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):

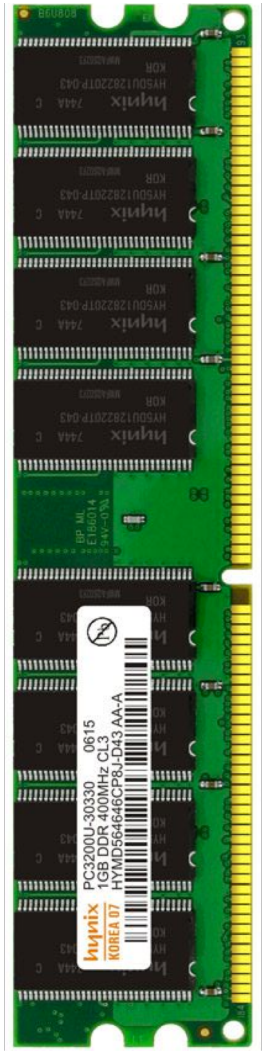
  CSC: Wednesday October 17th
  SDS: Tuesday October 23rd
  (both held in Ford Atrium)

- 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 ←→ 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"
```

- This is called **slicing**

Recap: slicing **`strings`**

- Special slices:

```
s = "Computer Science"
s[:9] = "Computer"
```


```
s[10:] = "Science"
```

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 ]
```

