twoD = [ [1,2,3], [4,5,6], [7,8,9] ]
```

1	2	3
4	5	6
7	8	9

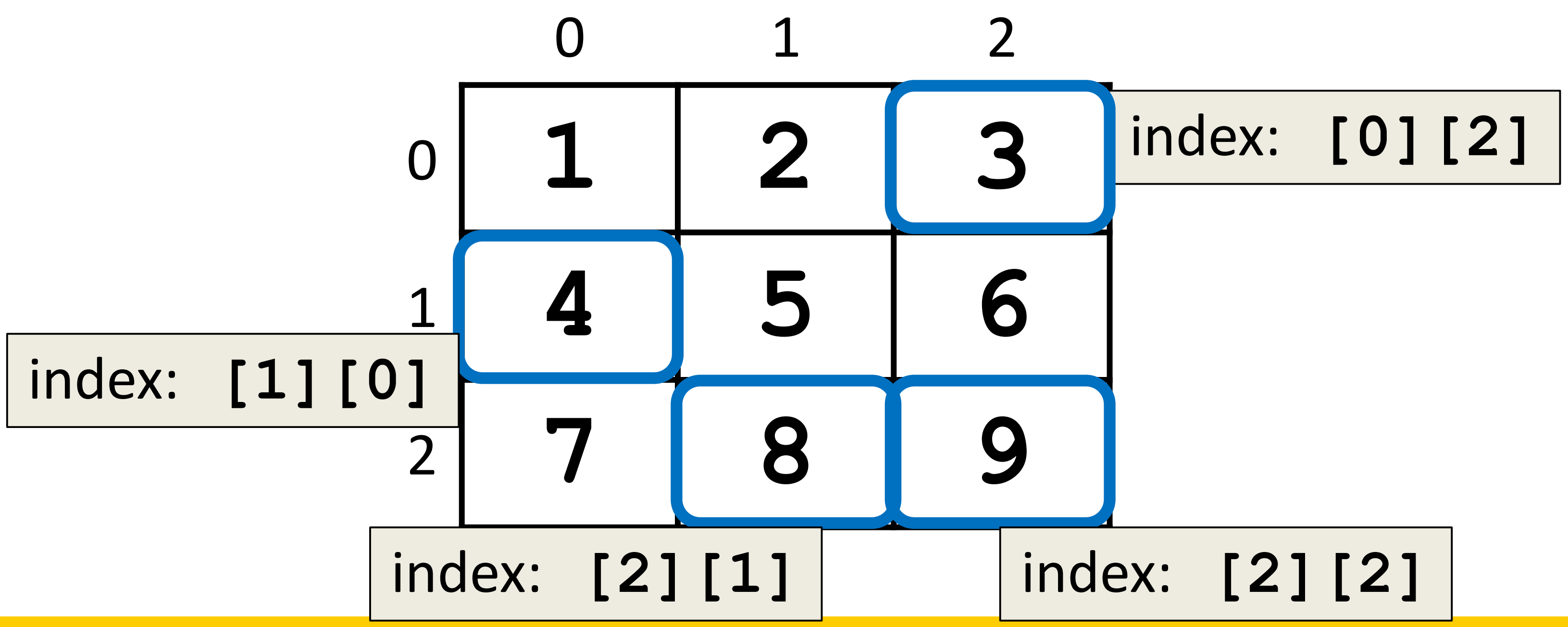
Two-Dimensional Lists: A Grid

- You access an element by the index of its row, and then the column
  - Remember – indexing starts at 0!

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Two-Dimensional Lists: A Grid

- You access an element by the index of its row, and then the column
  - Remember – indexing starts at 0!



Lists of Strings

- Remember, a string is like a list of characters
- So what is a list of strings?
  - Like a two-dimensional list!

- We have the index of the string (the row)
- And the index of the character (the column)

Lists of Strings

- Lists in Python don't have to be rectangular
  - They can be jagged (rows of different lengths)

- Anything we could do with a one-dimensional list, we can do with a two-dimensional list
  - Slicing, index, appending

	0	1	2	3	4
0	`A`	`l`	`i`	`c`	`e`
1	`B`	`o`	`b`		
2	`E`	`v`	`a`	`n`	

`names`

Three-Dimensional Lists

- How would you declare a 3D list?
