MIRC Sector Momentum Sample Algo (Part 1)

This notebook was created as a demonstration for **McMaster's MIRC** quant club. For this project, we hypothesize that a **company's earning reports** contains information about the **performance of rest of the sector**, and will attempt to **exploit any alpha** if such a relationship exists.

As an example, if we believe that Apple will post good earnings numbers (earnings per share, growth in revenue, etc.) then other tech companies also will post good earnings. Generally good earnings correlate with a jump in the stock price the next day (and vice versa with poor earnings numbers), so we will use Apple as an indicator for the rest of the sector (assuming Apple is the first in the sector to report).

Importing Relevant Packages

Here I will place any important packages that are neccessary for the functions in the notebook to run.

In [1]:
```
from quantopian.pipeline import Pipeline # Used to import from Quantop
ian
from quantopian.research import run_pipeline #
from quantopian.pipeline.factors import Returns, MarketCap # Want stoc
k performance and mkt cap (SP500)
from quantopian.pipeline.data import USEquityPricing, Fundamentals # W
ant stock pricing and earnings

from quantopian.pipeline.filters import QTradableStocksUS, StaticAsset
s # Want to know if the stock

# is tradeable

from quantopian.pipeline.classifiers.fundamentals import Sector # Need
the sector of each company

# From Part 2 #########################################
from quantopian.pipeline.data.factset.estimates import Actuals
from quantopian.pipeline import Pipeline
import quantopian.pipeline.data.factset.estimates as fe
from quantopian.pipeline.domain import US_EQUITIES
from quantopian.research import run_pipeline
########################################################


# Other conventional packages #########################
import numpy as np # Numeric calculations
import pandas as pd # Dataframe calculations
import matplotlib.pyplot as plt # Plots and charts
```

Importing the Data

We will use **Quantopian's API** to store the constituents of the **S&P500** and save the data of each date into a separate dictionary entry. Since the S&P500 is an index that tracks the performance of the 500 largest companies by market cap, it is easy to create such a pull.

In [2]:
```
def make_pipeline():

    # Pipeline factors
    close_price = USEquityPricing.close.latest # Close price of each s
tock
    market_cap = MarketCap() # Market cap of each stock

    # Pipeline Filters
    QTU = QTradableStocksUS()
    top_500_market_cap = market_cap.top(500)

    QTU_top_500 = QTU & top_500_market_cap

    # Want to make sure that the stock is listed and tradeable on that
day
    has_pricing_data = close_price.notnull()

    return Pipeline(
        # columns takes a dict of {'name of factor goes here':  factor
_variable} arguments (usually)
        columns={'close_price': close_price},
        # screen takes filter arguments
        screen=QTU_top_500 & has_pricing_data
    )
```

Test the Data

Let us take a look at the output and **make sure we have everything we need** before we dedicate the time to pull a much larger dataset.

In [3]:
```
# Want to have this in its own cell so it only needs to be run once
pipeline_test = run_pipeline(make_pipeline(), '2017-1-1', '2017-1-5')
```

Pipeline Execution Time: 8.12 Seconds

In [4]:
```
pipeline_test.keys()
```

Out[4]:
```
Index(['close_price'], dtype='object')
```

So it looks like the **pipeline_test dict** stores everything in the key ["close_price"]. We can **change/add other outputs** (which we will need later) in the columns arg in the **return Pipeline() part of the function above.** Now we will look further into the dict to see how we can access more of the data.

```
In [5]: pipeline_test["close_price"].keys()[0]

Out[5]: (Timestamp('2017-01-03 00:00:00+0000', tz='UTC', offset='C'),
         Equity(24 [AAPL]))
```

Since I picked a short time frame at the beginning of the year, there was only one day where the market was open, which is why we see only one key here. We will explore further:

```
In [6]: pipeline_test["close_price"]["2017-01-03"].head() # .head() shows just
        the first 5 entries

Out[6]: Equity(24 [AAPL])    115.84
        Equity(62 [ABT])      38.42
        Equity(64 [GOLD])     15.99
        Equity(67 [ADSK])     74.01
        Equity(76 [TAP])      97.33
        Name: close_price, dtype: float64
```

So it seems like this is what we are looking for. Above are the close prices for some of the stocks listed on the S&P500 on that day. For example, we now know that Apple's close price (AAPL) on January 3, 2017 was $115.84.

We will continue with our research project in part 2 of the MIRC tutorials.

Part 2 -- Wrangling the Data

In Part 1 we learned how to **initialize a pipeline**, use that pipeline to **pull basic price data**, and **read that data into a dictionary** for further analysis.

If we are trying to look at **sector momentum*, we will need** more data**, both in terms of** range **(rule of thumb is something like 10 years of data) and in terms of** what data **we need (for sector momentum we will need some sort of gauge of "success," one example being** earnings surprise **i.e. how much the company** beat/missed earnings** previously predicted by analysts).

Creating our Factors

We will take the pipeline function we defined earlier and modify it to match the needs listed above (using the documentation found here https://www.quantopian.com/docs/data-reference/estimates_actuals (https://www.quantopian.com/docs/data-reference/estimates_actuals))

Note there are more packages that need to be downloaded for the next step (see the link). These will be added to the cell at the very top under the Part 2 comment

First, we must create our earnings surprise gauge using the pipeline (literally copied and pasted from the link):

In [7]:
```
# Slice the PeriodicConsensus and Actuals DataSetFamilies into DataSet
s. In this context,
# fq0_eps_cons is a DataSet containing consensus estimates data (e.g.
what analysts on Wall St and
# beyond believe the EPS of the company will be in the next quarter) a
bout EPS for the
# most recently reported fiscal quarter. fq0_eps_act is a DataSet cont
aining the actual
# reported EPS for the most recently reported quarter.
fq0_eps_cons = fe.PeriodicConsensus.slice('EPS', 'qf', 0)
fq0_eps_act = fe.Actuals.slice('EPS', 'qf', 0)

# Get the latest mean consensus EPS estimate for the last reported qua
rter.
fq0_eps_cons_mean = fq0_eps_cons.mean.latest

# Get the EPS value from the last reported quarter.
fq0_eps_act_value = fq0_eps_act.actual_value.latest

# Define a surprise factor to be the relative difference between the e
stimated and
# reported EPS.
fq0_surprise = (fq0_eps_act_value - fq0_eps_cons_mean) / fq0_eps_cons_
mean
```

Next we define a custom factor for sector (see link https://www.quantopian.com/posts/figure-out-sector (https://www.quantopian.com/posts/figure-out-sector)):

In [8]:
```
sector = Fundamentals.morningstar_sector_code.latest
```

Now that we have "made" our factors, we will plug it into the function that we declared in Part 1 and change a few things to match our goals.

