Warmup problem (L17)

Turn a CSV file into a dictionary of *arrays* of numbers. The dictionary should associate each column name with an array.

Array type

- The array type is in `numpy.typing`

  ```
  import numpy.typing as npt
  ```

- It’s called “`NDArray`” for reasons we’ll get to… eventually
  (ND stands for “*n*-dimensional”)

```
arr: npt.NDArray[np.float64]
```

This needs to be a NumPy dtype, not (e.g.) float

Array type

- The array type is in `numpy.typing`

  ```
  import numpy.typing as npt
  ```

- It's called "`NDArray`" for reasons we'll get to… eventually
  (ND stands for "*n*-dimensional")

```
arr: npt.NDArray[np.float64]
```

The dtypes are in `numpy` (`np`), *not* `numpy.typing` (`npt`)
(designed by monkeys, this library)

Other ways of arrays

CS114 M7

Arrays

- Since you can’t append to produce an array, how do you make big ones?
- Use functions that create simple arrays!
  - `np.zeros`
  - `np.ones`
  - `np.linspace`

np.zeros, np.ones

```
np.zeros(4) → np.array([0.0, 0.0, 0.0, 0.0])

np.ones(4) → np.array([1.0, 1.0, 1.0, 1.0])

np.zeros(4, dtype=int) → np.array([0, 0, 0, 0])

np.ones(4, dtype=bool) →
          np.array([True, True, True, True])
```

- The main use of `.zeros` isn't the zero[e]s, it's creating an array of a given size without appending!

`np.linspace`

- `range` only works with `int`s
  - This is because of mathematical difficulties with the step; remember, floats are approximate!
- For something like `range` to work over floats, we can't use a step
- Instead, we need a *count* (i.e., divide this range into this many parts)

np.linspace

```
np.linspace(0, 4, 5) →
      np.array([0.0, 1.0, 2.0, 3.0, 4.0])
np.linspace(1.1, 2.2, 3) →
               np.array([1.1, 1.65, 2.2])
```

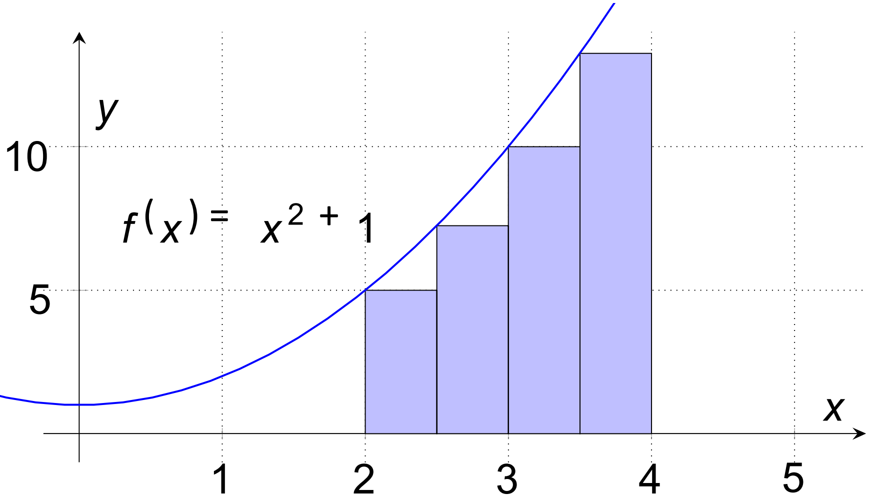
- Unlike `range`, upperbound *inclusive*
  - Optional parameter `endpoint=False` to turn this off
- Gives evenly spaced floats in the range

`np.arange`

- NumPy *does* have `.arange` (array-range), which behaves more like `range`
- `.arange` does work with floats
- Often not a good idea because float rounding can lead weird places late in the range

Example (finally!)

Let’s do some pre-calculus: get the Riemann sum of a curve (the approximate area under the curve by splitting it into rectangles and summing their areas)



```python
def approxArea(
    f: typing.Callable, lowerbound: float,
    upperbound: float, bins: int
) -> float:
    xs = np.linspace(
        lowerbound, upperbound, bins,
        endpoint=False
    )
    ys = f(xs)
    areas = ys * (upperbound - lowerbound) / bins
    return np.sum(areas)

def f(x: npt.NDArray) -> npt.NDArray:
    return x**2 + 1

print(approxArea(f, 2, 4, 4))
```

Vectorization

CS114 M7

Let’s `sin`!

- If we try to pass in `math.sin` as `f`, we get a long and confusing error
- While NumPy does things element-wise, to get that, you have to stick to NumPy functions!
- `np.sin` works with both numbers *and* arrays

```
print(approxArea(np.sin, 0, math.pi, 1000))
```

New terminology

- Functions that work element-wise are called *vectorized functions*
- Why? Because in linear algebra, we call these arrays *vectors*

Vectorizing our own code

- Let’s try to sum under the curve of a step function

```
def stepFunction(x: float) -> float:
    if x < 0:
        return -1
    elif x < 2:
        return 0
    else:
        return 1
```

Vectorizing our own code

- We got a big, confusing error! Why?
- We can do exponents or adding element-wise, but what would it mean to do “**`if`**” element-wise?
- To make this function to work, it needs to be run for each element in the array
- We could do this ourselves, but because this need is common, NumPy can do it for us!

