

### 3\_Financial Data Statistics

October 6, 2021

```
[2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
[3]: data = pd.read_csv('all_stocks_5yr.csv', parse_dates=True)
```

```
[4]: data.head()
```

```
[4]:
```

	date	open	high	low	close	volume	Name
0	2013-02-08	15.07	15.12	14.63	14.75	8407500	AAL
1	2013-02-11	14.89	15.01	14.26	14.46	8882000	AAL
2	2013-02-12	14.45	14.51	14.10	14.27	8126000	AAL
3	2013-02-13	14.30	14.94	14.25	14.66	10259500	AAL
4	2013-02-14	14.94	14.96	13.16	13.99	31879900	AAL

```
[5]: '''
Next, we're going to extract a new data frame by filtering all the rows where
→the name is SBUX.
'''

sbux=data[data['Name']=='SBUX'].copy()
sbux.head()
```

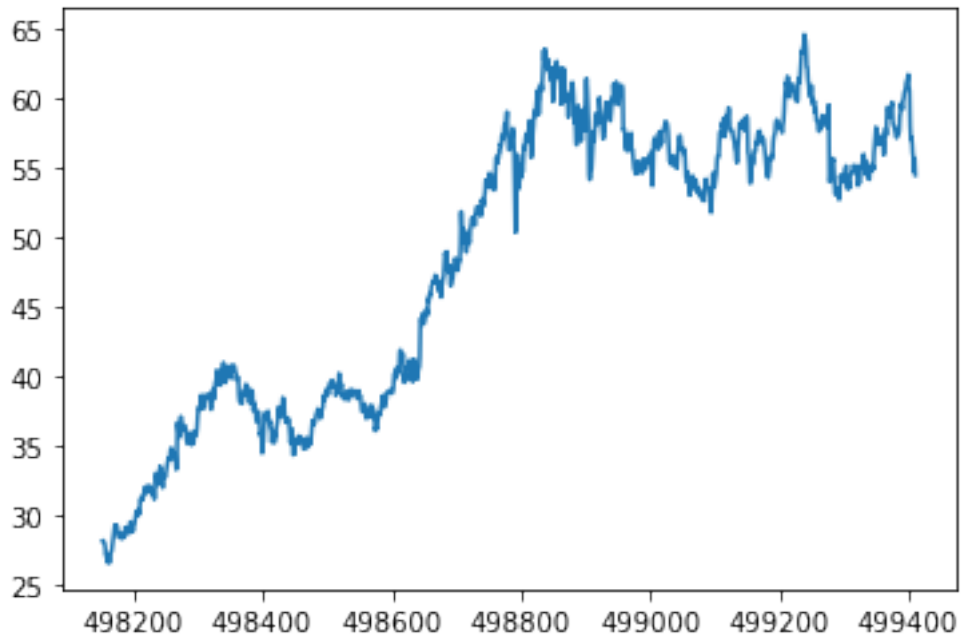
```
[5]:
```

	date	open	high	low	close	volume	Name
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX

```
[6]: '''
Next, we can call the plot function on the clothes column to look at the stock
→price as a time series.
'''

sbux['close'].plot()
```

```
[6]: <AxesSubplot:>
```



```
[7]: '''
Now, to get down to the real work, we know that in order to calculate the
    ↳return, we need the current
close price as well as the previous close price.

What we want to have is the previous close price in the same row as the close
    ↳price to do this, we
can call the shift function, we pass on the value one.

To say that we want to shift the close call by one will assign this new column
    ↳to be prev close.

If we use the head, come in again, we see that our new column of clothes has
    ↳been created.

Notice how all the items in the clothes column are just the items from the
    ↳clothes column shifted up
by one.
Since there is nothing to shift up, buy one for the very first value. This
    ↳value must necessarily be NaN.
'''

sbux['prev_close']=sbux['close'].shift(1)
sbux.head()
```

```
[7]:
```

	date	open	high	low	close	volume	Name	prev_close
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915

```
[8]: '''
In the next block of code, we calculate the return as discussed previously.

That's the close column divided by the previous column and then minus one will
→assign this value to
a column called Return.
'''

from IPython.display import display, Math, Latex
display(Math(r'R = {P_t \over P_{t-1}}-1'))
```

$$R = \frac{P_t}{P_{t-1}} - 1$$

```
[9]: %%\latex
\begin{align}
&\backslash\newline
R= \{{P_{t}}\over P_{t-1}} -1\}
&\backslash\end{align}
```

$$R = \frac{P_t}{P_{t-1}} - 1 \quad (1)$$

```
[10]: from IPython.display import Latex
Latex(r"""\begin{eqnarray} R= \{{P_{t}}\over P_{t-1}} -1\}
&\backslash\end{eqnarray}""")
```

```
[10]:
```

$$R = \frac{P_t}{P_{t-1}} - 1 \quad (2)$$

```
[11]: '''
Note the use of vectorized operations here.
There's no need to do something like a for loop through each row, calculating
→the return one by one.

If you're trained in programming, that's probably your first thought for how to
→calculate the return,
given a two dimensional array of items.
```

