InfPALS

Data Science - numpy & matplotlib (30')

Useful links / warmup

numpy & scipy	python	matplotlib
docs.scipy.org/doc/	docs.python.org/3.5/	matplotlib.org/contents

Have a look through some documentation and get comfortable with how to navigate it, if you get stuck during the activity, refer back to these for guidance.

Matrices are cool (20 minutes)

Why though?

- Matrices let us do operations like, really quick. According to this post (https://github.com/rougier/from-python-to-numpy/blob/master/04-code-vectorization.rst#introduction) on vectorization, "not only is the second approach faster, but it also naturally adapts to the shape of Z1 and Z2." 60x speed up, anyone?
- Matrices store data in meaningful shapes - these could be points from a grid, **rows of data points**, columns of vectors, simultaneous equations, etc. This is useful for referring back to it systematically.

Basic Matrix stuff / Let's get physical (5 minutes)

Let's start a new Jupyter Notebook, your turn to come up with a funny name. Let's start with the imports at the very top of the notebook (again, different blocks should be in separate cells),

```python
import numpy as np
```

Nothing too special here, however note that we alias to a shorter name to save some typing later.

```python
v = np.array([[5, 1, 8, 5, 4]]) # double square brackets = matrix
```

Okay, so far so good, now let's go all the way. Type:

```python
v.T
```

Whoa… Whoa whoa whoa… Is that what you were expecting? Did it just do what you think it did? I found this worth a solid "That's cool" when I first tried it. Okay, how about:

```python
(v.shape, v.T.shape)
```

So, when making an array/matrix using the numpy command, we get a bunch of extra info at our disposal should we need it later. Let's have a few more cells and you can work out what they should be:

```python
w = np.array([[2, 3, 4, 5, 6]])
```

```python
(v + w, v * w)
```

See what it did there? Cool, now let's apply a function to a whole array:

```python
np.sqrt(v)
```

Or, say you'd like to apply your own transformation of sorts:

```
v * 23 - 8
```

Check with your neighbours to see what kind of output they have, and if it matches your own.

Matrix multiplication is hard (5 minutes)

The previous functions all worked linearly through each element of the matrix, now let's move on to more involved procedures. Feel free to continue adding cells to your current notebook, and maybe you could even comment your code to make sure you remember what you were thinking at the time. Set a 3x3 matrix:

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

Forget the pen and paper and doing the arithmetic in your head - you could make all kinds of silly mistakes - simply:

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

In [10]: m @ m                              In [10]: np.matmul(m,m)
Out[10]: array([[ 30,  36,  42],           Out[10]: array([[ 30,  36,  42],
                [ 66,  81,  96],                           [ 66,  81,  96],
                [102, 126, 150]])                          [102, 126, 150]])
```

Left or Right? ... the world's your oyster
Note: "@" operator only works on python >=3.5

Likewise for dot product:

```
np.dot(v,w)
```

Then make your own vectors and try out a few transformations on them - points for the most creative. See
docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html

for ideas.

Note: numpy has two similar, yet different types of "matrix" - `numpy.array` and `numpy.matrix` - most notably:

> "Numpy matrices are strictly 2-dimensional, while numpy arrays (ndarrays) are N-dimensional. Matrix objects are a subclass of ndarray, so they inherit all the attributes and methods of ndarrays."
>
> *thomas-arildsen @ stackoverflow*
>
> stackoverflow.com/a/4151251

Being picky (5 minutes)

So we can transpose, multiply, add… What if our matrix contains column vectors, how do we extract them? Your turn! Make a 3x3 matrix and grab the second column. Don your Google hat and search for an answer. Try out different sources. As before, there are multiple approaches - try searching for multiple ways and you'll have more flexibility in future.

Dire Straits - So Far Away … how far exactly? (5 minutes)

How about we take two vectors and find their straight line separation in space - euclidean distance? You know the basic algorithm (if not, Google!) - let's generalise. Take the vectors:

```python
a = np.array([2,5,7])
b = np.array([4,7,9])
```

Again, multiple ways to tackle it - either brute force it just to give it a try, or do some research. Remember the benefits of vectorization and keep this in mind when crafting your solution, as it the reason behind numpy - it avoids lengthy amounts of loops and makes your code quicker, cleaner, and more concise.

Data and Plots (10 minutes)

Matplotlib? (5 minutes)

Basically, it's your Excel graph maker's geeky older brother. Make a new notebook and let's try out some scatter plots:

```
%matplotlib notebook  # this is called a "magic" - makes it extra fancy
import matplotlib.pyplot as plt
import numpy as np
```

Define two sequences of points which will define our (x,y) coordinates for each point.

```
x = np.arange(10)
y = np.arange(0,20,2)
```

Et voilà!

```
plt.scatter(x,y)
```

The "magic"[1] we define on line 1 lets us interact with the plot - click the [Pan] button and use the mouse to move around!

Almost anything you would (intelligibly) want to plot can likely be plotted. This looks super cool:

matplotlib.org/gallery/mplot3d/custom_shaded_3d_surface

You could then tie-in with the previous InfPALS workshop and use some TeX:

matplotlib.org/gallery/text_labels_and_annotations/tex_demo

Let's move on to making a 3D plot!

---

[1] "Magics" are any line which begins with a **%**. These are part of IPython and can be found here: ipython.readthedocs.io/en/stable/interactive/magics.html

Heard you like 3D (5 minutes)

For this we'll need one more import, but then it's smooth sailing:

```python
from mpl_toolkits.mplot3d import Axes3D
```

A bit more elaborate on the specifics: we create a figure containing a subplot which we can add to later on. We can do this in one go:

```python
z = np.arange(0,30,0.4)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(z,z*z,np.sqrt(z))  # this is where we pass it the points
plt.show()
```

Okay, quite a few new things: first, `arange` is basically like the built-in `range` function. Next, create a figure, and a set of axes. Then, scatter points according to the function specified, and display. Play around with the inputs to `ax.scatter` and / or the `arange` passed to z to get a feel for how it behaves. If you find another cool example, paste it to a cell and play around with it.

What did I say about Excel's geeky older brother?

A final note

At this point, you might be getting the slight feeling you've done something similar before - you definitely will have done if you are taking Inf2B (and keeping up with the Learning labs). Some of the above examples were inspired by those labs, as a means to try and bridge the gap between MATLAB and Jupyter. As a challenge / recommendation if you'd like to learn more, fire up the Inf2B labs, but instead of using MATLAB, get a Jupyter notebook running. There is a bit of translation involved, but it's not *too* radical. When you get stuck, check Google to see if there's a straightforward solution, read the documentation, or play around with other ideas.

On a side note, part of the reason why python and Jupyter are so popular in data science is that they are lightweight and open-source (remember how this is taking up around 200 MB of space, whereas MATLAB "can be 3-4 GB for a typical installation"). Also worth mentioning is how Jupyter, being based on a web browser interface, does not necessarily have to run locally. If you grabbed the `short-mandelbrot.ipynb` notebook earlier on, you'll have come across Azure notebooks - a cloud-based notebook server that could even run on your phone.

Good work! Keep on pythoning!