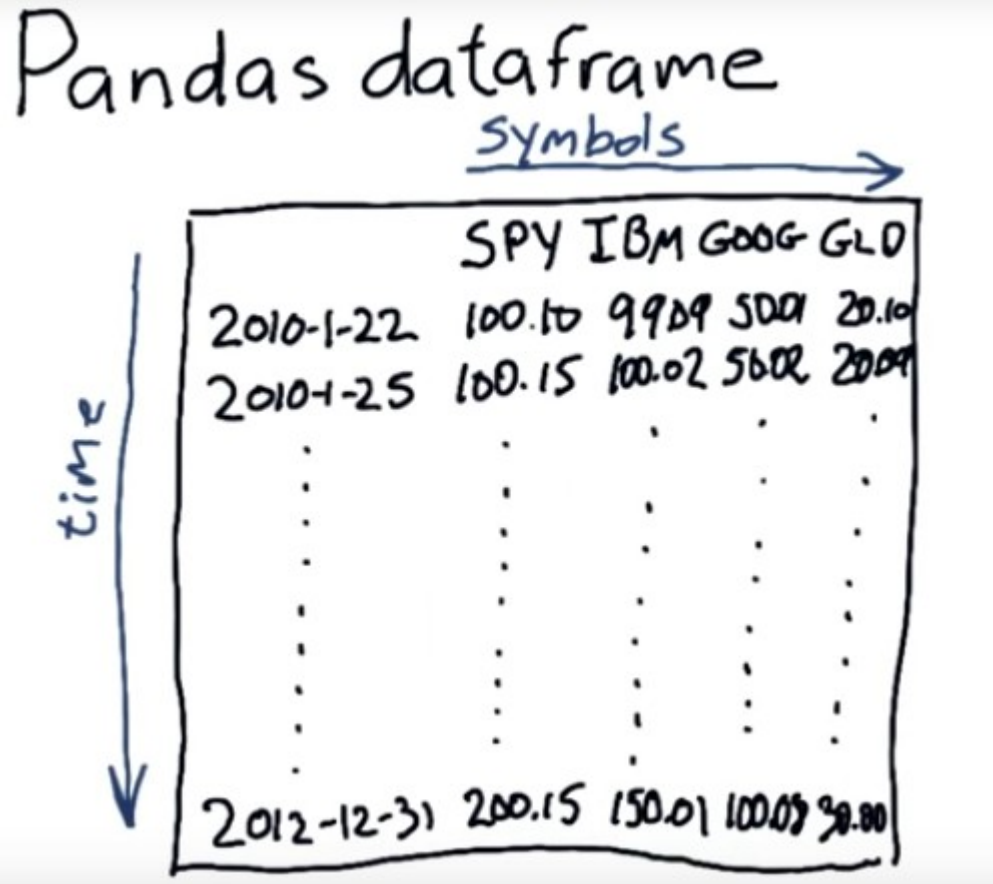
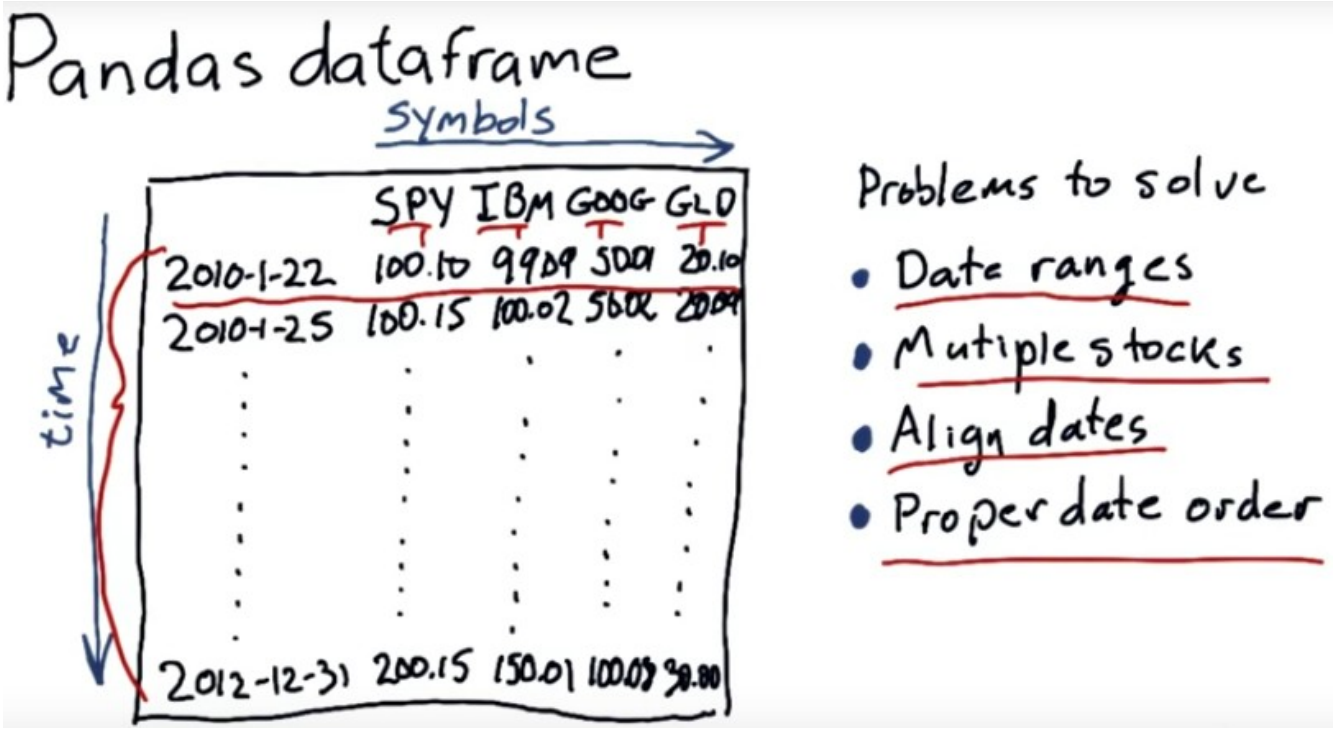
1. So far in this course we've been working with one stock at a time. >> Yes, professor. >> Dave, are you a juggler? >> No. >> Can you juggle lots of things at once? >> What? >> [Laughter] >> I can juggle two. >> Sorry, I can't do this. >> Allright. We're not jugglers.



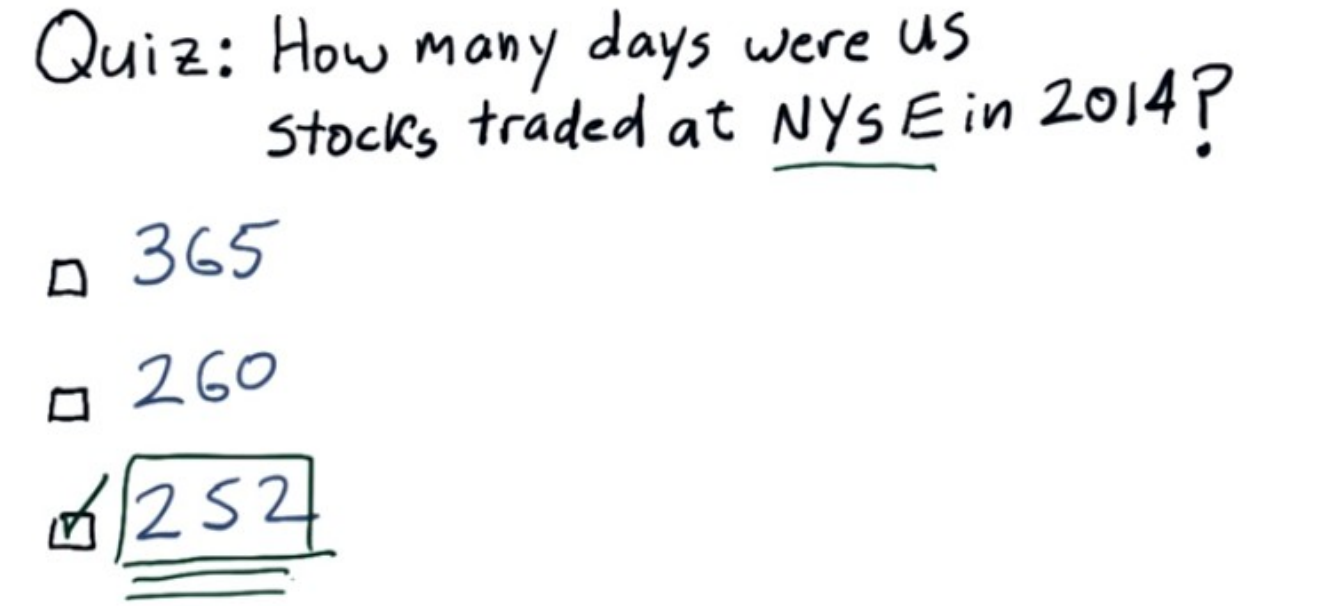
上圖的 dataframe 就是我們要得到的東西, 即目標.