But luckily, pandas make this operation very easy by allowing us to calculate  
 ↪ all of the returns at once.

If we do a `sbux.head()` again to see what our new data frame looks like, we see  
 ↪ our return.

Collum note again how the first row is NaN This must be the case since we've  
 ↪ `prev_close` is NaN and therefore it's not possible to calculate  
 the return on this date.

Also, notice how small the returns are, as mentioned, financial engineers are  
 ↪ pretty accustomed to  
 working with very small numbers like this, and that's why we use units such as  
 ↪ basis points.

```
'''
sbux['return']=sbux['close']/sbux['prev_close'] -1
sbux.head()
```

```
[11]:
```

	date	open	high	low	close	volume	Name	prev_close \
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915

	return
498152	NaN
498153	-0.004080
498154	0.002138
498155	-0.007643
498156	-0.005015

```
[12]: '''
The way to do this is we call the pct_change function.
We pass in the argument one to mean that we want to calculate the percent
↪ change over one timestep.

Upon inspection, we see that both the return and return2 to columns are the
↪ same, verifying that our
calculation of the return is correct.
'''
sbux['return2']=sbux['close'].pct_change(1)
sbux.head()
```

```
[12]:
```

	date	open	high	low	close	volume	Name	prev_close	\
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN	
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185	
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070	
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130	
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915	

	return	return2
498152	NaN	NaN
498153	-0.004080	-0.004080
498154	0.002138	0.002138
498155	-0.007643	-0.007643
498156	-0.005015	-0.005015

```
[13]: '''
Although up until now, we've been plotting Time series, what we are often
    ↳ interested in when we look
at returns is the distribution of returns.

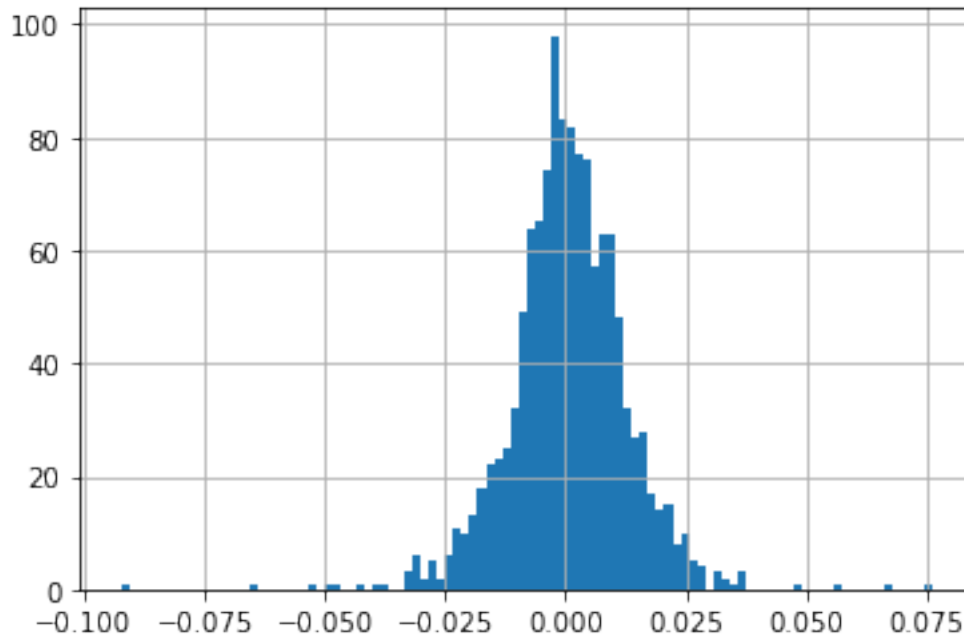
One quick visualization we can do to get a feel for the distribution of a set
    ↳ of numbers is the histogram in pandas.

All we need to do is call the highest hist on our data frame or series object.
We pass in the bins argument to specify how fine grained we want the histogram
    ↳ to be.

So as you can see, what we get is this typical bell shaped curve.
'''

sbux['return'].hist(bins=100)
```

```
[13]: <AxesSubplot:>
```



```
[14]: '''
Another thing we can do with our series of returns is calculate statistics such
↳as the sample mean and
the sample variance.

The STD function actually gives us the standard deviation, which is the square
↳root of the variance.

As you can see, the return is very small, very close to zero, and the standard
↳deviation is also
quite small, about zero point zero one.

'''

sbux['return'].mean(),sbux['return'].std()
```

```
[14]: (0.0006002332205830914, 0.012360934026133882)
```

```
[15]: '''
Now, since we learned about log returns as well, let's try doing the same
↳process, but on the log
returns, luckily numpy operations, broadcast over pandas data structures.

So all we need to do is take the return coloumn at once and called log on that.
```

