Mitchell Rosenthal

Following 228 Followers About

Crash Course — Python & Pandas for Trading and Investing (Part 1)

Mitchell Rosenthal Dec 22, 2020 · 14 min read

Python and Pandas make it pretty easy to analyze and visualize time series data, even if you're a beginner. In this crash course, you'll learn about:

- Importing packages
- Making a random time series
- Transforming data to make new columns
- Creating logical conditions and visualizing them as signals
- Simulating a trading strategy and considering slippage

To learn code for more advanced functions, like predictive regressions and conditional distributions, check out this post here.

Let's dive in.

Getting Started & Initial Imports

here.

Once we open up a new Colab file, we can import some helpful tools using the code below. (PS: a notebook with just the code [no comments] is here).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
import statistics
import scipy.stats as stats
import re
from datetime import datetime
from datetime import date
import random
import seaborn as sns
sns.set()
```

You may also want to include the lines below, which prevent outputs from getting truncated.

```
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

Making a Random Time Series

Next, we're going to make a random data series and use it to make graphs and calculate some interesting statistics. In another article, I'll show you how to import data from URLs and avoid some typical errors, but that's not the focus of this post.

First, we'll make the index of our data using "date_range." We have to decide the start date, the frequency, and the number of rows. I picked today's date, December 21, as the start date, used a daily frequency, and created 2500 rows.

Next, we'll make an empty DataFrame that uses the date range we created as its index. To do that, we'll use "pd.DataFrame()." I called this DataFrame "df", but you can call it whatever you want.

```
df = pd.DataFrame(index = our_index)
```

Now we can make some columns. I'm planning on generating fake data for a stock price's closing value, so I'll make one called "Close."

To make a random series that looks like a stock price, I'll use four steps.

> *1: Fill a column with random, small decimal numbers. They represent the logarithm of the daily changes in the stock price, and we'll sample them from a random distribution.*
>
> *2: Replace each number (N) with e^N, so our column will be filled with values near 1 (like 0.99, 1.01, 0.97, etc.). These values represent 1 plus the % change experienced that day.*
>
> *3: Set the first row equal to 1, which is our starting stock price.*
>
> *4: Take the cumulative product of the column.*

Overall, the code looks like this:

```
df['Close'] = np.random.normal(loc=0.0002, scale=0.011, size=2500)
df['Close'] = np.exp(df['Close'])
df['Close'].iloc[0] = 1
df['Close'] = df['Close'].cumprod()
```

Why do we do step 4? Suppose our first three values are 1, 1.01, and 1.02. We want the value of the stock on day 2 to equal (1)*(1.01). The value of it the next day should equal (1)*(1.01)*(1.02). This is what the cumulative product does for us.

over time.

To make sure this worked, let's quickly plot this column using the line below. Hopefully, it will look like a stock price.

```
df['Close'].plot(figsize=(7,5), title='Our Fake Stock Price')
```

Looks good! Note that the plot above doesn't have a legend that tells you what the name of this line is. When you plot a DataFrame, it will automatically include a legend. But we didn't plot a DataFrame, we plotted df['Close'], which is a series (a column of a DataFrame). If we want to plot it as a DataFrame, we would use double brackets, like this:

Another option is to plot the series, df['Close'], and just add legend=True inside plot().

Next, we will make another column that equals our stock price after being transformed somehow, and we'll run some calculations on it.

Calculating a Moving Average

Since a stock price can be volatile, it can be helpful to look at its moving average, which appears smoother and gives us an idea of the overall trend. I decided to find the average price over the past 100 days. Here's the code, and a plot that shows 'Close' and its moving average.

```
df['100MA'] = df['Close'].rolling(100).mean()
df[['Close','100MA']].plot(figsize=(7,5), title='Our Fake Stock
Price')
```

So far so good. Keep in mind, it's usually better to plot non-stationary (trendy) time series using log scale so that every % change takes up the same vertical distance. To do that, just add "logy=True" inside plot().