In [9]:
```
# Add the surprise factor to the pipeline.
def make_pipeline():

    # Pipeline factors -- same as before
    close_price = USEquityPricing.close.latest # Close price of each s
tock
    market_cap = MarketCap() # Market cap of each stock

    # Pipeline Filters -- same as before
    QTU = QTradableStocksUS()
    top_500_market_cap = market_cap.top(500)
    QTU_top_500 = QTU & top_500_market_cap

    # Want to make sure that the stock is listed and tradeable on that
day
    # same as before
    has_pricing_data = close_price.notnull()

    # this is the only part that needs to be changed, add our new fact
or to the list
    return Pipeline(
        columns={
        'eps_surprise_factor': fq0_surprise,# New factor
        'sector': sector, # new factor
        'actuals': fq0_eps_act_value, # separate factor for actuals, w
ill see why later
        'close_price': close_price # Same as before
    },
    screen=QTU_top_500 & has_pricing_data)
```

In [10]:
```
# Want to have this in its own cell so it only needs to be run once
# Same as before, just to test to make sure this is what we are lookin
g for
pipeline_test = run_pipeline(make_pipeline(), '2017-1-1', '2017-1-5')
```

Pipeline Execution Time: 3.45 Seconds

In [11]:
```
pipeline_test.keys()
```

Out[11]:
```
Index(['actuals', 'close_price', 'eps_surprise_factor', 'sector'], d
type='object')
```

Let us now look at what our new eps factor looks like:

```
In [12]: pipeline_test["eps_surprise_factor"]["2017-01-03"].head()
```

```
Out[12]: Equity(24 [AAPL])    0.007769
         Equity(62 [ABT])     0.014876
         Equity(64 [GOLD])    0.191487
         Equity(67 [ADSK])   -0.248384
         Equity(76 [TAP])     0.007497
         Name: eps_surprise_factor, dtype: float64
```

We can see that the estimate average for AAPL comes very close in this case to the actual realized value (estimates were off the actuals by less than a percent (I wonder if this is a reasonable assumption for all the estimates?)) vs, for example Autodesk (ADSK) where the estimate average was off by almost 25%.

And now we will look at what the sector factor looks like:

```
In [13]: pipeline_test["sector"]["2017-01-03"].head()
```

```
Out[13]: Equity(24 [AAPL])    311
         Equity(62 [ABT])     206
         Equity(64 [GOLD])    101
         Equity(67 [ADSK])    311
         Equity(76 [TAP])     205
         Name: sector, dtype: int64
```

There seems to be a code that Morningstar/FactSet uses for their sectors, so I will create a dictionary to easily map the code to a string (see https://spotlightstockmarket.com/media/6145/morningstar.pdf (https://spotlightstockmarket.com/media/6145/morningstar.pdf)):

```
In [14]: # Find the codes (GICS Level 1 usually has only 11)
         pipeline_test["sector"]["2017-01-03"].unique()
```

```
Out[14]: array([311, 206, 101, 205, 207, 103, 309, 310, 102, 308, 104])
```

```
In [15]: sectorNumToString = {}
         sectorNumToString[311] = "Technology"
         sectorNumToString[206] = "Healthcare"
         sectorNumToString[101] = "Basic Materials"
         sectorNumToString[205] = "Consumer Staple" # aka Consumer Defensive
         sectorNumToString[207] = "Utilities"
         sectorNumToString[103] = "Financial Services"
         sectorNumToString[309] = "Energy"
         sectorNumToString[310] = "Industrials"
         sectorNumToString[102] = "Consumer Cyclical"
         sectorNumToString[308] = "Communication Services"
         sectorNumToString[104] = "Real Estate" # iirc this one is tricky becau
         se it was introduced later

         # quick test
         sectorNumToString[104]

Out[15]: 'Real Estate'
```

Finally, we will look at the actuals. We need this factor to find the first company to report in a given quarter. Since companies usually report on different days, often days or weeks after the official quarter end, we must find the first company to report for every quarter.

```
In [16]: pipeline_test["actuals"]["2017-01-03"].head()

Out[16]: Equity(24 [AAPL])     1.67
         Equity(62 [ABT])      0.59
         Equity(64 [GOLD])     0.24
         Equity(67 [ADSK])    -0.18
         Equity(76 [TAP])      1.03
         Name: actuals, dtype: float64
```

Looking at this data, we can say that AAPL's reported Earnings per Share on January 3, 2017 was $1.67. The value itself is not important, it is the reporting date that is.

Reading, Consolidating, and Wrangling the Data

Now that we have created all the factors that we need and have sanity checked to make sure our data pull gives us what we want, we can now pull the whole range of data and "wrangle" it to format our algorithm later

```
In [17]: # In a separate cell so we only need to run it once
         data = run_pipeline(make_pipeline(), '2010-1-1', '2019-1-1')
```

Pipeline Execution Time: 53.81 Seconds

Next, I will create separate DataFrames for each to make wrangling easier:

```
In [18]: data_close = pd.DataFrame(data["close_price"])
         data_close = data_close.unstack() # removes the structure imposed by t
         he pipeline
         data_close.index.names = ["Date"] # creates a title for the date index
         data_close.columns = data_close.columns.droplevel(0) # removes the ext
         ra unneccessary column level
         data_close.head()
```

Out[18]:

	Equity(2 [ARNC])	Equity(24 [AAPL])	Equity(53 [ABMD])	Equity(62 [ABT])	Equity(64 [GOLD])	Equity(67 [ADSK])	Equity(76 [TAP])	Equity [AD
Date								
2010-01-04 00:00:00+00:00	16.12	210.84	NaN	53.95	39.40	NaN	45.14	3
2010-01-05 00:00:00+00:00	16.64	214.18	NaN	54.49	40.37	NaN	45.96	3
2010-01-06 00:00:00+00:00	16.12	214.38	NaN	53.99	40.88	NaN	45.34	3
2010-01-07 00:00:00+00:00	16.94	210.94	NaN	54.34	41.77	NaN	45.28	3
2010-01-08 00:00:00+00:00	16.59	210.58	NaN	54.78	41.18	NaN	44.60	3

5 rows × 595 columns

Do the same for the other factors:

In [19]:

```
data_surp = pd.DataFrame(data["eps_surprise_factor"]).unstack()
data_surp.index.names = ["Date"]
data_surp.columns = data_surp.columns.droplevel(0)
data_surp.head()
```

Out[19]:

	Equity(2 [ARNC])	Equity(24 [AAPL])	Equity(53 [ABMD])	Equity(62 [ABT])	Equity(64 [GOLD])	Equity(67 [ADSK])	Equity(76 [TAP])	Equit [A
Date								
2010-01-04 00:00:00+00:00	-1.486871	0.238709	NaN	0.024963	1.267621	NaN	0.18213	0.0
2010-01-05 00:00:00+00:00	-1.486871	0.238709	NaN	0.024963	1.267621	NaN	0.18213	0.0
2010-01-06 00:00:00+00:00	-1.486871	0.238709	NaN	0.024963	1.267621	NaN	0.18213	0.0
2010-01-07 00:00:00+00:00	-1.486871	0.238709	NaN	0.024963	1.267621	NaN	0.18213	0.0
2010-01-08 00:00:00+00:00	-1.486871	0.238709	NaN	0.024963	1.267621	NaN	0.18213	0.0

5 rows × 595 columns

The actuals dataframe will be a little different because we want to include the sector of the company in this dataframe to make later calculations easier.