*you recall that when  $X$  is very small, it's approximately equal to  $\log(X+1)$ , and  
 ↳ we can see that  
 kind of behavior here.*

*Notice how the log returns are actually very close to the non log returns.  
 They only differ in about the last two decimal places.*

'''

```
sbux['log_return']=np.log(sbux['return']+1)
sbux.head()
```

```
[15]:
```

	date	open	high	low	close	volume	Name	prev_close	\
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN	
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185	
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070	
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130	
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915	

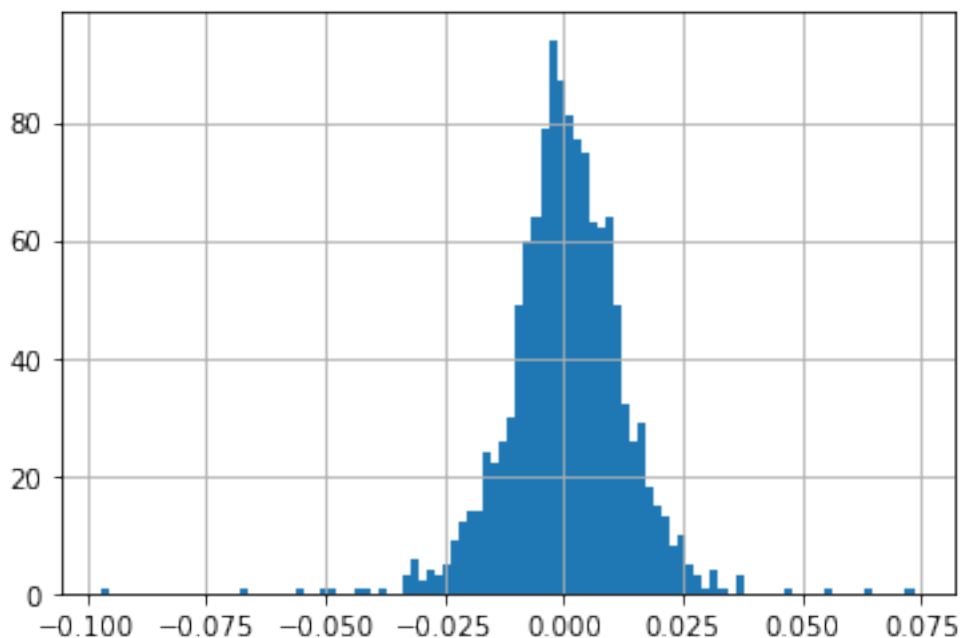
  

	return	return2	log_return
498152	NaN	NaN	NaN
498153	-0.004080	-0.004080	-0.004089
498154	0.002138	0.002138	0.002135
498155	-0.007643	-0.007643	-0.007672
498156	-0.005015	-0.005015	-0.005028

```
[16]: '''
Next, we're going to plot a histogram of our log returns using the highest_
↳ function, as you can see,
we get pretty much the exact same distribution that we got for the non log_
↳ returns.
'''

sbux['log_return'].hist(bins=100)
```

```
[16]: <AxesSubplot:>
```



```
[17]: '''
Finally, we can calculate the sample mean and the sample standard deviation of
↳the log return again

as before, we see that the mean is very close to zero
and the standard deviation is about zero point zero one.
'''
sbux['log_return'].mean(),sbux['log_return'].std()
```

```
[17]: (0.000523590274810868, 0.012381234216101253)
```

```
[18]: '''
Before we make any Q-Q plots, I want to start by simply drawing the normal PDF
↳over the histogram.

We know that theoretically the histogram will approach the true distribution as
↳the number of samples
collected approaches infinity.

So if the distribution we choose is a good fit, then it's histogram should
↳match up pretty closely
with the true distribution.
'''
from scipy.stats import norm
'''
```

