Jupyter Notebook Tips & Tricks

1. Native jupyter features (~20 minutes).
    - Keyboard Shortcuts
    - Magics!
2. Handy libraries (~10 minutes).

Navigation Keyboard Shortcuts

ESC takes you out of a cell (and into command mode)

ENTER returns you to editing a cell

CTRL + ENTER runs your current cell

SHIFT + ENTER runs your current cell and goes to the next one (inserting a new cell if you're at the end)

ESC + F finds in your code but not output

ESC + O toggles output

CTRL + SHIFT + P displays the command palatte (all commands)

SHIFT-TAB will show the docstring for whatever function is on the same line as your cursor

Command mode

A inserts a cell above

B inserts a cell below

M switches a cell to markdown

Y switches it back to code

CTRL + X cuts deletes a cell, **DD** (double tap on d) deletes it

Magics!

These are built-in juptyer functions that give you some nice IDE-like code-help functionality. A line magic goes in front of a line of code, and cell magic goes at the top of a cell. Line magics look like this: **%<command>**, cell magics look like this: **%%<command>**

?<function name>() will bring up the docstring for the function (this one isn't technically a magic...but it sort of acts like one)

SHIFT-TAB also works!

In [1]:
```
import pandas as pd
```

In [2]:
```
?pd.DataFrame()
```

I'm not going to go through all of them because there are a lot.

%lsmagic is how you list them:

In [3]:
```
%lsmagic
```

Out[3]:
```
Available line magics:
%alias  %alias_magic  %autocall  %automagic  %autosave  %bookmark  %cd  %clear  %cls  %colors  %config  %conne
ct_info  %copy  %ddir  %debug  %dhist  %dirs  %doctest_mode  %echo  %ed  %edit  %env  %gui  %hist  %history  %
killbgscripts  %ldir  %less  %load  %load_ext  %loadpy  %logoff  %logon  %logstart  %logstate  %logstop  %ls
%lsmagic  %macro  %magic  %matplotlib  %mkdir  %more  %notebook  %page  %pastebin  %pdb  %pdef  %pdoc  %pfile
%pinfo  %pinfo2  %popd  %pprint  %precision  %profile  %prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab
  %qtconsole  %quickref  %recall  %rehashx  %reload_ext  %ren  %rep  %rerun  %reset  %reset_selective  %rmdir
%run  %save  %sc  %set_env  %store  %sx  %system  %tb  %time  %timeit  %unalias  %unload_ext  %who  %who_ls  %
whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%cmd  %%debug  %%file  %%html  %%javascript  %%js  %%latex  %%perl  %%
prun  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time  %%timeit
%%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

The cell magics let you write code in a few different languages. One that's missing there is cython, which can be handy, but first you have to load the extension:

In [4]:
```
%load_ext Cython
```

Now you can write in cython!

In [5]:
```
%%cython --annotate

cdef int a = 0
for i in range(20):
    a += i
print(a)
```

```
190
```

Out[5]:

Generated by Cython 0.26

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
 1:
+2: cdef int a = 0
+3: for i in range(20):
+4:     a += i
+5: print(a)
```

Get excited, because you'll get a bit9 error, do the right thing and just click *allow* without reading! This is because cython compiles and then runs.

Why should you care?

Cython can really speed up your code, if you're running into repetitive, menial tasks and have exhausted options for revising the python code then cython may be a good option. Used well you can get an order-of-magnitude increase in speed.

There are also some nice magics for seeing what variables are cluttering your namespace, **%%env** shows environment variables and **%%who** shows global scope variables.

My favorite magics are the timing magics, **%%timeit**, **%%time** and their line-magic variants. **%%timeit** give you a best of 3 runs and **%%time** just measures how long a cell took to run.

In [6]:
```
import numpy as np
```

In [7]:
```
%time big_ole_range = np.random.rand(3000,20000)
```

```
Wall time: 827 ms
```

In [8]:
```
%%time
print(big_ole_range.mean())
print(big_ole_range.std())
```

```
0.499985242074
0.288685223146
Wall time: 685 ms
```

In [9]:
```
%%timeit
squarer = lambda t: t ** 2
squares = np.array([squarer(xi) for xi in big_ole_range])
```

```
1 loop, best of 3: 632 ms per loop
```

In [10]:
```
squarer = lambda t: t ** 2
v_squarer = np.vectorize(squarer)
%time squares = v_squarer(big_ole_range)
```

```
1 loop, best of 3: 25.8 s per loop
```

In [11]:
```
%%cython --annotate
cimport numpy as np
def cy_squarer(np.ndarray x):
    return x ** 2
```

Out[11]:

Generated by Cython 0.26

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
+1: cimport numpy as np
+2: def cy_squarer(np.ndarray x):
+3:     return x ** 2
```

In [12]:
```
%timeit squares = [cy_squarer(xi) for xi in big_ole_range]
```