In [20]:

```
data_actu = pd.DataFrame(data["actuals"]).unstack()
data_actu.index.names = ["Date"]
```

In [21]:
```
data_actu.head()
```

Out[21]:

	Equity(2 [ARNC])	Equity(24 [AAPL])	Equity(53 [ABMD])	Equity(62 [ABT])	Equity(64 [GOLD])	Equity(67 [ADSK])	Equity(76 [TAP])	Equity [AD
Date								
2010-01-04 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.14	
2010-01-05 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.14	
2010-01-06 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.14	
2010-01-07 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.14	
2010-01-08 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.14	

5 rows × 595 columns

We want to be able to **group the data above by sectors**, so we will take the sector data that we have pulled and turn it into a single-column dataframe where **index=stock/company name and value=sector**.

In [22]:
```
data_sector = pd.DataFrame(data["sector"]).unstack()
data_sector = data_sector.bfill() # Backwards fill so each company's s
ector code is the first row
data_sector.columns = data_sector.columns.droplevel(0)
data_sector = data_sector.T["2010-01-04"]
```

In [23]:
```
data_sector.head()
```

Out[23]:
```
Equity(2 [ARNC])     101.0
Equity(24 [AAPL])    311.0
Equity(53 [ABMD])    206.0
Equity(62 [ABT])     206.0
Equity(64 [GOLD])    101.0
Name: 2010-01-04 00:00:00+00:00, dtype: float64
```

In [24]:
```
for x in data_sector.index:
    data_sector[x] = sectorNumToString[data_sector[x]]
```

```
In [25]: data_sector.head()
```

```
Out[25]: Equity(2 [ARNC])     Basic Materials
         Equity(24 [AAPL])         Technology
         Equity(53 [ABMD])         Healthcare
         Equity(62 [ABT])          Healthcare
         Equity(64 [GOLD])    Basic Materials
         Name: 2010-01-04 00:00:00+00:00, dtype: object
```

Part 3 -- More Data Wrangling

We want to finish formatting our data so that our calculations are easy to run. We will continue working on the data_actu/data_sector combination

```
In [26]: # Remove hidden column multi-index in dataframe
         data_actu.columns = list(data_actu.columns.droplevel([0]))
```

I want to have a Multi-Index on the data_actu dataframe so I can group the stocks by each of their respective sectors. From there, it will be a lot easier to find the company that reported first for each sector for each quarter.

```
In [27]: # Create a list for the multiindex in the data_actu dataframe
         # use list comprehension here to make life easier, for more info see:
         # https://www.pythonforbeginners.com/basics/list-comprehensions-in-pyt
         hon
         data_actu.columns = [(data_sector[x],x) for x in data_actu.columns]

         # Change the columns in the data_actu to the new multiindex and sort t
         o look organized
         data_actu.columns = (pd.MultiIndex.from_tuples(data_actu.columns, name
         s=['sector', 'stock'])
                                          .sort_values())
```

In [28]:
```
data_actu.head()
```

Out[28]:

sector							
stock	**Equity(2 [ARNC])**	**Equity(64 [GOLD])**	**Equity(154 [AEM])**	**Equity(460 [APD])**	**Equity(1595 [CLF])**	**Equity(2119 [DD])**	**Equity(226 [DOW]**
Date							
2010-01-04 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-05 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-06 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-07 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-08 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1

5 rows × 595 columns

This looks a lot more like what I'm looking for. Now I can simply slice the dataframe by sector and work with those slices:

In [29]:
```
data_actu["Basic Materials"].head()
```

Out[29]:

stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226 [DOW]
Date							
2010-01-04 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-05 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-06 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-07 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1
2010-01-08 00:00:00+00:00	0.12	1.82	NaN	0.919999	0.54	NaN	1.1

5 rows × 38 columns

From here, we will turn the data into a binary dataframe to see exactly when a company reports as follows:

- Assign 1 if the value is new e.g. it is a new earnings release
- 0 if the value has not changed

To do this, we will:

- Take the percent change day-over-day. If the value has not changed, the percent change will be 0 and if it has changed, it will be a number whose absolute value is greater than zero.
- Divide that dataframe by itself, which will populate all the 0's (representing no change, or the stock hasn't been listed) with NaN's and all the nonzero numbers (representing when a company reports) with 1's

In [30]:
```
report = (data_actu.pct_change() / data_actu.pct_change()).fillna(0)
report.head()
```

Out[30]:

sector							
stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226: [DOW]
Date							
2010-01-04 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(

5 rows × 595 columns

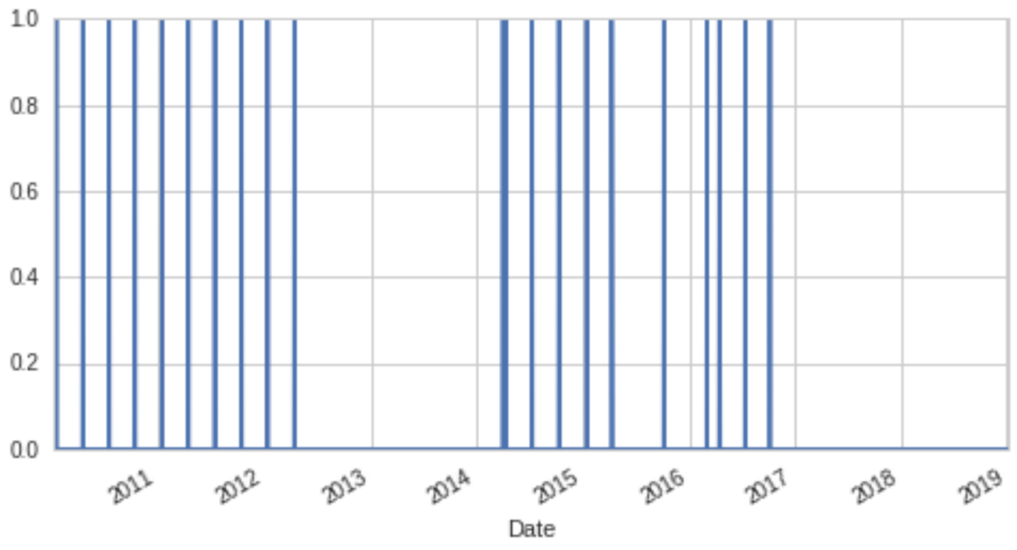
Let's pick a random stock and see how it looks in this new dataframe:

