

Lecture 16:

LISTS

CSC111: Introduction to CS through Programming

R. Jordan Crouser

Assistant Professor of Computer Science

Smith College

Announcements

- Presentation(s) of the Major(s):

CSC: Wednesday October 17th

SDS: Tuesday October 23rd

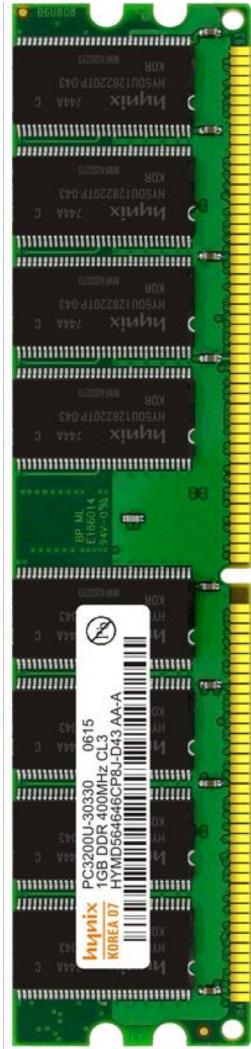
(both held in Ford Atrium)

- Grades for A1-3, Labs 1-5 posted to Moodle (labs pending attendance adjustments), A4 posted later today
- Heads up: guest speakers next week (Jordan in Berlin):
 - Monday 10/22: Tiff Xiao '20 – Google Cloud Platform
 - Wednesday 10/24: Prof. John Foley – Recursion Pt. 1
 - Friday 10/26: Prof. Katie Kinnaird – Music Information Retrieval
 - Monday 10/29: Prof. John Foley – Recursion

Outline

- Recap: strings:
 - indexing
 - slicing
- Lists
 - the basics
 - iterating
 - methods
- Dictionaries
 - motivation
 - defining a dictionary
 - converting multiple lists \leftrightarrow dictionaries

Recap: storing stuff in memory



**collections of things
in “numbered boxes”**

Recap: strings

- Collections of **characters**:

```
name = "Jordan"  
      ≈ [ 'J', 'o', 'r', 'd', 'a', 'n' ]  
           0   1   2   3   4   5
```

- To access the letter at position 2:

```
name[2] = "r"
```

- Can also use **negative** indexing (i.e. start at the end):

```
≈ [ 'J', 'o', 'r', 'd', 'a', 'n' ]  
    -6   -5   -4   -3   -2   -1
```

- To access the letter at position -2:

```
name[-2] = "a"
```

Check in

There are two ways to access the **last letter** in a string:
what are they?



Recap: slicing strings

- Sometimes we want to access a specific part of the string (more than a single letter, but less than the whole thing)
- e.g. to access the letters in positions **4 through 7**:

```
s = "Computer Science"
```

```
s[3:6] = "put"
```



remember:
not inclusive

- This is called **slicing**

Recap: slicing strings

- Special slices:

`s = "Computer Science"`

`s[:9] = "Computer"`



“start at the beginning”

`s[10 :] = "Science"`



“continue until the end”

Okay, so...

strings are collections of **characters**

defined using
“ quotes ”

Okay, so...

i.e. just about
anything



lists are collections of **objects**

defined using
[square brackets]

list of integers

```
[ 1, 2, 3, 4, 5, 6 ]
```

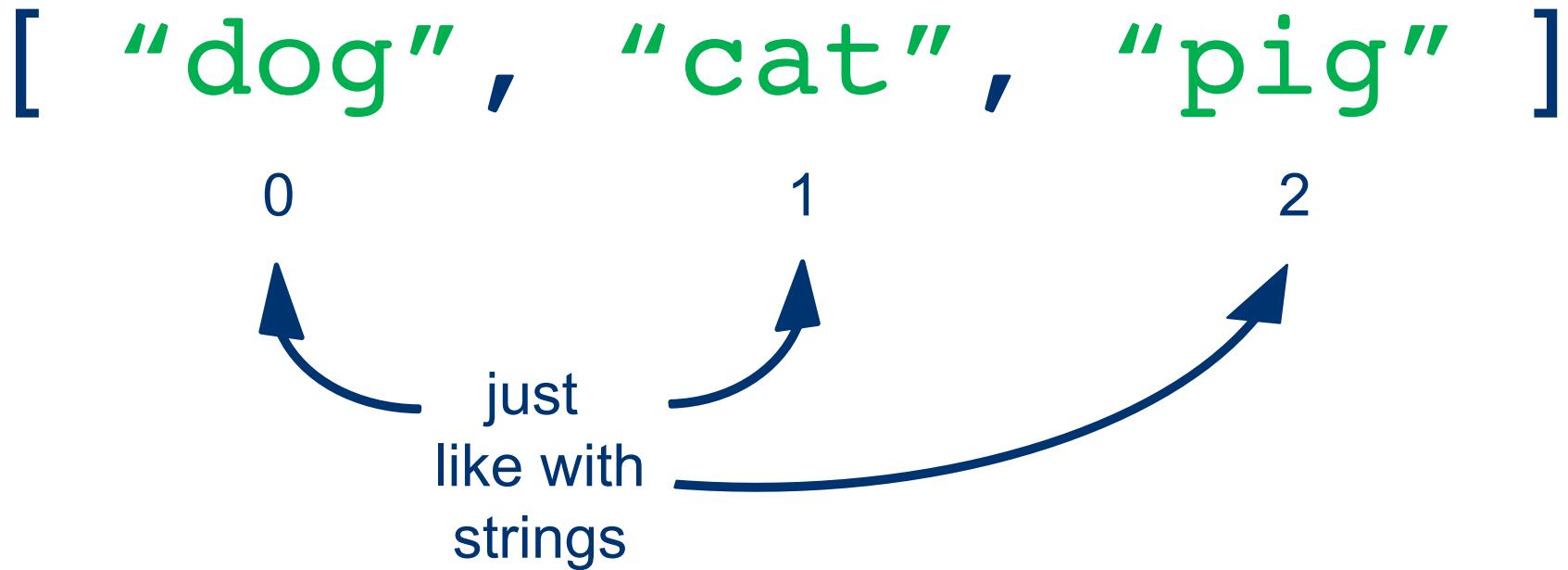
list of floats

```
[ 1.2, 3.5, 0.7, 7.8 ]
```

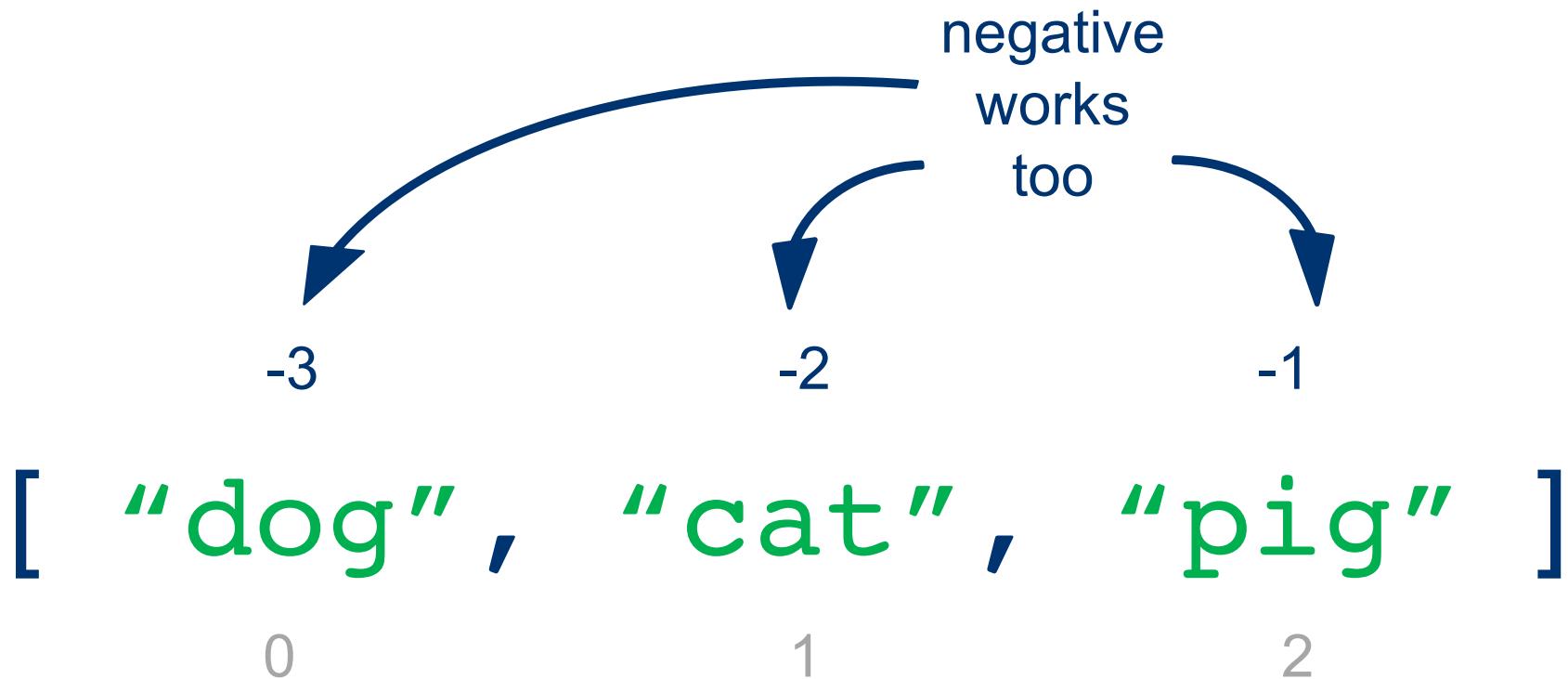
list of strings

```
[ "dog", "cat", "pig" ]
```

Indexing a list



Indexing a list



Weird **python** thing

in **python**, lists can contain **mixed types**:

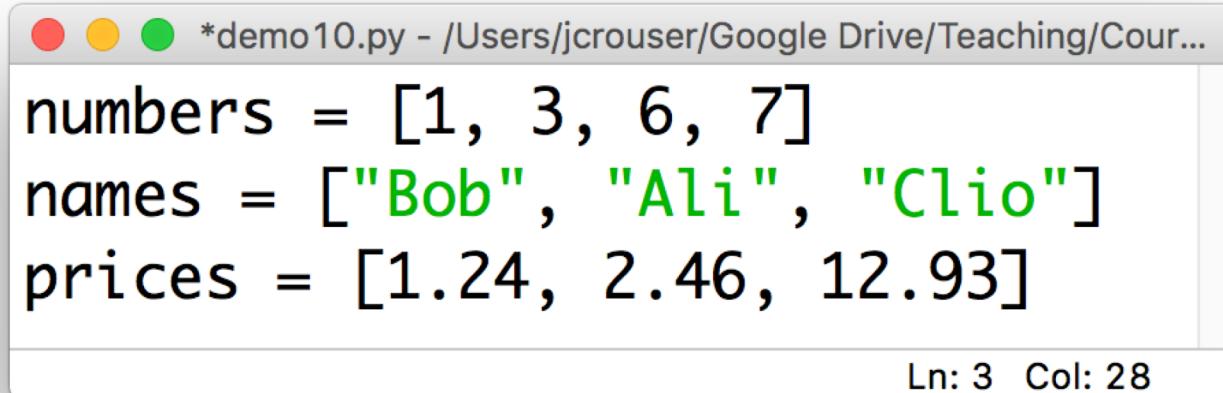
```
[ 1, "cat", 7.8 ]
```

this is
not allowed

in many other languages
(so be careful!)

Naming convention

- Remember: it's always a good idea for variable names to be **descriptive**
- Because lists contain collections of things, we'll generally label them with a **plural noun**, e.g.



A screenshot of a terminal window titled "*demo10.py - /Users/jcrouser/Google Drive/Teaching/Cour...". The window contains three lines of Python code:

```
numbers = [1, 3, 6, 7]
names = ["Bob", "Ali", "Clio"]
prices = [1.24, 2.46, 12.93]
```

The terminal also displays the current line and column information at the bottom right: "Ln: 3 Col: 28".

Iterating through items **in** a list

```
*Untitled*
```

```
animals = ["dog", "cat", "pig"]  
  
for animal in animals:  
    print(animal)
```

Ln: 4 Col: 0

Checking membership **in** a list

```
*Untitled*
```

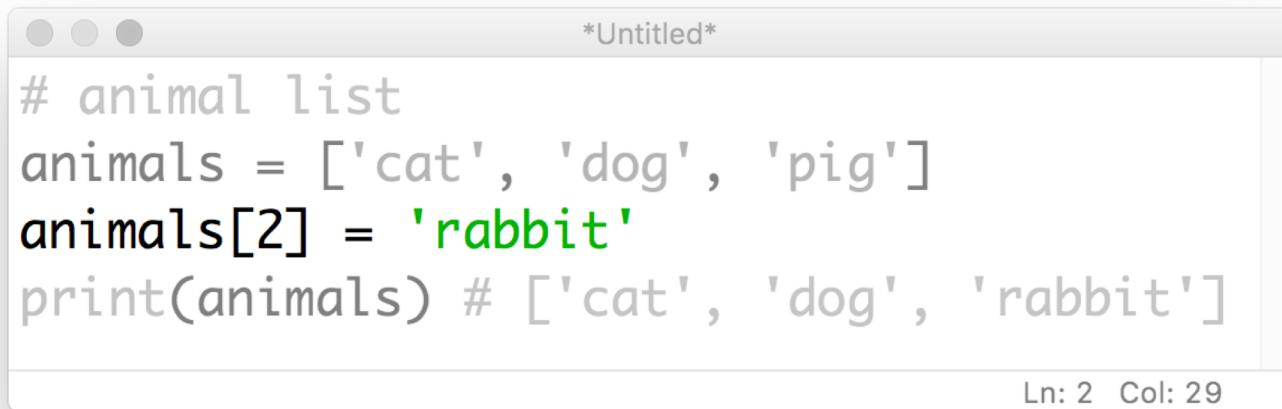
```
animals = ["dog", "cat", "pig"]
new_animal = input("Animal? ")

if new_animal in animals:
    print("I know that one!")
else:
    print("Hmm, don't know it.")
```

Ln: 3 Col: 0

Overwriting an item in a **list**

- If we want to overwrite an item in a **list**, we can use indexing combined with the **=** operator:



The image shows a screenshot of a code editor window titled "*Untitled*". The code in the editor is as follows:

```
# animal list
animals = ['cat', 'dog', 'pig']
animals[2] = 'rabbit'
print(animals) # ['cat', 'dog', 'rabbit']
```

The line `animals[2] = 'rabbit'` is highlighted in green, indicating it is the part of the code being demonstrated. The status bar at the bottom right of the editor window displays "Ln: 2 Col: 29".

Discussion

What happens when we try to do this
with a **string**?



Discussion

```
Python 3.6.5 Shell
>>> animal = 'pig'
>>> animal[1] = 'u'
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    animal[1] = 'u'
TypeError: 'str' object does not support item assignment
Ln: 65 Col: 4
```



mutable vs. **immutable**

- **strings** are **immutable** (which means we cannot change them in memory, we have to overwrite them completely)
- **lists** defined with [...] are **mutable** (which means we can change them in memory)
- if we want an **immutable lists**, we can define them with (...) instead

list methods: .append()

- If you want to **add a new item** to the end of a **list**:



```
# animal list
animals = ['cat', 'dog', 'pig']

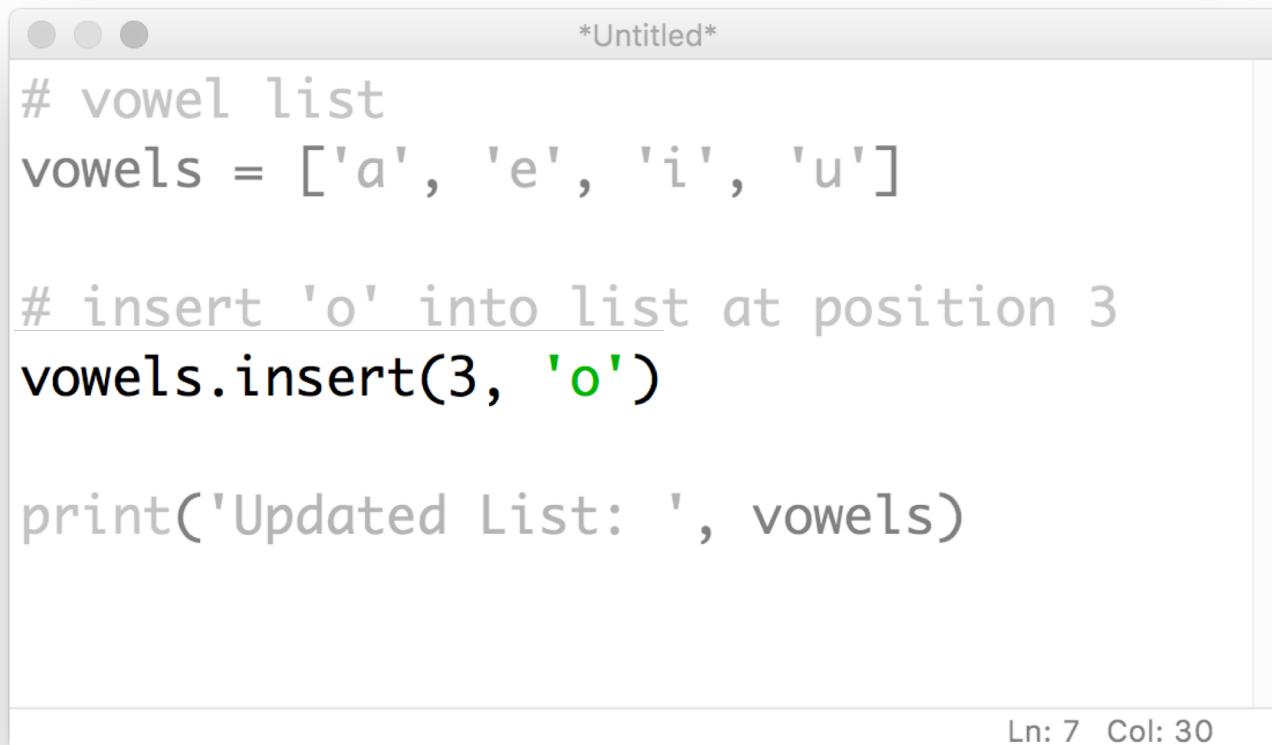
# add an element
animals.append('guinea pig')

# print updated list
print(animals)
```

Ln: 8 Col: 6

list methods: `.insert()`

- If you want to **add a new item** into a **list** at a specific position:



```
*Untitled*
```

```
# vowel list
vowels = ['a', 'e', 'i', 'u']

# insert 'o' into list at position 3
vowels.insert(3, 'o')

print('Updated List: ', vowels)
```

Ln: 7 Col: 30

list methods: .remove()

- If you want to **remove** an item from a **list**:

```
*Untitled*
```

```
# animal list
animal = ['cat', 'dog', 'rabbit',
          'guinea pig']

# 'rabbit' element is removed
animal.remove('rabbit')

#Updated Animal List
print('Updated animal list: ', animal)
```

Ln: 3 Col: 10

list methods: .remove()

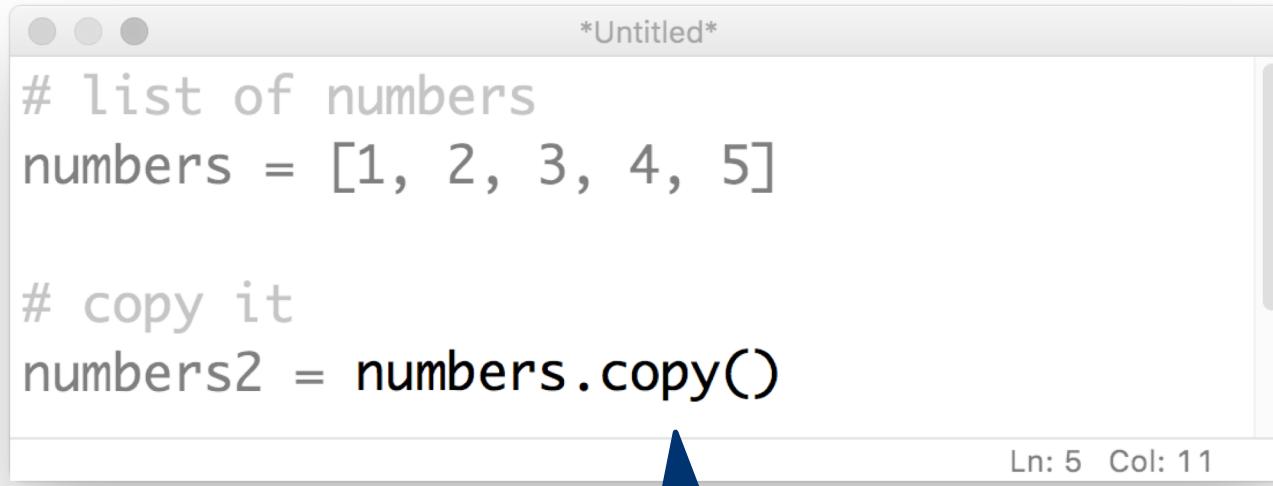
- If you try to **remove** an item that isn't in the **list**, the interpreter will throw a **ValueError**:



```
Python 3.6.5 Shell
>>> # animal list
animal = ['cat', 'dog', 'rabbit',
           'guinea pig']
>>> animal.remove("elephant")
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    animal.remove("elephant")
ValueError: list.remove(x): x not in list
Ln: 42 Col: 4
```

list methods: .copy()

- If you want to **copy** the **list**:



```
*Untitled*
```

```
# list of numbers
numbers = [1, 2, 3, 4, 5]

# copy it
numbers2 = numbers.copy()
```

Ln: 5 Col: 11

wait... why?

An important note about copying a `list`

- Usually when we want to copy a string or a number, we just say something like:

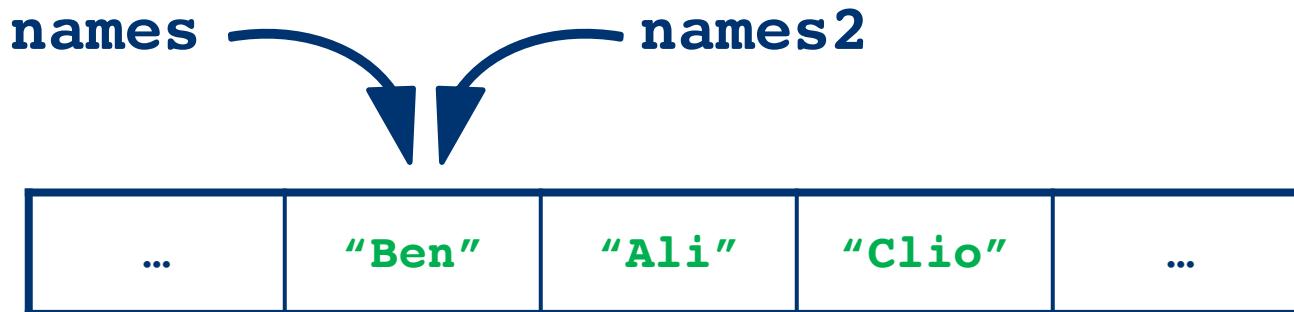
`x2 = x1`

- Copying a list this way, both the original and the copy point to the **same spot** in memory
- This can cause some unexpected behavior... remember when we said lists were **mutable**?

An important note about copying a list

- Let's say we have a **list** stored in memory:

```
names = ["Ben", "Ali", "Clio"]
```

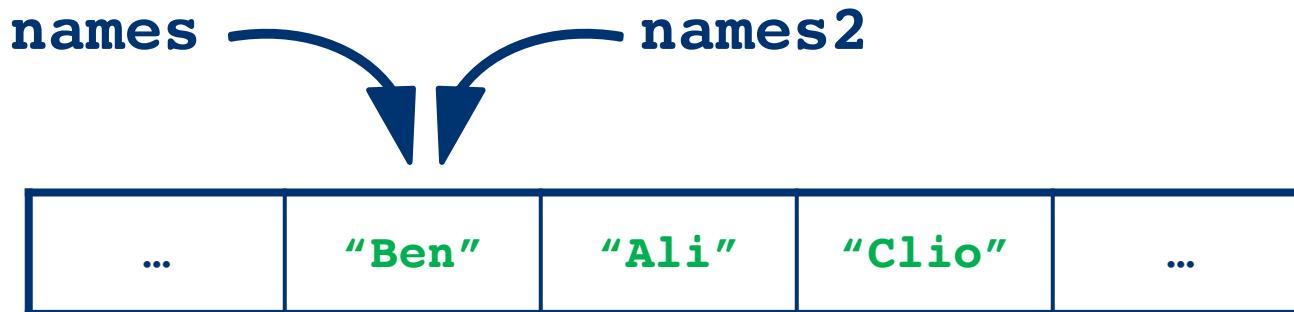


- And then we say `names2 = names`

An important note about copying a list

- Let's say we have a `list` stored in memory:

```
names = ["Ben", "Ali", "Clio"]
```



- And then we say `names2 = names`

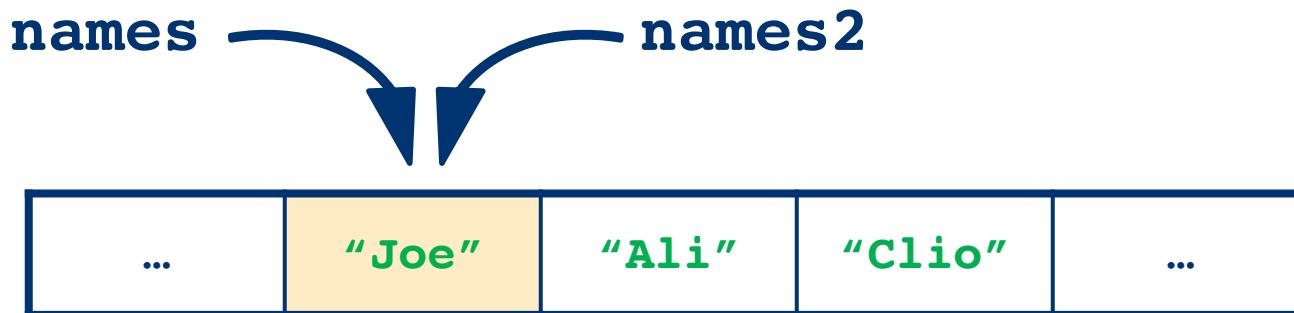
- If we then say:

```
names2[0] = "Joe"
```

An important note about copying a list

- Let's say we have a `list` stored in memory:

```
names = ["Ben", "Ali", "Clio"]
```



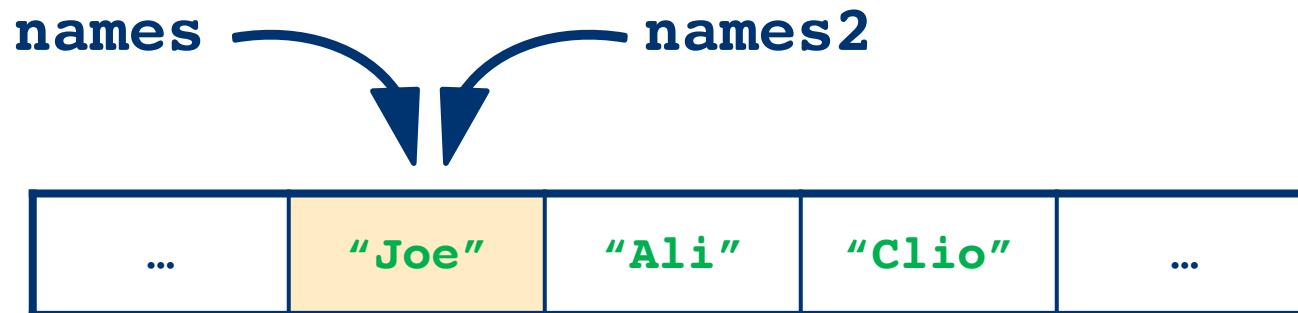
- And then we say `names2 = names`

- If we then say:

```
names2[0] = "Joe"
```

An important note about copying a list

What happens if we then ask for `names[0]`?



Recap: copying lists

```
*demo10.py - /Users/jcrouser/Google Drive/Teaching/Course Materi...
names = ["Joe", "Ali", "Clio"]

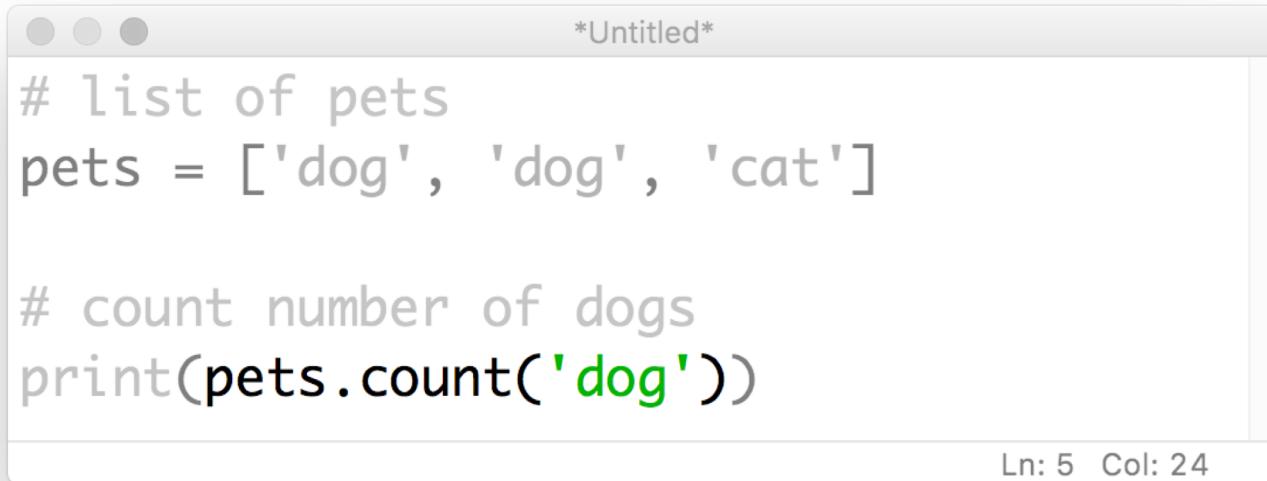
names2 = names # points to same
              # place in memory

names2 = names.copy() # list is
                      # duplicated

Ln: 7 Col: 23
```

list methods: .count()

- If you want to **count how many times an item appears in the list**:



The image shows a screenshot of a code editor window titled "Untitled*". The code in the editor is as follows:

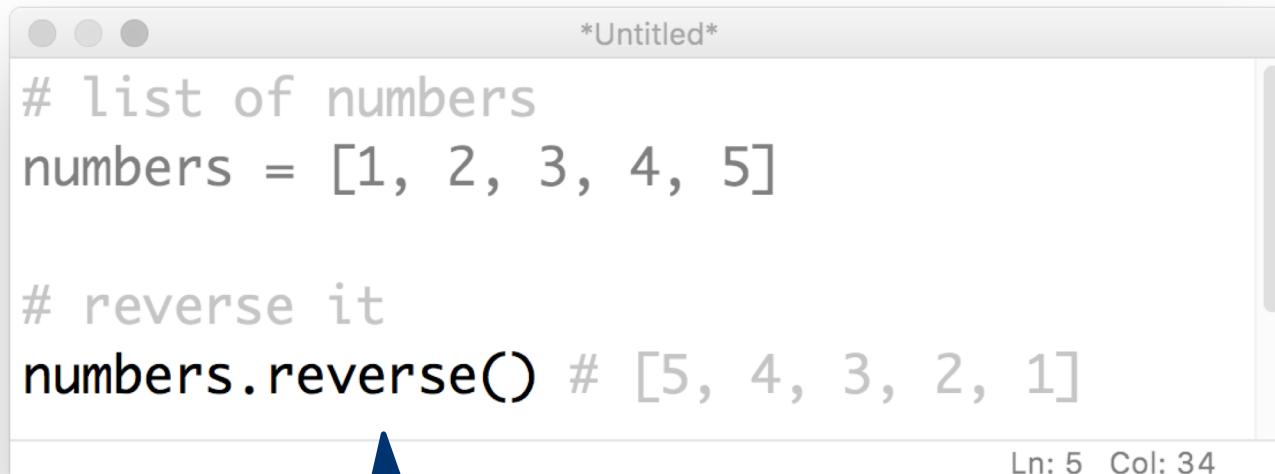
```
# list of pets
pets = ['dog', 'dog', 'cat']

# count number of dogs
print(pets.count('dog'))
```

At the bottom right of the code editor, it says "Ln: 5 Col: 24".

list methods: `.reverse()`

- If you want to **reverse** the **list**:



A screenshot of a code editor window titled "*Untitled*". The code is as follows:

```
# list of numbers
numbers = [1, 2, 3, 4, 5]

# reverse it
numbers.reverse() # [5, 4, 3, 2, 1]
```

The line `numbers.reverse()` is highlighted in black. In the bottom right corner of the editor, it says "Ln: 5 Col: 34".



this **changes**
the original list!

list methods: .sort()

- If you want to **sort** the **list**:



The image shows a screenshot of a code editor window titled "Untitled*". The code is as follows:

```
# list of names
names = ["Ben", "Ali", "Clio"]

# sort it
names.sort()

# print it
print(names)
```

A blue arrow points from the text "this also changes" to the line "names.sort()", indicating that the sort operation affects the original list.

Ln: 8 Col: 12

15-minute exercise:

Write a program that:

- asks the user to `input()` names one at a time
- adds each new name to a list called `friends`
- and after each new name is added prints the list in alphabetical order

The program should loop until the user types “**DONE**”

Quick taste of list comprehensions

- In my demo code last week, you saw something that looked like this:

```
grids = [[0]*n for row in range(n)]
```

- This is called a **list comprehension**, and it's a shorthand way to define a list according to a rule
- This is kind of advanced, but I'll show you how it works when we get to the lab

Assignment 6: music library

- In this assignment, you'll use lists (and other collections) to create a **music library** interface
- Using your interface, the user should be able to:
 - add a song (title, artist, album, link to youtube video, etc.)
 - delete an existing song
 - play a selected song's youtube video
 - print the contents of the library to the screen
- Hint: this week's lab will be **very helpful** in getting started!

Coming up

✓ Recap: strings:

- ✓ indexing
- ✓ slicing

✓ Lists

- ✓ the basics
- ✓ iterating
- ✓ methods

- Dictionaries

- motivation
- defining a dictionary
- converting multiple lists \leftrightarrow dictionaries