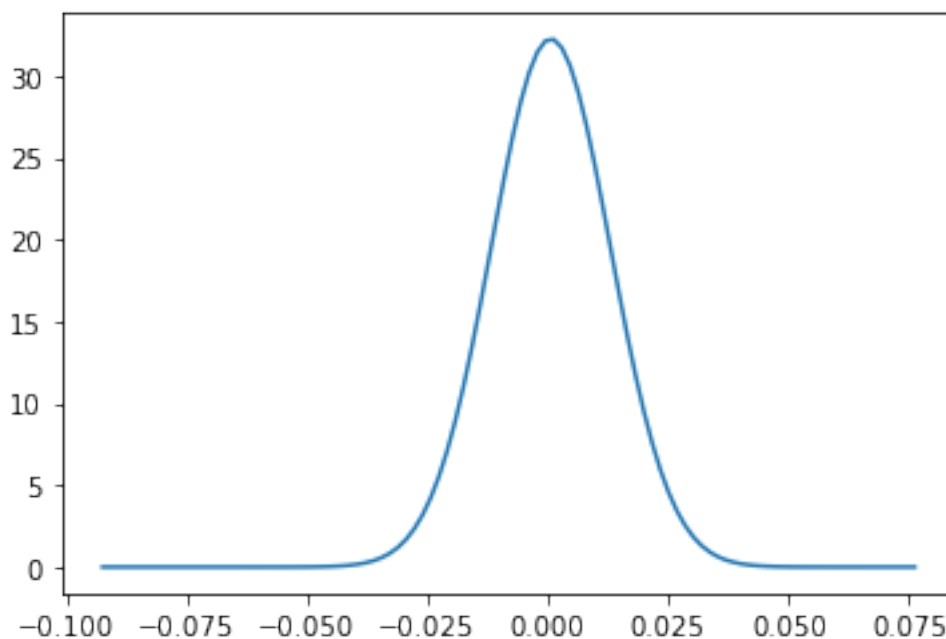


*We can create a list of X coordinates which will span from the minimum return, to the maximum return, with 100 points in between, as you recall. This can be accomplished with the linspace function.*

```
'''
x_list=np.linspace(sbox['return'].min(), sbox['return'].max(),100)
```

```
[21]: plt.plot(x_list,y_list)
```

```
[21]: [<matplotlib.lines.Line2D at 0x7fc84a083d90>]
```



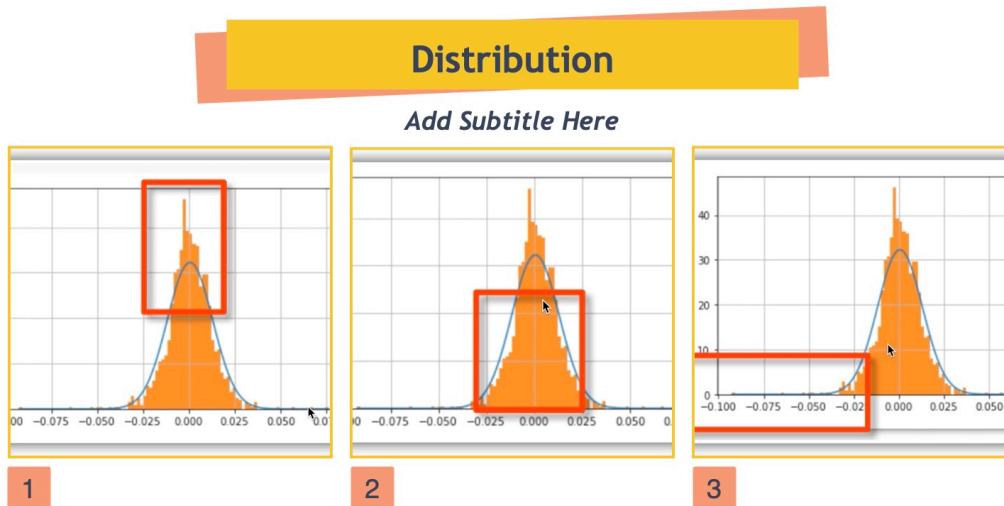
```
[19]: '''
Next, we're going to generate the normal PDF with mean and standard deviation,
equal to the sample mean
and sample standard deviation of our returns.

We can accomplish this by calling the function norm.
pdf(x_list,loc=sbox['return'].mean(),scale=sbox['return'].std())

the first argument is the X coordinates.
The second argument is the mean and the third argument is the scale, which for
the normal distribution
is the standard deviation.
'''
y_list=norm.pdf(x_list,loc=sbox['return'].mean(),scale=sbox['return'].std())
```

```
[27]: from IPython.display import Image
Image(filename='/Users/subhasish/Documents/Apple/Snagit/Distribution.jpg')
```