2. Okay, in this lesson, we're going to dive into how do we fill a dataframe with the data that we want. And we're also going to touch upon a couple issue that we had that were pointed out in the previous lesson. For instance, remember when we had that issue that the dates were actually in the reverse order when we loaded them in, we're going to fix that. We're going to fix a couple other things too. Okay, just to refresh your memory, here's the kind of stuff that goes in a dataframe, our columns are the particular symbols of stocks that we're interested in, and our rows are dates. So time, goes from the top of the dataframe down to the bottom, and symbols, from left to right.

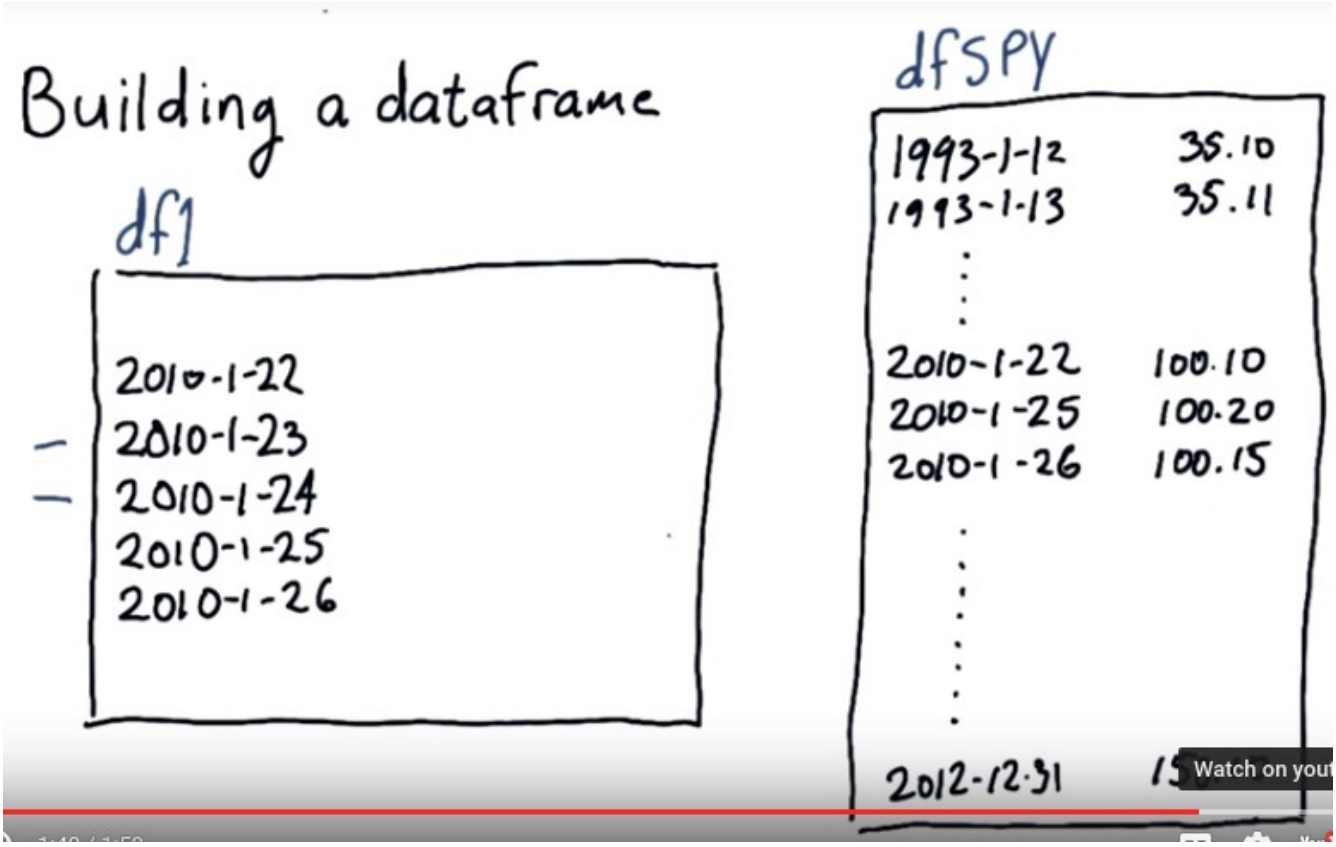


3. So in order to get to a dataframe like this, we've got a couple of problems to solve. And we're going to solve them in this lesson. So one of the first issues is we want to be able to read in particular date ranges. So if you remember from the last lesson, we just read all of it in. But what if we just want to read in a certain part of it, instead of from 1995 to 2012. What if we want, say, just 2010 to 2012? Well, we gotta figure out how to solve that. Something that's very special about Pandas, and one of the things that makes it very powerful, is that we can index the rows by dates. We don't have to just use a single number like zero for instance. We need to be able to read in multiple stocks, instead of just, say having one. We want to have SPY, IBM, Google, and Gold all at once. We need to be able to align dates. For instance, If GLD traded on particular days and SPY traded on particular days we want to make sure that those line up (即 SQL 中的 join), so that for each row we have the correct information for each equity on that particular date. Finally, we need to undo that problem that we discovered in the last lesson. Namely that these dates and the files we read them from are in reverse order. So, we want to be sure that we have them in the right order when we're processing them. So, these are the tasks we're going to dig into in this lesson.

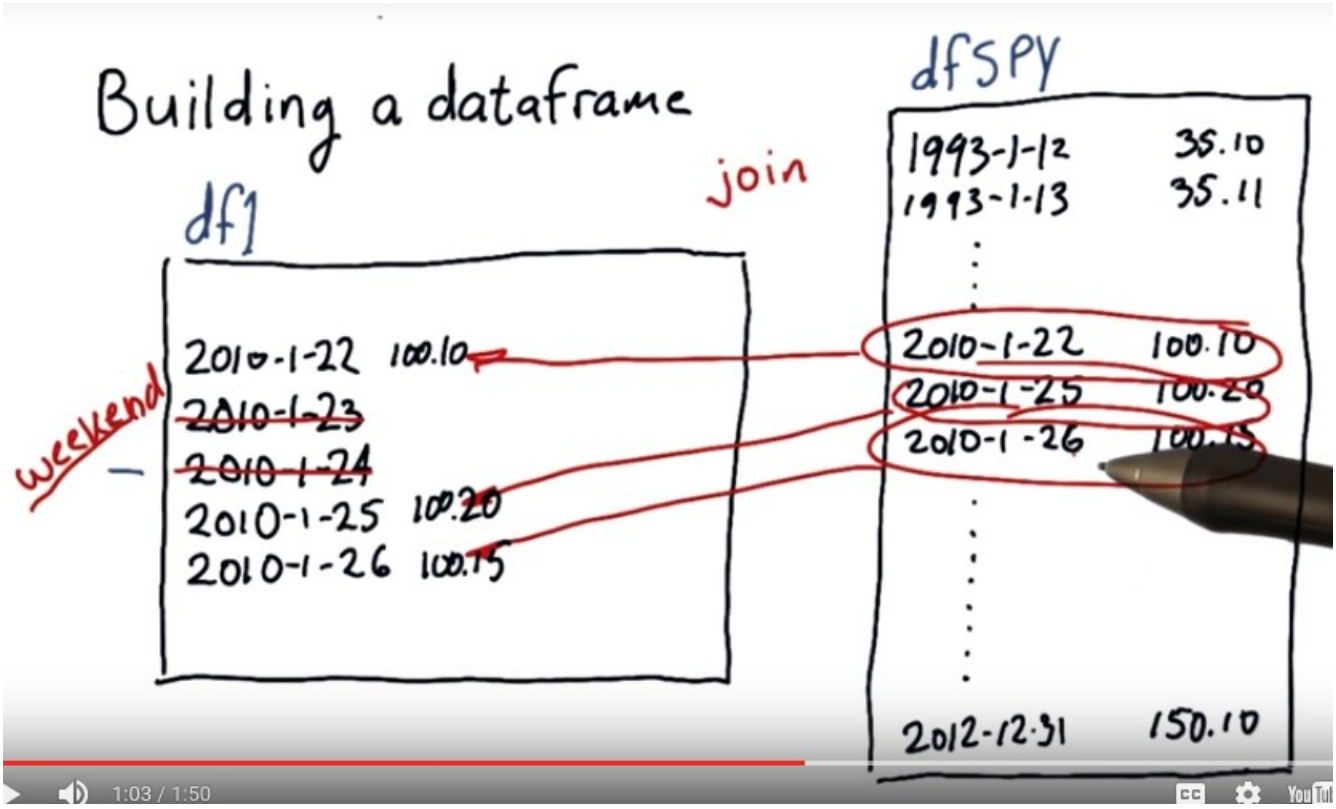
4. Here's a quiz, and something important to think about. How many days were US stocks traded at the New York Stock Exchange in 2014? Here are three possible answers.



5. The correct answer is 252. The reason is, you need to factor in weekends and holidays. And interestingly, the New York Stock Exchange has a set of holidays that occasionally differ from the US Government holidays. In almost all cases, almost every year, we've got 252 trading days and this number is going to come up a lot, as we continue to look at days and calculating statistics about stocks.