- Square brackets for the list, row, and cells

overall list brackets

one cell

```
threeD = [ [ [201,"B"], [202,"B"], [203,"C"] ],
           [ [313,"C"], [331,"C"], [341,"C"] ] ]
```

one row

Three-Dimensional Lists

- Don’t think of the third dimension as “depth”
- Instead, it’s simply the “contents” of the cells

The first two dimensions give us a 2 row, 3 column list

```
threeD = [ [ [201,"B"], [202,"B"], [203,"C"] ],
           [ [313,"C"], [331,"C"], [341,"C"] ] ]
```

Mutability

Mutable and Immutable

- In python, certain structures cannot be altered once they are created and are called ***immutable***
  - These include integers, tuples, and strings

- Other structures can be altered after they are created and are called ***mutable***
  - These include lists

Lists and Mutability

- When you assign one list to another, it is by default a “shallow” copy of the list
- Any “in place” changes that are made to the shallow copy show up in the “original” list
  - Sort of like a pseudonym: one variable can be accessed with two separate names
- The other option is a “deep” copy of the list, but you must specify this is what you want

Shallow and Deep Copies

- A ***shallow copy*** is when the new variable actually points to the old variable, rather than making an actual copy

- A ***deep copy*** is the opposite, creating an entirely new list for the new variable

Shallow Copy Example

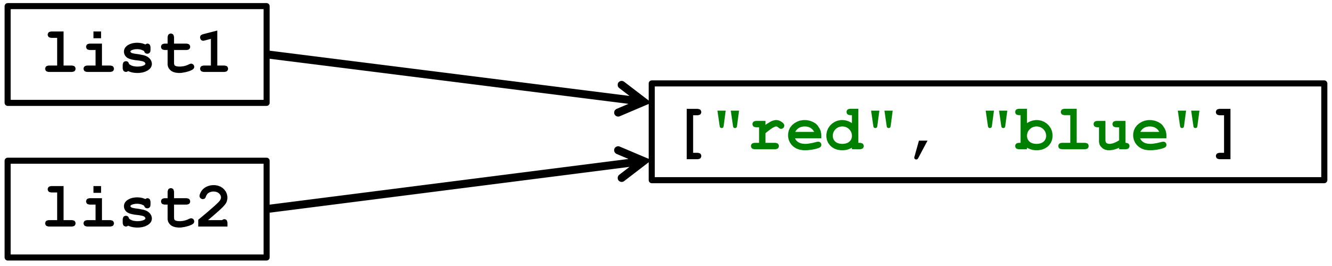
- A shallow copy and its effects on the original:

```
list1 = ["red", "blue"]
list2 = list1
list2.append("green")
list2[1] = "yellow"
print("original:      ", list1)
print("shallow copy: ", list2)
```

```
original:       ['red', 'yellow', 'green']
shallow copy:   ['red', 'yellow', 'green']
```

Shallow Copy

- When we make a shallow copy, we are essentially just giving the same list two different variable names
  - They both ***reference*** the same place in memory



Deep Copy

- There are two easy ways to do a deep copy:
  - Use slicing, and "slice" out the entire list
  - Cast the original as a list when assigning
- With these, Python returns a brand new copy that you can then assign to the new variable
  - Now you have two separate, individual lists!

