Intro in class Python lab

January 24, 2018

This is our first, in class Python lab. As such, it's pretty straight forward. We'll just familiarize ourselves with the software a bit and do a few basics. If you've got Jupyter running Python 3.6 already on your laptop - great! If not, try logging into `https://cocalc.com/` and running from it from there. You should be able to use your UNCA credentials since it's just a gmail account.

1 The basics

Let's start with a few basic computations. Note that the `computer type looking stuff` after the `In[]` prompts is the actual code I'd like you to type in to practice.

```
In [ ]: (2+3)/42
```

```
In [ ]: 2**1234-1
```

```
In [ ]: 2.0**1234-1
```

```
In [ ]: x = 3/4
        y = x**3 - 2*x + 1
```

```
In [ ]: y
```

```
In [ ]: r = 10
        A = pi*r**2
```

While simple, there are already some important points to note.

- Syntax is important!
  - a^b is represented as `a**b`.
  - Defining a variable as in `a=2` supresses the output. You can always execute just `a` to see the output.
- Although Python is dynamically typed, it *is* typed. There are several different number types.
  - Integers have unlimited precision (which is why `2**1234` makes sense) and the can be used for things like indexing arrays. They are typically represented internally by a C type `int`.
  - Floats represent real numbers and are typically represented internally by a C type `double`. These are the types of numbers that we will mostly work with in this class.

  - 2+3 returns the integer 5 but 2/3 returns a float that approximates 2/3. This behavior was modified in the change from Python 2 (where 2/3 rounds down to the integer 0) to Python 3.

- Some seemingly standard quantities (like π) are not defined; but *a lot* of stuff is available via libraries.

Here's an example of an import from the Numpy library:

```
In [ ]: import numpy as np
        r = 10
        A = np.pi*r**2
        A
```

Note that NumPy is quite large and we will often choose to import it into its own namespace like this. Thus, all functionality from NumPy will be accessed via `np.function_we_want`. Here's how to find out how many items have been imported:

```
In [ ]: len(dir(np))
```

1.1 Formatting text and TeX in the notebook

This lab isn't *just* about Python - it's about the Jupyter notebook, as well. Note the `Cell` menu near the top of the window. This allows you to specify the `Cell Type`. If you choose `Markdown`, you can use markdown, HTML, and even TeX to format your cell. Once you hit the enter button, it should look really spiffy. That's how I got the list above, `computer type`, and groovy math like

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}.$$

1.1.1 More on markdown

Markdown is an awesome and relatively easy tool to create formatted prose from simple plaintext. It's like MarkUp that's more succinct and easier to read. It's becoming much more widespread as it's used in

- GitHub - the open source code repository.
- Many discussion forums, like
  - Stackexchange
  - Discourse, which runs NumericTalk
  - Vanilla Forums, which runs A-CalcTalk
- Programming environments, like
  - RStudio
  - The Sage Notebook
  - The Jupyter Notebook!!!

There are loads of tutorials out there so I'll just recommend GitHub's Mastering Markdown and demonstrate a couple of things in class. There are a number of different versions of markdown but the Jupyter notebook's version is GitHub flavored.

A few things to keep in mind:

Section headings Use hashmarks like:

```
# Level 1
## Level 2
### Level 3
```

Math Just type a LaTeX snippet. Note that you *cannot* type a full LaTeX document. Sectioning and overall formatting is left to markdown - not LaTeX.

Non-executable code for illustration For a short piece of inline code like `x=2` type '`x=2`'. For a code block, indent four spaces like:

```
    def f(x) = x**2-2
    f(2)
```

Code intended for execution in a Jupyter notebook should be entered into a `code` cell.

Prose Just type as normal!

1.2 Defining, graphing, and iterating a function

Let's play with the function $f(x) = \sin(2x)$. First, let's graph the function, together with the line $y = x$. To do so, we'll use the MatPlotLib library, as well as NumPy. In fact, here are three lines that should often go near the top of one of our Jupyter notebooks:

```
In [ ]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
```

The first line with the % sign is called a magic that makes MatPlotLib work well with the Jupyter notebook.

Now, we'll define the function and check to see if it looks like it works:

```
In [ ]: def f(x): return np.sin(2*x)
        f(np.pi)
```

Well, I wish that was zero but maybe we can live with that by now.

Now, plotting has three essential steps:

- Generate a list of `x` values over which to plot (often using the `np.linspace` command),
- generate the corresponding `y` values by applying `f` to the `x` values, and
- plot `x` versus `y` using `plt.plot`.

Those three steps correspond to the following three lines of code:

```
In [ ]: x = np.linspace(-1.1,1.3,100)
        y = f(x)
        plt.plot(x,y)
        plt.plot(x,x);
```

I know what you're thinking - that's four lines! True, but the first three lines correspond to the outline. The fourth line plots the line `y=x`. (Can you see why?)

Now it looks like f has three fixed points. Two near ± 1 are equal but opposite. Let's iterate f starting near $x_1 = 1.0$ and see if we can generate that fixed point to high precision.

A common scheme in Python is to use a `for` loop to iterate over all values in an iterable. For example, `range(n)` generates an iterator with `n` terms. So here's a way to iterate `f` 20 times and view the results:

```
In [ ]: x = 1.0
        for i in range(20):
            x = f(x)
            print(x)
```

It looks like we're converging towards something, but it's not clear how long we'll need to go until we get a reasonable amount of precision. Let's suppose that we know we'd like 15 digits of precision and let's write a loop that will iterate until we reach that precision. I'll also implement a counter to ensure that the iteration does stop eventually.

```
In [ ]: x1 = 1.0
        x2 = f(x1)
        cnt = 0
        while np.abs(x1-x2)>10**-15 and cnt<1000:
            x1 = x2
            x2 = f(x2)
            cnt = cnt+1
        (x2,cnt)
```

Question: What happens above if you request 17 digits of precision?? Why?
Exercise: Try the above with a couple of other functions. For example:

- $f(x) = \sin(2x^2)$ (remember that x^2 is entered as `x**2` in Python)
- Your personal function.

There might be surprises!