

 This repository

[Pull requests](#) [Issues](#) [Gist](#)

 [quantopian](#) / [research_public](#)

 Watch ▾

95

 Star

181

 Fork

106

 Code

 Issues 0

 Pull requests 0

 Projects 0

 Wiki

 Pulse


 Graphs

Branch: master ▾

[research_public](#) / [lectures](#) / [Introduction to Research.ipynb](#)

Find file

Copy path

 **CaptainKanuk** BUG: Removed incorrect content from intro notebook. a5711fe on Jan 19


1 contributor


551 lines (551 sloc) 13.1 KB


Raw

Blame

History







Introduction to the Research Environment

The research environment is powered by IPython notebooks, which allow one to perform a great deal of data analysis and statistical validation. We'll demonstrate a few simple techniques here.

Code Cells vs. Text Cells

As you can see, each cell can be either code or text. To select between them, choose from the 'Cell Type' dropdown menu on the top left.

Executing a Command

A code cell will be evaluated when you press play, or when you press the shortcut, shift-enter. Evaluating a cell evaluates each line of code in sequence, and prints the results of the last line below the cell.

```
In [ ]: 2 + 2
```

Sometimes there is no result to be printed, as is the case with assignment.

```
In [ ]: X = 2
```

Remember that only the result from the last line is printed.

```
In [ ]: 2 + 2  
3 + 3
```

However, you can print whichever lines you want using the `print` statement.

```
In [ ]: print 2 + 2  
3 + 3
```

Knowing When a Cell is Running

While a cell is running, a `[*]` will display on the left. When a cell has yet to be executed, `[]` will display. When it has been run, a number will display indicating the order in which it was run during the execution of the notebook `[5]`. Try on this cell and note it happening.

```
In [ ]: #Take some time to run something  
c = 0  
for i in range(10000000):  
    c = c + i  
c
```

Importing Libraries

The vast majority of the time, you'll want to use functions from pre-built libraries. You can't import every library on Quantopian due to security issues, but you can import most of the common scientific ones. Here I import numpy and pandas, the two most common and useful libraries in quant finance. I recommend copying this import statement to every new notebook.

Notice that you can rename libraries to whatever you want after importing. The `as` statement allows this. Here we use `np` and `pd` as aliases for numpy and pandas. This is a very common aliasing and will be found in most code snippets around the web. The point behind this is to allow you to type fewer characters when you are frequently accessing these libraries.

```
In [ ]: import numpy as np  
import pandas as pd  
  
# This is a plotting library for pretty pictures.  
import matplotlib.pyplot as plt
```

Tab Autocomplete

Pressing tab will give you a list of IPython's best guesses for what you might want to type next. This is incredibly valuable and will save you a lot of time. If there is only one possible option for what you could type next, IPython will fill that in for you. Try pressing tab very frequently, it will seldom

time. If there is only one possible option for what you could type next, IPython will fill that in for you. By pressing tab very frequently, it will seldom fill in anything you don't want, as if there is ambiguity a list will be shown. This is a great way to see what functions are available in a library.

Try placing your cursor after the `.` and pressing tab.

```
In [ ]: np.random.
```

Getting Documentation Help

Placing a question mark after a function and executing that line of code will give you the documentation IPython has for that function. It's often best to do this in a new cell, as you avoid re-executing other code and running into bugs.

```
In [ ]: np.random.normal?
```

Sampling

We'll sample some random data using a function from numpy.

```
In [ ]: # Sample 100 points with a mean of 0 and an std of 1. This is a standard normal distribution.
X = np.random.normal(0, 1, 100)
```

Plotting

We can use the plotting library we imported as follows.

```
In [ ]: plt.plot(X)
```

Squelching Line Output

You might have noticed the annoying line of the form `[<matplotlib.lines.Line2D at 0x7f72fdbc1710>]` before the plots. This is because the `.plot` function actually produces output. Sometimes we wish not to display output, we can accomplish this with the semi-colon as follows.

```
In [ ]: plt.plot(X);
```

Adding Axis Labels

No self-respecting quant leaves a graph without labeled axes. Here are some commands to help with that.

```
In [ ]: X = np.random.normal(0, 1, 100)
X2 = np.random.normal(0, 1, 100)

plt.plot(X);
plt.plot(X2);
plt.xlabel('Time') # The data we generated is unitless, but don't forget units in general.
plt.ylabel('Returns')
plt.legend(['X', 'X2']);
```

Generating Statistics

Let's use numpy to take some simple statistics.

```
In [ ]: np.mean(X)
```

```
In [ ]: np.std(X)
```

Getting Real Pricing Data

Randomly sampled data can be great for testing ideas, but let's get some real data. We can use `get_pricing` to do that. You can use the `?` syntax as discussed above to get more information on `get_pricing`'s arguments.

```
In [ ]: data = get_pricing('MSFT', start_date='2012-1-1', end_date='2015-6-1')
```

Our data is now a dataframe. You can see the datetime index and the columns with different pricing data.

```
In [ ]: data
```

This is a pandas dataframe, so we can index in to just get price like this. For more info on pandas, please [click here](http://pandas.pydata.org/pandas-docs/stable/10min.html) (<http://pandas.pydata.org/pandas-docs/stable/10min.html>).

```
In [ ]: X = data['price']
```

Because there is now also date information in our data, we provide two series to `.plot`. `X.index` gives us the datetime index, and `X.values` gives us the pricing values. These are used as the X and Y coordinates to make a graph.

```
In [ ]: plt.plot(X.index, X.values)
plt.ylabel('Price')
plt.legend(['MSFT']);
```

We can get statistics again on real data.

```
In [ ]: np.mean(X)
```

```
In [ ]: np.std(X)
```

Getting Returns from Prices

We can use the `pct_change` function to get returns. Notice how we drop the first element after doing this, as it will be NaN (nothing -> something results in a NaN percent change).

```
In [ ]: R = X.pct_change()[1:]
```

We can plot the returns distribution as a histogram.

```
In [ ]: plt.hist(R, bins=20)
plt.xlabel('Return')
plt.ylabel('Frequency')
plt.legend(['MSFT Returns']);
```

Get statistics again.

```
In [ ]: np.mean(R)
```

```
In [ ]: np.std(R)
```

Now let's go backwards and generate data out of a normal distribution using the statistics we estimated from Microsoft's returns. We'll see that we have good reason to suspect Microsoft's returns may not be normal, as the resulting normal distribution looks far different.

```
In [ ]: plt.hist(np.random.normal(np.mean(R), np.std(R), 10000), bins=20)
plt.xlabel('Return')
```