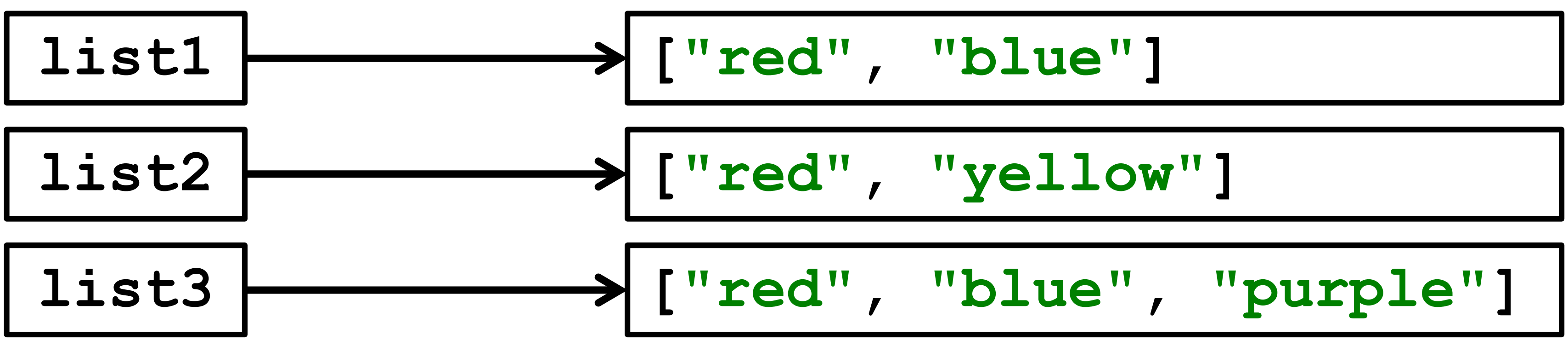
Deep Copy Example

```
list1 = ["red", "blue"]
list2 = list1[:]           # use slicing to copy
list2[1] = "yellow"
list3 = list(list1)        # use casting to copy
list3.append("purple")
print("original:     ", list1)
print("deep copy1:   ", list2)
print("deep copy2:   ", list3)
```

```
original:       ['red', 'blue']
deep copy1:     ['red', 'yellow']
deep copy2:     ['red', 'blue', 'purple']
```

Deep Copy

- Creates a copy of the entire list's contents, not just of the list itself
- Each variable now has its own individual list



Mutability and Functions

Lists, Functions, and Mutability

- When actual parameters are passed to a function, they are assigned to the formal parameters using the assignment operator

- So does the function have a deep copy?
  - No, it has a *shallow* copy!
  - It's a ***reference*** to the original list

Python Is "Lazy"

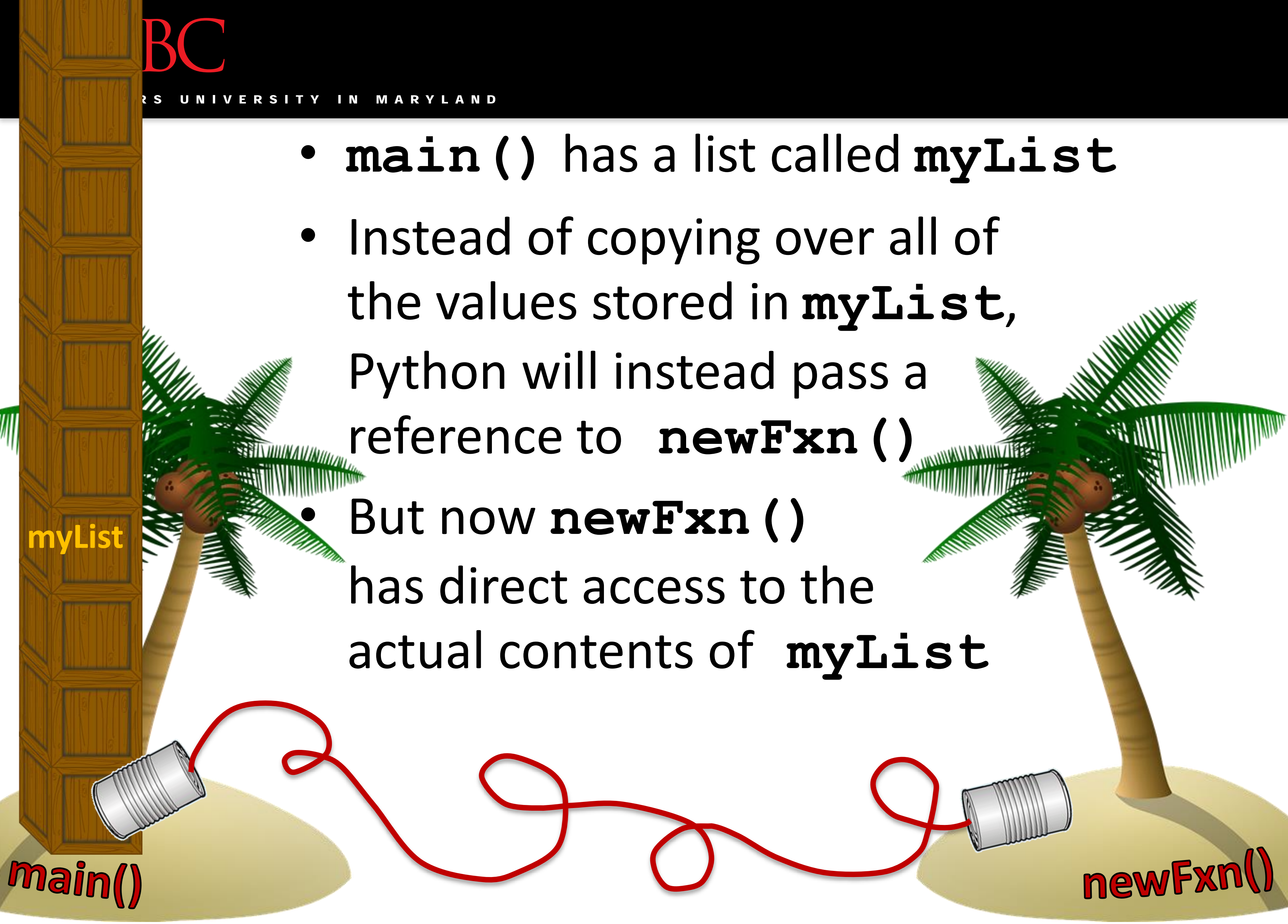
- Lists can be a lot bigger than Booleans, integers, or even strings!
- When we pass a list as a parameter, Python doesn't want to copy the entire thing
  - Copying can take a <u>lot</u> of memory and time
- Instead, when we pass a list to a function, Python actually sends a ***reference*** to the list

References

