Time Series Data in Python

CS 7646 "Part 1"
Instructor: David Byrd

May 30, 2016

1 Market Data Basics

```
Date,Open,High,Low,Close,Volume,Adj Close
2012-09-12,144.39,144.55,143.90,144.39,87607000,144.39
2012-09-11,143.60,144.37,143.56,143.91,88647200,143.91
2012-09-10,144.19,144.44,143.46,143.51,86458500,143.51
2012-09-07,144.01,144.39,143.88,144.33,107272100,144.33
2012-09-06,141.76,143.78,141.75,143.77,158272500,143.77
2012-09-05,141.09,141.47,140.63,140.91,100660300,140.91
```

In this class, we use historical S&P 500 data from Yahoo! Finance. The data comes in files with name IBM.csv, MSFT.csv, etc. Each row of the CSV file contains information about the high-level activity of the given stock on a single historical date. The columns are:

0	Date	Date of this information, format YYYY-MM-DD.
1	Open	Price at which the stock opened at 9:30 AM ET.
2	High	The highest price at which the stock *traded* during the day.
3	Low	The lowest price at which the stock *traded* during the day.
4	Close	Price of the final trade of the day, as recorded *on that day*.
5	Volume	Number of shares that changed hands during the day.
6	Adj Close	Stock closing price after adjusting for all *future* splits and dividends.

Note that High and Low only consider successful trades, not merely offers to buy or sell at a price. Volume counts the shares in a trade only once (not once for the "buy" and once for the "sell"), but if a trade *cascades* (forces other trading activity), then each subsequent trade will be counted separately. Close and Adjusted Close will be the same from today backwards to the most recent split or dividend payment, then they will diverge (more details in the finance section of the course). Use Adjusted Close to avoid "jumps" (discontinuities) in your data for accurate backtesting.

2 Key Pandas Concepts

2.1 What you get from pandas.read_csv()

Index	Date	Open	High	Low	Close	Volume	Adj Close
0	(Today)						
1	(Yesterday)						
2	(Day before Yesterday)						
	...						
N	(Earliest date available)						

`df = pandas.read_csv(filename)` will read a CSV file and return a reasonable DataFrame object with the contents of the file. It will automatically assign integer indices to rows and use the first row of the file (if non-numeric) as string column indices. Rows will be in the same order they appeared in the file, as seen above.

```
Date    Open    High     Low   Close     Volume  Adj Close
0  2012-09-12  144.39  144.55  143.90  144.39   87607000     144.39
1  2012-09-11  143.60  144.37  143.56  143.91   88647200     143.91
2  2012-09-10  144.19  144.44  143.46  143.51   86458500     143.51
3  2012-09-07  144.01  144.39  143.88  144.33  107272100     144.33
4  2012-09-06  141.76  143.78  141.75  143.77  158272500     143.77
```

That was easy, but it is not what you want.

2.2 What you want

	SPY	IBM	GOOG
2011-07-01			
2011-07-05			
2011-07-06			
2011-07-07			

You want a DataFrame that contains a subset of dates you can specify (start→end) and only days the stock market was open. You want those date indices sorted in *ascending* order (oldest to newest). You want only the Adj Close price, but you want it for multiple stock symbols, properly lined up. You

want the date column to be used as the row index, and to be "real" date variables (not strings). You want missing values to be represented as Python/Pandas `NaN` values (not the string "NaN").

2.3 How you get there

1. Build a date range object with `pd.date_range(start,end)`
2. Use it to create an empty `DataFrame` with the dates you want using `pd.DataFrame(index)`
3. Read the SPY data file using `read_csv()`. Be sure to check out these parameters:
   (a) `index_col`
   (b) `parse_dates`
   (c) `na_values`
   (d) `usecols`
4. We read SPY first, because it is the S&P 500 Stock Market Index. It has prices for a given day if (and only if) the market was open that day.
5. Put the two `DataFrame` objects together using `DataFrame.join()`
   (a) The `DataFrame` on which you call `join()` is considered the "left" table.
   (b) The `how` parameter gives SQL-like join types (`left, right, inner, outer`).
   (c) Either use an `inner` join or use `DataFrame.dropna()` to eliminate dates the market was not open.
6. You have a combined `DataFrame` with your dates and the `'Adj Close'` column of SPY.
   (a) Be sure to use `DataFrame.rename(columns)` to change the name of the new column to `'SPY'`!
7. Read another stock data file, add it to your main `DataFrame` using `join`, rename the new column, and repeat until done.