[27]:



Created by SUBHASISH BISWAS | 6 October 2021

Made with TechSmith Snagit™

```
[28]: '''
Next, we can draw a plot of our PDF along with the histogram, simply by calling
↳ the plot function
and the hist function separately.
```

Note that for the hist function we need to pass in the argument, density =  
↳ true, so that the  
histogram is normalized by default histogram.

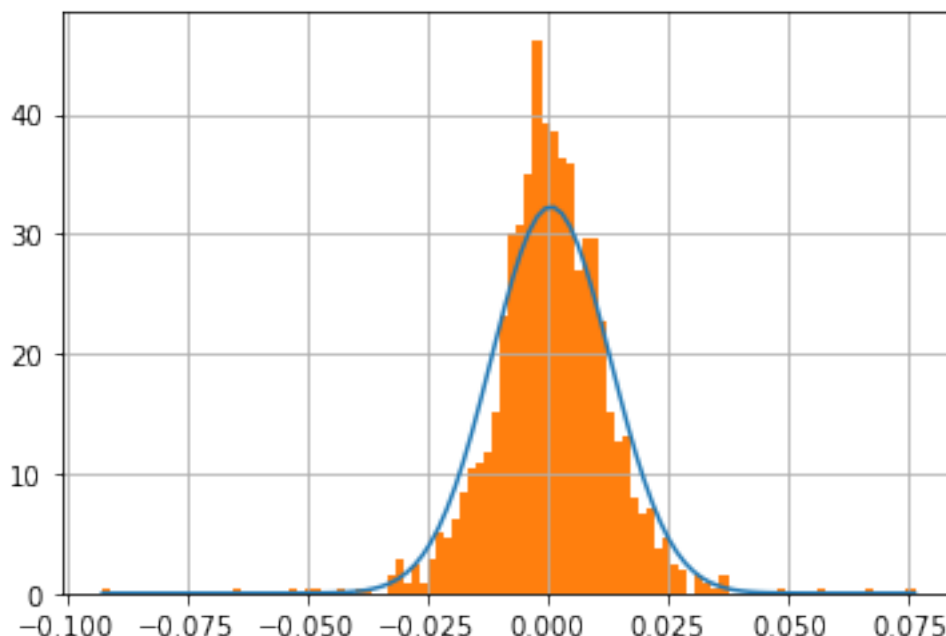
So what do we see, as you can see, this is probably not quite a good fit.  
It's reasonable, but there are some areas of the plot which don't look nice.

For example, in the center, the data has a much higher frequency than predicted,  
↳ by the normal distribution.  
There also appear to be pretty significant gaps in the shoulders of the  
↳ distribution.

And thirdly, the returns seem to take on pretty extreme values, which should be very unlikely according to the normal distribution.

```
'''  
plt.plot(x_list,y_list)  
sbux['return'].hist(bins=100,density=True)
```

[28]: <AxesSubplot:>



[29]: 

```
'''  
Let's now see how we can generate a Q-Q plot to verify what we've seen, one method of doing this is to simply use the probplot function from scipy.stats.
```

As you recall, another name for the Q-Q plot is the probability plot in the next block, we call the probplot function.

The first argument is the data That's the sbux returns, but we call drop in a first so that we only pass in actual numbers.