```
1 loop, best of 3: 255 ms per loop
```

Debugging code... Let's be honest, I haven't done much with this, but I probably should have...

The internet (https://davidhamann.de/2017/04/22/debugging-jupyter-notebooks/) seems to like the **set_trace** function:

- **n** in the prompt goes to the next line
- **c** continues the program
- You can type variable names as usual in a debugger

In [13]:
```
from IPython.core.debugger import set_trace

def add_to_life_universe_everything(x):
    answer = 42
    set_trace()
    answer += x

    return answer

add_to_life_universe_everything(12)
```

```
> <ipython-input-13-5d554bb8cb23>(6)add_to_life_universe_everything()
      4     answer = 42
      5     set_trace()
----> 6     answer += x
      7
      8     return answer

ipdb> n
> <ipython-input-13-5d554bb8cb23>(8)add_to_life_universe_everything()
      6     answer += x
      7
----> 8     return answer
      9
     10 add_to_life_universe_everything(12)

ipdb> n
--Return--
54
> <ipython-input-13-5d554bb8cb23>(8)add_to_life_universe_everything()
      6     answer += x
      7
----> 8     return answer
      9
     10 add_to_life_universe_everything(12)

ipdb> n
--Call--
> c:\development\anaconda\lib\site-packages\ipython\core\displayhook.py(236)__call__()
    234         sys.stdout.flush()
    235
--> 236     def __call__(self, result=None):
    237         """Printing with history cache management.
    238

ipdb> c
```

Out[13]: 54

Running other scripts!

This can be extra handy if you're trying to run a multiprocess program--they don't tend to run in notebooks very nicely. It's simple, just use the **%run** command! Specifically, **%run -i <your_program.py>**, which loads the scirpt *into your notebooks namespace.*

This is what **script.py** looks like:

```
my_super_secret_variable = 42
print('My super secret variable loaded!')
```

In [35]:
```
%run -i script.py
```

```
My super secret variable loaded!
```

In [36]:
```
my_super_secret_variable
```

Out[36]: 42

According to stack overflow this is hack-ey, to which I say bah humbug.

Fun Libraries!

My absolute favorite is tqdm!

In [19]:
```
from tqdm import tqdm_notebook

for i in tqdm_notebook(range(300)):
    blah = np.random.rand(3000,3000)
    blah = 0
```

That's the cute notebook implementation--it can slow things down, but the normal one sometimes looks bad (but it's perfectly fast):

In [20]:
```
from tqdm import tqdm

for i in tqdm(range(300)):
    blah = np.random.rand(3000,3000)
    blah = 0
```

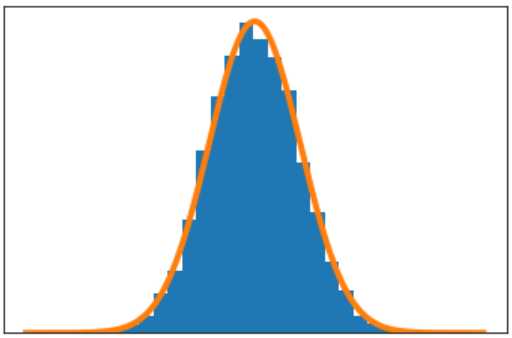
```
100%|██████████████████████████████████████████████████| 300/300 [00:38<00:00,  7.70it/s]
```

This is most useful if you're running a bunch of iterations and you would like a ballpark estimate of how long it will take.

Plotting

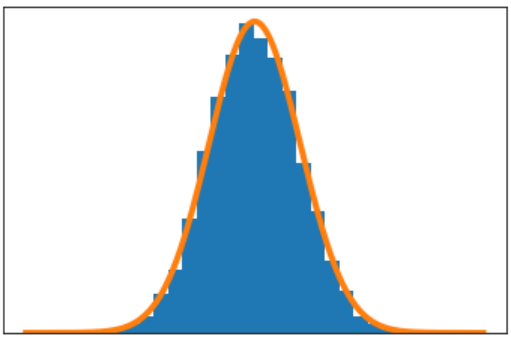
There are a million options for plotting, so let your personal preference guide you. I use **bokeh** for interactive charts and because it really helps with server setup. In terms of notebooks I would recommend starting with **matplotlib**. It's obtuse and complicated, but it's been around forever, so support (in the form of google: "How to do ____ with matplotlib") is really good.

In [21]:
```
import matplotlib.pyplot as plt
import numpy as np


random_state = np.random.RandomState(19680801)
X = random_state.randn(10000)

fig, ax = plt.subplots()
ax.hist(X, bins=25, normed=True)
x = np.linspace(-5, 5, 1000)
ax.plot(x, 1 / np.sqrt(2*np.pi) * np.exp(-(x**2)/2), linewidth=4)
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```



You can use the **%matplotlib nbagg** magic to get some additional functionality...

In [25]:
```
%matplotlib nbagg
random_state = np.random.RandomState(19680801)
X = random_state.randn(10000)

fig, ax = plt.subplots()
ax.hist(X, bins=25, normed=True)
x = np.linspace(-5, 5, 1000)
ax.plot(x, 1 / np.sqrt(2*np.pi) * np.exp(-(x**2)/2), linewidth=4)
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```



However, if you really need control over interaction I find that matplotlib's complexity can begin to get daunting, so other libraries might be best.

Ipython Libraries

There's a whole universe of ipython libraries that can help spruce up the ouput, I'll just go over the simplest one, **HTML**, which requires the **display** library. There is way more you can do in terms of adding interaction and widgets (http://ipywidgets.readthedocs.io/en/stable/examples/Widget%20List.html) and all that.

In [31]:
```
from IPython.core.display import display, HTML

url = 'https://www.mcmaster.com/mvA/gfx/home/Fastening-and-Joining-Fasteners-sprite-60.png?ver=1491990952'
display(HTML('<img src="{}">'.format(url)))
```

That's all folks!