In [31]:
```
report["Basic Materials"].columns[0]
```

Out[31]:
```
Equity(2 [ARNC])
```

```
In [32]: # Plug in the above value into the slice, plot the slice of that stock
         report["Basic Materials"][report["Basic Materials"].columns[0]].plot(f
         igsize=(8,4))
```

```
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3f5663908>
```



We can see exactly when this company reports, and see the gaps when the stock must have been delisted from the S&P 500.

Basic Visualization of the Strategy (to be cont'd)

Here I will show a visualization of the strategy. Let's take the above dataframe of Basic Materials and try to find the first company in the sector to report for each sector.

```
In [33]: test_sector = report["Basic Materials"]
         test_sector.head()
```

Out[33]:

stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226: [DOW]
Date							
2010-01-04 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(

5 rows × 38 columns

```
In [34]: # Create a column at the end of the dataframe called Qdate with the co
         rresponding quarter start date
         test_sector['Qdate'] = [date - pd.tseries.offsets.QuarterBegin(startin
         gMonth=1)
                                 for date in test_sector.index]
```

```
/venvs/py35/lib/python3.5/site-packages/ipykernel_launcher.py:3: Set
tingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
  This is separate from the ipykernel package so we can avoid doing
imports until
```

```
In [35]: test_sector.head()
```

Out[35]:

stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226: [DOW]
Date							
2010-01-04 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(

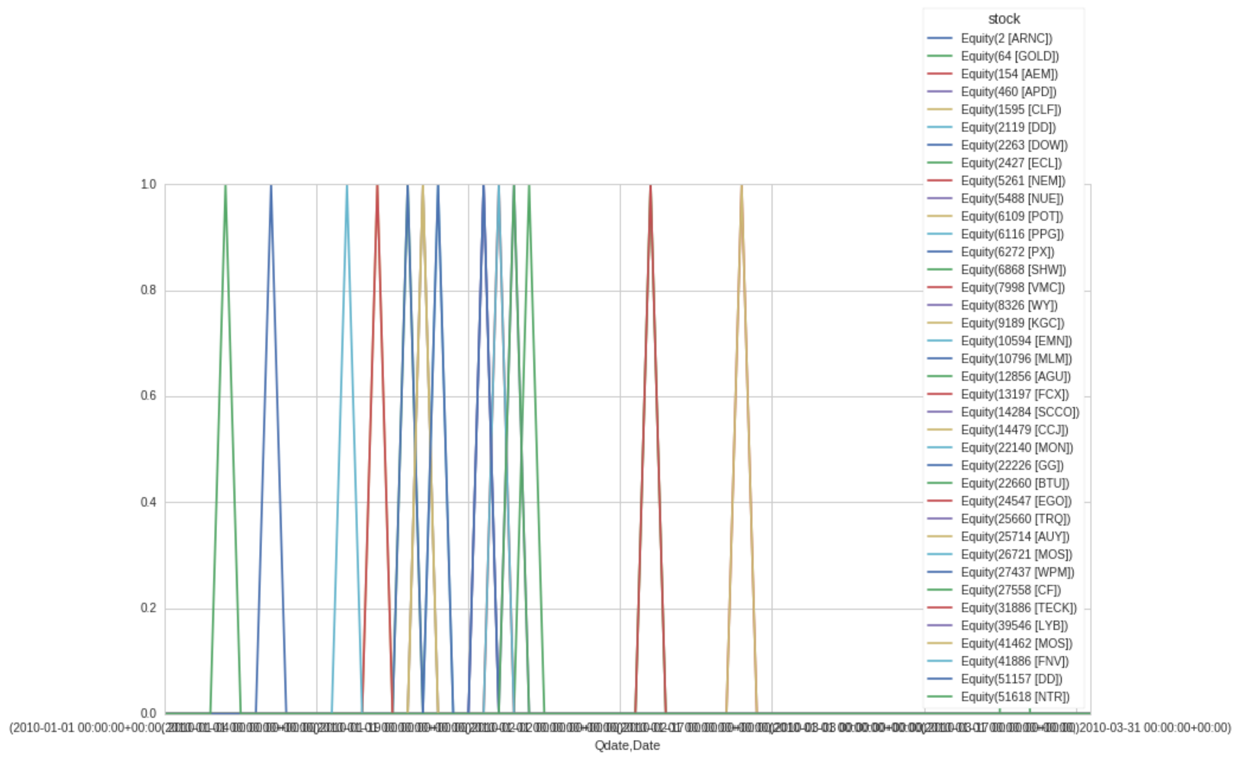
5 rows × 39 columns

We can see at the end that the Qdate column has worked. We will now group by the Qdate and take only the first value of that group:

```
In [36]: test_sector = test_sector.reset_index().groupby(["Qdate", "Date"]).fir
         st()
```

```
In [37]: # Visualize reporting schedule for quarter 1, 2010
         test_sector[test_sector.index.get_level_values(0) == "2010-01-01"].plo
         t()
```

```
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3f5cb0898>
```



We want to isolate the first green value on the far left of the chart. Let's create a dummy dataframe and try to simply get that first value alone:

In [38]:
```
sample = test_sector[test_sector.index.get_level_values(0) == "2010-01
-01"]
sample.head()
```

Out[38]:

	stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2 [I
Qdate	Date						
	2010-01-04 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
	2010-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
2010-01-01 00:00:00+00:00	2010-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
	2010-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
	2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	

5 rows × 38 columns

In [39]:
```
# Delete any rows that are all zero
sample = sample[~(sample==0).all(axis=1)]

# Isolate the first row with a nonzero value
sample = sample.iloc[0,:]

# Convert to DataFrame and transpose
sample = pd.DataFrame(sample).T
```

In [40]:
```
sample.head()
```

Out[40]:

	stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2 [I
2010-01-01 00:00:00+00:00	2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	

1 rows × 38 columns

If we scroll all the way to the end of this transposed matrix, we will see that stock NTR was the first to report for this quarter, and the report was on January 8, 2010. So now we want to see how these earnings compared to the estimates. If there was a big surprise and the company reported a high earnings per share, we will buy every stock in the sector, assuming that all the others will also report favourably. Vice-versa if the sector is performing badly.

Let's see what the results were for NTR on this date by looking at our data_actu dataframe:

```
In [41]: # Carve out only Basic Materials, on January 8, 2010, for stock 51618,
         or NTR
         data_actu["Basic Materials"][data_actu["Basic Materials"].index == "20
         10-01-08"][51618]
```

```
Out[41]: Date
         2010-01-08 00:00:00+00:00    0.58
         Name: Equity(51618 [NTR]), dtype: float64
```

We see that NTR reported favourably on that date, so we will long all stocks in the Basic Materials sector for the rest of that quarter.

The only step now is to extend this idea to all quarters, using all sectors. Once this signal is generated, it will be easy to see how well (or poorly) this algorithm performs.

Part 4 -- End of Data Wrangling

As mentioned above, we will extend the single-sector, single-quarter example constructed above to all sectors and all quarters. First we need to transform our data_actu dataframe to match what with

```
In [42]: report['Qdate'] = [date - pd.tseries.offsets.QuarterBegin(startingMont
         h=1)
                                 for date in report.index]

         report = report.reset_index().groupby(["Qdate", "Date"]).first()
```

```
/venvs/py35/lib/python3.5/site-packages/pandas/core/base.py:320: Per
formanceWarning: dropping on a non-lexsorted multi-index without a l
evel parameter may impact performance.
  return self.obj.drop(self.exclusions, axis=1)
```

In [43]:
```python
report.head()
```

Out[43]:

	sector						
	stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2 [I
Qdate	Date						
	2010-01-04 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
	2010-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
2010-01-01 00:00:00+00:00	2010-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
	2010-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	
	2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	

5 rows × 595 columns

In [44]:
```
# Create a dictionary to store the values
first_report = {}
is_empty = bool

sector_df = pd.DataFrame()

# Loop through each sector
for sector in report.columns.get_level_values(0).unique():
    first_report[sector] = {}

    # Within each sector, loop through each quarter
    for quarter in report[sector].index.get_level_values(0).unique()[:
-1]:

        sample = report[sector][report[sector].index.get_level_values(
0) == quarter]

        # Delete any rows that are all zero
        sample = sample[~(sample==0).all(axis=1)]

        # It could be possible that data wasn't collected for some sec
tors until a certain
        # point in time, so we will put a try/except statement here in
case there are some
        # empty quarters
        try:
            # Isolate the first row with a nonzero value
            sample = sample.iloc[0,:]

            # Convert to DataFrame and transpose
            sample = pd.DataFrame(sample).T

        except:
            pass

        first_report[sector][quarter] = sample
```

Now let's take a look at a random sector combined into a single dataframe:

```
In [45]: pd.concat(first_report["Basic Materials"]).head()
```

Out[45]:

		stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equi
2010-01-01 00:00:00+00:00	2010-01-01 00:00:00+00:00	2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	
2010-04-01 00:00:00+00:00	2010-04-01 00:00:00+00:00	2010-04-09 00:00:00+00:00	0.0	0.0	0.0	0.0	
2010-07-01 00:00:00+00:00	2010-07-01 00:00:00+00:00	2010-07-14 00:00:00+00:00	1.0	0.0	0.0	0.0	
2010-10-01 00:00:00+00:00	2010-10-01 00:00:00+00:00	2010-10-11 00:00:00+00:00	1.0	0.0	0.0	0.0	
2011-01-01 00:00:00+00:00	2011-01-01 00:00:00+00:00	2011-01-12 00:00:00+00:00	1.0	0.0	0.0	0.0	

5 rows × 38 columns

When we combine the dictionary entries into a single dataframe, we have to make some changes. But we can see the quarters in the second column and the first reported date in the third column. Let's make some changes:

```
In [46]: test_sector = pd.concat(first_report["Basic Materials"])
         test_sector.index = test_sector.index.get_level_values(2)
         test_sector.head()
```

Out[46]:

stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226: [DOW]
2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-04-09 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-07-14 00:00:00+00:00	1.0	0.0	0.0	0.0	0.0	0.0	0.(
2010-10-11 00:00:00+00:00	1.0	0.0	0.0	0.0	0.0	0.0	0.(
2011-01-12 00:00:00+00:00	1.0	0.0	0.0	0.0	0.0	0.0	0.(

5 rows × 38 columns

The final step in formatting this is to now reindex to match the surprises:

In [47]:
```python
test_sector = test_sector.reindex(data_surp.index)
test_sector.head()
```

Out[47]:

stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226 [DOW]
Date							
2010-01-04 00:00:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	Na
2010-01-05 00:00:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	Na
2010-01-06 00:00:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	Na
2010-01-07 00:00:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	Na
2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.

5 rows × 38 columns

And finally fill NAs with 0:

In [48]:
```python
test_sector = test_sector.fillna(0)
test_sector.head()
```

Out[48]:

stock	Equity(2 [ARNC])	Equity(64 [GOLD])	Equity(154 [AEM])	Equity(460 [APD])	Equity(1595 [CLF])	Equity(2119 [DD])	Equity(226 [DOW]
Date							
2010-01-04 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.
2010-01-05 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.
2010-01-06 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.
2010-01-07 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.
2010-01-08 00:00:00+00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.

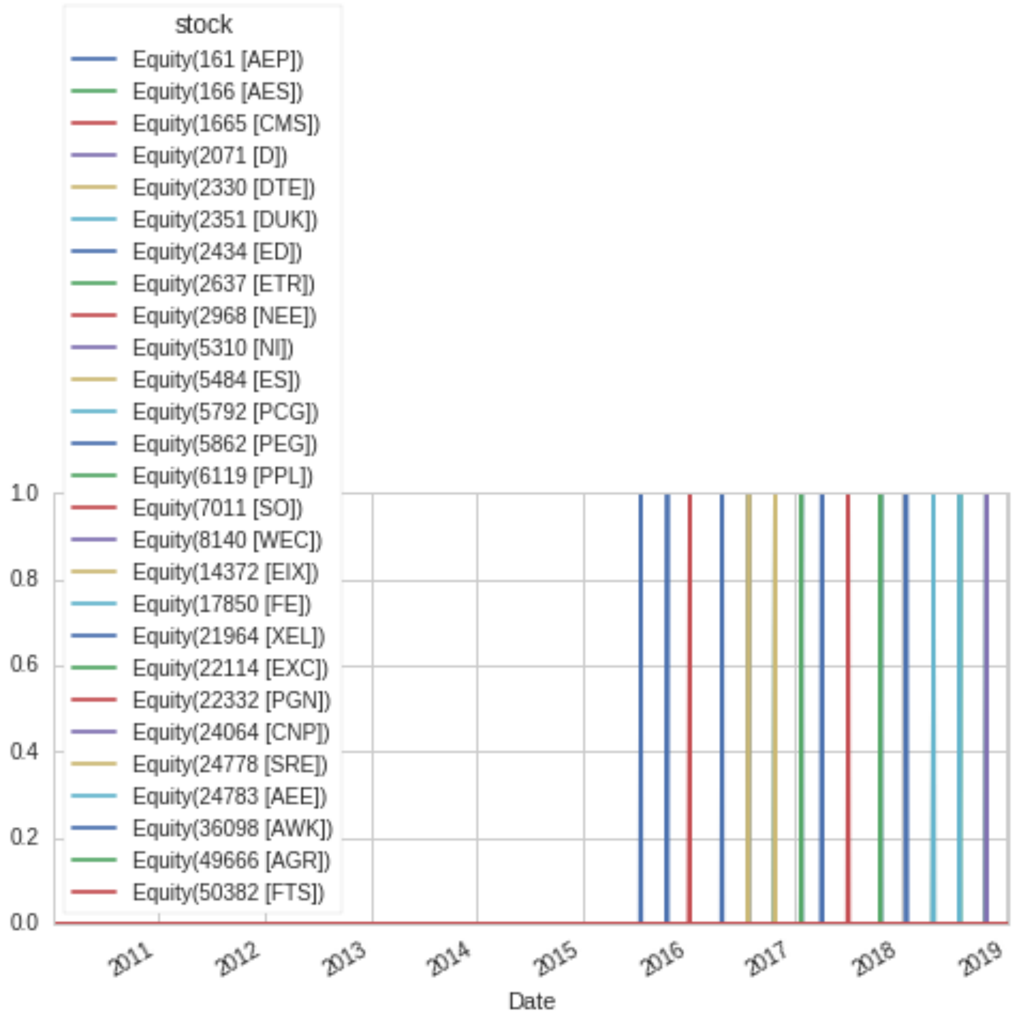
5 rows × 38 columns

So now we extend this same idea to each sector:

```
In [49]: for sector in first_report.keys():
             first_report[sector] = pd.concat(first_report[sector])
             first_report[sector].index = first_report[sector].index.get_level_
         values(2)
             first_report[sector] = first_report[sector].reindex(data_surp.inde
         x)
             first_report[sector] = first_report[sector].fillna(0)
```

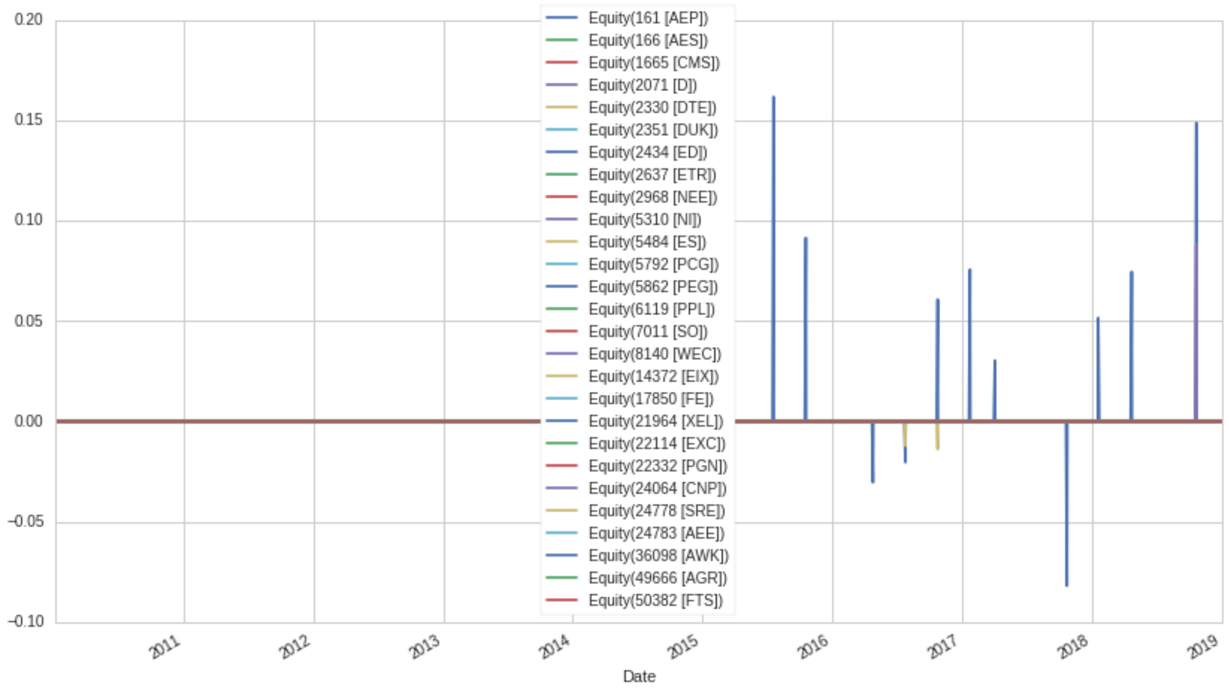
```
In [50]: # Random sanity check
         first_report["Utilities"].plot(figsize=(8,4))
```

```
Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3f5323e48>
```



From this point onwards, it will all just be matrix multiplication. We will multiply our sector dataframes (indicating the company that reported first) by the surprise of that company:

```
In [51]: # Slice only the columns that are in each of the first_report datafram
         es:
         (data_surp[first_report["Utilities"].columns] * first_report["Utilitie
         s"]).fillna(0).plot()

Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3f4d00550>
```