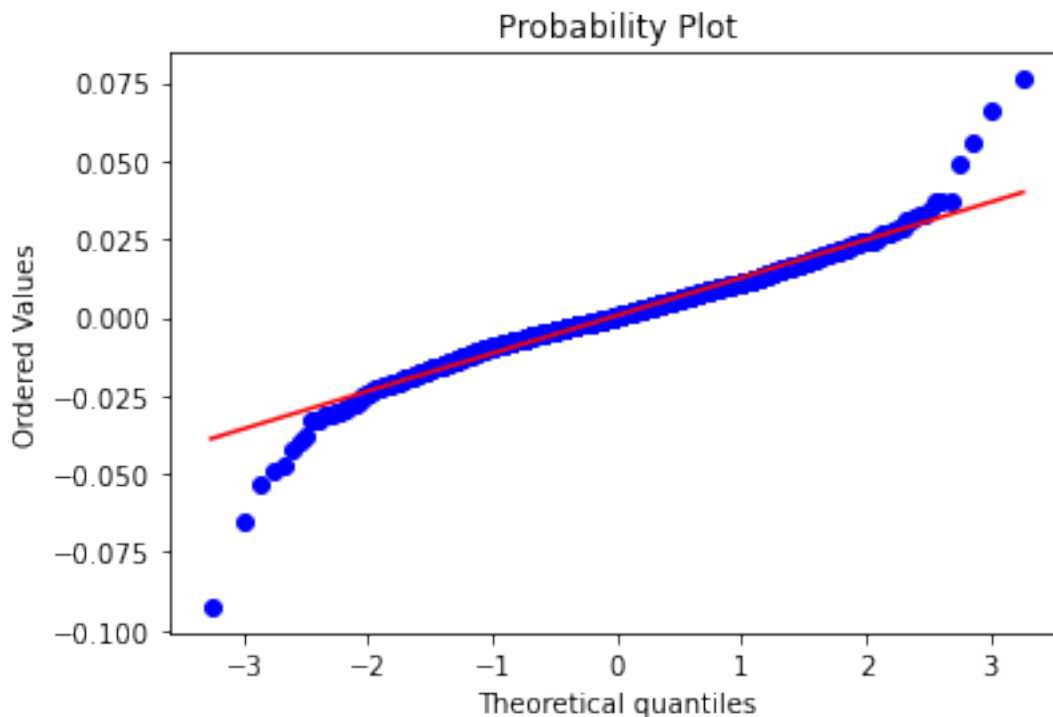
Next, we say dist='norm', To say that we want to compare our data with the normal distribution

*we say fit=True so that the function will find the best parameters for the  
→normal distribution that will match with our data.*

*Finally, we pass in a plot equals party, which corresponds to matplotlib, which  
→we imported earlier  
'''*

```
from scipy.stats import probplot
probplot(sbox['return'].dropna(),dist='norm',fit=True,plot=plt)
```

```
[29]: ((array([-3.26318411, -3.00291115, -2.85798028, ...,  2.85798028,
              3.00291115,  3.26318411])),
       array([-0.09243697, -0.06519128, -0.05321627, ...,  0.05617538,
              0.06623157,  0.0761332 ])),
       (0.012069021175478525, 0.0006002332205830934, 0.9745536080830226))
```

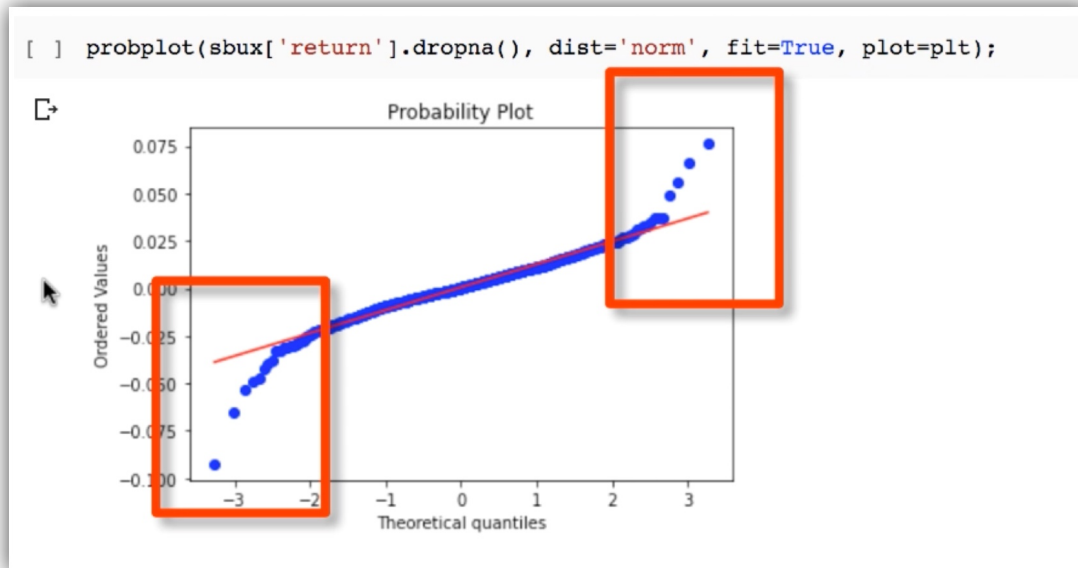


[30]: *'''*  
*So if we look at the probability plot, what do we see?*

*Well, we see pretty significant divergence at the ends of the probability plot.  
This suggests that our data has much heavier tails than expected if it came  
→from the normal distribution,  
which confirms what we were seeing earlier.*

```
'''
Image('/Users/subhasish/Documents/Apples/Snagit/Q-QPlot.jpg')
```