Now, we're going to make one final column that we'll later use for some interesting calculations. This column is called '100MAdist', and it equals the distance between 'Close' and its moving average.

```
df['100MAdist'] = -1 + (df['Close'] / df['100MA'])
```

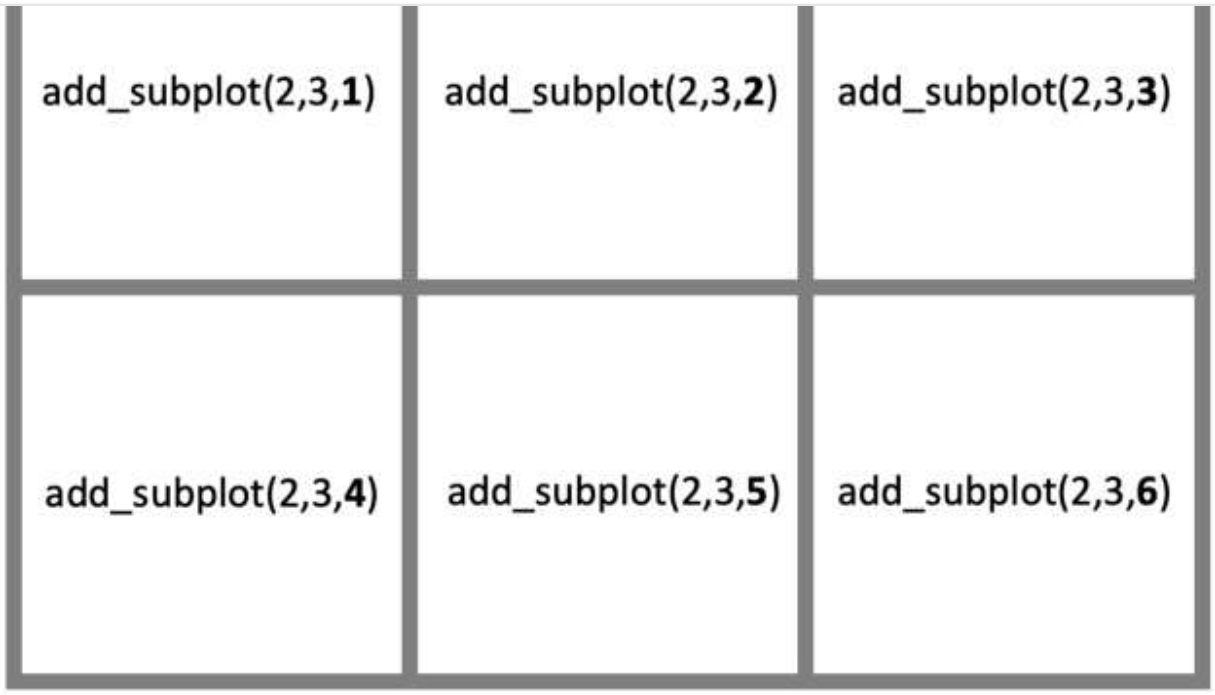
Making Multiple Plots

To see what this looks like, we'll make two plots. The first one shows 'Close' and its moving average. The one below that shows '100MAdist.' First, we make a figure called f1 and specify its width and height. Then, we add two subplots, called ax1 and ax2, using add_subplot().

```
f1 = plt.figure(figsize=(7,5))

ax1 = f1.add_subplot(2, 1, 1)
ax2 = f1.add_subplot(2, 1, 2)
```

The first two numbers inside each add_subplot() tell us how many rows and columns we want our figure to have. In this case, we used 2 rows and 1 column, so we could put one plot in the first row and one plot in the second row.

The third number inside add_subplot() specifies the position of each subplot. This value starts at 1 and gets bigger as you move to the right and down. When it's 1, it refers to row 1, column 1. When it's 2, it refers to the row 1, column 2, and so forth. Once you run out of columns, it moves on to the next row down. To get a sense of how this third number works, check out the image below, which shows a situation with 2 rows and 3 columns.