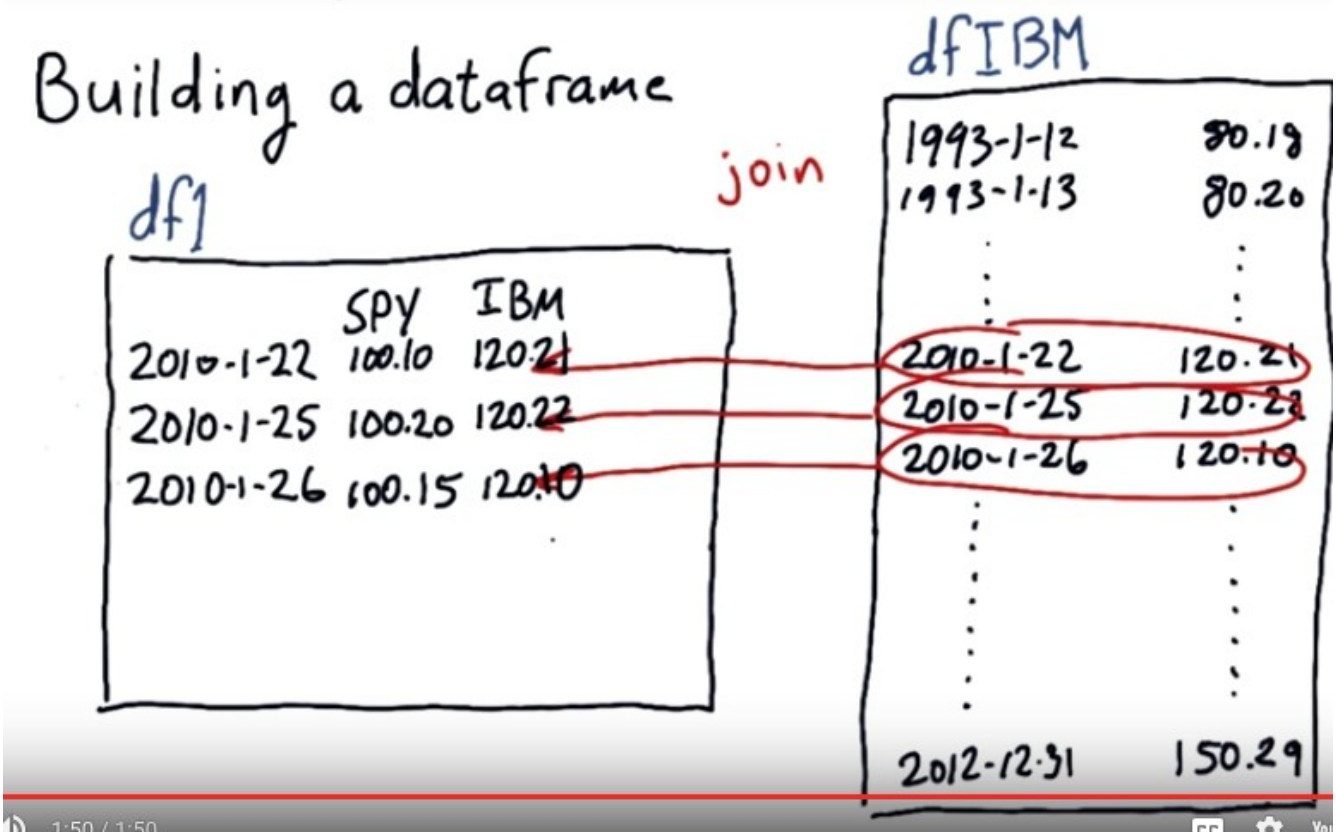


6. Now here's how we're going to build that data frame. Alright we start by constructing an empty data frame that has all the dates we're potentially interested in (即我們想到的那些 dates 組成一個單獨的 dataframe). And Dave will show you the syntax for doing that later on in the lesson. We'll call this df1 for data frame 1 of course. Now, we want to load this data frame up with a column of data for SPY, for IBM, for Google and Gold, and I'm going to show you step by step how we do that. Okay, so we separately read in SPY. And again, Dave will show you how to do that. Now when we read SPY, we get all the potential dates and all the prices that go with that. And in this case, we're loading adjusted closed data. Note that there's many more dates here (df1) than there are here (dfSPY). And this is our target data frame that we loaded with the particular dates that we're interested in. One other thing to mention, that I didn't mention before, is these two days (2010-1-23, 2010-1-24) are weekend days. So you can go ahead and check your calendar, go back to 2010, and see if I'm right. Interesting thing, or just obvious thing about weekends, is the markets are not open on weekends. So if you compare this with this, look in our SPY history, we don't have the 23rd and the 24th because SPY did not trade that day. So, we've got a little bit of mismatch there that we need to deal with. And this actually is one of the important reasons that we use SPY as a reference, because if SPY traded, it meant the market was open. And if the market was open, SPY traded. So, SPY, the S&P 500 ETF, is our reference for many, many things (SPY 即 500 強, 可以代表大多數公司, 故可以用它來做 reference).



7. So pandas has many very powerful features, among those features are the ability to do many operations that you may be familiar with from databases. In particular, one that we're going to leverage here is join. So we're going to join this data from SPY and our empty data frame. And we're going to do a special join that says, look, we're only interested in dates that are present both in SPY and df1. And what happens is, we end up with this data. According to its date and this position in our target dataframe, and, of course, the other two rows as well. But note, because of the join, these two days were missing from SPY and the result of the join is they are eliminated from our dataframe. They're eliminated because it's the weekend. There was no trading, because we know that because SPY is our

reference.



All right, so here's our original data frame now after we've loaded an SPY. And those weekend days that were here are gone because of that join. Now we can add more columns from other stocks, here's one for instance, IBM, by performing additional joins. So after this join with data frame IBM, bump, we've got this new data over here in our empty data frame. We can repeat this process for each additional symbol that we want to add. So we'll add Google and GLD. And again, Dave will show you the syntax for doing that and pandis in just a few moments.

```
1  ''' Build a dataframe in pandas'''
2  import pandas as pd
3
4  def test_run():
5      start_date='2010-01-22'
6      end_date='2010-01-26'
7      dates=pd.date_range(start_date,end_date)
8      print dates
9
10
11 if __name__ == "__main__":
12     test_run()
```

8. Let's try to build the data frame professor outline. Starting with the things we need to populate the data frame with are firstly dates. We used pandas date range method which takes two parameters, that is start and end date. For this code, we will take a small date range that is from 22nd Jan 2010 to 26th Jan 2010. We then call the date range function as I mentioned before, passing two parameters, start date and the end date. Let's run this code to see what variable dates has in it.

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2010-01-22, ..., 2010-01-26]
Length: 5, Freq: D, Timezone: None
```

The output you see is not the list of strings, but the list of date time index objects. Now, what do you mean by date time index object? Let's extract the first element of this list.

```
def test_run():
    start_date='2010-01-22'
    end_date='2010-01-26'
    dates=pd.date_range(start_date,end_date)
    print dates[0]
```

You can get the first element of the list by writing dates[0]. Let's go ahead and run this code.

```
2010-01-22 00:00:00
```

This is the first element of the list which a date/time index object. The trailing zero zeros for each object is the default time stamp. The index for a stock data only consists of dates. We can ignore the time stamps for now.

```
1  ''' Build a dataframe in pandas'''
2  import pandas as pd
3
4  def test_run():
5      start_date='2010-01-22'
6      end_date='2010-01-26'
7      dates=pd.date_range(start_date,end_date)
8      df1=pd.DataFrame(index=dates)
9
10
11
12 if __name__ == "__main__":
13     test_run()
```

Next we define an empty dataframe df1 with these dates as index. We use the parameter index to supply the dates. Note that without this parameter the dataframe will have an index of integers 0,1,2 as seen before. Let's print this now.

```
Empty DataFrame
Columns: []
Index: [2010-01-22 00:00:00, 2010-01-23 00:00:00, 2010-01-24 00:00:00, 2010-01
-25 00:00:00, 2010-01-26 00:00:00]
```

So here's your DataFrame, DF1. It's an empty DataFrame with no columns. However, as we pass the index parameter, we have an index as dates. And you can see that it's a date time index object. Two major steps have now been completed.

```
''' Build a dataframe in pandas'''
import pandas as pd

def test_run():
    #Define date range
    start_date='2010-01-22'
    end_date='2010-01-26'
    dates=pd.date_range(start_date,end_date)

    #Create an empty dataframe
    df1=pd.DataFrame(index=dates)

    #Read SPY data into temporary dataframe
    dfSPY = pd.read_csv("data/SPY.csv")

    #Join the two dataframes using DataFrame.join()
    df1=df1.join(dfSPY)
    print df1
```

9. Continuing on, let's read the csv file for SPY. In dfSPY, a temporary DataFrame. The next step is the heart of building the final DataFrame. We combine the empty DataFrame, df1, with the temporary DataFrame, dfSPY. We use the join function of the DataFrame for this purpose. Let's do it. DataFrame.join does a left join by default. So if we write a.join b, it will read in all the rows from a, but only those rows from b whose index values are present in a's index. For the remaining rows, that is the index values, not present in b, pandas introduce nans. So in this case, all the rules from the df1 will be retained and we will get all the values for the prizes from dfSPY for the given range we defined above. So in our case, all the rows from the df1 will be retained and only those rows of dfSPY, which is present in df1, will be retained. This will give us all the prices for the stock SPY in the defined date range. Let's use join df1 one after the join step to make it clear. You should expect to see all the values of SPY for the given dates.

	Date	Open	High	Low	Close	Volume	Adj Close
2010-01-22 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-23 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-24 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-25 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-26 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Observe the output. We did not get any values from the dfSpy. What do you think? What could be the

reason?