- A ***reference*** essentially states where the list is stored in the computer's memory
  - Mutable objects are always passed by reference
- Since lists are ***mutable***, that means that the function the list was passed to now has direct access to the "original" list
  - And can change its contents!!!

- `main()` has a list called `myList`
- Instead of copying over all of the values stored in `myList`, Python will instead pass a reference to `newFxn()`

- **`main()`** has a list called **`myList`**
- Instead of copying over all of the values stored in **`myList`**, Python will instead pass a reference to **`newFxn()`**
- But now **`newFxn()`** has direct access to the actual contents of **`myList`**



Images from pixabay.com and flickr.com

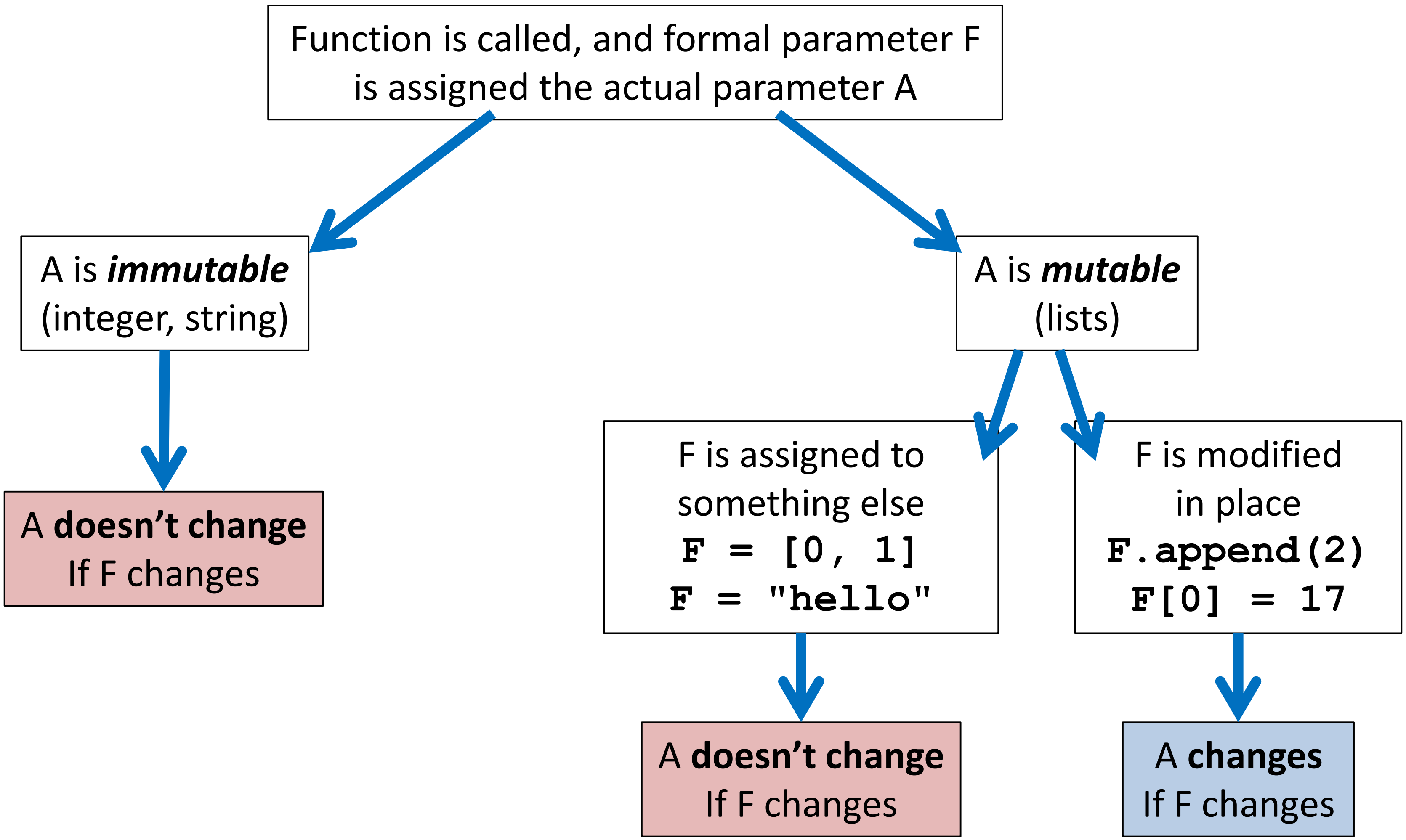
Mutability in Functions

- When a parameter is passed that is ***mutable***, it is now possible for the second function to directly access and change the contents
- This only works if we change the variable “in place” – assigning a whole new value to the variable will override the mutability
  - Any “in place” changes that are made to the shallow copy show up in the “original” list

Scope and Mutability in Functions

- A good general rule for if a change is "in place":
- When you use something like **`.append()`** on it, that's an "in place" change
- When you use the ***assignment operator***, the that's not an "in place" change
  - Unless you are editing one element, like in a list

Scope and Mutability in Functions



From http://stackoverflow.com/a/25670170

Using Mutability

- Shallow copies are not always a bad thing!

- Being able to
  - Pass a list to a function
  - Have that function make changes
  - And have those changes "stick"
- Can be very useful!

LIVECODING!!!

Cloning and Adopting Dogs

- Write a program that contains the following:

- A **`main()`** with a list of dogs at an adoption event
  - Use deep copy to "clone" the dogs by creating a second, unique list (and a third one as well)
- An **`adopt()`** function that takes in a list of dogs, and replaces all of their names with "adopted!"
  - These changes should "stick" in main() as well, without the function returning anything

Announcements

- HW 5 out on Blackboard Wednesday night
  - Must re-take the Academic Integrity Quiz to see it
  - Due *next* Friday, April 7th @ 8:59:59 PM

- Discussions start again next week
  - Remainder of labs will be in-person
  - Pre Lab quiz will come out Friday morning
- Exam pick-up at the front