[30]:



[35]:

```
'''
The next thing I want to do in this script is to show you how to make the exact
↳ same Q-Q plot by using
stat's models rather than scipy

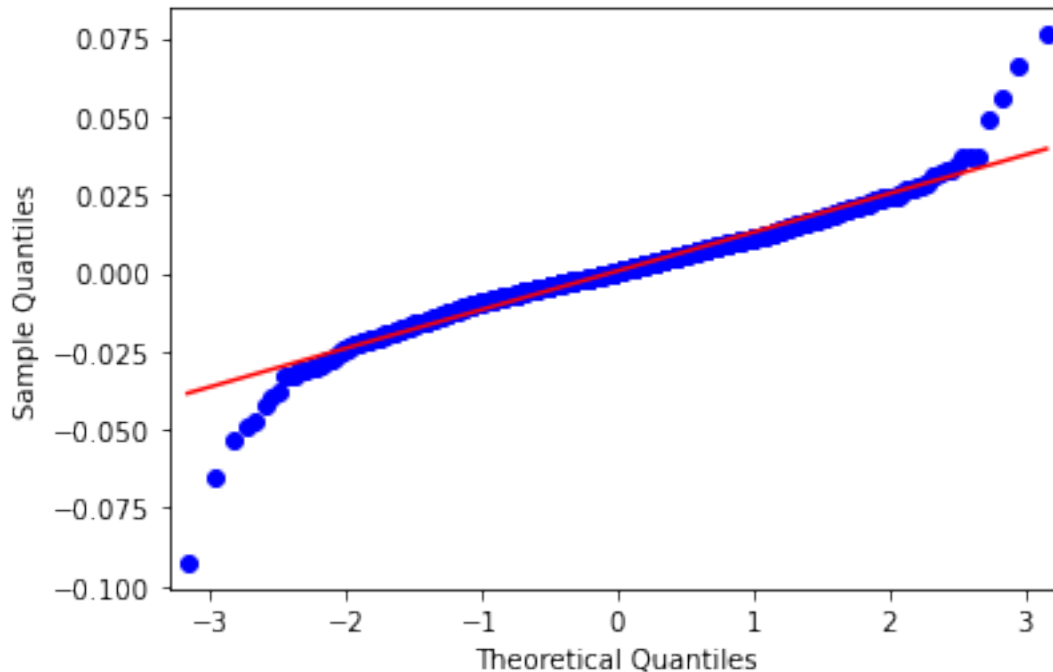
As input, we pass in our data again, first calling drop in to remove any NaN
↳ values and line='s'

So what does line='s' mean?

S means that the line is standardized.
That is, its scaled and shifted by the standard deviation and mean of our data.

There are other possibilities such as R, which means to fit a regression line
↳ if you want to learn
about the different arguments you can pass into this function.
'''

import statsmodels.api as sm
sm.qqplot(sbox['return'].dropna(), line='s');
```



```
[36]: '''
Now, as you know, in finance, sometimes we like to work with the log returns
↳ rather than the returns.
So let's see if anything changes when we use the log returns instead.

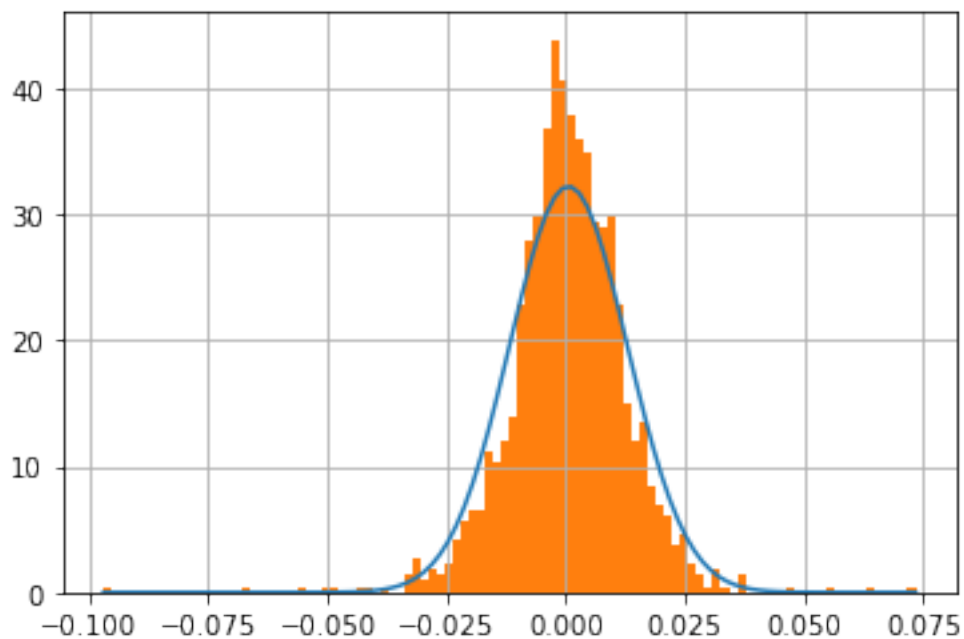
Well, we see that the picture pretty much looks exactly the same.