```
''' Build a dataframe in pandas'''
import pandas as pd

def test_run():
    #Define date range
    start_date='2010-01-22'
    end_date='2010-01-26'
    dates=pd.date_range(start_date,end_date)

    #Create an empty dataframe
    df1=pd.DataFrame(index=dates)

    #Read SPY data into temporary dataframe
    dfSPY = pd.read_csv("data/SPY.csv")
    print dfSPY

    #Join the two dataframes using DataFrame.join()
    #df1=df1.join(dfSPY)
    #print df1
```

Let's print dfSPY to investigate. Commenting out the join lines. So let's see what dfSPY has.

	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
5	2012-09-05	141.09	141.47	140.63	140.91	100660300	140.91

We told pandas to join df1, and dfSPY. But dfSPY has an index of integers, which is not the same as the dates index that df1 uses.

```
def test_run():
    #Define date range
    start_date='2010-01-22'
    end_date='2010-01-26'
    dates=pd.date_range(start_date,end_date)

    #Create an empty dataframe
    df1=pd.DataFrame(index=dates)

    #Read SPY data into temporary dataframe
    dfSPY = pd.read_csv("data/SPY.csv",index_col="Date",parse_dates=True)
    print dfSPY

    #Join the two dataframes using DataFrame.join()
    #df1=df1.join(dfSPY)
    #print df1
```

We fix this by informing the read_csv function, that the date column in the csv file should be used as index. We do this by using the index_col parameter. Make sure you give the correct column name. We also want the dates present in the DataFrame to be converted into date time index objects. This can be done by setting the value for the parse_dates parameter to True. Let's see if this works.

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

Notice that the Date column is now being used as the index. There is no separate integer index on the left.

```
''' Build a dataframe in pandas'''
import pandas as pd

def test_run():
    #Define date range
    start_date='2010-01-22'
    end_date='2010-01-26'
    dates=pd.date_range(start_date,end_date)

    #Create an empty dataframe
    df1=pd.DataFrame(index=dates)

    #Read SPY data into temporary dataframe
    dfSPY = pd.read_csv("data/SPY.csv",index_col="Date",
                        parse_dates=True,usecols=['Date','Adj Close'],
                        na_values=['nan'])

    #Join the two dataframes using DataFrame.join()
    df1=df1.join(dfSPY)
    print df1
```

Now let's see what the resulting join DataFrame looks like by uncommenting these lines. But before that, let's add some more parameters to the read_csv function. Note that we are interested in just two columns, which is the Date and the Adj Close. We can get rid of the other column, by using the usecols parameter of the read_csv. We pass a list of column name we are interested in, which are Date and Adj Close.

	Adj Close
2010-01-22	104.34
2010-01-23	NaN
2010-01-24	NaN
2010-01-25	104.87
2010-01-26	104.43

Now let's see what's there in df1. You see now we have just the Adj Close for the SPY for a given date range. Also observe that weekend dates have NaN value. Before we go ahead, let's understand that csv NaN as string, so we need to tell the read_csv that NaN should be interpreted as not a number. This is how you indicate it.

```
def test_run():
    #Define date range
    start_date='2010-01-22'
    end_date='2010-01-26'
    dates=pd.date_range(start_date,end_date)

    #Create an empty dataframe
    df1=pd.DataFrame(index=dates)

    #Read SPY data into temporary dataframe
    dfSPY = pd.read_csv("data/SPY.csv",index_col="Date",
                        parse_dates=True,usecols=['Date','Adj Close'],
                        na_values=['nan'])

    #Join the two dataframes using DataFrame.join()
    df1=df1.join(dfSPY)

    # Drop NaN Values
    df1 = df1.dropna()
```

3:55 / 4:10

One last step. We just want the date, on which SPY traded, so we can add more stocks based on these dates. Let's drop the rows where SPY is NaN. For this, we use the dropna function. df1.dropna will drop all the rows which has NaN values for the SPY. Let's go ahead and print this.

	Adj Close
2010-01-22	104.34
2010-01-25	104.87
2010-01-26	104.43

Now we have built a clean DataFrame filled with SPY data using a selected date range and keeping only the dates that SPY traded on.

10. Note that we use two steps for combining the data frames. That is left joining the empty data frame with dfSPY, and then dropping the rows that SPY did not trade on. This can be done in a single step using the how argument when calling join. Can you figure out what the appropriate value of how

should be? Type it in the box.

11. We are essentially trying to do an inner join, that is, we only want to retain rows common to both dataframes. Note that this is not the default, hence, we have to mention it explicitly. So what is the default value? What effect does it have on the result? Find out using the documentation link provided in the instructor notes.

```
11      df1=pd.DataFrame(index=dates)
12
13      #Read SPY data into temporary dataframe
14      dfSPY = pd.read_csv("data/SPY.csv",index_col="Date",
15                          parse_dates=True,usecols=['Date','Adj Close'],
16                          na_values=['nan'])
17
18      #Join the two dataframes using DataFrame.join(), with how='inner'
19      df1=df1.join(dfSPY,how='inner')
20
21      #Read in more stocks
22      symbols = ['GOOG','IBM','GLD']
23      for symbol in symbols:
24          df_temp=pd.read_csv("data/{}.csv".format(symbol), index_col='Date',
25                              parse_dates=True,usecols=['Date','Adj Close']
26                                                      ,na_values=['nan'])
27          df=df1.join(df_temp) #use default how='left'
28
29      print df1
```

12. So we want to read in more stocks into a combined dataframe. Start with the code we used to build our dataframe with the SPY data. Then define a list with the required stock symbols. Now we can write a for loop to read and join each stock into the dataframe just like SPY. So here's the for loop which takes each symbol in the symbols list and joins it to our main dataframe. Let's go ahead and print this.

```
columns overlap but no suffix specified: Index([u'Adj Close'], dtype='object')
```

Oops. There is an error. Reading the error message carefully, we observe that index column Adj Close has an overlap. What is happening here is that, irrespective of the stock the column we are extracting each time is named Adj Close. So the join method is confused as to what to name it in the result. Column names must be unique.

```
13    #Read SPY data into temporary dataframe
14    dfSPY = pd.read_csv("data/SPY.csv",index_col="Date",
15                        parse_dates=True,usecols=['Date','Adj Close'],
16                        na_values=['nan'])
17
18    #Rename 'Adj Close' column to 'SPY' to prevent clash
19    dfSPY = dfSPY.rename(columns={'Adj Close':'SPY'})
20
21    #Join the two dataframes using DataFrame.join(), with how='inner'
22    df1=df1.join(dfSPY,how='inner')
23
24    #Read in more stocks
25    symbols = ['GOOG','IBM','GLD']
26    for symbol in symbols:
27        df_temp=pd.read_csv("data/{}.csv".format(symbol), index_col='Date',
28                            parse_dates=True,usecols=['Date','Adj Close']
29                                                    ,na_values=['nan'])
30         # rename to prevent clash
31        df_temp = df_temp.rename(columns={'Adj Close': symbol})
32        df1=df1.join(df_temp) #use default how='left'
```

As professor described earlier, we would like each stock symbol as the corresponding column name or header. So we add these two lines. This renames the column Adj Close to the respective stock symbol.

	SPY	GOOG	IBM	GLD
2010-01-26	104.43	542.42	119.85	107.56
2010-01-25	104.87	540.00	120.20	107.48
2010-01-22	104.34	550.01	119.61	107.17

Now let's see the output. Here you go, everything is finally in place.

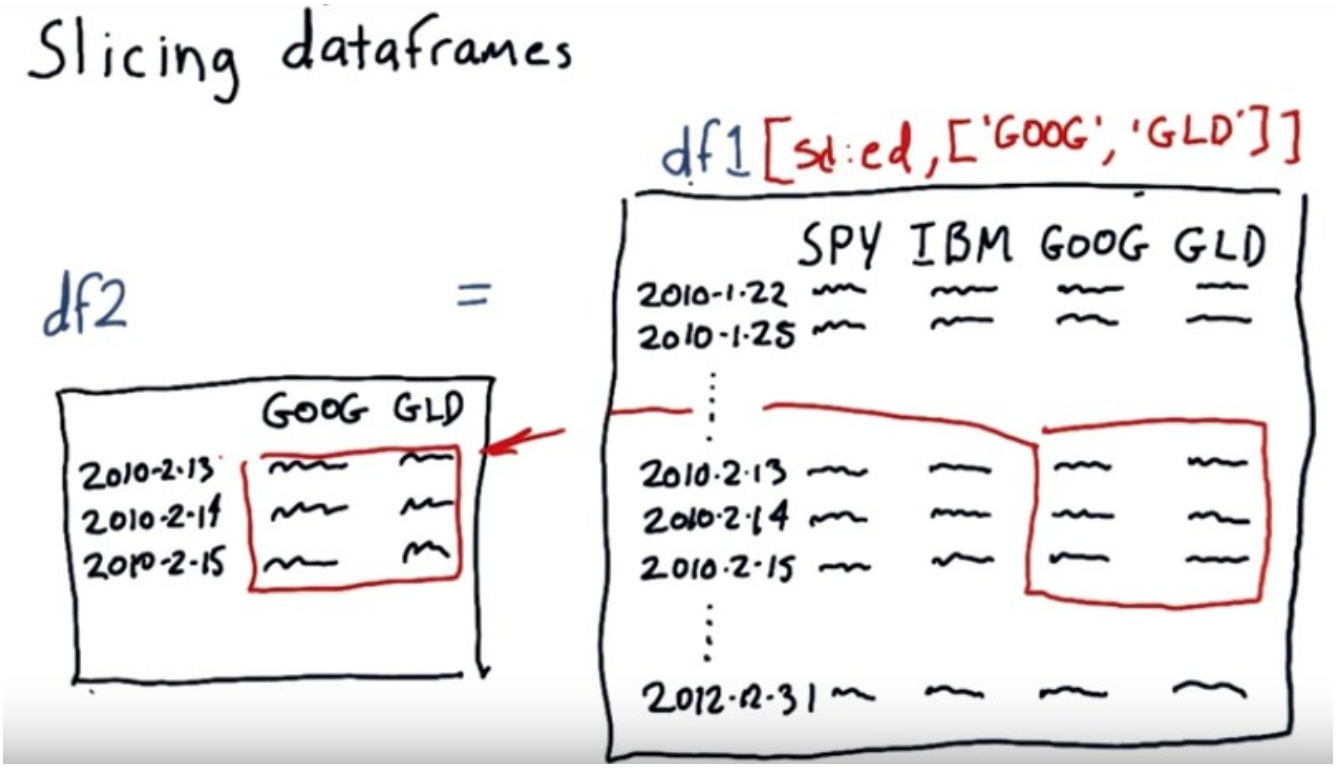
13. You must have noticed how we are carrying out essentially the same operation in different places. Why not write some utility function that we can use going forward? I have implemented one function for you, symbol_to_path. It accepts a symbol name as a string and returns the path to the corresponding CSV file, assuming it is stored under data by default. For example, symbol_to_path(IBM) will return data/IBM.csv. Can you finish the implementation for get_data? It take a list of symbols and dates as index and is supposed to return a data frame with stock data for each symbol within the given date range. SPY is inserted into the list, if not already present, in order to solve as a reference. Note that you must ensure the column for SPY does not have any nulls. That is, the data frame should only contain dates when SPY actually traded in the given date range. Type in your code here. You can use test run to execute your code and submit to evaluate it against our test cases. Don't worry, there is no limit to how many times you can try. Good luck.

```
6  def symbol_to_path(symbol, base_dir="data"):
7      """Return CSV file path given ticker symbol."""
8      return os.path.join(base_dir, "{}.csv".format(str(symbol)))
9
10
11 def get_data(symbols, dates):
12     """Read stock data (adjusted close) for given symbols from CSV files."""
13     df = pd.DataFrame(index=dates)
14     if 'SPY' not in symbols:  # add SPY for reference, if absent
15         symbols.insert(0, 'SPY')
16
17     for symbol in symbols:
18         df_temp = pd.read_csv(symbol_to_path(symbol), index_col='Date',
19                 parse_dates=True, usecols=['Date', 'Adj Close'], na_values=['nan'])
20         df_temp = df_temp.rename(columns = {'Adj Close': symbol})
21         df = df.join(df_temp)
22         if symbol == 'SPY': #drop dates SPY did not trade
23             df = df.dropna(subset=["SPY"])
```

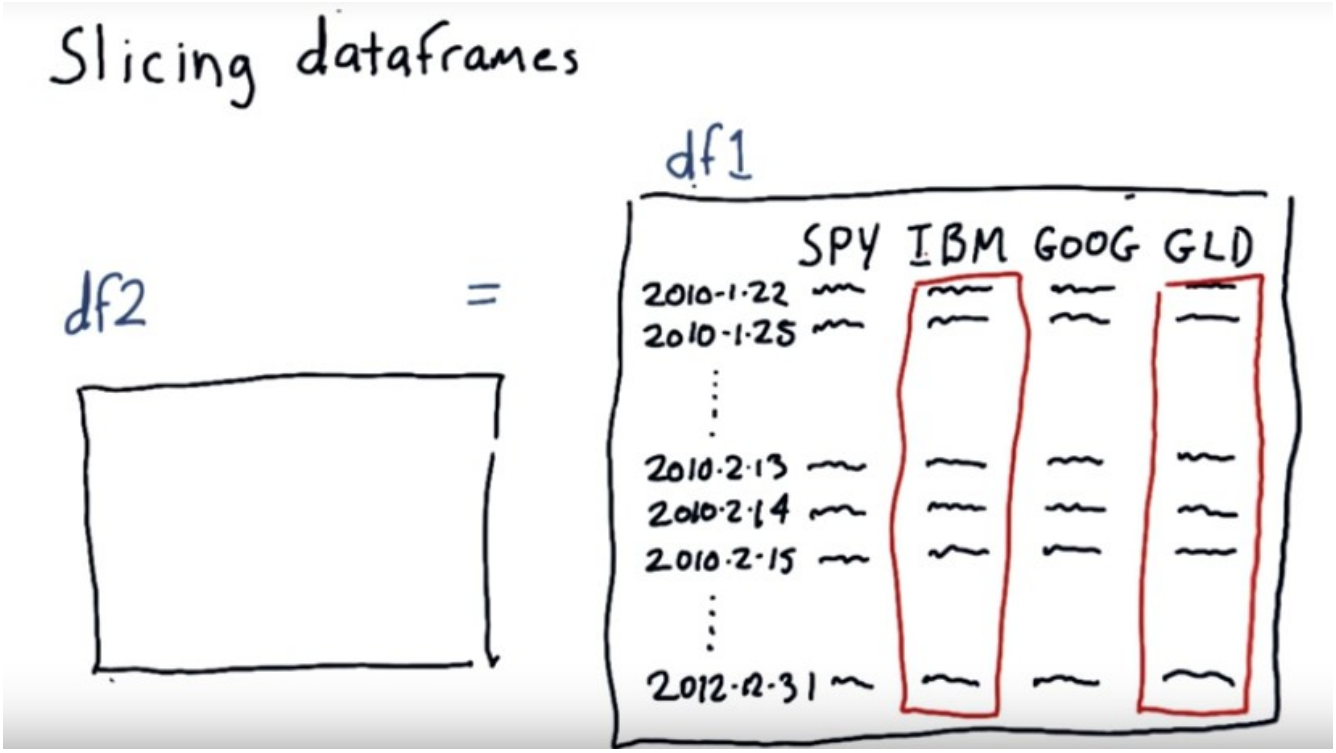
14. As show in the previous demo, there are three main steps to implement inside the for loop. The first one is to read in the data from the symbol. Make sure you specify all the parameters. Also notice how I have used symbol_to_path function to get the path to the CSV file. The next step is to rename the adjacent close column to the symbol name. And the last step is to join this new data with the new data frame. Now, we have to take care of one important thing. That is, dropping off the lines from the SPY. Subset is equal to SPY will ensure that only those rows will be dropped where SPY is none. Also the statement ensures that SPY is used as a reference. And that we do not have any non-values in the SPY column.

	SPY	GOOG	IBM	GLD
2010-01-22	104.34	550.01	119.61	107.17
2010-01-25	104.87	540.00	120.20	107.48
2010-01-26	104.43	542.42	119.85	107.56

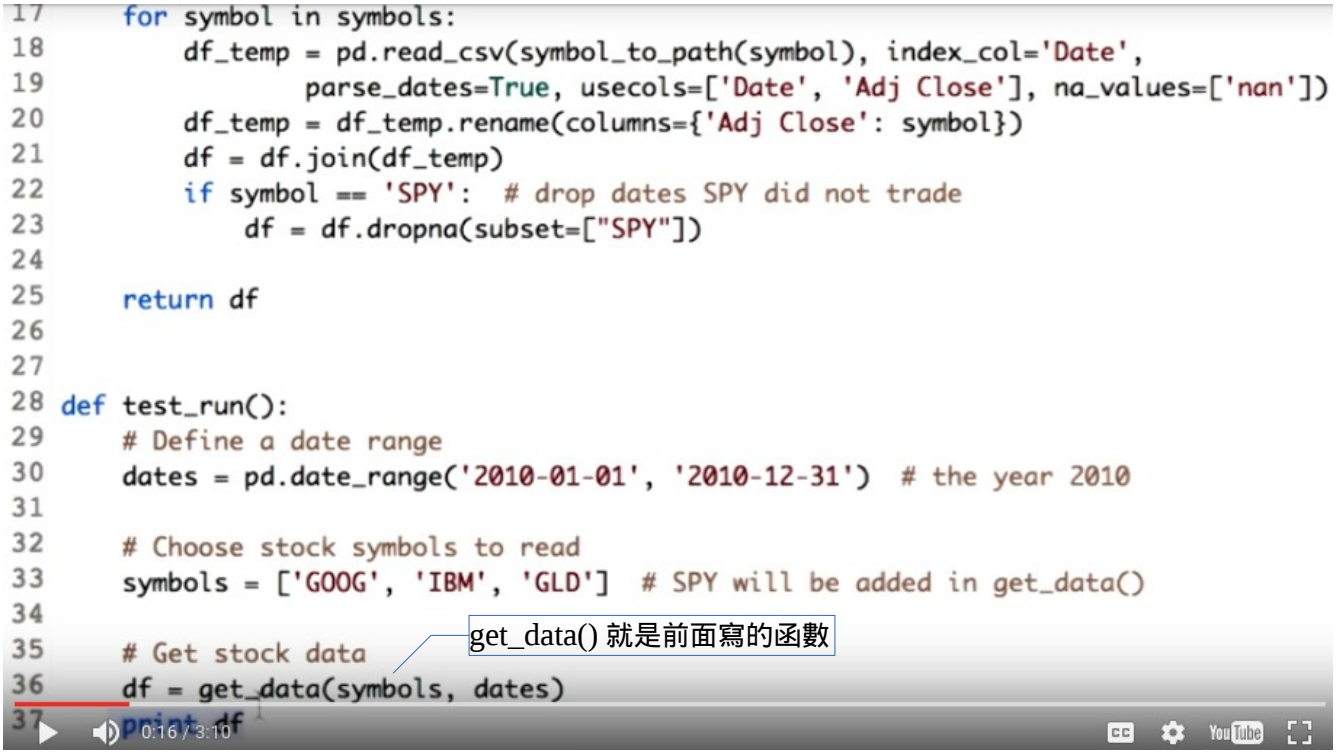
Let's run it. So here's the output, same as before.



15. Okay, let's suppose we have a nice big beautiful pandas dataframe and this time read in a lot of data. We didn't just focus on a few days, we've got the data all the way from the beginning of 2010 to 2012 and we got data for SPY, IBM, Google and GLD. Now suppose we want to focus on just a subset of that data. In fact, we might call that a slice. For instance, what if we wanted to look at just the values for Google and GLD between these dates, February 13, 2010 through February 15, 2010, and we want to again just look at Google and GLD. Well there's very beautiful syntax in Pandas that allows you to do that. So we're going to be learning a lot more about this syntax in a later lesson. But this is sort of a preview of the very nice things you can do. To select these rows we do a simple statement that they will show you later to create a date time object. And we just put the start date colon end date. And if we just write df1[sd:ed] where this is start date and that's end date, then we end up With these three rows, but we want to be more selective. We want to focus on these three rows and these columns and we need to add one more piece of syntax that indicates these columns. So this statement will extract these rows and these columns, and leave you just with this sub-portion or slice of df1. So, if we were to execute the statement df2 equals df1, and then this additional syntax, we will end up with this little morsel of data right here in our new dataframe.



Now there's lots of different ways you can slice the data, you don't have to take a group of pieces of data that are right next to each other. You can grab any different columns you want and any different rows you want. So you might build a new dataframe by taking GLD and IBM. And it'll just take those two columns and splat them into df2. So that's slicing. This is just a brief introduction. We're going to go into some deeper examples when we get to the lesson on numbpie.



16. To do any kind of analysis on the data, we need significant amount of data. So let's read data for each stock for a period of one year, that is, the year 2010. You can do this by changing the stock and the end date as shown here.

	SPY	GOOG	IBM	GLD
2010-01-04	108.27	626.75	126.24	109.80
2010-01-05	108.56	623.99	124.71	109.70
2010-01-06	108.64	608.26	123.90	111.51
2010-01-07	109.10	594.10	123.47	110.82
2010-01-08	109.46	602.02	124.71	111.37
2010-01-11	109.61	601.11	123.41	112.85
2010-01-12	108.59	590.48	124.39	110.49

Let's see the data frame contains now. So this is your data frame which has stock prices for the symbol SPY, Google, IBM and gold for a year 2010.