Now you have a single `DataFrame` object with all of the stock symbols you care about as columns and the date range you want as the rows. There are only rows for days the market was open, and stocks that did not trade on a given day will be `NaN`.

Now you can do date math on "market open days" trivially. Row `n+5` will be five market days after row `n`, regardless of how many calendar days that is.

3 Key Python differences

- `import library_name`
  - use functions as `library_name.function_name()`
- `import library_name as xyz`
  - use functions as `xyz.function_name()`
- `from library_name import function_name()`
  - use function as `function_name()` directly
- `def some_function (a=10, b=['x','y'], c=None):`
  - can call function with any/all/no parameters, even out of order if names provided
  - e.g. `some_function(b=[1,2,3])`
    - `a` will have its default value of 10
    - `c` will have its default value of None
- `list[3:6]`
  - slicing in Python is [inclusive, exclusive)
  - above example gets elements 3, 4, 5
  - since Python is 0-based, this is the fourth, fifth, and sixth elements
- `for item in list:`
  - Python's for loop always iterates over a list
  - there is no C-like `for (int i=0; i<10; i++) { }`
- `for i in range(min,max,step):`
  - the `range` function allows simulating a C-style for loop
  - e.g. `for i in range(0,10):`
  - the max is exclusive as normal
- Python is strongly typed (but dynamically typed)
  - Variable types "figured out" by first use
  - Variables do not have to be pre-declared
  - Once a variable *has* a type, it can only be that type
  - `''High''+''Five''` results in `''HighFive''`
  - `1+5` results in `6`
  - `''High''+5` throws a type mismatch error, can't add string and int
  - `''High''+str(5)` results in `''High5''`

4 Key Numpy concepts

- Numpy is "Python's Matlab". It is written in C and Fortran and is very fast (~600x faster than normal Python code).
- Pandas is a wrapper around numpy, extending it with date indices and useful time series functions
- `DataFrame.values` will get the `numpy.ndarray` from a Pandas `DataFrame`

4.1 Slicing ndarray

Assume you have two 2-D numpy arrays named `a` and `b`:

- `a[0]` gets the first row of a
- `a[:,0]` gets the first column of a
- `a[0,0]` gets the first element of a
- `a[1:3,1:3]` gets a 2-D slice, elements `(1,1),(1,2),(2,1),(2,2)`
- `a[1:3,-1]` gets the last element of rows 1 and 2
- `a[1:10:2]` gets every other row from 1 to 10 (exclusive)
- `a[2:4]=b[1:3]` replaces the contents of rows 2,3 of a with rows 1,2 of b

4.2 Creating ndarray

You can use the following to create a new `ndarray` with three rows and four columns:

- `np.empty([3,4])`
  - uninitialized values
- `np.zeros([3,4])`
  - zero-filled
- `np.ones([3,4])`
  - one-filled
- `np.random([3,4])`
  - filled with random floats from 0 to 1 (exclusive)
- `np.normal(100,50,size=[3,4])`
  - filled with a normal distribution of mean 100, std 50

- `np.randint(0,10,size=[3,4])`
  - filled with random ints from 0 to 9

You can also use `np.array(list)` to create an `ndarray` from any list-like or array-like object you already have. You can use `np.random.seed(int)` to fix the random number generator for repeatable testing.