How can this be?

Well, recall that when the values of the returns are very nearly zero, adding
↳ one and taking the log
does not change its value by a lot.

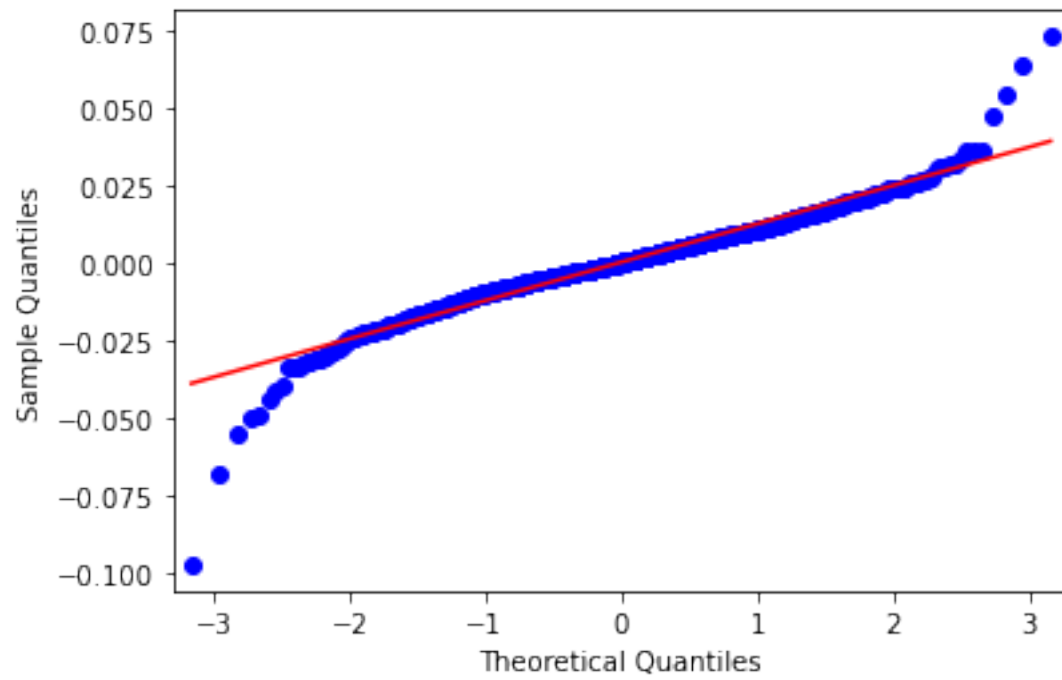
In other words,  $X$  is approximately equal to  $\log(1+X)$  when  $X$  is near zero.
Again, we see the same pattern where the histogram is taller than the
↳ theoretical distribution and
has much more extreme values than the theoretical distribution would admit.
'''
x_list=np.linspace(sbox['log_return'].min(), sbox['log_return'].max(),100)
y_list=norm.pdf(x_list,loc=sbox['log_return'].mean(),scale=sbox['log_return'].
↳std())
```

```
[38]: plt.plot(x_list,y_list)
sbox['log_return'].hist(bins=100,density=True)
```

[38]: <AxesSubplot:>



```
[39]: '''  
If we look at the Q-Q plot, we again see the same pattern, the points diverge_  
→at the ends because  
it has heavier tails compared to the theoretical distribution.  
'''  
sm.qqplot(sbox['log_return'].dropna(),line='s');
```



[ ]: