1 Review

Last time:

- Python miscellany
- Questions?
- Today: Numpy

2 Numpy Basics

- NumPy (Numerical Python) is a python package for working with numerical data
- numpy is the standard for data analysis and is used in MANY other scientific packages—it's very important that you're familiar with Python
- Numpy should come with your Anaconda distribution.
- If you don't have numpy, install on the command line: `pip install numpy`
- To import numpy

```
import numpy as np
```

3 ndarray

- An n-dimensional array of items of the SAME type.
- ndarrays are faster and take up less memory than Python lists
- It is MUCH easier to perform calculations using ndarrays (we'll discuss this more!)
- numpy arrays have a shape given by a tuple. `(rows, cols)` for 2d arrays.

ndarray example:

```
import numpy as np
a = array([[ 0,  1,  2,  3,  4],
           [ 5,  6,  7,  8,  9],
           [10, 11, 12, 13, 14]])

>>> a.shape
(3,5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.size
15
```

Creating arrays:

- We can create arrays from an existing list `a = np.array([[2,3,4], [5,6,7])`
- Ones and zeros:

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])

>>> # dtype can also be specified
>>> # This is a 3d array! (we'll talk about axes in a bit!)
>>> np.ones((2,3,4), dtype=np.int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

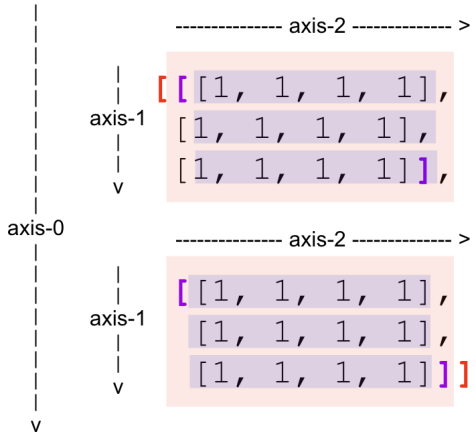
- Create a seqeunce:

```
# evenly spaced range arange([start], stop, [step])
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])

# specify a number of elements across a range
# linspace(start, stop, num)
>>> np.linspace(0, 1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

numpy axes:

- Often a very confusing topic!



- Each dimension corresponds to an axis
- The axis convention is **OUT TO IN**. For example: the 3d array in the figure above has 3 degrees of freedom:
  - **axis-0**: which one of the **two** 2d arrays? (`a[0]` or `a[1]`)
  - **axis-1**: given one of the 2d arrays, which of the **three** 1d arrays (`a[i,0]` or `a[i,1]` or `a[i,2]`)
  - **axis-2:** given one of the 1d arrays, which of the **four** elements? (`a[i,j,0]` or `a[i,j,1]` or `a[i,j,2]` or `a[i,j,3]`)
- For 2d array: `(row, col)`. For 3d array: `(matrix, row, col)`.
- This is why the array has shape `(2,3,4)`

Practice with array indexing:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])

# What is a[1]?
# What is a[1,2]?
# What is a[1,2,3]

# Notice: array slicing still works!
# Notice: a[1] = a[1,:,:] = a[1:...]

# REMEMBER: read left to right, out to in
# What is a[0,0,2:]?
# What is a[0,:,2:]?
# What is a[:,:2,2:]?
```

Manipulating arrays (you can read up on your own):

- `np.reshape`
- `np.concatenate`
- `np.stack`, `np.vstack`, `np.hstack`, `np.vsplit`, `np.hsplit`, `np.column_stack`

4 Vectorized Operations

- Often times, we want to perform element-wise operations on an array. (For example, we want to multiply all elements by some factor). This is called a VECTORIZED operation.
- How would we do this with lists?

```
a = [[1,2,3], [4,5,6]]
# what happens when we do 3*a? (it ends badly)

for row in range(len(a)):
    for col in range(len(a[0])):
        a[i][j] = 3 * a[i][j]
```

  This is SLOW and ANNOYING to write. It's also hard to quickly understand what the code is doing!

