Intro in class Python lab

January 12, 2016

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 haven't already, the first thing you should do is download the Anaconda Python distribution: https://www.continuum.io/downloads.

Be sure to grab Python 3.5! Once it's installed, fire up the Jupyter notebook by typing `jupyter notebook` into your terminal. On Windows, you might need to activate the Anaconda Prompt to do this.

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` suppresses 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.
  - Try `1 == 1.0` and `1 is 1.0` at the prompt.
  - 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))
```

2 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

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

It's easiest to just demo this in class, so let's take some time to do so!

3 The Collatz conjecture: Or functions, booleans, and loops

Define a function on the natural numbers by

$$f(n) = \begin{cases} 3n+1 & \text{if } n \text{ is even} \\ n/2 & \text{if } n \text{ is odd.} \end{cases}$$

Now suppose that we iterate f starting from some initial seed. The Collatz conjecture states that the resulting sequence will eventually land at 1. From there, it's easy to see that $1 \to 2 \to 4 \to 1$, so that the process essentially terminates. As this is really an integer based problem, it might not be an ideal example in numerical analysis. Nonetheless, it's cool problem that leads naturally to the important concepts of functions, booleans and loops. Furthermore, the basic techniques are exactly the kinds of things we'll be doing with floats.

3.1 Functions

We use the `def` keyword to define functions. For example, we might represent the mathematical function $f(x) = x^2$ by

```
In [ ]: def f(x): return x**2
        f(-2)
```

It's important to understand, though, that a function is Python is not necessarily a mathematical function; it's any organized, reusable block of code that's wrapped up in this fashion.

3.2 Booleans

The term *boolean* (named after logician George Boole) refers specifically to the Python keywords `True` and `False` and, more generally to truth testing in computer science motivated by Boolean Algebra. `True` and `False` are values in Python and Boolean valued operators typically return one or the other. For example:

```
In [ ]: a = 10
        a == 10
        # Should return True
```

```
In [ ]: a < 0
        # Should return False
```

We often use these with `if` statements to direct flow. We can combine these ideas with the mod operator (denoted `%`) like so:

```
In [ ]: if(a%2 == 0):
            print('even')
        else:
            print('odd')
```

3.3 The Collatz function

OK, let's use this to define the actual Collatz function!

```
In [ ]: def f(n):
            if n%2 == 0:
                return int(n/2)
            else:
                return 3*n+1
        f(7)
```

`w00t`!! How exciting is that? I guess not very actually. Maybe we should do something more.

3.4 A loop

To demonstrate the Collatz conjecture, let's start at $x_0 = 7$ and iterate *until* we reach 1. To do so, we'll use a `while` statement. Rather, than just printing the output, we'll define a list called `orbit` and `append` the results to `orbit` as we go along.

```
In [ ]: x0 = 7
        orbit = [x0]
        while x0 != 1:
            x0 = f(x0)
            orbit.append(x0)
        orbit
```

Well, that's a bit cool, at least!

4 Loops, iterables, and vectorization

Often, we'll loop through a sequence, list, or other iterable type. For example, here's the cosine of the first 1000 integers:

```
In [ ]: loop_values = []
        for x in range(1000):
            loop_values.append(np.cos(x))
```

If we want to investigate the contents of a lengthy output like this, we might grab just a portion via a notation like so:

```
In [ ]: loop_values[:10]
        # Should return just the first 10 values.
```

Perhaps, that's not super illuminating. A plot might help? We'll investiagate that soon. First, let's look at another approach to genenerate this same list, we'll just apply NumPy's cosine function to the whole list.

```
In [ ]: vector_values = np.cos(range(1000))
```

We can check to see if we got the same result:

```
In [ ]: loop_values == vector_values
        # Should be a whole array of Trues
        # we can allways wrap that result in ‘all‘
```

This is a common theme in NumPy called *vectorization*. Note that this approach is often much faster. Let's time the two approaches using the `%%timeit` magic function. And don't ask me what a magic function is - that's why it's called magic!

```
In [ ]: %%timeit
        vector_values = np.cos(range(1000))
```

```
In [ ]: %%timeit
        loop_values = []
        for x in range(1000):
            loop_values.append(np.cos(x))
```

By the way, a microsecond μs or one millionth of a second is *much* smaller than a millisecond `ms` - a thousand times smaller, in fact!

5 Plotting with matplotlib

Of course, it's nice to plot results. Let's plot the points we just computed.

```
In [ ]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.plot(np.cos(range(1000)),'.')
```

Well, that's pretty darn cool!

Often,we'll like to plot a function over a particular interval. NumPy provides some nice tools for generating ranges of numbers. The two most commonly used ones are:

- `np.linspace(a,b,n)` - generates `n` numbers even spaced out from `a` to `b`
- `np.arange(a,b,dx)` - generates $\{a + i\,dx, 0 \leq i < (b - a)/dx\}$

Also, multiple calls to `plt.plot` draws to the same object. Thus, we can plot $f(x) = x\cos(x^2)$ together with the line $y = x$ as follows:

```
In [ ]: def f(x): return x*np.cos(x**2)
        xs = np.linspace(-2,2,100)
        plt.plot(xs,f(xs))
        plt.plot(xs,xs)
```