Vectorizing our own code

```
stepFunction(np.linspace(
    -5, 5, 1000
)) # CRASH!

vectorizedStepFunction =
          np.vectorize(stepFunction)
vectorizedStepFunction(np.linspace(
    -5, 5, 1000
)) # Works!

print(approxArea(
    np.vectorize(stepFunction),
    -5, 5, 1000
)) # -2.0
```

`vectorize` is a weird function

- What are the types of `np.vectorize`?
- This is a function that takes a function as an argument and returns a function!
- Remember: functions are values!
- This function is a great example of *why* functions are values

In-lecture quiz (L17)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- After this code, what is `x` (floats approx.)?

```
x = np.linspace(0, 5, 4)
```

```
A. np.array([0.0, 4.0])
B. np.array([0.0, 1.0, 2.0, 3.0, 4.0])
C. np.array([0.0, 1.667, 3.333, 5.0])
D. np.array([0.0, 1.25, 2.5, 3.75])
E. np.array([0.0, 1.333, 2.667, 4.0])
```

Multi-dimensional arrays

CS114 M7

Multi-dimensional arrays

- Like lists, you can make multi-dimensional arrays by putting lists in lists

```
board = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
```

Terminology

- NumPy actually calls these (and all arrays) *n-dimensional arrays*
- Hence why the type is `NDArray`

n-dimensional arrays

- Like 1-dimensional arrays, *n*-dimensional arrays do things element-wise

```
print(board * 2)
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

Indexing

- Unlike lists, you index with both the *y* and *x* position in one go

```
print(board[0, 2])
3
```

- This would also work with `[0][2]`, but…

Slicing

- You can slice in both axes at the same time!

```
print(board[:, 1:2])
[[2]
 [5]
 [8]]

print(board[1:, :])
[[4 5 6]
 [7 8 9]]
```

- This would need loops with lists

Huge slicing caveat

- Slicing *lists* copies
- Slicing *arrays* does not
- If you want a copy, you want the `.copy()` method
- We won’t go in depth into the implications of slices not copying; if you want more detail, the keyword to search for is “array *view*”

Larger *n*-dimensional arrays

- `np.zeros` and `np.ones` can be given a shape, instead of just a size:

```
a = np.zeros((2, 3))
print(a)
[[0. 0. 0.]
 [0. 0. 0.]]
```

Be careful of parentheses! This is a function that takes a tuple as an argument, not a function with two arguments!

`len` and n-dimensional arrays

- `len(a)`, where `a` is an *n*-dimensional array, is the length of the *first* dimension
- `a.shape` is a tuple of the length of *every* dimension (just like the tuple for `np.zeros`)

n-dimensional arrays

- I only showed two-dimensional arrays, but three-dimensional arrays and so on are also possible
- In linear algebra,
  - One-dimensional arrays are called vectors,
  - two-dimensional arrays are called matrices, and
  - three-and-higher-dimensional arrays are called tensors

n-dimensional arrays

- You’re unlikely to need 3D or higher-dimensional arrays for this course
- You *will* need them for linear algebra, so it was worth mentioning them

Plotting

CS114 M7

matplotlib

- A popular module for drawing plots is matplotlib
- It’s extremely powerful (almost an entire programming language on its own), but we only need one part of it: pyplot
- It has its own excellent tutorial, so we’ll follow (our own version of) that

pyplot tutorial

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/pyplot.ipynb
```

(Original from https://matplotlib.org/stable/tutorials/pyplot.html)

NOTE: This tutorial is considered part of the course materials! If you're reading these slides, make sure you follow the tutorial!

Format strings

- Each character in the string is a part of the style with which the line will be plotted (and a few two-character styles)
- Colors: **b**lue, **g**reen, **r**ed, **c**yan, **m**agenta, **y**ellow, blac**k**, **w**hite
- Markers: "**.**" point, "**o**" circle, **s**quare, **^>v<** triangles, "**+**" plus, "**x**" cross
- Lines: "**-**" solid, "**--**" dashed, "**:**" dotted
- There are more than just these

Format strings

- Default is `"b-"`, but the color and markers/line are separate
  - E.g., if you do `"r"` it'll draw `"r-"`, and if you do `"o"` it'll do `"bo"`
- Markers and lines can be combined but don't need to be

Restarting pyplot

- Every time you call `.show()`, pyplot shows the current plot and resets, ready for a new plot
- If you mess something up, all that internal plotting state will still be there in the kernel
- Use `plt.figure()` to reset it
    - Or, just make sure you always `.show()`!

Let’s use it!

Let’s plot the temperature over time from nino34.csv

```
import csv
import matplotlib.pyplot as plt

xs = []
ys = []
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    for row in nino:
        xs.append(
            (int(row["YR"])-1950)*12 +
            (int(row["MON"])-1)
        )
        ys.append(float(row["TOTAL"]))

plt.plot(xs, ys)
plt.xlabel("Months since January 1950")
plt.ylabel("Temperature (c)")
plt.show()
```

Module summary

CS114 M7

Module summary

- NumPy arrays are like lists but more consistent
- NumPy arrays are component-wise
- Component-wise makes lots of things much easier and clearer
- *n*-dimensional arrays
- matplotlib/pyplot for plotting