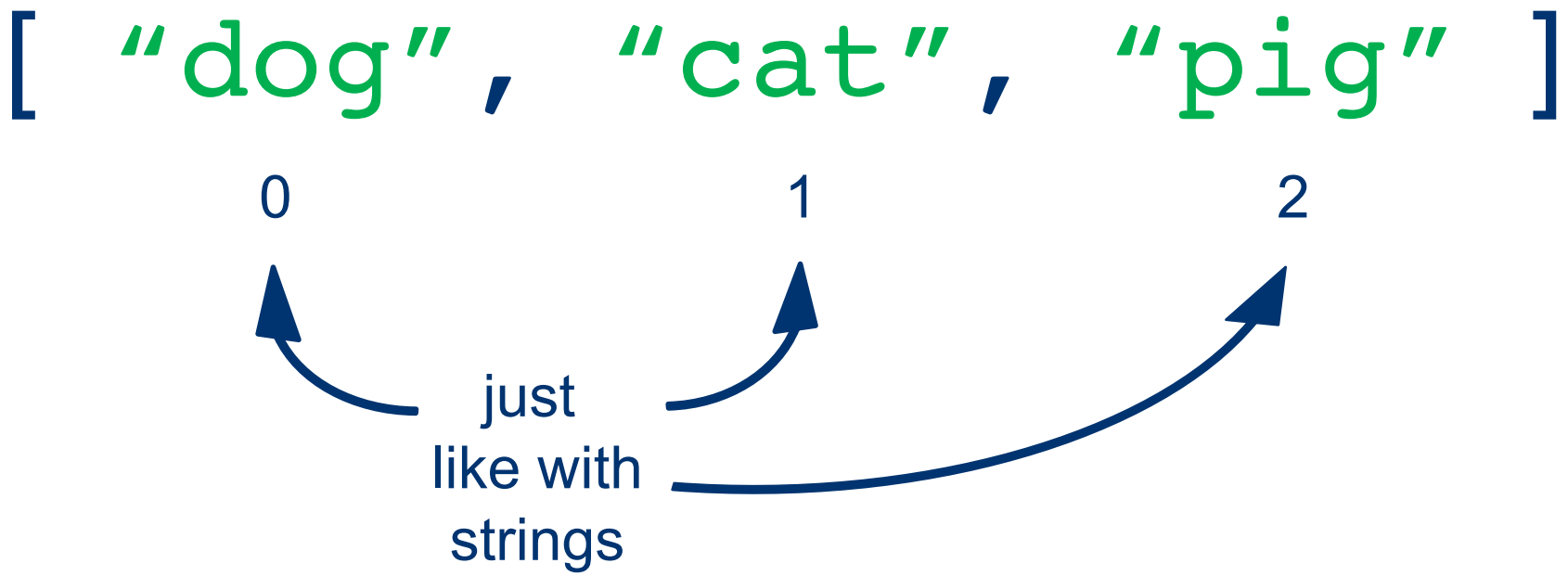
`list` of **`floats`**

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

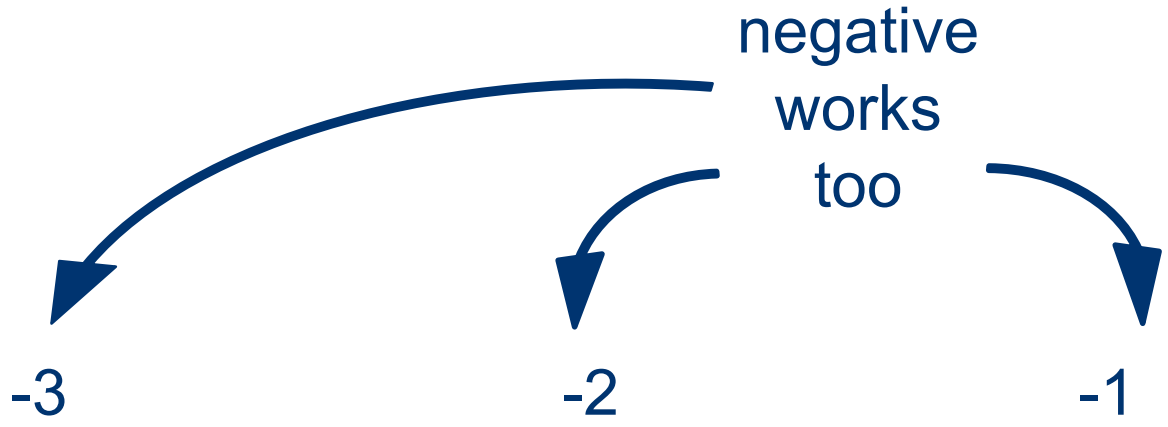
list of **strings**

```
[ “dog”, “cat”, “pig” ]
```