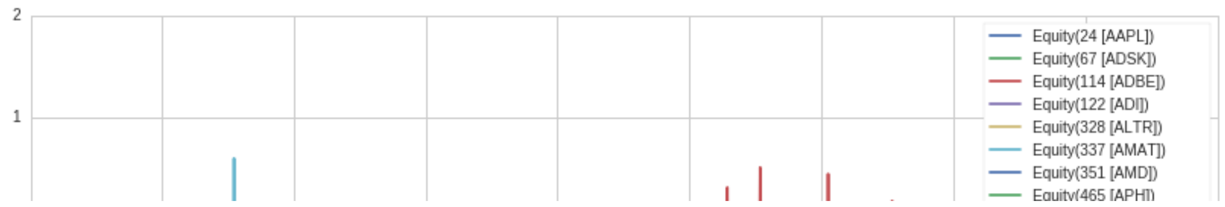


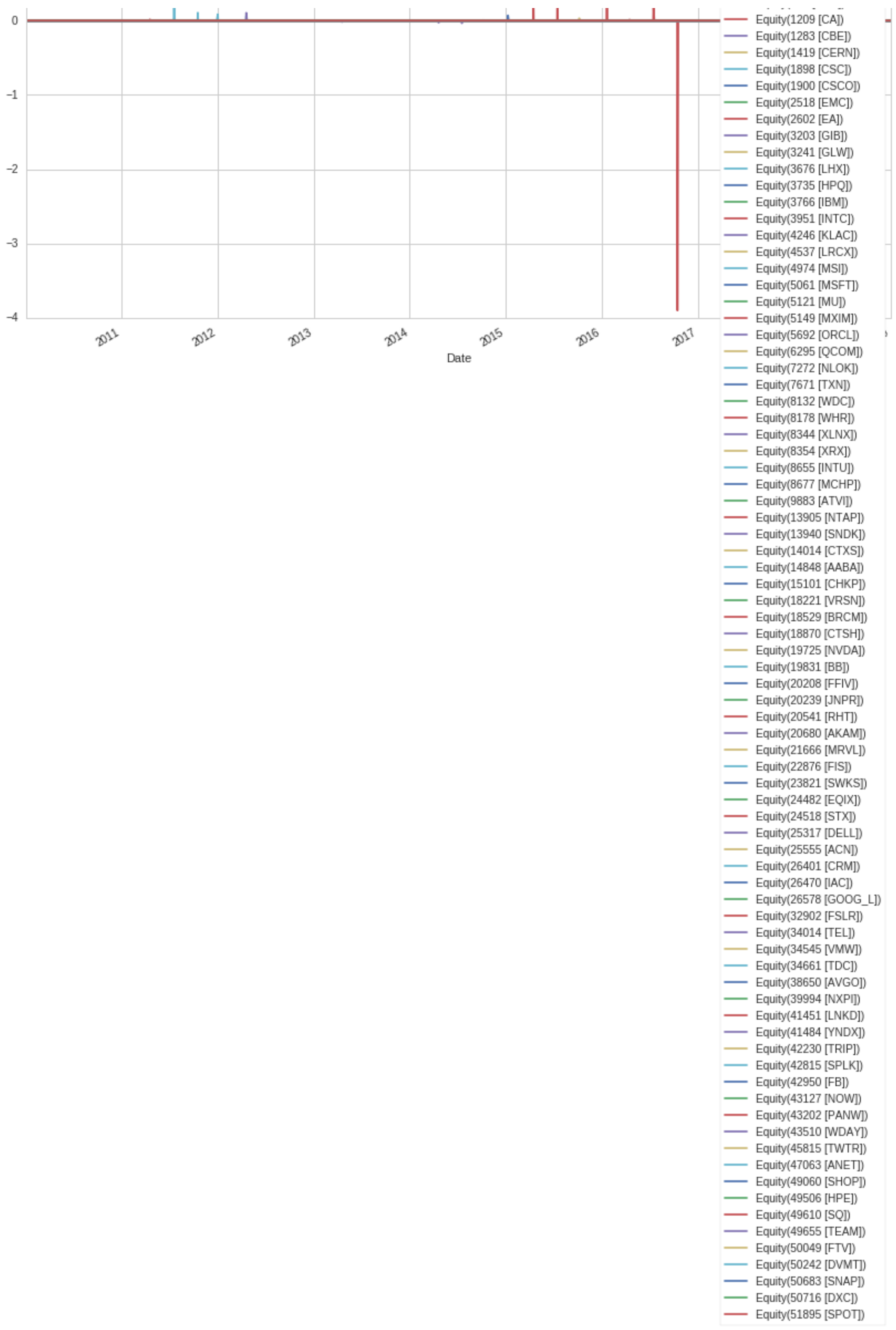
```
In [52]: # Do the above calculation for each sector
         for sector in first_report.keys():
             first_report[sector] = (data_surp[first_report[sector].columns] *
         first_report[sector]).fillna(0)
```

In some cases, there appears to be more than one company reporting first, on the same day, so we will take the mean across all first-to-reports

Here, we will need to get creative in order to construct our buy and sell signals. What we want is, for the quarters where the first report was positive, a +1 for each stock indicating buy, and a -1 if the report was negative, indicating a sell or short.

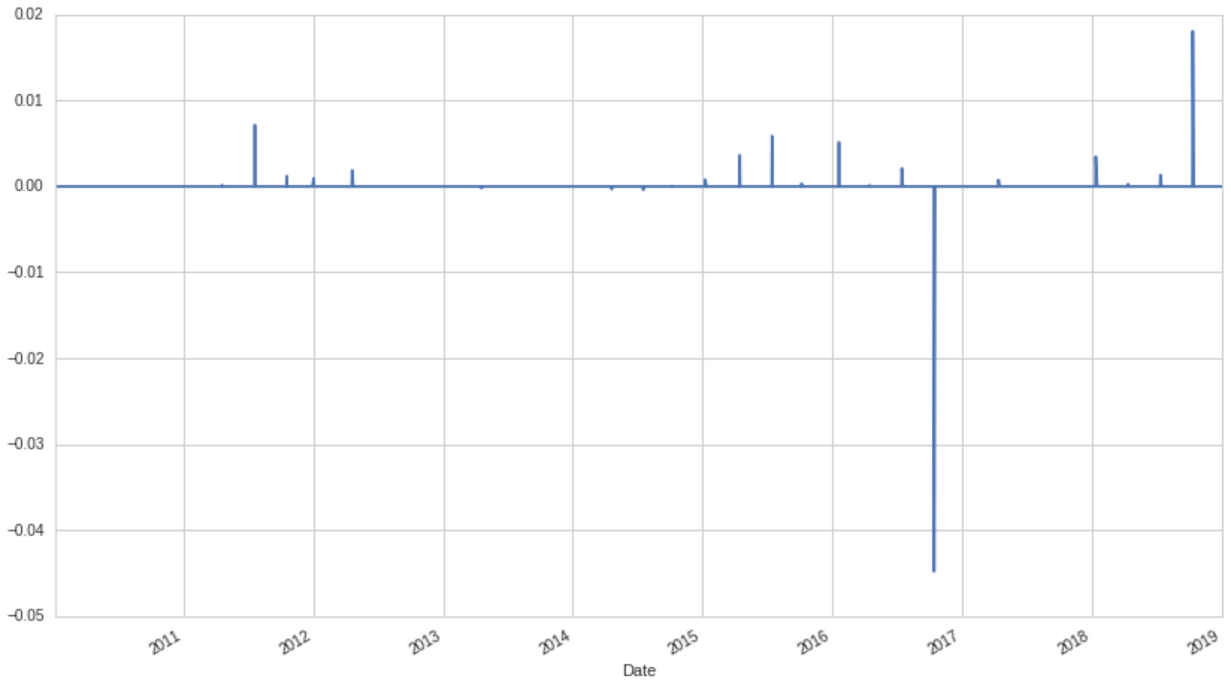
```
In [53]: signal_sample = first_report["Technology"]
         signal_sample.plot()

Out[53]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3f584cf60>
```