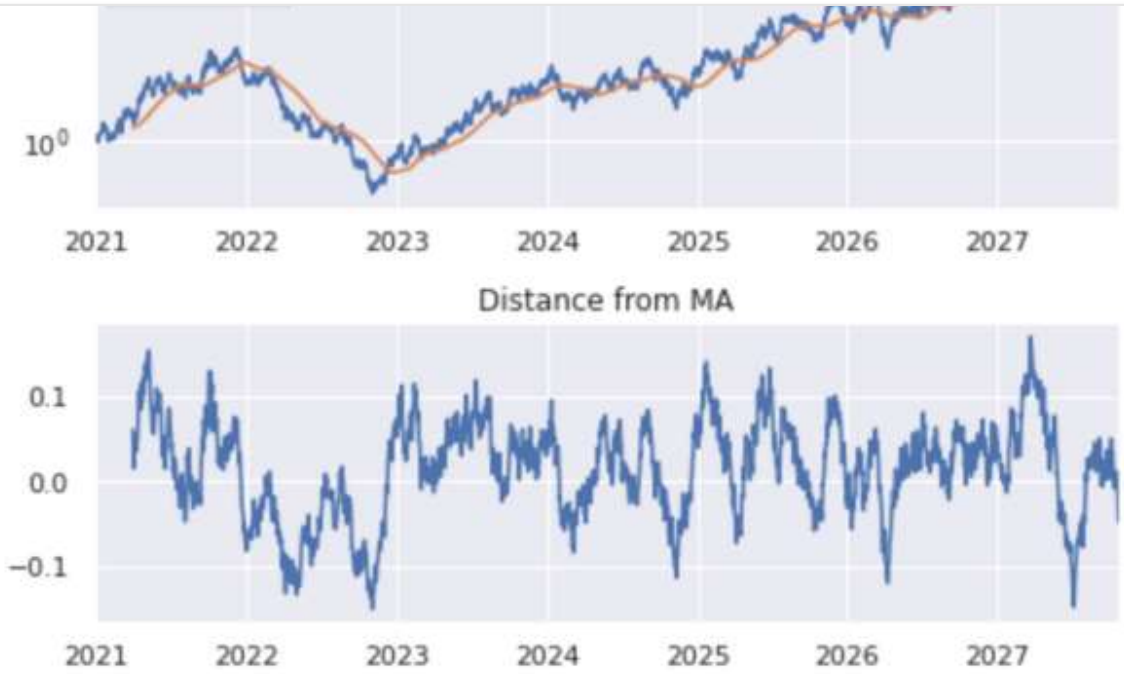


After making the subplots ax1 and ax2, we add two more lines of code, one for each plot.

```
f1 = plt.figure(figsize=(7,5))
ax1 = f1.add_subplot(2, 1, 1)
ax2 = f1.add_subplot(2, 1, 2)

df[['Close','100MA']].plot(title='Our Fake Stock Price', ax=ax1,
logy=True)
df['100MAdist'].plot(title='Distance from MA', ax=ax2)
plt.tight_layout()
```

The first one plots a DataFrame that contains the ‘Close’ and ‘100MA’ columns, and it specifies that we want it shown on ax1. The next one plots the ‘100MAdist’ column and specifies that we want it shown on ax2. Finally, we use plt.tight_layout() to make sure our labels don’t overlap.

The results look like this:



Visualizing Signals

Now, we'll apply some kind of logical condition and see what happens when we try to make buy and sell decisions based on it. To keep things simple, we'll test a simple strategy that stays invested only when the stock price is above its moving average.

First, we'll visualize the signal. Using np.where(), we can make a column that equals "1" when our condition is satisfied and "0" otherwise. I'll call this column 'MAsignal.'

```
df['MAsignal'] = np.where(df['Close']>df['100MA'], 1, 0)
```

The code above creates a column that equals 1 when the 'Close' column is greater than the '100MA' column, and 0 otherwise.

Next, we can visualize this signal by putting all the x-values where the signal is True into a list. For each x-value in that list, we'll plot a vertical line using "axvline."

