To the next level functions

February 29, 2016

Here's an example of some 'to-the-next-level' functions in Python and Jupyter.

```
In [1]: #This is a code cell. Use a hashtag for comments!
        x = 3
        2 + 2
        def multiplication(x,y):
            z = 3
            return x*y
```

double-click me to see text cells
this is a text

1 this is a header

If you use a % before a command, it will be special to Jupyter. They are called "magics".
You can learn about them more below:
http://jupyter.cs.brynmawr.edu/hub/dblank/public/Jupyter%20Magics.ipynb
http://blog.dominodatalab.com/lesser-known-ways-of-using-notebooks/
'%whos' gives you all the variables in the namespace.

```
In [2]: %whos

Variable          Type        Data/Info
---------------------------------------
multiplication    function    <function multiplication at 0x0000003F2CDF1E18>
x                 int         3
```

You can turn anything into a string.

```
In [3]: str([1,2,3])

Out[3]: '[1, 2, 3]'
```

Here's how to import libraries! (If this doesn't work, try typing `conda install numpy` at the Command Prompt or Terminal.)

```
In [4]: import numpy
```

They come with pre-written functions, depending on the library.

```
In [5]: numpy.cos(0)

Out[5]: 1.0
```

Guido Van Rossum's designing philosophies of Python are hidden in a secret poem here:

```
In [6]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

These are strings! This is the "type" that Python understands for text.

```
In [7]: my_string = "hello world"
```

Certain functions only work on strings — because they only make sense for strings!

```
In [8]: my_string.islower()
```

```
Out[8]: True
```

```
In [9]: type(my_string)
```

```
Out[9]: str
```

Lists are a different "type" of thing.

```
In [10]: my_list = [1,3,"a",42,"hello"]
```

```
In [11]: type(my_list)
```

```
Out[11]: list
```

"Loops" can go over each element in the list, and do things systematically. Indentation matters!

```
In [12]: for element in my_list:
             #do stuff here
             print(element)
```

```
1
3
a
42
hello
```

Let's check our namespace with %whos

```
In [13]: %whos
```

```
Variable         Type        Data/Info
--------------------------------------
element          str         hello
multiplication   function    <function multiplication at 0x0000003F2CDF1E18>
my_list          list        n=5
my_string        str         hello world
numpy            module      <module 'numpy' from 'C:\<...>ges\\numpy\\__init__.py'>
this             module      <module 'this' from 'C:\\<...>envs\\py3\\lib\\this.py'>
x                int         3
```

This magic is useful to make plotting happen in-line.

```
In [14]: %matplotlib inline
```

If you look at http://matplotlib.org/gallery.html, you'll find a collection of different plotting examples. If you click on an image you like, then right-click on the link to "Source code", and copy the link address, then you can put the code in your Jupyter notebook quickly:

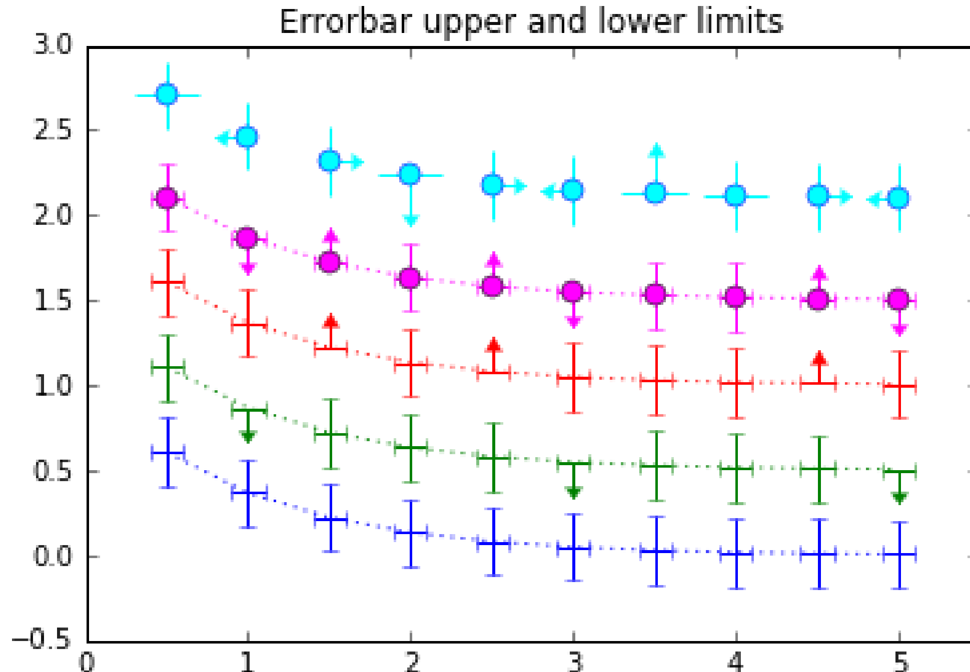
```
In [15]: #use %load then paste the link
```

```
In [ ]: % load http://matplotlib.org/mpl_examples/statistics/errorbar_limits.py
```

Running the above block will turn into something like the below block.

```
In [17]: # %load http://matplotlib.org/mpl_examples/statistics/errorbar_limits.py
         """
         Demo of the errorbar function, including upper and lower limits
         """
         import numpy as np
         import matplotlib.pyplot as plt

         # example data
         x = np.arange(0.5, 5.5, 0.5)
         y = np.exp(-x)
         xerr = 0.1
         yerr = 0.2
         ls = 'dotted'

         fig = plt.figure()
         ax = fig.add_subplot(1, 1, 1)

         # standard error bars
         plt.errorbar(x, y, xerr=xerr, yerr=yerr, ls=ls, color='blue')

         # including upper limits
         uplims = np.zeros(x.shape)
         uplims[[1, 5, 9]] = True
         plt.errorbar(x, y + 0.5, xerr=xerr, yerr=yerr, uplims=uplims, ls=ls,
                      color='green')

         # including lower limits
```

```
lolims = np.zeros(x.shape)
lolims[[2, 4, 8]] = True
plt.errorbar(x, y + 1.0, xerr=xerr, yerr=yerr, lolims=lolims, ls=ls,
             color='red')

# including upper and lower limits
plt.errorbar(x, y + 1.5, marker='o', ms=8, xerr=xerr, yerr=yerr,
             lolims=lolims, uplims=uplims, ls=ls, color='magenta')

# including xlower and xupper limits
xerr = 0.2
yerr = np.zeros(x.shape) + 0.2
yerr[[3, 6]] = 0.3
xlolims = lolims
xuplims = uplims
lolims = np.zeros(x.shape)
uplims = np.zeros(x.shape)
lolims[[6]] = True
uplims[[3]] = True
plt.errorbar(x, y + 2.1, marker='o', ms=8, xerr=xerr, yerr=yerr,
             xlolims=xlolims, xuplims=xuplims, uplims=uplims, lolims=lolims,
             ls='none', mec='blue', capsize=0, color='cyan')

ax.set_xlim((0, 5.5))
ax.set_title('Errorbar upper and lower limits')
plt.show()
```



Here's another example, that you can run yourself.

```
In [ ]: %load http://matplotlib.org/mpl_examples/lines_bars_and_markers/barh_demo.py
```