Warmup problem (L16)

Read a file “in.txt”, and write it out to “out.txt” in order from the shortest line to the longest line.

NumPy and matplotlib

CS114 M7

NumPy for science

CS114 M7

NumPy

- NumPy (prounounced “num-py” if you’re boring or “nump-ee” to be just *slightly* irritating) is a Python module for *numerics*
- In particular, it’s very useful for *linear algebra*
- … which you’ll learn in PHYS249, not here.
- Point being: NumPy’s not going to offer anything you couldn’t already do, but that’s because we’re not covering the fun stuff ☹

Linear algebra

- My favorite discipline of continuous mathematics!
- Deals in *vectors*, *matrices*, and (sometimes) *tensors*
  - A vector is a list of numbers,
  - a matrix is a 2D field of numbers, and
  - a tensor is a higher-dimensional group of numbers
- Includes a large set of rules, e.g., how to multiply whole matrices of numbers

NumPy module

```
import numpy as np
```

- We haven’t seen an “**`as`**” import yet
- It does what you think it does: imports the `numpy` module, but gives it the name “`np`” instead of “`numpy`”
- You can **`import`**-**`as`** any module, but `numpy` is almost *always* imported as `np`

NumPy does numerics!

```
np.sin(1) → 0.8414709848078965
np.abs(-42) → 42
np.sum([2, 4, 6, 0, 1]) → 13
```

- But, simple functions over numbers isn’t its purpose

NumPy Arrays

CS114 M7

NumPy arrays

- The main thing that NumPy gives us is the *NumPy array*
- A NumPy array is like a list, but *fassssster*
- It gets more speed by:
  - *Fixed types.* Even though we prefer that our lists be of only one type, Python doesn't stop us from mixing. Arrays are one type.
  - *Fixed size.* Once an array is created, it cannot be expanded or contracted. There is no `append` or `pop`.

Reference types!

- Like lists, NumPy arrays are *mutable* and *reference types*
- Remember all that that implies:
  - You can index to change values
  - If you set `y = x`, changes in `x` are visible in `y`
  - If you pass an array to a function, it can change your array

It’s gonna be a while…

- Because NumPy arrays are like lists but more restrictive, I need to do a lot of setup before I can do examples
- Nearly every list example I’ve written can be swapped for arrays with no other change
- Arrays are sequences, so all sequence examples too (including loops, of course!)

Pedantry corner

- “List” and “array” are terms used in almost all programming languages
- … and Python is the only one I know of that uses them like this
  - What Python calls lists, most languages either don’t have, or call arrays or arraylists
  - What NumPy calls arrays is all that many languages have, and the rest call them “typed arrays” or “machine arrays”

Pedantry corner (cont.)

- Saying “NumPy arrays” every time is going to get dull, so I’ll usually just say “arrays”
- There’s nothing else called “array” in Python, so when I say “array”, I mean “NumPy array”

Creating an array

- There is no new syntax for arrays
- You create one by either
  - converting any other sequence into an array, or
  - using a function that creates simple arrays (e.g. an array of all zeroes)

Creating an array

- Converting a list:
  - `np.array([2, 4, 6, 0, 1])`
  - `np.array([1, 2, 3.14159])`
    - This will convert the 1 and 2 into floats

  -
    ```
    lst = []
    while …:
        … lst.append(x) …
    np.array(lst)
    ```
    - (You can create the list programmatically!)

The dtype

- When you create an array, it has a *dtype*
    - This stands for *datatype*, and is the type of the data in the array
    - `arr.dtype` to see it
- The dtypes should look similar to the types we've seen, but with extra info
    - int*64*, float*64*
- The number is the size in bits
    - We shouldn't usually have to care, but that's what it means if you're curious

The dtype

- Values retain their NumPy-specific dtype when you pull them out of the array
- The dtypes are compatible with their normal types
- E.g.,

```
arr = np.array([1.0, 2.0, 3.0])
arr.dtype # float64
x = arr[1] # Still a float64,
           # but you can pass it
           # to a function that
           # wants float
```

Creating silly arrays

- `np.array` will let you create arrays of strings, but it’s extremely confusing
- E.g., if I do this:
  `np.array(["one", "two", "three"])`
  the dtype is now `"<U5"`
- That means “strings that are less than or equal to five characters”
- If I try to put `"seventy"` in there, it becomes `"seven"`

Sensible arrays

- Just stick to arrays of numbers!

What’s the point?

- Arrays do math *component-wise*

```
x = np.array([1.1, 2.2, 3.3])
y = np.array([2.0, 4.0, -1.0])
x * 2 → np.array([2.2, 4.4, 6.6])
x < 3 → np.array([True, True, False])
x – 1 → np.array([0.1, 1.2, 2.3])
x * y → np.array([2.2, 8.8, -3.3])
```

Compound assignment operators

- With a list or array, `x*2` gives a *new* list or array
- What if I wanted to component-wise multiply an array by 2 in place?
- Python has *compound assignment* for this:

```
x *= 2
print(x) # np.array([2.2, 4.4, 6.6])
x -= 1
print(x) # np.array([1.2, 3.4, 5.6])
```

Compound assignment operators

- Compound assignment operators aren’t just for arrays!
- We’ve had them all along, just this is the first time when they’re really important

```
n = 42.0
n /= 7.0
print(n) # 6.0
l = [1, 2, 3]
l *= 2
print(l) # [1, 2, 3, 1, 2, 3]
```

One weird caveat

```
x = np.array([1, 2, 3])
y = x / 2 # Works
x /= 2 # Crashes
```

- Huh?
- Remember, an array has a fixed type
- Updating it in place can’t change that type
- Division makes floats, so this is a no-go

In-lecture quiz (L16)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q1: What numbers are printed?

```
x = np.array([1, 2, 3])
y = x
z = x * 2
y *= 2
print(x)
```

A. Nothing or an error

B. 1, 2, 3

C. 2, 4, 6

D. 4, 8, 12

In-lecture quiz (L16)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q2: At the end of this code, what is the type of x?

```
import math
a = np.array([math.pi, 2, 1])
x = a[1]
```

A. This code has an error (no type)
B. float64
C. float
D. int64
E. NumPy array of float64s

NumPy functions

- And now you know why NumPy has functions that were already in `math`

```
y = np.array([2.0, 4.0, -1.0])
np.sin(y) → np.array([2.0, 4.0, 1.0])
math.sin(y) # CRASH!
```

Testing with arrays

- Component-wise gives us a new way to test!

```
assert np.all(
    np.abs(makesAnArray(…) – expected) < 0.001
), "Checks that all the booleans are True"
```