- In numpy, mathematical operations are vectorized!

```
a = np.array(a)
a = 3 * a

>>> a
array([[ 3,  6,  9],
       [12, 15, 18]])
```

  Numpy's vectorized operations are implemented in C, so they are FAST! With numpy, it is easier to perform element-wise operations and it's easier to read!

Unary Function: $f(x)$	NumPy Function
$\|x\|$	`np.absolute`
$\sqrt{x}$	`np.sqrt`
Trigonometric Functions	
$\sin x$	`np.sin`
$\cos x$	`np.cos`
$\tan x$	`np.tan`
Logarithmic Functions	
$\ln x$	`np.log`
$\log_{10} x$	`np.log10`
$\log_2 x$	`np.log2`
Exponential Functions	
e^x	`np.exp`

Binary Function: $f(x, y)$	NumPy Function	Python operator
$x \cdot y$	`np.multiply`	`*`
$x \div y$	`np.divide`	`/`
$x + y$	`np.add`	`+`
$x - y$	`np.subtract`	`-`
x^y	`np.power`	`**`
$x\%y$	`np.mod`	`%`

Sequential Function: $f(\{x_i\}_{i=0}^{n-1})$	NumPy Function
Mean of $\{x_i\}_{i=0}^{n-1}$	`np.mean`
Median of $\{x_i\}_{i=0}^{n-1}$	`np.median`
Variance of $\{x_i\}_{i=0}^{n-1}$	`np.var`
Standard Deviation of $\{x_i\}_{i=0}^{n-1}$	`np.std`
Maximum Value of $\{x_i\}_{i=0}^{n-1}$	`np.max`
Minimum Value of $\{x_i\}_{i=0}^{n-1}$	`np.min`
Index of the Maximum Value of $\{x_i\}_{i=0}^{n-1}$	`np.argmax`
Index of the Minimum Value of $\{x_i\}_{i=0}^{n-1}$	`np.argmin`
Sum of $\{x_i\}_{i=0}^{n-1}$	`np.sum`

Example:

```
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])

>>> y = 5*x + 1
>>> y
array([ 1,  6, 11, 16, 21])

# operations between two numpy arrays are STILL element-wise
>>> x + y
array([ 1,  7, 13, 19, 25])
>>> x * y
array([ 0,  6, 22, 48, 84])

>>> np.max(y)
21
>>> np.mean(y)
11.0
```

We can specify the axis of an operation:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])

>>> np.sum(a)
276
>>> # sum ALONG the 0 axis
>>> np.sum(a, axis=0)
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

Numpy also supports vectorized logical operations:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> a%2 == 0
array([[[ True, False,  True, False],
        [ True, False,  True, False],
```