To get those x-values, we need to filter our DataFrame and only keep the rows where our signal is True. The general way to filter a DataFrame is shown below:

The condition we are interested in is the 'MAsignal' column being equal to 1. So we replace SomeConditionHere with df['MAsignal']==1.

```
dfFiltered = df[df['MAsignal']==1]
```

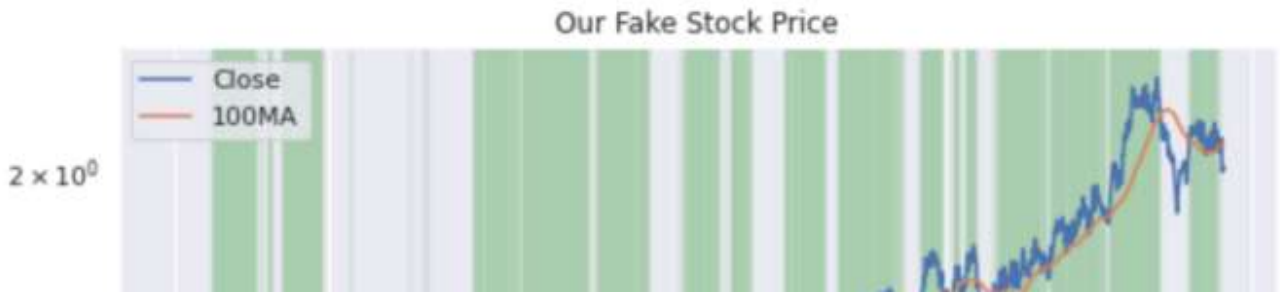
Now, we save the dates of this filtered DataFrame as a list. Dates are located in the index of our DataFrame. So we need to convert the index of dfFiltered into a list.

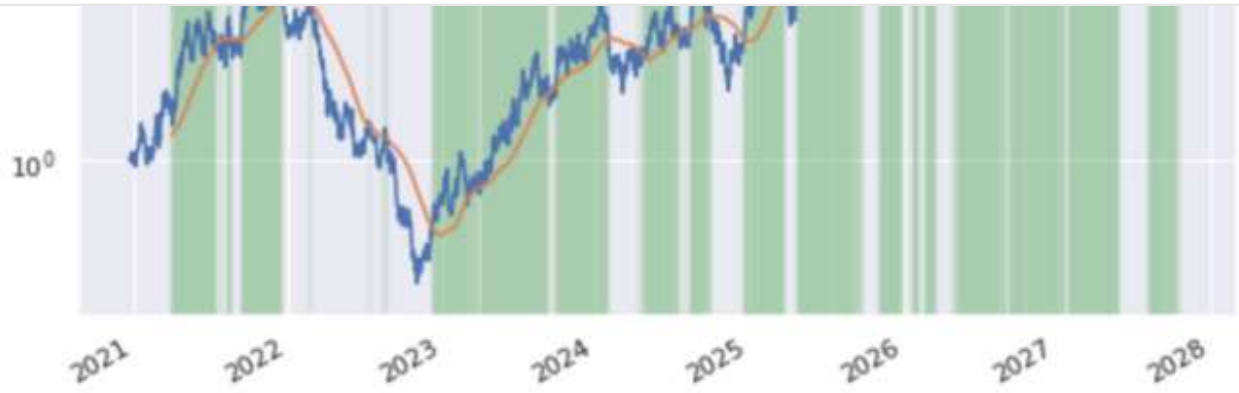
```
truedates = dfFiltered.index.tolist()
```

Now we can start plotting.

```
f1 = plt.figure(figsize=(9,6))
ax1 = f1.add_subplot(1, 1, 1)

df[['Close','100MA']].plot(title='Our Fake Stock Price', ax=ax1,
logy=True)

for x in truedates:
    ax1.axvline(x, color='tab:green', alpha = 0.27, linewidth = .25,
linestyle='-')
```

Note that axvline's "alpha" refers to the opacity of each vertical line. The closer it is to 0, the more transparent the line is. Also, it's important to call plot(ax=ax1) before calling ax1.axvline to avoid formatting problems.



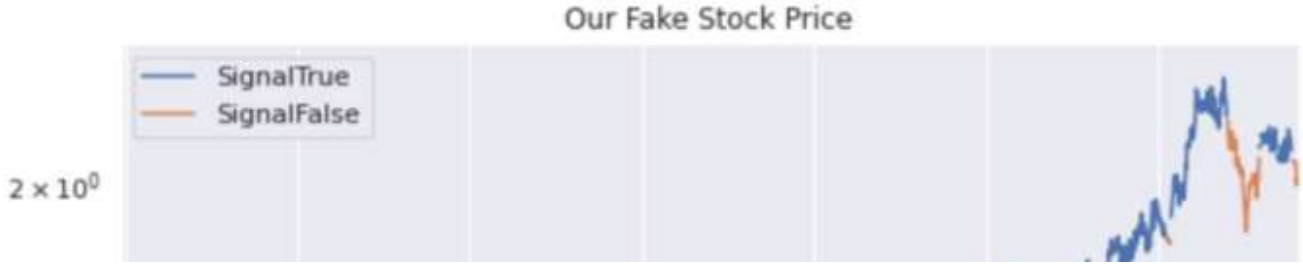


Another way to do this is to make new columns that only exist when our condition is True or False, and plot them.

```
df['SignalTrue'] = np.where(df['Close']>df['100MA'], df['Close'],
np.NaN)

df['SignalFalse'] = np.where(df['Close']>df['100MA'], np.NaN,
df['Close'])
```

The ‘SignalTrue’ column equals the ‘Close’ column, but only when our condition is satisfied. Otherwise, it doesn’t exist (it equals np.NaN, “not a number”). The ‘SignalFalse’ column equals the ‘Close’ column, but only when our condition is false. When we plot them together, each will have a different color.

```
f1 = plt.figure(figsize=(9,6)
ax1 = f1.add_subplot(1, 1, 1)

df[['SignalTrue','SignalFalse']].plot(title='Our Fake Stock Price',
ax=ax1, logy=True)
```



Some folks have trouble distinguishing between different colors. To take this into account, we can differentiate the lines by giving them different levels of opacity. In the code below, we specify a lower opacity than usual for the 'SignalFalse' line by saying "alpha=0.50."

```
f1 = plt.figure(figsize=(9,6))
ax1 = f1.add_subplot(1, 1, 1)

df[['SignalTrue']].plot(title='Our Fake Stock Price', ax=ax1,
logy=True)
df[['SignalFalse']].plot(ax=ax1, alpha=0.50, logy=True)
```

Testing a Trading Signal

Simulating trading strategies can be a lot of fun. Let's learn how to do it in a realistic way. This means considering slippage and being precise about when trades take place.

We should remember that in the real world, securities fluctuate throughout the day. They start trading at one price ("open"), close at another price ("close"), and reach some maximum ("high") and minimum ("low") during the time in between.

Our signal is based on the close: if the close is above the MA, we want to be in the market. The problem is, by the time we know the close price, that trading increment is over; we can't enter until the open of the next time increment. We need to take that into account.

Making OHLC Data

To start, I'm going to generate OHLC data. Here are the steps:

> *Generate the stock price data in 30 minute increments.*
>
> *Filter the DataFrame, keeping times when the market is open.*
>
> *Resample the filtered data using resampler('D').ohlc().*

First, we make our index. Earlier, we created 2500 days. Each day has 24 hours, and each hour has two 30-minute increments. So to have data covering the same time period, we need to make 2500*24*2 rows of 30-minute increments. We'll set our freq as "30T" to specify 30-minute time increments.

```
our_index = pd.date_range(start='2020-12-21', periods=2500*24*2,
freq='30T')
df = pd.DataFrame(index = our_index)
```

Earlier, we set the mean change from the end of the previous day to the end of the current day to be (0.0002). Since we are making higher-frequency data, we need to reduce this mean % change to preserve the original rate of growth. Recall that there are 48 30-minute increments per day.

The new % change, called R, relates to the old one (r) in this formula:

$(1+R)^{48} = (1+r)^{1}$

$(1+R) = (1+r)^{\wedge}(1/48)$

$R = [(1+r)^{\wedge}(1/48)]-1$

What about stdev? For stocks that follow a random walk, the variance of their returns is directly proportional to time. Standard deviation is the square root of variance, so it is proportional to the square root of time. So the new stdev (STDEV) is related to the old one (stdev) like so:

$(STDEV)/(sqrt(1/48)) = (stdev)/(sqrt(1))$

$STDEV = (stdev)(sqrt(1))(sqrt(1/48))$

Ok, now we can make a new series for our stock price:

```
R_old = 0.0002
Stdev_old = 0.011

R_new = (1+R_old)**(1/48)-1
Stdev_new = (Stdev_old)*(1**0.5)*((1/48)**0.5)

df['Close'] = np.random.normal(loc=R_new, scale=Stdev_new,
size=2500*24*2)
df['Close'] = np.exp(df['Close'])
df['Close'].iloc[0] = 1
df['Close'] = df['Close'].cumprod()
df['Close'].plot(figsize=(7,5))
```

filtered DataFrame. I decided to rename the columns so each one starts with a capital letter. Then I used df.head() to view the first 5 rows of the DataFrame to see if things worked. Here's the code.

```
df = df.between_time('09:30', '16:00')
df = df['Close'].resample('1D').ohlc()
df.rename(columns=
{'open':'Open','high':'High','low':'Low','close':'Close'},
inplace=True)

df.head()
```

The result:

	Open	High	Low	Close
2020-12-21	0.996568	1.001727	0.995915	0.995915
2020-12-22	0.987578	0.992072	0.986215	0.989944
2020-12-23	1.001950	1.008166	1.001950	1.004320
2020-12-24	1.009122	1.019739	1.007000	1.014211
2020-12-25	1.000228	1.004986	1.000022	1.004986

Looks good. Now we'll track a portfolio that implements our strategy.

Betting on the Strategy

In this strategy, we compare each day's close with its moving average. If the close is above it, we want to be in the market, if not, we want to be out of the market. Buy

transactions incur slippage.

For any given day, there are four potential % changes our portfolio might experience.

1. If the signal equaled 1 at the end of yesterday's close, and 0 the day before, then today we buy at the open. Today, we experience the change from the open to the close.
2. If the signal equaled 1 at the end of yesterday's close and the day before, we continue to hold through today. We experience the change from yesterday's close to today's close.
3. If the signal equaled 0 at the end of yesterday's close and 1 the day before, then today we exit (sell) at the open. Today, we experience the change from yesterday's close to today's open. This is the overnight change.
4. If the signal equaled 0 at the end of yesterday's close and 0 the day before, we do nothing today and experience no change. This is our default setting.

First, we'll create our three main conditions, one for buying, one for holding, and one for selling. Note that shift(1) refers to the value in the row above. For a DataFrame in ascending order, lower rows are more recent, so shift(1) refers to an older value.

```
df['100MA'] = df['Close'].rolling(100).mean()
df['MAsignal'] = np.where(df['Close']>df['100MA'], 1, 0)

condition1 = (df['MAsignal'].shift(1)==1) &
(df['MAsignal'].shift(2)==0)

condition2 = (df['MAsignal'].shift(1)==1) &
(df['MAsignal'].shift(2)==1)

condition3 = (df['MAsignal'].shift(1)==0) &
(df['MAsignal'].shift(2)==1)
```

Now we can create a column of the daily changes experienced by our portfolio when betting on this strategy. I'll call it 'PortChng'. I'll use np.where() to consider these

```
df['PortChng'] = 0 # default setting

df['PortChng'] = np.where(condition1, -1 + df['Close']/df['Open'],
df['PortChng'])

df['PortChng'] = np.where(condition2, -1 +
df['Close']/df['Close'].shift(1), df['PortChng'])

df['PortChng'] = np.where(condition3, -1 +
df['Open']/df['Close'].shift(1), df['PortChng'])
```

Finally, we'll create a column of our portfolio value over time. All we have to do is add 1 to the column of daily changes, and then take the cumulative product.

```
df['Port'] = df['PortChng'] + 1
df['Port'] = df['Port'].cumprod()
```

Let's plot this and compare it to just being in the stock price the whole time. I'm going to specify that the legend of the plot is in the upper left. I'm also going to add green vertical lines to mark when our strategy is in the market at any point in a given day; in other words, when conditions 1, 2, or 3 are satisfied. The "|" symbol is used to mean "or."

```
f1 = plt.figure(figsize=(9,6))
ax1 = f1.add_subplot(1, 1, 1)

true_indexvals = df[condition1|condition2|condition3].index.tolist()

df[['Close','Port']].plot(title='Testing the 100MA Strategy', ax=ax1,
logy=True)

for x in true_indexvals:
  ax1.axvline(x, color='tab:green', alpha = 0.22, linewidth = .25,
linestyle='-')

ax1.legend(loc='upper left')
```

As you can see, when we are out of the market, our portfolio is unchanged (flat) and there are no vertical green lines.

Considering Slippage

All of your transactions go through a market maker, whose job it is to buy and sell the stock at all times. Market makers adjust their prices to try to satisfy as many orders as possible. When you buy, you have to pay their asking price (“ask”), and when you sell, you have to receive their bidding price (“bid”). In exchange for their troubles, they make money from the bid/ask spread; they sell at the ask, and buy at the bid, which is lower.

When backtesting a strategy, we should assume that we will lose some money to this bid/ask spread. The historical OHLC data we are looking at shows actual transaction prices. To account for slippage, we can pretend these prices represent the “mid” price, which is halfway between the bid and the ask. Then, we make a parameter that assumes some bid/ask spread. This allows us to generate a realistic bid and ask price for any given mid price. We will assume that we have to buy at the ask and sell at the bid.

This bid/ask spread is often quoted like this:

The mid price equals:

> *mid = (ask+bid)/2*

Rearranging, we can solve for the bid and the ask as a function of "mid" and "BidAskPct":

> *ask = [(2)(mid)]/[2-BidAskPct]*
>
> *bid = (2)(mid)-ask*
>
> *bid = (2)(mid)-[(2)(mid)]/[2-BidAskPct]*

We can use this formula to find the bid and ask associated with each transaction price we have: open, high, low, and close. I'll store these key columns in a list, and find each column's bid and ask. When backtesting, we usually assume we can only transact at the open or the close, so my list will only have those two.

Remember that to make a new column, we do:

```
df[ColumnName] = something
```

Column name is some string, like "NewColumn." If I want to instead call it "NewColumnBig", I can do this:

```
df[ColumnName+'Big'] = something
```

This is a neat trick for when we use loops to make new columns. Let's try it out below.

```
BidAskPct = 4/(100*100) # (ask-bid)/ask = 4 basis points
list_cols = ['Open','Close']

for i in list_cols:
  df[i+'Ask'] = (2*df[i])/(2-BidAskPct)
```

To see if it works, let's look at the first five rows of our DataFrame, but look only at columns that contain the word "Open" or "Close." To do this, we can use a handy tool called Regex. Before using it, type "import re".

```
import re
df.filter(regex='Open|Close').head()
```

The result:

	Open	Close	OpenAsk	OpenBid	CloseAsk	CloseBid
2020-12-21	0.996568	0.995915	0.996767	0.996368	0.996114	0.995715
2020-12-22	0.987578	0.989944	0.987776	0.987381	0.990142	0.989746
2020-12-23	1.001950	1.004320	1.002151	1.001750	1.004521	1.004119
2020-12-24	1.009122	1.014211	1.009324	1.008920	1.014414	1.014008
2020-12-25	1.000228	1.004986	1.000428	1.000028	1.005187	1.004785

Looks good. CloseAsk is higher than Close, which we pretend is the mid. CloseBid is below it. Same thing for Open.

Now, we need to redefine our 'PortChng' column so that buy and sell transactions include slippage. When we buy the open, we need to pay the asking price of the market maker, "OpenAsk." When we sell the open, we need to receive the bid price, "OpenBid."

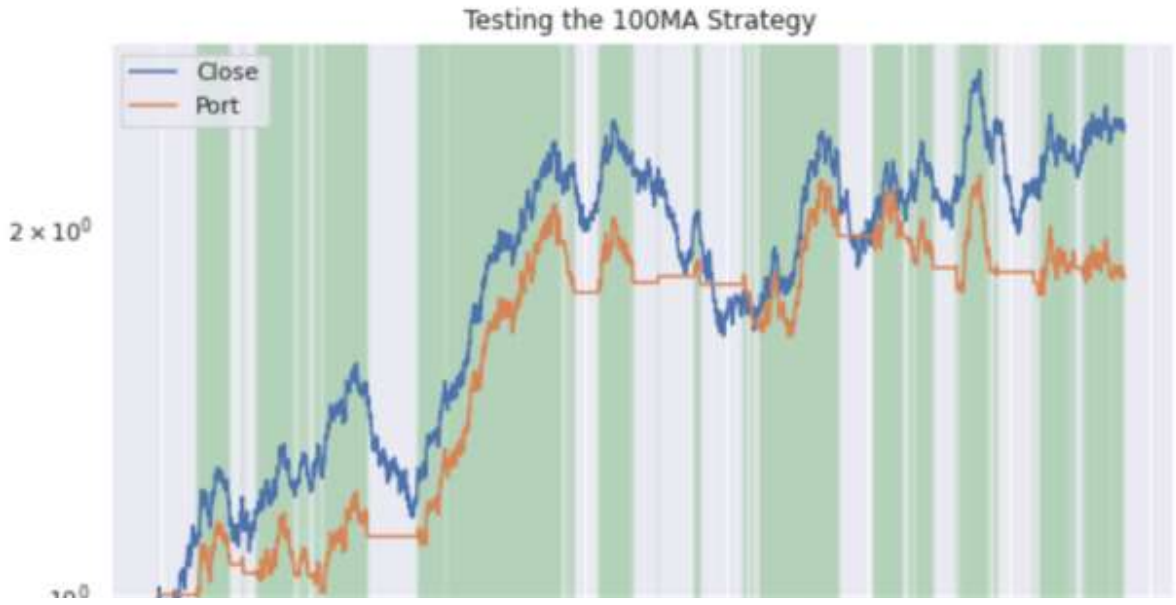
```
df['PortChng'] = 0 # default setting

df['PortChng'] = np.where(condition1, -1 + df['Close']/df['OpenAsk'],
df['PortChng'])
```

```
df['PortChng'] = np.where(condition3, -1 +
df['OpenBid']/df['Close'].shift(1), df['PortChng'])

df['Port'] = df['PortChng'] + 1
df['Port'] = df['Port'].cumprod()
df['Port'].plot(figsize=(7,5))
```

Then, we can graph our portfolio just as we did before.

```
f1 = plt.figure(figsize=(9,6))
ax1 = f1.add_subplot(1, 1, 1)

true_indexvals = df[condition1|condition2|condition3].index.tolist()

for x in true_indexvals:
  ax1.axvline(x, color='tab:green', alpha = 0.22, linewidth = .25,
linestyle='-')

df[['Close','Port']].plot(title='Testing the 100MA Strategy', ax=ax1,
logy=True)

ax1.legend(loc='upper left')
```

Here's the final result:



Note that this doesn't look very different than our previous plot. That's because we didn't have very many buy and sell transactions. But for higher-frequency strategies, slippage will make a remarkable difference on performance.

Nice work! I hope you enjoyed this crash course. The next one will cover how to import and clean data stored on Google Drive, GitHub, or some other URL, and it'll show you how to run some statistical studies.

Until next time,

Mitch

Python Trading Data Science Statistics Investing