Indexing a `list`



Indexing a `list`

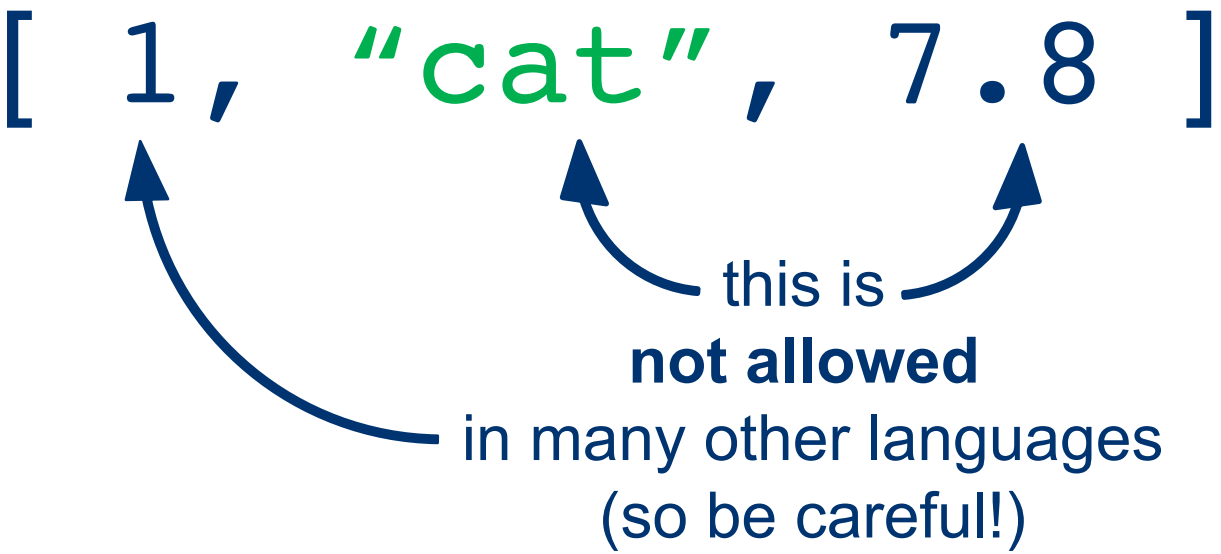



```
[ "dog", "cat", "pig" ]
```

0 1 2

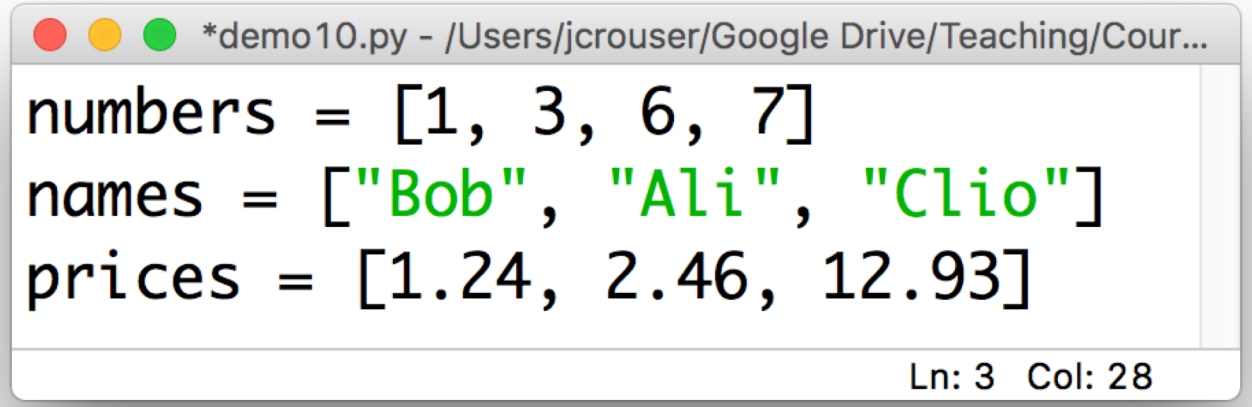
Weird **python** thing

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



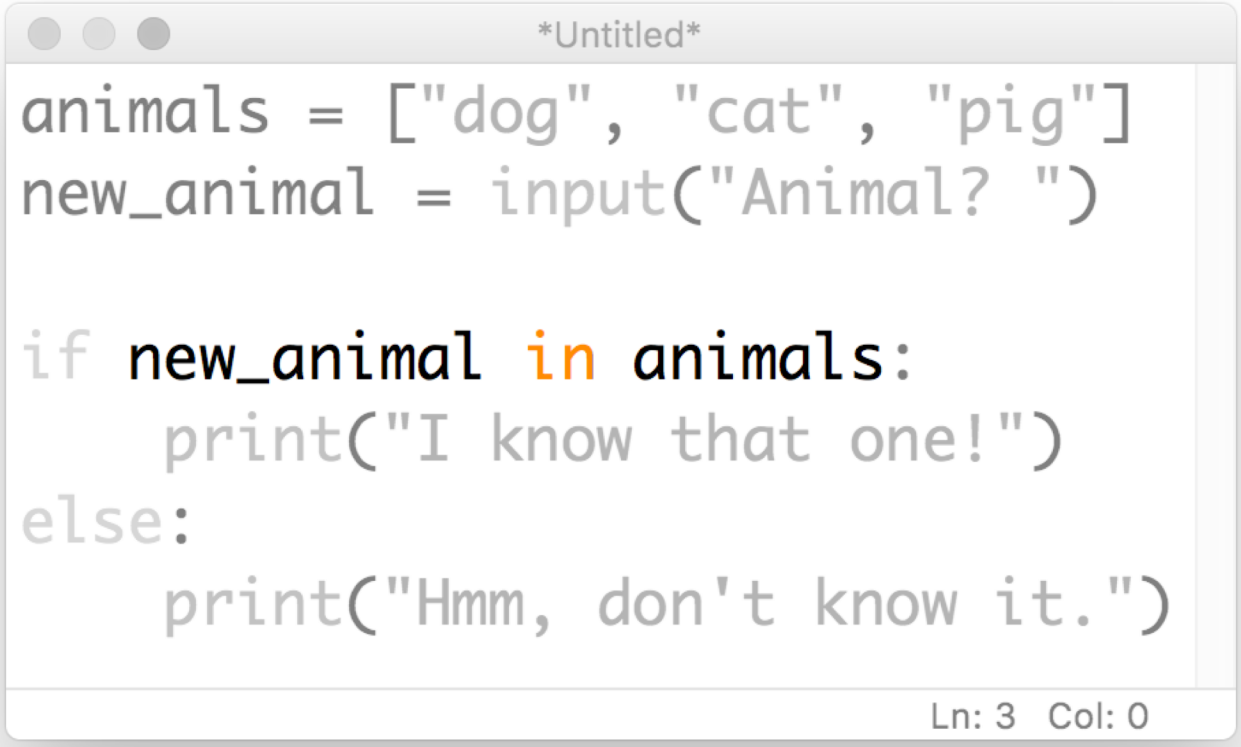
Naming convention

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



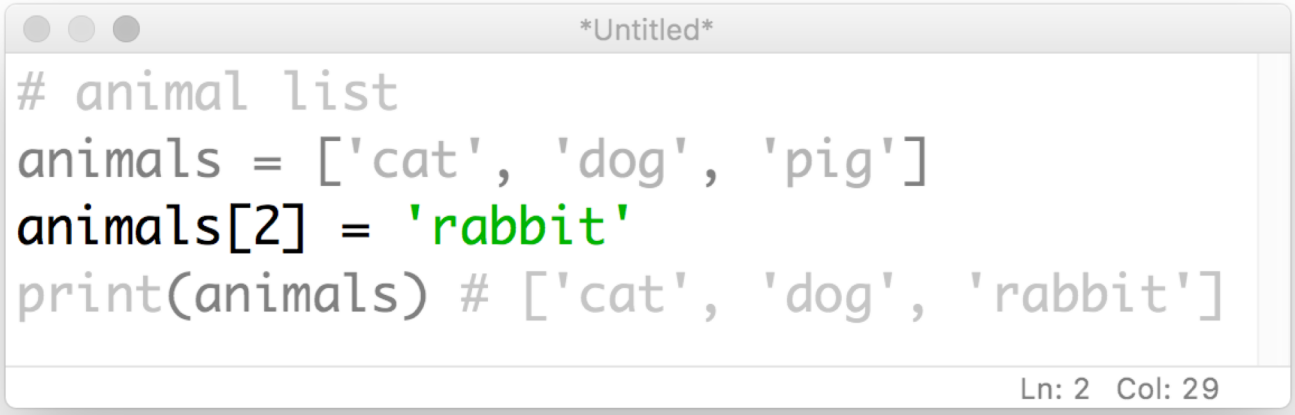
Iterating through items `in` a `list`

Checking membership `in` a `list`



Overwriting an item in a `list`

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



Discussion

What happens when we tr 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`:

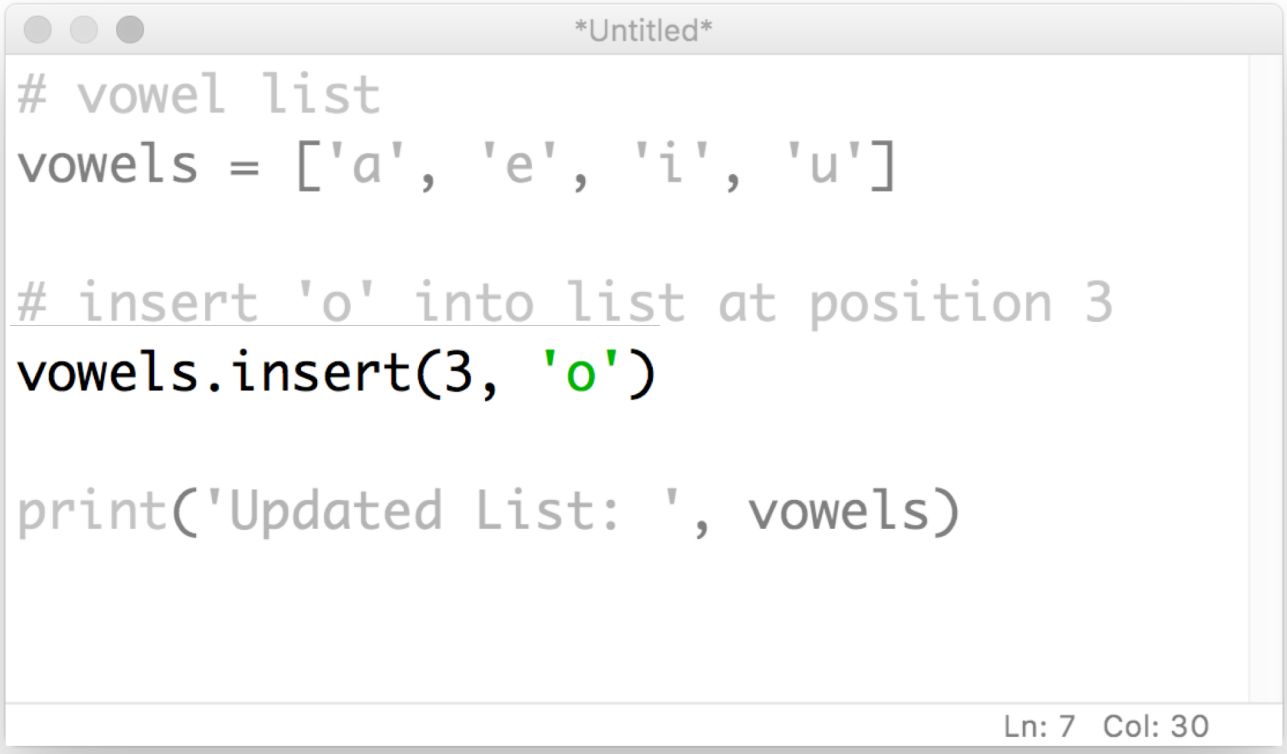
Untitled

```
# 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:



`list` methods: `.remove()`

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

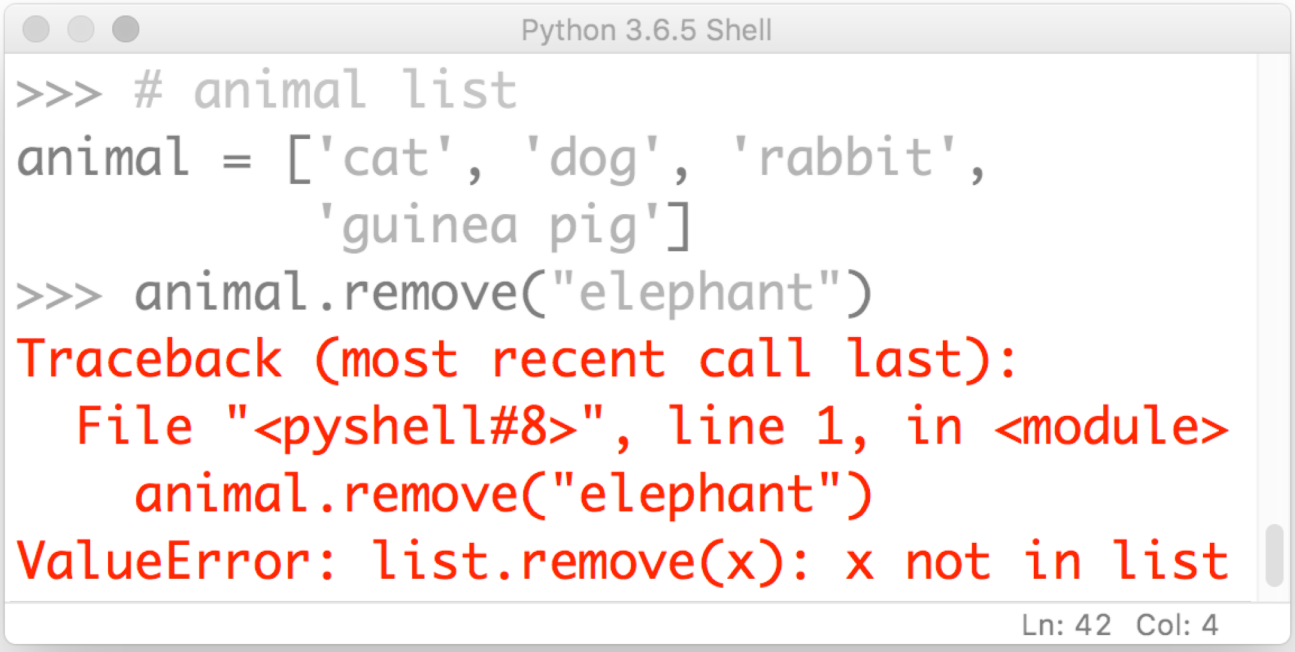
Untitled

```
# 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

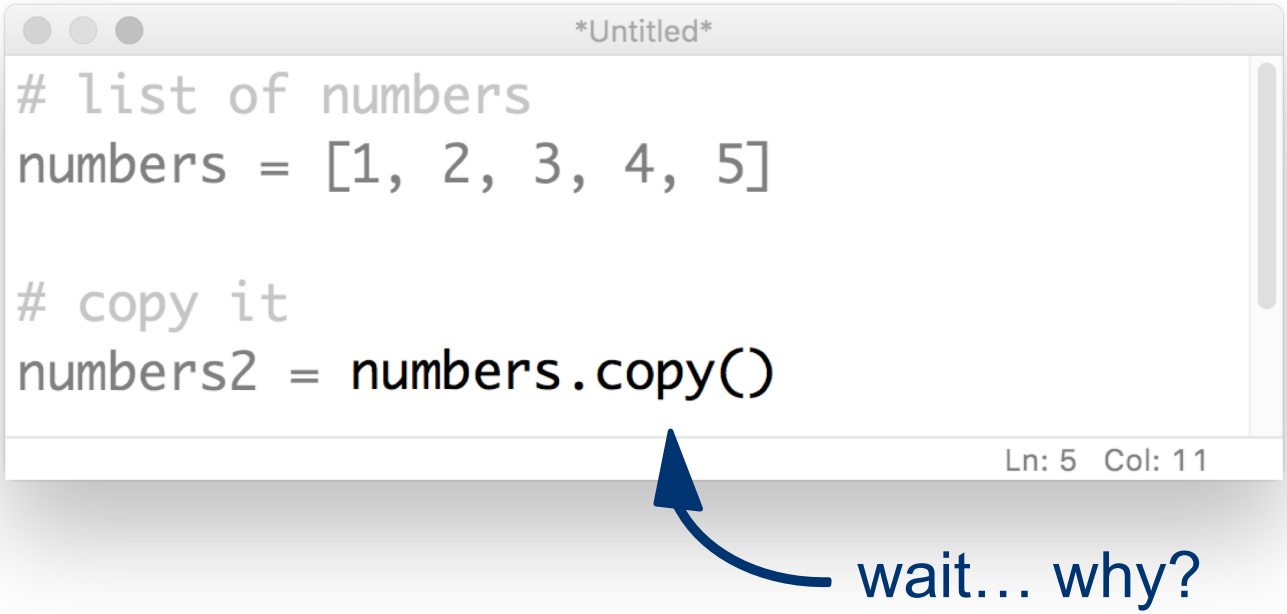
`list` methods: `.remove()`

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



`list` methods: `.copy()`

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



An important note about copying a `list`

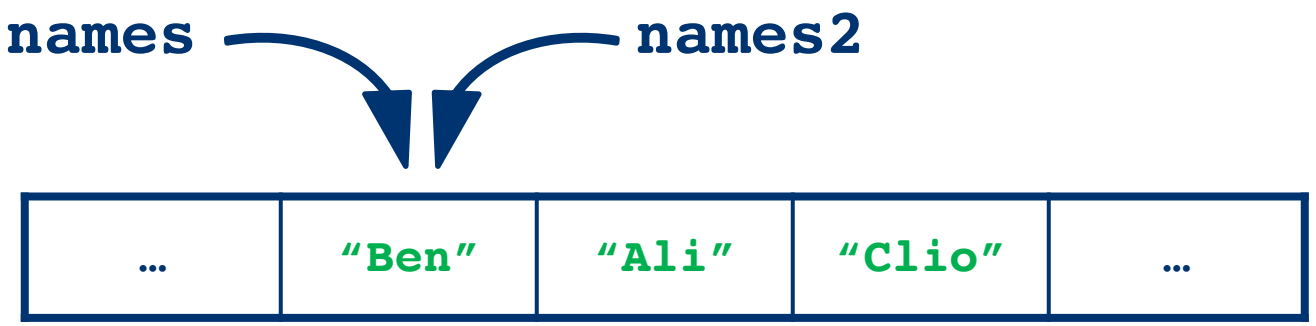
- Usually when we want to copy a string or a number, we just say something like:

```
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`

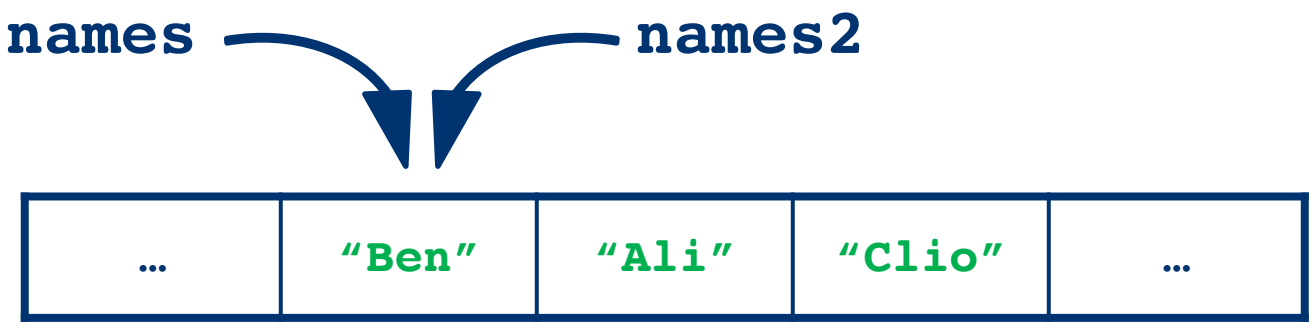
- Let's say we have a **`list`** stored in memory:
  `names = ["Ben", "Ali", "Clio"]`



- 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"]`

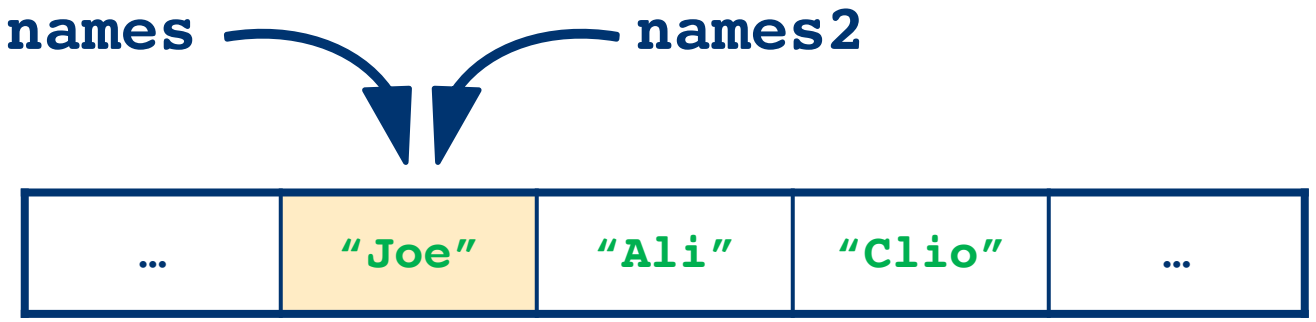


- And then we say `names2 = names`
- If we then say:

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

An important note about copying a `list`

- Let's say we have a `list` stored in memory:
  `names = ["Ben", "Ali", "Clio"]`

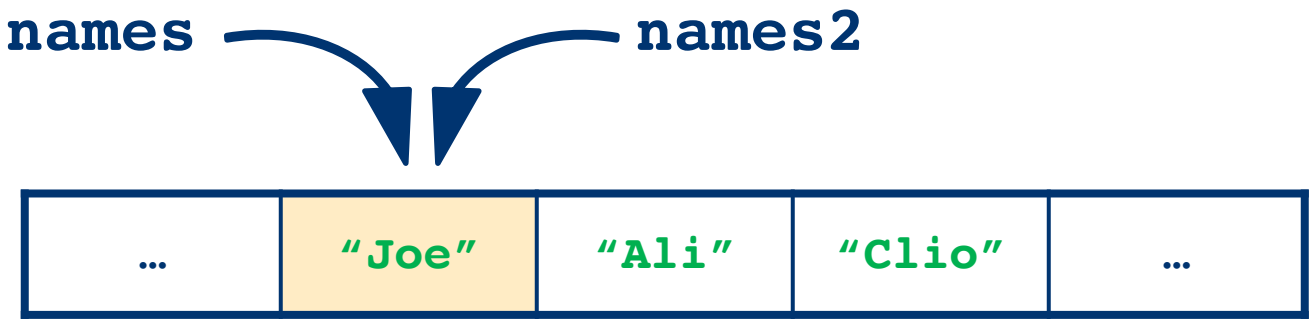


- And then we say `names2 = names`
- If we then say:

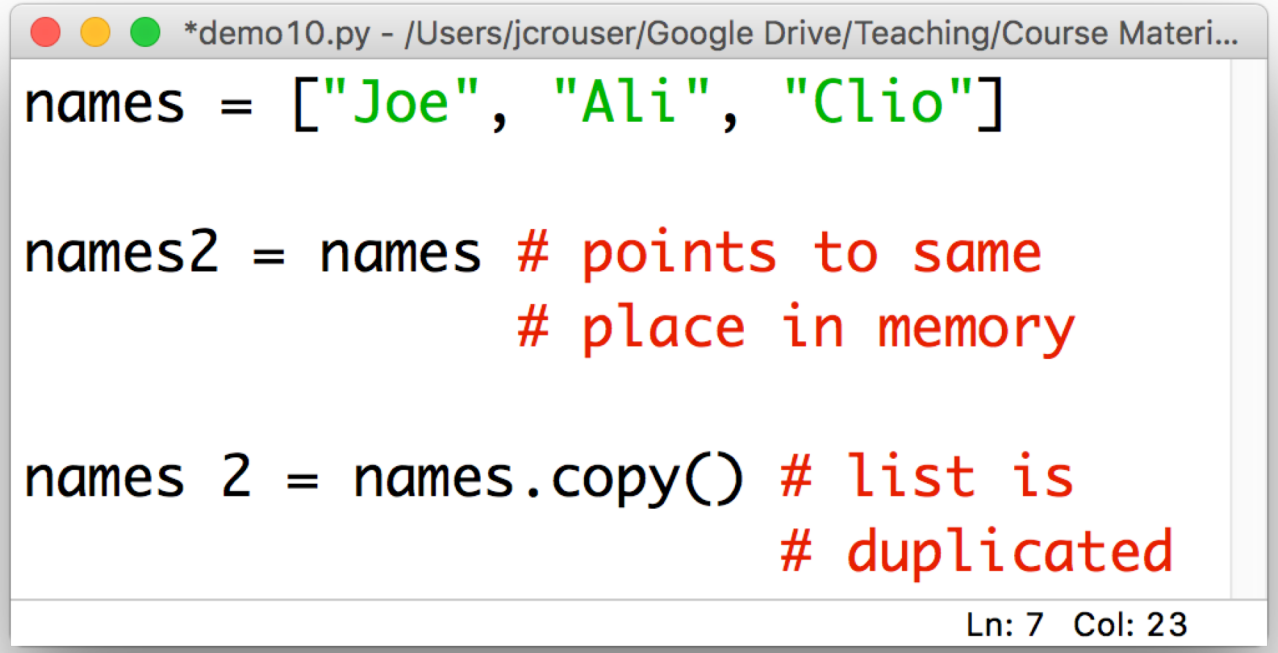
```
names2[0] = "Joe"
```

An important note about copying a `list`

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

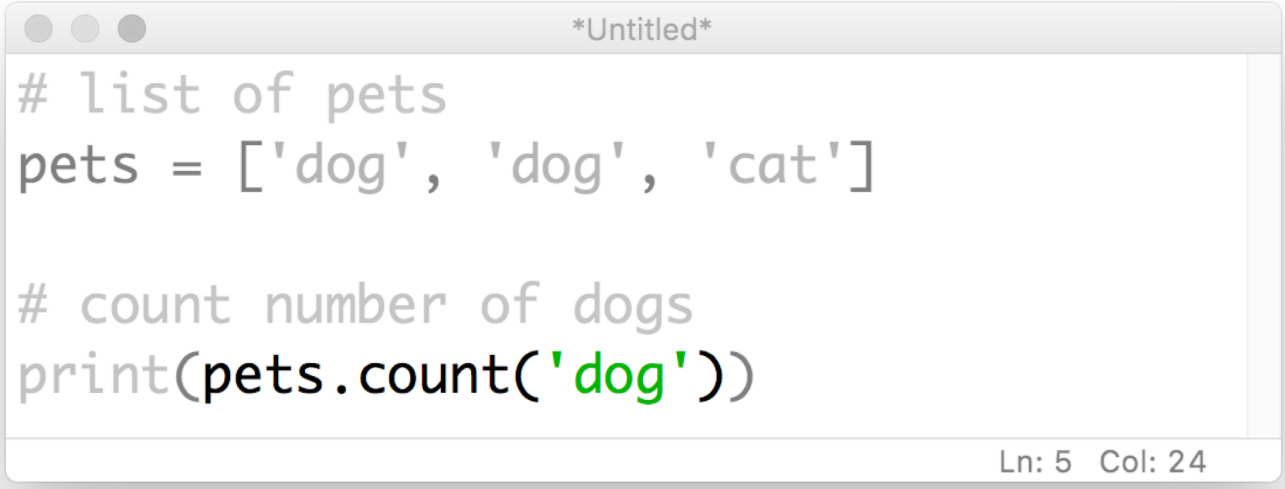