```
        [ True, False,  True, False]],

       [[ True, False,  True, False],
        [ True, False,  True, False],
        [ True, False,  True, False]]])
>>> np.where(a%2 == 0)
(array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]),
 array([0, 0, 1, 1, 2, 2, 0, 0, 1, 1, 2, 2]),
 array([0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2]))
>>> a[np.where(a%2 == 0)]
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22])
```

5 Additional

Numpy has much more functionality that we won't have the chance to cover (to name a few):

- Random
- Linear algebra (linalg)
- Fast Fourier Transfor (fft)

Read more about Numpy:

- https://numpy.org/doc/stable/contents.html
- https://www.pythonlikeyoumeanit.com/module_3.html

6 Next Time

- Pandas
- Plotting in python with Matplotlib

A (Optional) Array Broadcasting

When two arrays have the same shape, we can easily understand what an element-wise operation does—it applies some operation between corresponding elements of the arrays!

For example (both arrays have `shape=(2,3)`),

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} + \begin{pmatrix}1 & 1 & 1\\ 2 & 2 & 2\end{pmatrix} \xrightarrow{element-wise} \begin{pmatrix}2 & 3 & 4\\ 6 & 7 & 8\end{pmatrix}$$

However, we can use BROADCASTING to perform operations on arrays of UNEQUAL shapes IF they are broadcastable. For example, adding a `(3,2)` array to a `(2,)` array:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} + \begin{pmatrix} 1 & 2 \end{pmatrix} \xrightarrow{broadcast} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \end{pmatrix} \xrightarrow{element-wise} \begin{pmatrix} 2 & 4 \\ 4 & 6 \\ 6 & 8 \end{pmatrix}$$

Broadcasting steps (using example above):

1. Determine the shape of both arrays: `(3,2)` and `(2,)`
2. If the shapes are not the same size, **prepend** the smaller shape with 1's until both arrays have the same number of dimensions. In this case, the first shape has dimension 2, so we must extend the second shape: `(2,)` $\rightarrow$ `(1,2)`
3. Align the array dimensions:

   (3, 2)
   (1, 2)

   Each of the respective dimensions must be (a) equal or (b) unequal but one dimension is 1. **Otherwise, the arrays can not be broadcast.** In this case, axis-0 (3 and 1) are not equal but contains one 1, and axis-1 (2 and 2) are equal.
4. For each of the dimensions, if the respective dimensions are equal, the operation is applied element wise. If the respective dimensions are not equal, but one dimension is 1, then the dimension of size 1 is broadcast to the larger dimension.

This is fully generalized. However, broadcasting can get nasty with high dimensions. As always, if you're getting lost in the complexity of your data structure, you should probably reconsider it's format.

B (Optional) Python Speedups Using C

Earlier, we mentioned that Numpy is fast because it is implemented in C. What does this mean? After all, Numpy is a *Python* package.

In short, the core functionality of Numpy (ndarrays and vectorized operations) is written in the programming language C with C syntax, C variables, etc. However, Numpy provides Python wrapper functions that allows the user to call Python functions (e.g. np.ndarray()) to perform operations in what is really C behind the scenes.

Why are we going to all this trouble? *Speed.* Python has considerably more overhead than C. This means that for the same basic operation, Python needs to perform more work and use more memory than C.

For example, Python is dynamically typed: a variable can hold any data type. While this is convenient for us, the programmer, it takes additional memory and operations behind-the-scenes to implement. On the other hand, C is strongly typed: you must define beforehand the variable type. Thus, the program will only allocate as much memory as the type requires

and there is no run-time ambiguity about the data type a variable contains. This is reflected in the fact that Numpy arrays can only store a single type of data.

While these optimizations may seem minor, they add up. For example, let's compare a function that computes the Fibonacci sequence written in both C and Python. First, we implement `fib()` in C,

```
# fib.c
int fib(int n) {
    if(n<2) {
       return n;
    }
    return fib(n-1) + fib(n-2)
}
```

Making use of the `ctypes` module, we can write our own wrapper function to use the `fib` function above in Python.

```
# fib.py
import time
import ctypes

# C implementation of fib with Python wrapper
# gcc -shared -o fib.so -fPIC fib.c
_libc = ctypes.CDLL('fib.so')
c_fib = _libc.fib
c_fib.restype = ctypes.c_int
c_fib.argtypes = [ctypes.c_int]
```

Let's now implement the exact same function, but in Python.

```
# python implementation of fib
def py_fib(n):
    if n<2:
        return n
    return py_fib(n-1) + py_fib(n-2)
```

Comparing the run times,

```
start = 0
def tik():
    global start
    start = time.time()
def tok():
    end = time.time()
    print(f'\t{end-start:3f} sec')
```

```
print('py_fib(36): ')
tik()
print(f'\t{py_fib(36)}')
tok()

print('c_fib(36): ')
tik()
print(f'\t{c_fib(36)}')
tok()
```

```
>>> python fib.py
py_fib(36):
    result: 14930352
    time: 5.697 sec
c_fib(36):
result: 14930352
    time: 0.110 sec
```

`c_fib` is considerably faster than `py_fib`—this is why packages such as Numpy and Scipy are so popular. We can take advantage of the speedups of using C while retaining the convenience of writing code in Python!