Extra Material (Optional): NumPy

Matplotlib with NumPy

NumPy is used for numerical processing, comparing its fundamental data type (a "numpy array") with lists

You will also see some NumPy in Lab 06 (provided)

NumPy is very useful when combined with Matplotlib! You won't be required to use it for Part C, but it is useful to know the basics of NumPy and how you could incorporate it with Matplotlib for future work

Resources

You don't need to know more than what is presented in this slide deck/provided code, but here are a few recommended resources if you're curious about going deeper into NumPy (in general, we don't recommend resources that you find on your own, unless you know how to identify the "good" from the "bad")

This Quick Start was posted as a reading, which is the recommended reading for getting started with NumPy (you don't need to know more than what we cover in class, but it is a helpful resource to go deeper into the basics). Make sure you do not follow the instructions for conda, which we strongly **do not** recommend for CS1.

A *very* nice Visual Guide to NumPy and ND-Arrays/Matrices (credit cited for some of the visualization on these slides!)

Matplotlib with Numpy

We have seen how to plot data with `(x, y)` points and two lists of `[xs]`, `[ys]`

In practice, Matplotlib is most commonly used with another library called NumPy, which provides *a ton* of useful features for data processing

While you don't need numpy to use Matplotlib, it is common enough that we'll learn some of the basics (you don't need to know more than what we teach!)

NumPy

NumPy is one of the most popular Python libraries, with many applications:

- Scientific/mathematical computing
- Serves as the "backend" to a lot of other libraries, including Pandas
- Image processing
- Machine learning

There are a lot of features in NumPy, but when integrating with Matplotlib we leverage its special "NumPy" array, which is analogous to a Python list

Visualizations are *very* useful when it comes to learning NumPy, and credit is given to this Visual Guide for some diagrams featured in this slide deck.

A Basic NumPy Array

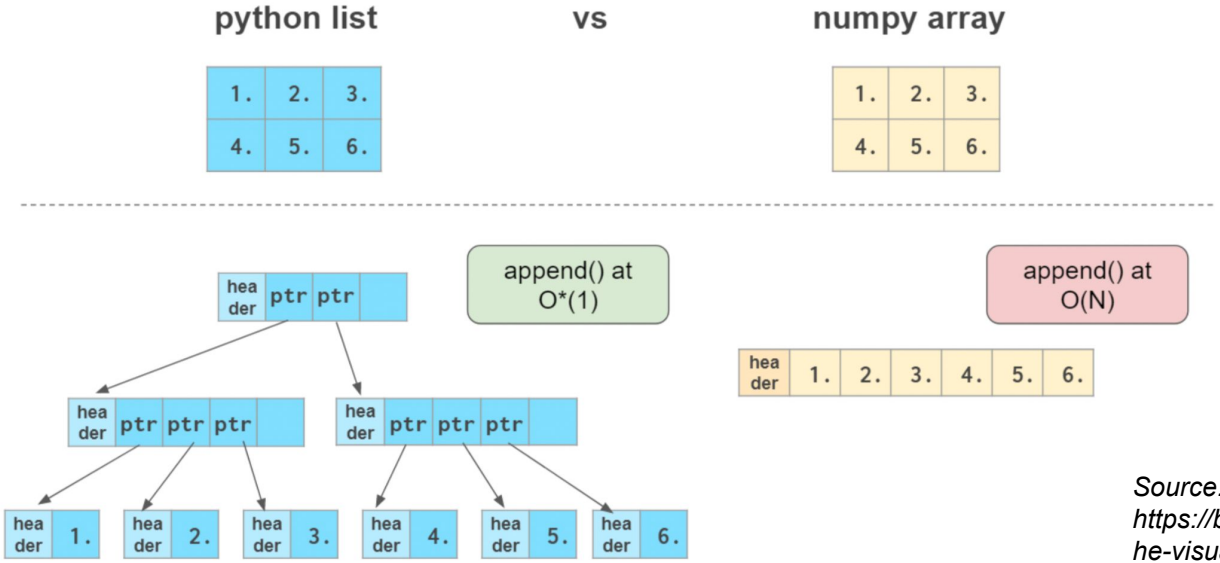
The easiest way to construct a numpy array is using **`np.array(lst)`**

- By default, the datatype matches that of the datatypes passed (they must be homogenous, meaning the same type); this can also be specified with a keyword argument, dtype=<datatype> but only if it is different than (but compatible with) the value types of the passed list
- The array has a "shape" in terms of rows and columns, just like in matrix terminology; the below array has 3 values in a single list (1D)

```
a = np.array([1., 2., 3.])
```

NumPy Arrays vs. Python Lists

You don't need to know about O(n) yet, but here's a good visualization (this slide was discussed in more detail in lecture) about the difference between arrays and lists under the hood:



Source: https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d

NumPy Arrays vs. Python Lists

Unlike Python lists, NumPy arrays:

- Require fixed types (e.g. all ints, all floats, all strs) and do not allow for mixed types
- Are *generally* faster than Python lists
  - No type checking/conversion required when iterating over
  - Less memory taken under the hood (contiguous memory)
  - Caveat: Appending to lists is usually faster than appending to a NumPy array
- Have easy-to-specify "shapes" to represent N-Dimensional arrays (matrices)
- Have a ton of useful operator overloading features for common matrix operations
- Note: There is nothing a NumPy array can do that a list cannot

Q: If NumPy arrays are faster than lists, why don't we always use NumPy? (they aren't always faster!)