Recap: copying **`lists`**



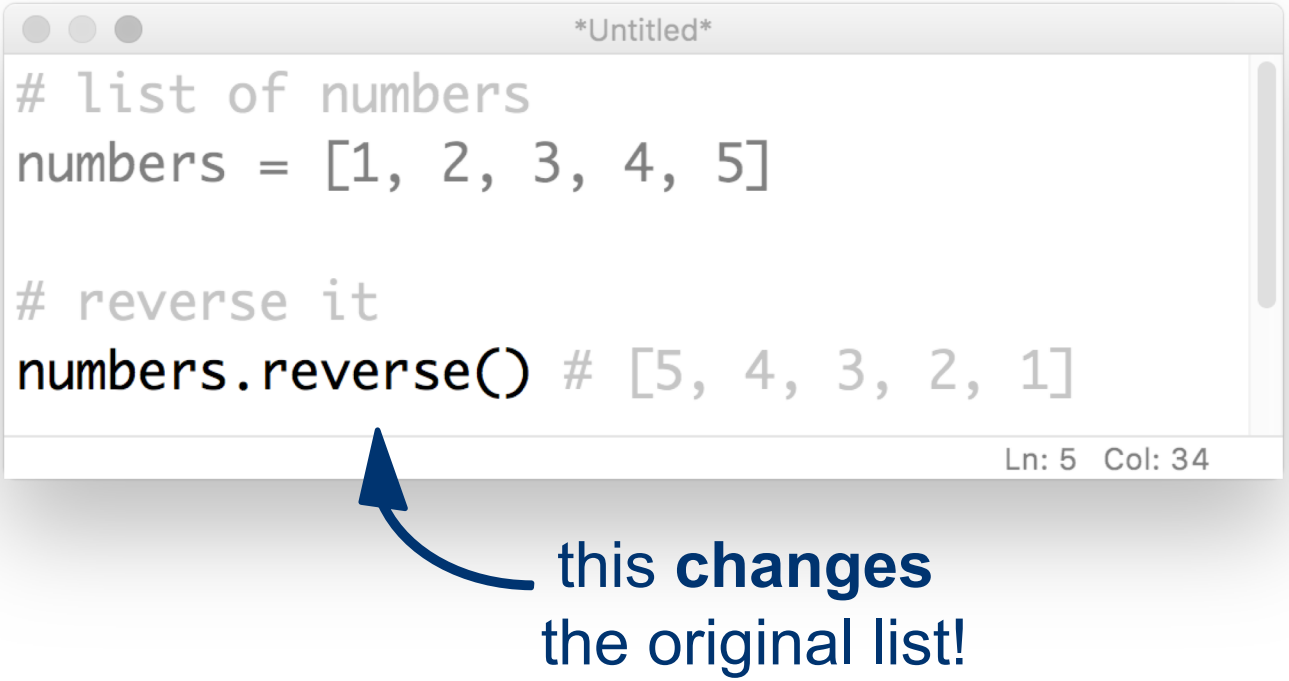
`list` methods: `.count()`

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



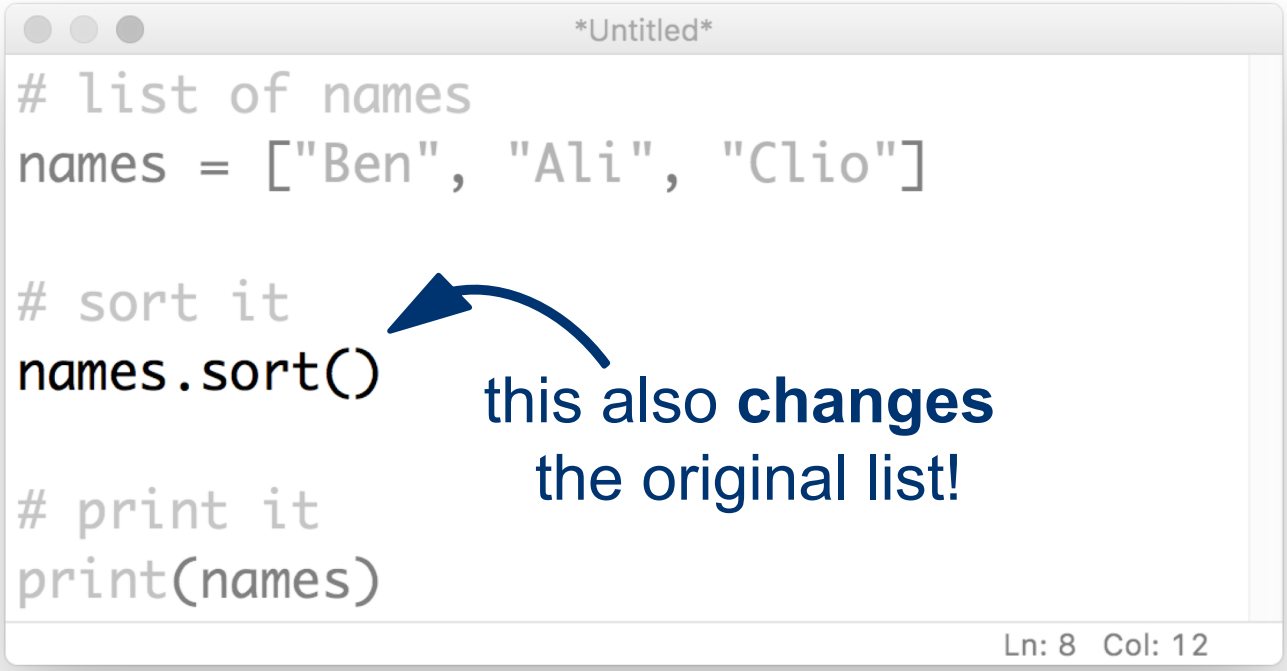
`list` methods: `.reverse()`

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



`list` methods: `.sort()`

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



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 ←→ dictionaries