4.3 Attributes and Functions of ndarray

Some useful attributes:

- `ndarray.shape`
  - returns shape of array in each dimension, e.g. `(3,4)`
- `ndarray.shape[1]`
  - gets length of dimension 1 (columns), e.g. `4`
- `ndarray.size`
  - gets total number of elements, regardless of shape, e.g. `12`
- `ndarray.dtype`
  - gets data type of ndarray, e.g. `float64`

Some useful functions:

- `ndarray.sum()`
  - sums all elements to a single scalar number
- `ndarray.sum(axis=0)`
  - sums across axis 0 (rows), i.e. sums along columns, returns vector of column sums
- `ndarray.min()`
- `ndarray.max()`
- `ndarray.mean()`
  - these can all take an axis parameter to return a vector instead of a single scalar number

4.4 Advanced indexing and broadcasting

- Numpy will repeat a scalar or vector automatically to make operator arguments the same size
  - e.g. `a[:,1]=4` will set the entire second column of values to the number 4
- `ndarray[ndarray]` does work for indexing!
  - `a[a.argmax()]=a.min()` will replace the max value in `a` with the min value
  - `a[np.array[1,4,1,0]]` will return a four-element view into `a` containing the values at indices 1, 4, 1 again, and 0
  - `a[a < a.mean()]=0` *will work* to replace all elements in `a` less than the mean with zeros

4.5 Notes on mathematical functions

- Numpy matrix operations are always *element-wise* by default.
- You must use `ndarray.dot()` and `ndarray.cross()` for "matrix multiplication"

5 Time Series Statistics

5.1 Global Statistics

When we refer to "global statistics" on a time series, we mean statistical values that are computed to a single number for the entire time series. For example, the "global mean" for IBM's close price in 2009 is the single scalar value that represents its average price in 2009.

This is fine for long-term thinking about IBM, but what if we were trying to trade IBM during this period? If IBM rose steadily during the first two quarters and fell during the last two quarters, knowing that its mean for the year did not change from the previous year's mean is not very helpful. It does not help us decide whether to be long or short on IBM in February.

- `df.values.mean()` averages all stocks in the `DataFrame` together. That isn't very helpful.
- `df.mean()` automatically returns a 1-D series of values, the mean of each stock across our time range.

5.2 Rolling Statistics

We primarily use rolling statistics for time series data. Rolling statistics answer the question: "How has the series been performing *lately*?" You will select a window size (in days) for how far back to look when computing a rolling statistic. That window "slides forward" each day as you perform your computation.

- You want the 5-day rolling mean of IBM in some time period, called the SMA-5 (simple moving average)
  - `rolling_mean[5] = IBM_prices[0:5].mean()`
  - `rolling_mean[6] = IBM_prices[1:6].mean()`
  - and so forth
- Note that you can't compute the SMA-5 until the sixth day!
- Pandas supports dozens of rolling statistics via the `pandas.stats` package.
  - You pass in a `DataFrame` or `Series` and a window size. It gives you a `Series` back.

5.3 Daily Returns

- GOOG trades in the $700 range
- LC (Lending Club) trades in the $5 range
- How to easily compare? A $1 change in GOOG means nothing. For LC, it is *huge*!
- Compute a series of daily returns (zero-based, percent change from previous day)
  - `daily_rets[t] = (price[t] / price[t-1]) - 1`
  - e.g. daily return of -0.1 means price dropped 10% from previous day
- Now you have "apples to apples" data to compare between GOOG and LC
- It is also helpful to plot daily returns of a stock vs the daily returns of the SPY
  - Makes it clear when this stock "zigs" as the market "zags"

5.4 Cumulative Returns

- Stocks rise and fall constantly during the year
  - So does your investment portfolio
- How to simply evaluate overall performance as of some date?
- Compute a series of cumulative returns (zero-based, percent change from day 0)
  - `cumulative_rets[t] = (price[t] / price[0]) - 1`
  - e.g. cumulative return of 0.15 means stock rose 15% from day 0 to day `t`
  - Some people use one-based cumulative returns (e.g. 0.9 means stock fell 10%)

5.5 Problems with Stock Data

- Assumptions
  - Stock data is perfect
  - Recorded moment by moment
  - Always present when it should be, no gaps
- Reality
  - Stock data is a "best estimate" at any time
  - Many gaps in data for many reasons
  - There may not even be a "real" stock price
- What to do with all those `NaN` values?
  - Fill forward, then fill backward
  - `DataFrame.fillna()` will help!
  - Filling forward first ensures your trading strategy does not "see the future"

6 Portfolio Statistics

For most of this class, what you really want to evaluate is not just one stock, but some aggregate performance of an entire portfolio of different allocations to different stocks. We will assume allocations represent percentages that sum to 1 (or 100%).

- We have $1M and buy-and-hold this portfolio on day 0 of our evaluation window (2009-2011):
  - 40% SPY
  - 40% XOM
  - 10% GOOG
  - 10% GLD
- How can we evaluate our performance at the end of 2011?
- Build a `DataFrame` for all these stocks, as we have discussed before.
  - We start with raw prices for the stocks
  - Normalize the price to cumulative returns.
    - Now the top row is all 1s
  - Multiply everything by our allocations
    - Now the top row is the allocations we bought
  - Multiply by our starting cash
    - Now the top row is the dollar value of each position we bought
    - And every row is the dollar value of the position on that day
  - Our daily portfolio value is just the sums across the stocks on a given day
  - Now you can calculate daily returns and cumulative returns on the portfolio dollar values

The key portfolio statistics you will use the most in this class are cumulative return (especially on the final day), average daily return (mean of daily returns), standard deviation of daily returns, and the Sharpe ratio of the portfolio.

6.1 Sharpe Ratio

Sometimes it is clear which stock (or portfolio) is the better risk-vs-reward.

- Portfolio A returns 10% in a year with low variance.
- Portfolio B returns 5% in a year with similar variance.
  - Portfolio A is better (more return for similar risk)
- Portfolio C returns 10% in a year with high variance.
- Portfolio D returns 10% in a year with low variance.
  - Portfolio D is better (same return for lower risk)
- Portfolio E returns 10% in a year with higher variance.
- Portfolio F returns 9% in a year with lower variance.
  - Hard to say which is better for sure.

William Sharpe proposed a quantitative measure of risk-adjusted return to answer this question. It has since become known as the Sharpe Ratio.

$$SR = \frac{\mathbb{E}[R_p - R_f]}{\text{std}[R_p - R_f]}$$

where R_p is the return of our portfolio and R_f is the risk-free rate of return (e.g. short-term US Treasury Bonds or the LIBOR rate). This formulation is *ex ante* (looking forward). We usually calculate it historically:

$$SR = \frac{\text{mean}(dailyrets - dailyriskfree)}{\text{std}(dailyrets - dailyriskfree)}$$

Since the risk-free rate of return does not change much on a daily basis (it is a very long-term economic factor), it is common to approximate it with a constant. Once this is done, we can place it outside the expectation in the numerator, and we can eliminate it entirely in the denominator (altering a vector by a constant can not change its variance).

In fact, however, for many years now the risk-free rate of return in the United States and elsewhere has been *zero*.

6.1.1 Annualized Sharpe Ratio

- Sharpe Ratio varies widely based on selected sampling frequency (annual, daily, monthly)
- SR was designed as an annual measure, so values are always *annualized*
- $SR_{annualized} = K \cdot SR$
- $K = \sqrt{samples/year}$ or the sampling rate (number of samples is irrelevant)
- Common values of K
  - Daily sampling: $\sqrt{252}$ (on average, 252 market trading days per year)
  - Weekly sampling: $\sqrt{52}$
  - Monthly sampling: $\sqrt{12}$
  - Yearly sampling: 1
- **Important:** only sampling *rate* matters, not the actual number of samples you have taken!

So the final formulation of Sharpe Ratio we will typically use is:

$$SR = K\left[\frac{\text{mean}(dailyrets)}{\text{std}(dailyrets)}\right]$$