NumPy with Matplotlib

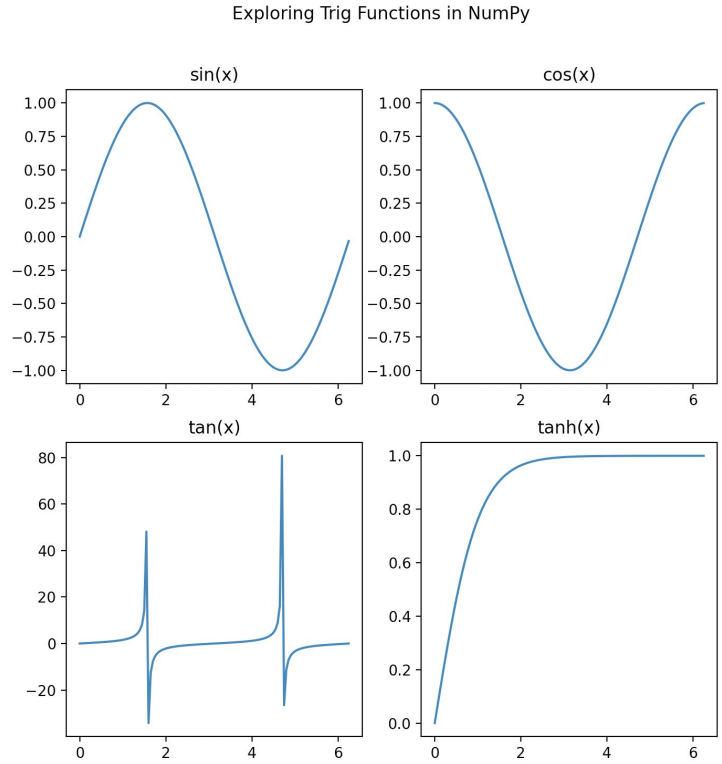
```
# Use np's arange to create a range of float radians

xs = np.arange(0, math.pi*2, 0.05)
# Using numpy's basic trignometric functions
ys1 = np.sin(xs)
ys2 = np.cos(xs)
ys3 = np.tan(xs)
ys4 = np.tanh(xs)

# Quadrant (2 x 2) plot
fig, axs = plt.subplots(2, 2)

# (2, 2) -> 2D list of 2 rows, each holding two cols
(ax1, ax2), (ax3, ax4) = axs

# Plot sin, cos, etc. functions
ax1.plot(xs, ys1)
ax2.plot(xs, ys2)
...
```

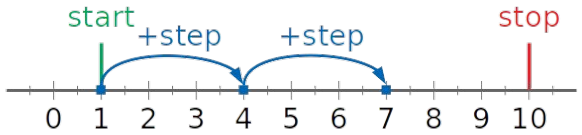


`numpy.arange([start, ]stop, [step, ]dtype=None, …)`

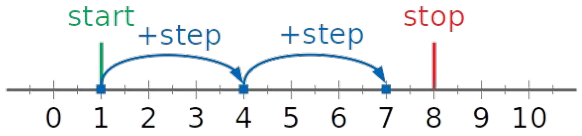
In Python, we have seen how to use **`range`** to generate a sequence of integers

There is a similar function in NumPy called **`arange`** ("array" range), which also returns a sequence of values, but supports float values as well (where as range requires ints)

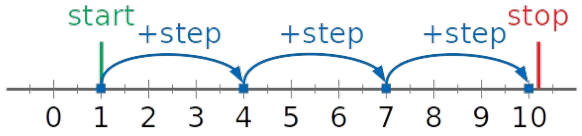
```
>>> np.arange(1, 10, 3)
array([1, 4, 7])
```



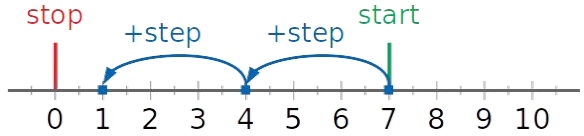

```
>>> np.arange(1, 8, 3)
array([1, 4, 7])
```




```
>>> np.arange(1, 10.1, 3)
array([1., 4., 7., 10.])
```




```
>>> np.arange(7, 0, -3)
array([1, 4, 7])
```



Source: https://realpython.com/how-to-use-numpy-arange/

`range(...)` vs. `np.arange(...)`

`range(...)`

- Default function in Python
  - Independent of modules, so can be more efficient for simple use cases
- Returns a successive collection of numbers which are individual values represented as a range object
- **`range`** is often faster than arange() when used in Python for loops, especially when there's a possibility to break out of a loop soon. This is because range generates numbers in the lazy fashion, as they are required, one at a time.

`np.arange(...)`

- Part of the NumPy library (requires importing numpy)
- Returns a special ndarray (a NumPy array)
- Useful for large datasets and fast numerical processing with NumPy
- Can be used with many NumPy features, including special *, +, -, etc. operations

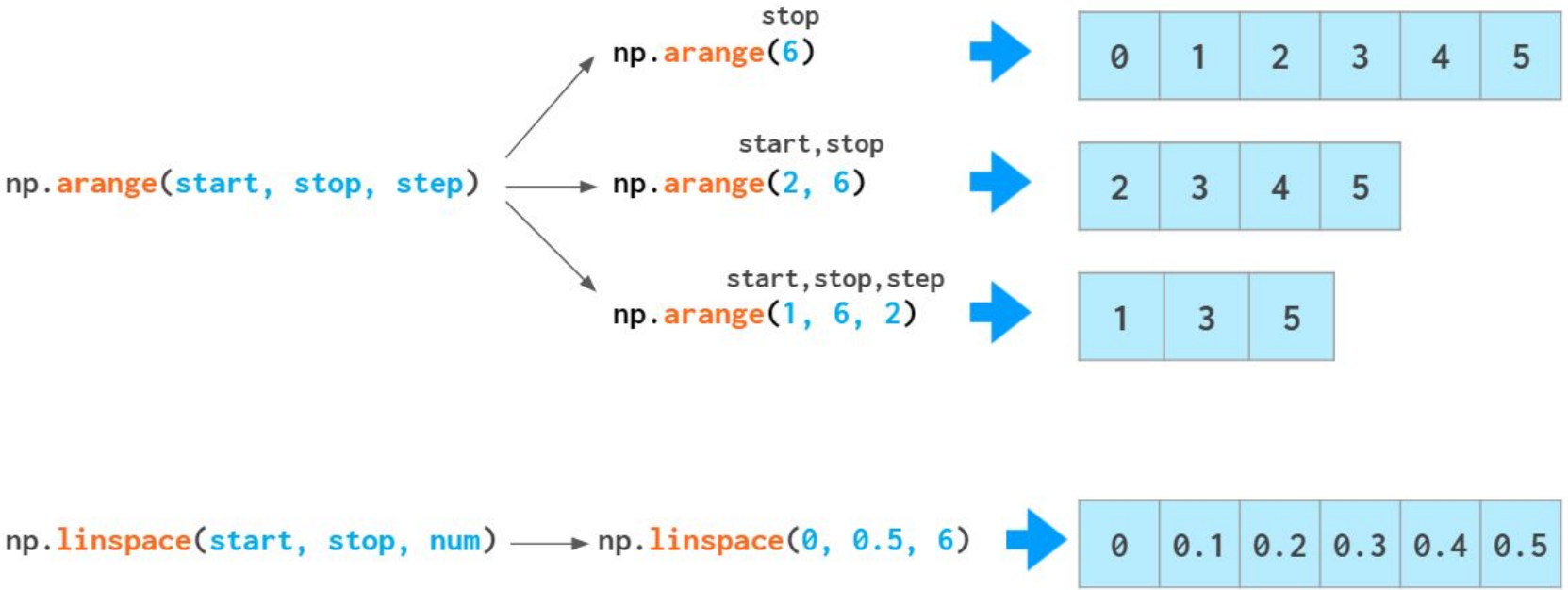
`np.arange` vs. `np.linspace`

`np.arange` and **`np.linspace`** are the two most popular functions to quickly generate an array of values in NumPy

Both support **`float`**s (different than Python's **`range`**), but:

- **`np.arange`** is used when you have a start, stop (exclusive), and step size
- **`np.linspace`** ("linear space") is used when you want to generate a sequence evenly separated between start and stop, and stop is **inclusive**

`np.arange` vs. `np.linspace` ("linear space")



Wrapping Up

You don't need to know more than what we covered in this lecture but here are the key takeaways:

- The fundamental datatype in NumPy is the array (sometimes called "ndarray" for n-dimensions), which unlike Python's list type:
  - Takes up less memory because all types are homogenous (the same) and the consecutive memory allocated is guaranteed to be enough
  - Is faster for matrix operations as a result, ***only*** if you are not changing the length of the array (if you are, the Python list will be faster, since the memory is not consecutive and thus we don't have to reallocate consecutive space)
  - Can be used with `*`, `-`, `+`, `np.cos`, etc. to quickly compute operations on the N-dimensional array
  - Is very useful to pass to `Matplotlib.pyplot` plotting functions that require a list
- `np.arange` and `np.linspace` are popular functions to generate a sequence of values, particularly with `float`s that cannot be passed to Python's built-in `range` function
- Just because you can do something in NumPy, does not mean every program should use it, as there is overhead to be aware of (discussed in class)