```
28 def test_run():
29     # Define a date range
30     dates = pd.date_range('2010-01-01', '2010-12-31')  # the year 2010
31
32     # Choose stock symbols to read
33     symbols = ['GOOG', 'IBM', 'GLD']  # SPY will be added in get_data()
34
35     # Get stock data
36     df = get_data(symbols, dates)
37
38     # Slice by row range (dates) using DataFrame.ix[] selector
39     print df.ix['2010-01-01':'2010-01-31']  # the month of January
```

We briefly explained slicing in the last lesson. In this lesson we will learn how to do slicing using the data frame we just created. There are basically three ways we can slice the data frame. First, row slicing. As the name suggest, it will give us the required rows along with all the columns. This is useful when you want to compare moment of all the stocks over subset of time. This is how you do it in the code. We use the function .ix of the data frame and just mention the start and the end date in the square brackets. Here we extract the moment of all the stocks in the month of Jan. Note that the start and the end date should be in the chronological order. If you write the 31st Jan date before 1st Jan, the date frame will give you an empty data frame. Even if you remove the .ix function and just to print DF passing the dates in the chronological order, you will get the same result. However, .ix is considered to be more Pythonic and robust, so we follow that.

	SPY	GOOG	IBM	GLD
2010-01-04	108.27	626.75	126.24	109.80
2010-01-05	108.56	623.99	124.71	109.70
2010-01-06	108.64	608.26	123.90	111.51
2010-01-07	109.10	594.10	123.47	110.82

Now let's run the code to see the stock prices for the month of January. You can observe that we get the data only for the January month but for all the symbols, this is known as row slicing.

```
29      # Define a date range
30      dates = pd.date_range('2010-01-01', '2010-12-31')  # the year 2010
31
32      # Choose stock symbols to read
33      symbols = ['GOOG', 'IBM', 'GLD']  # SPY will be added in get_data()
34
35      # Get stock data
36      df = get_data(symbols, dates)
37
38      # Slice by row range (dates) using DataFrame.ix[] selector
39      print df.ix['2010-01-01':'2010-01-31']  # the month of January
40
41      # Slice by column (symbols)
42      print df['GOOG']  # a single label selects a single column
43      print df[['IBM', 'GLD']]  # a list of labels selects multiple columns
```

Second way of slicing is useful when you want to view prices of only one stock over the date range, in this case you can use column slicing. We want to project the prices of Google for the entire year of 2010. Here is how we do it. A square bracket along with the name of the column and do not forget the colon. To retrieve a single column we just pass a single label. To select multiple columns we pass a list of labels.

	IBM	GLD
2010-01-04	126.24	109.80
2010-01-05	124.71	109.70
2010-01-06	123.90	111.51
2010-01-07	123.47	110.82

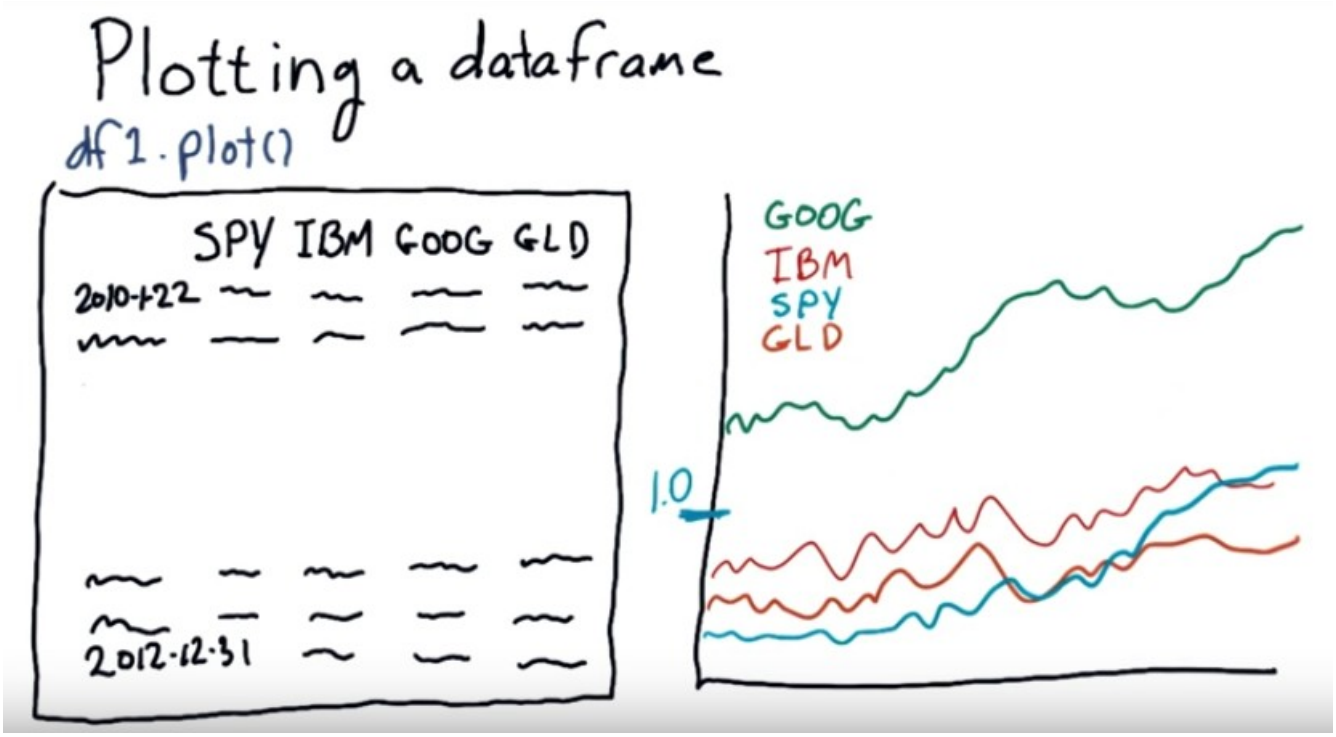
Let's print this. And this is the output for multiple columns.

```
29      # Define a date range
30      dates = pd.date_range('2010-01-01', '2010-12-31')  # the year 2010
31
32      # Choose stock symbols to read
33      symbols = ['GOOG', 'IBM', 'GLD']  # SPY will be added in get_data()
34
35      # Get stock data
36      df = get_data(symbols, dates)
37
38      # Slice by row range (dates) using DataFrame.ix[] selector
39      #print df.ix['2010-01-01':'2010-01-31']  # the month of January
40
41      # Slice by column (symbols)
42      #print df['GOOG']  # a single label selects a single column
43      #print df[['IBM', 'GLD']]  # a list of labels selects multiple columns
44
45       # Slice by row and column
46      print df.ix['2010-03-10':'2010-03-15', ['SPY', 'IBM']]
```

The last way of slicing is to slice through both dimensions, that is rows and columns. The most robust way to do this is using the IX selector of the data frame. Let's go ahead and use it. If you need more than one column, you define them in a list like this, and date range are separated with a colon.

	SPY	IBM
2010-03-10	109.84	120.26
2010-03-11	110.30	122.16
2010-03-12	110.31	122.48
2010-03-15	110.34	122.38

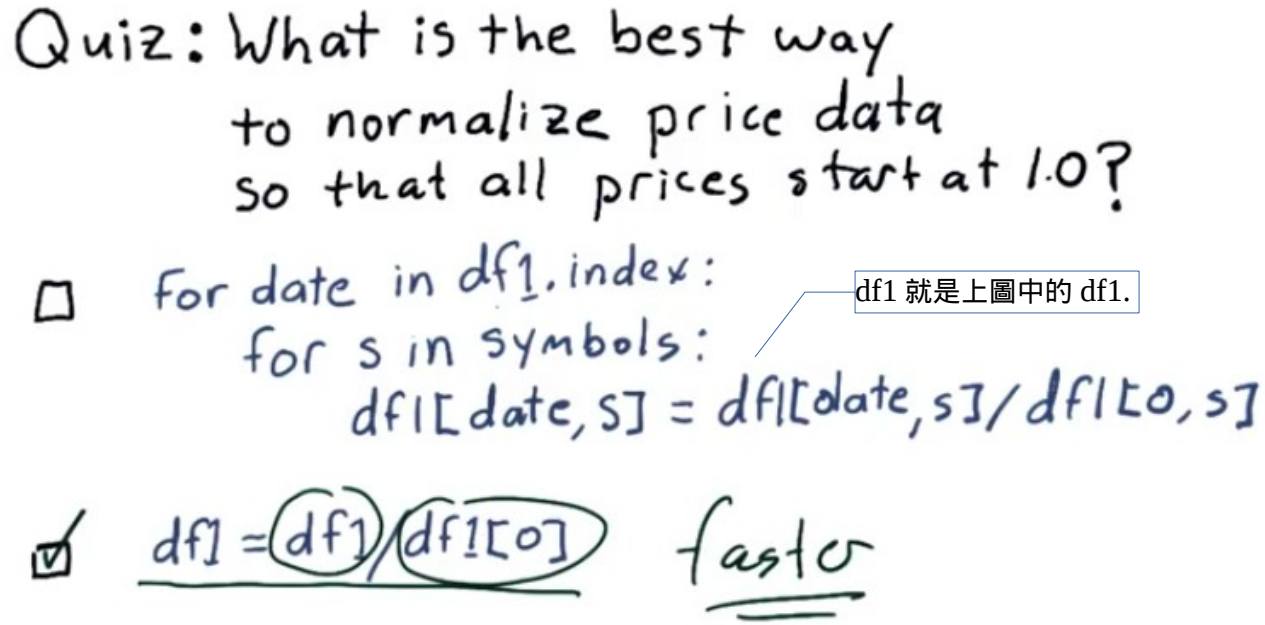
Here you go, the stock prices for the symbol SPY and IBM over the date range 10th March to 15th March. One application of this way of slicing is to compare multiple stocks over a period of time. Panda support many ways of slicing a data frame to suit different needs. Find out more using the link in the instructor notes.



17. Suppose you've got a data frame. It's pretty easy from the data frame to make a plot. In fact the syntax is as simple as this. Pandas will take this data and create a nice chart that looks about like this. It will give an individual color to each time series and give you a nice legend telling you which is which. Now one problem with viewing. Data like this is for instance, at this time, by the way, these numbers are made up. But it's often the case that stocks are priced at significantly different levels, so in this example, say, Google had a very high price and the other stocks had low prices, and it's hard to look at them sort of In a good way comparatively when they have these widely variant prices. So what we'd like to do instead is be able say to have them all start at one single point here, say that's 1.0. And then go out from there so we can compare them on an apples to apples comparison.

So we'd like to end up with a chart that looks like this, where everything starts at 1.0 (意思見下面的 quiz 就明白了), and we can compare them on an equal basis going forward.

18. Okay, let's try a quick quiz. What is the best way to normalize price data so that all prices start at 1.0? So, here are two choices: a nested for loop or a one line expression. Good luck.



19. Well, it was a little bit of a trick question because both of these will accomplish the same goal, namely normalizing everything according to the first row. This method uses two nested four loops to go through each date for each symbol and then make the division. This(第二個) is the preferred method however, and that's because it's elegant, just a single line. We divide the entire data frame by its first row. The other reason that this is the way to do it is because it's much much faster. This ends up being executed in C at lower levels. Whereas this will be executed at the higher level, the higher interpreter levels of Pandas and Python

```
18      for symbol in symbols:
19          df_temp = pd.read_csv(symbol_to_path(symbol), index_col='Date',
20                  parse_dates=True, usecols=['Date', 'Adj Close'], na_values=['nan'])
21          df_temp = df_temp.rename(columns={'Adj Close': symbol})
22          df = df.join(df_temp)
23          if symbol == 'SPY':  # drop dates SPY did not trade
24              df = df.dropna(subset=["SPY"])
25
26      return df
27
28  def plot_data(df):
29      '''Plot stock prices'''
30      df.plot()
31      plt.show() #must be called to show plots in some environments
```

20. Carly Fiorina rightly states that the goal is to turn data into information and information into insight. So let's better understand the data by improving our plots. We will need the matplotlib library for this purpose. Specifically we import the matplotlib.pyplot and rename it as plt for ease of use. First, let's define a trivial plot function. We define a plot data function which will essentially plot the contents of the data frame. We pass the data frame to it and use its plot function to plot all the data in it. Let's see the output. Remember, to see the graph, we have to call the show function from the matplotlib library. Observe that x axis has dates and y axis has prices for all the four stocks. The graph still looks incomplete. We need to add details like give a name or title to the graph, add x and y axis labels. We will give name to the graph by passing in parameter title.

```
18      for symbol in symbols:
19          df_temp = pd.read_csv(symbol_to_path(symbol), index_col='Date',
20                  parse_dates=True, usecols=['Date', 'Adj Close'], na_values=['nan'])
21          df_temp = df_temp.rename(columns={'Adj Close': symbol})
22          df = df.join(df_temp)
23          if symbol == 'SPY':  # drop dates SPY did not trade
24              df = df.dropna(subset=["SPY"])
25
26      return df
27
28  def plot_data(df,title="Stock prices"):
29      '''Plot stock prices'''
30      ax = df.plot(title=title,fontsize=2)
31      ax.set_xlabel("Date")
32      ax.set_ylabel("Price")
33      plt.show() #must be called to show plots in some environments
```

This should be customizable. And here we pass the value of the title to the parameter title of the dataframe. For x and y axis labels, we need a handler to the plot which the dataframe generates. The output of df.plot is such a handler. You can imagine it as an object. We name it ax for axis. Now we call set x label and set y label on this object to give the x and y axis some meaningful labels. The desired labels are passed as a parameter to the function set_xlabel and set_ylabel. Remember that set_xlabel and set_ylabel are the function of the object that we got from df.plot. You can also use the fontsize parameter in df.plot to make the graph more readable.

Now let's see the modified plot. So here is your detailed plot with x label, y label, and a title.

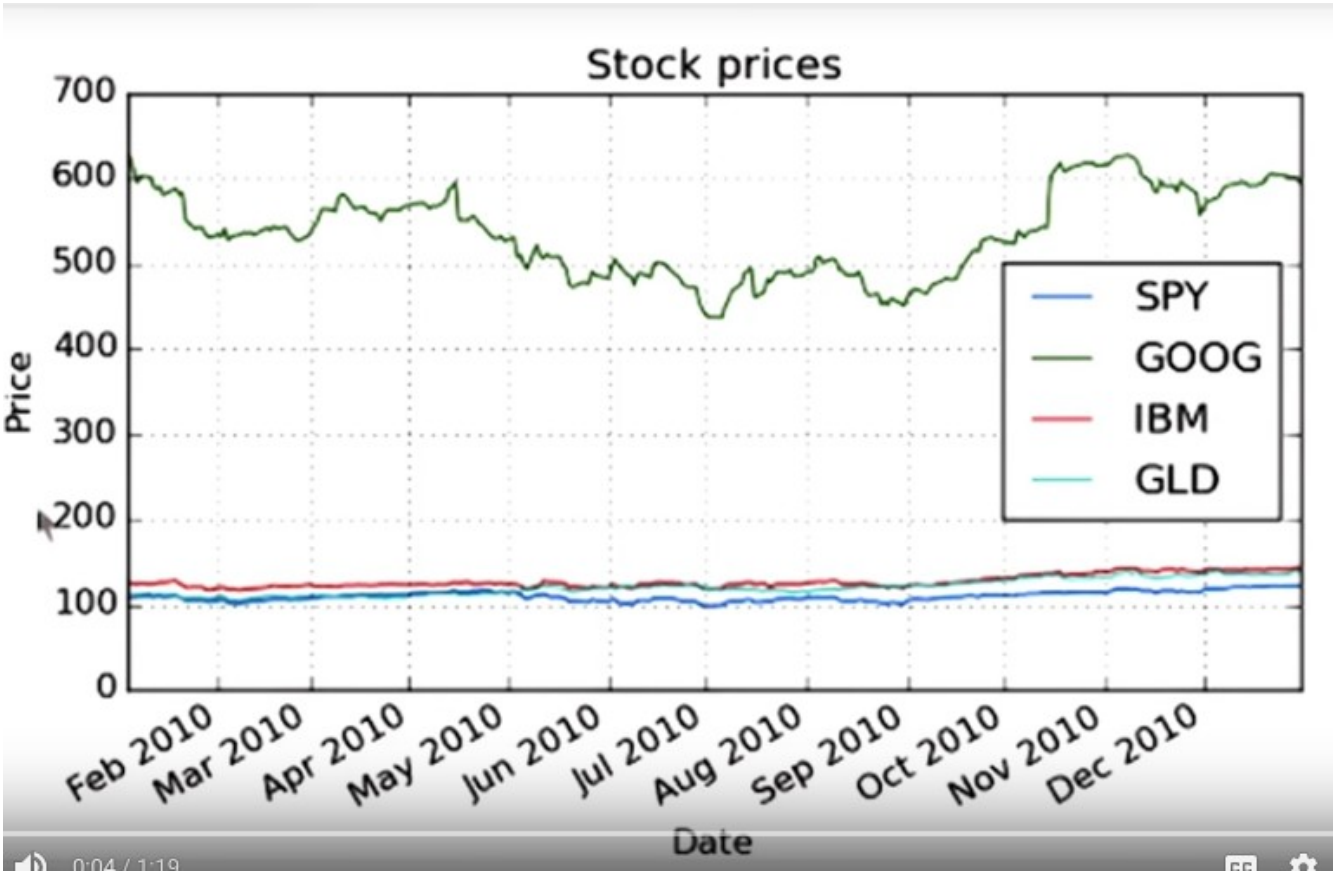
21. You now know how to read stock data, slice it, and plot it. The challenge for you now is to write the code to plot the values of SPY and IBM over the date range. Go ahead and write your code in this function. The clue to this is use slicing.

```
4  import pandas as pd
5  import matplotlib.pyplot as plt
6
7
8  def plot_selected(df, columns, start_index, end_index):
9      """Plot the desired columns over index values in the given range."""
10     plot_data(df.ix[start_index:end_index,columns],title="Selected data")
11
12
13 def symbol_to_path(symbol, base_dir="data"):
14     """Return CSV file path given ticker symbol."""
15     return os.path.join(base_dir, "{}.csv".format(str(symbol)))
16
17
18 def get_data(symbols, dates):
19     """Read stock data (adjusted close) for given symbols from CSV files."""
20     df = pd.DataFrame(index=dates)
21     if 'SPY' not in symbols:  # add SPY for reference, if absent
22         symbols.insert(0, 'SPY')
```

22. To complete this, four slides using the notation you learned earlier. Once you get the desired data set, the next step is to plot by calling the plot and the score data function we defined earlier. Recollect that, plot_data takes the parameter, data frame and title.

Let's see what we get now. Here's the plot showing the changes in the stock prices for SPY and IBM

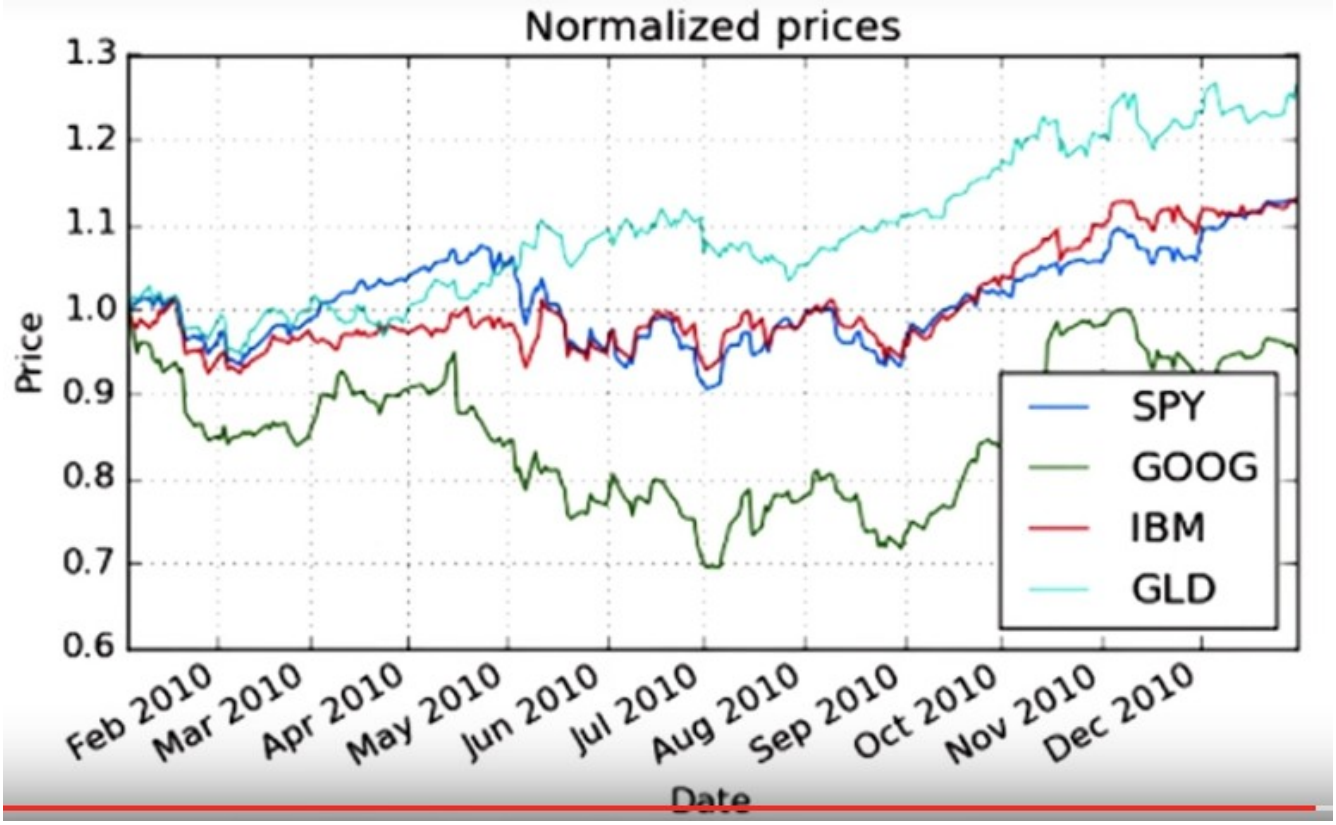
for the period of March.



23. Let's analyze the graph that we had plotted for the four stocks. You see the four stocks are all multiple ranges. But we need to observe the movement of the stock. By movement I mean how much the stock went up or down as compared to the others.

```
24          if symbol == 'SPY':  # drop dates SPY did not trade
25              df = df.dropna(subset=["SPY"])
26
27      return df
28
29
30  def normalize_data(df):
31      """Normalize stock prices using the first row of the dataframe."""
32      return df/ df.ix[0,:]
33
34
35  def plot_data(df, title="Stock prices"):
36      """Plot stock prices with a custom title and meaningful axis labels."""
37      ax = df.plot(title=title, fontsize=12)
38      ax.set_xlabel("Date")
39      ax.set_ylabel("Price")
40      plt.show()
```

To do this, we need to normalize the prices of all the stock. We do this by dividing the values of each column. By day one. This will ensure that all the stocks will start with $1. Power of pandas and Python is that we can do this in one line. Let's add it. We define the function: normalize data and pass the data frame through it. First, we want all the values of the data frame. Hence, we just type the name of the data frame, which is df. Now we want to divide all the values of this data frame by the first row of eight. So we extract the first row using row slicing. This(df.ix[0,:]) will give us the first row. Now we will just divide it.



Let's see how the graph changes. Observe, all the stocks start with price one. And now you can see the changes that is the stock movement. That's it for now, I will be back soon with more coding. Until then, enjoy the power of the Python and happy coding.