0
−1
−2
−3
−4
2011
2012
2013
2014
2015
2016
2017
Date
Equity(1209 [CA])
Equity(1283 [CBE])
Equity(1419 [CERN])
Equity(1898 [CSC])
Equity(1900 [CSCO])
Equity(2518 [EMC])
Equity(2602 [EA])
Equity(3203 [GIB])
Equity(3241 [GLW])
Equity(3676 [LHX])
Equity(3735 [HPQ])
Equity(3766 [IBM])
Equity(3951 [INTC])
Equity(4246 [KLAC])
Equity(4537 [LRCX])
Equity(4974 [MSI])
Equity(5061 [MSFT])
Equity(5121 [MU])
Equity(5149 [MXIM])
Equity(5692 [ORCL])
Equity(6295 [QCOM])
Equity(7272 [NLOK])
Equity(7671 [TXN])
Equity(8132 [WDC])
Equity(8178 [WHR])
Equity(8344 [XLNX])
Equity(8354 [XRX])
Equity(8655 [INTU])
Equity(8677 [MCHP])
Equity(9883 [ATVI])
Equity(13905 [NTAP])
Equity(13940 [SNDK])
Equity(14014 [CTXS])
Equity(14848 [AABA])
Equity(15101 [CHKP])
Equity(18221 [VRSN])
Equity(18529 [BRCM])
Equity(18870 [CTSH])
Equity(19725 [NVDA])
Equity(19831 [BB])
Equity(20208 [FFIV])
Equity(20239 [JNPR])
Equity(20541 [RHT])
Equity(20680 [AKAM])
Equity(21666 [MRVL])
Equity(22876 [FIS])
Equity(23821 [SWKS])
Equity(24482 [EQIX])
Equity(24518 [STX])
Equity(25317 [DELL])
Equity(25555 [ACN])
Equity(26401 [CRM])
Equity(26470 [IAC])
Equity(26578 [GOOG_L])
Equity(32902 [FSLR])
Equity(34014 [TEL])
Equity(34545 [VMW])
Equity(34661 [TDC])
Equity(38650 [AVGO])
Equity(39994 [NXPI])
Equity(41451 [LNKD])
Equity(41484 [YNDX])
Equity(42230 [TRIP])
Equity(42815 [SPLK])
Equity(42950 [FB])
Equity(43127 [NOW])
Equity(43202 [PANW])
Equity(43510 [WDAY])
Equity(45815 [TWTR])
Equity(47063 [ANET])
Equity(49060 [SHOP])
Equity(49506 [HPE])
Equity(49610 [SQ])
Equity(49655 [TEAM])
Equity(50049 [FTV])
Equity(50242 [DVMT])
Equity(50683 [SNAP])
Equity(50716 [DXC])
Equity(51895 [SPOT])

In [54]:
```
# Find the average of the rows
signal_sample.mean(axis=1).plot()
```

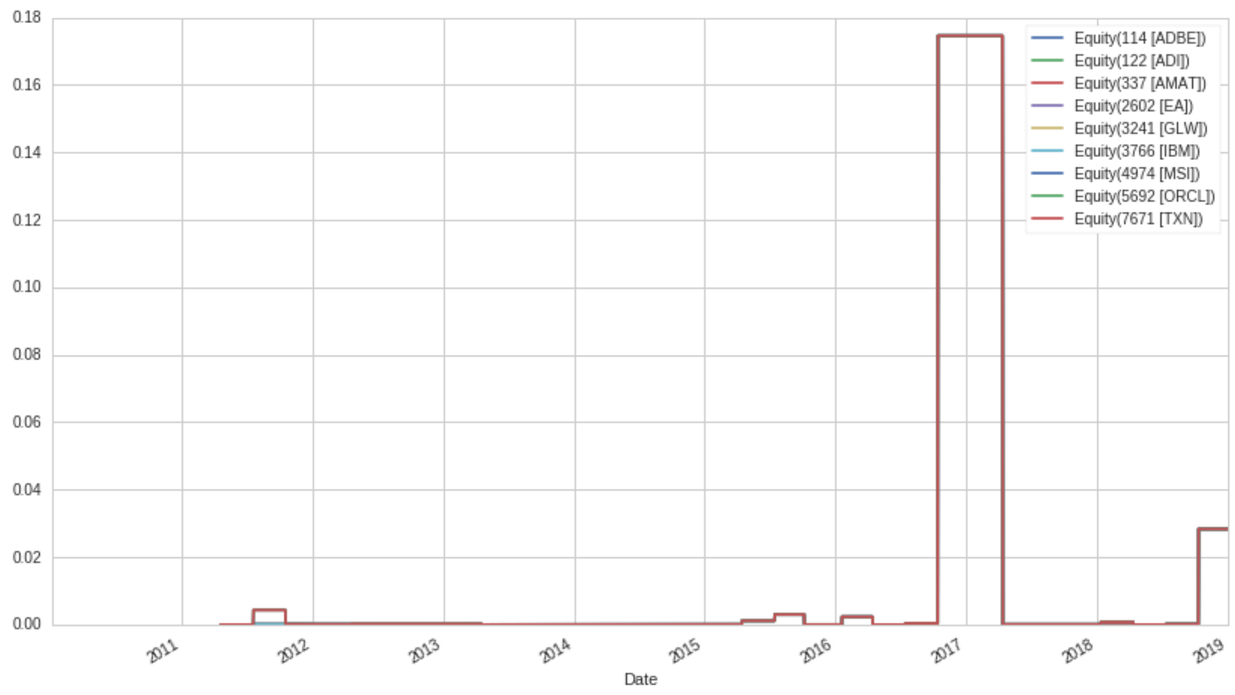
Out[54]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd3f555f9e8>
```



In [55]:
```
# Multiply this by the dataframe
signal_sample2 = (signal_sample.T * signal_sample.mean(axis=1)).T

# Change zeroes to NA
signal_sample2 = signal_sample2.replace(0, np.nan)
```

In [56]:
```
signal_sample2 = signal_sample2.dropna(how="all", axis=1).ffill(axis=1
).bfill(axis=1).ffill()
signal_sample2.plot()
```

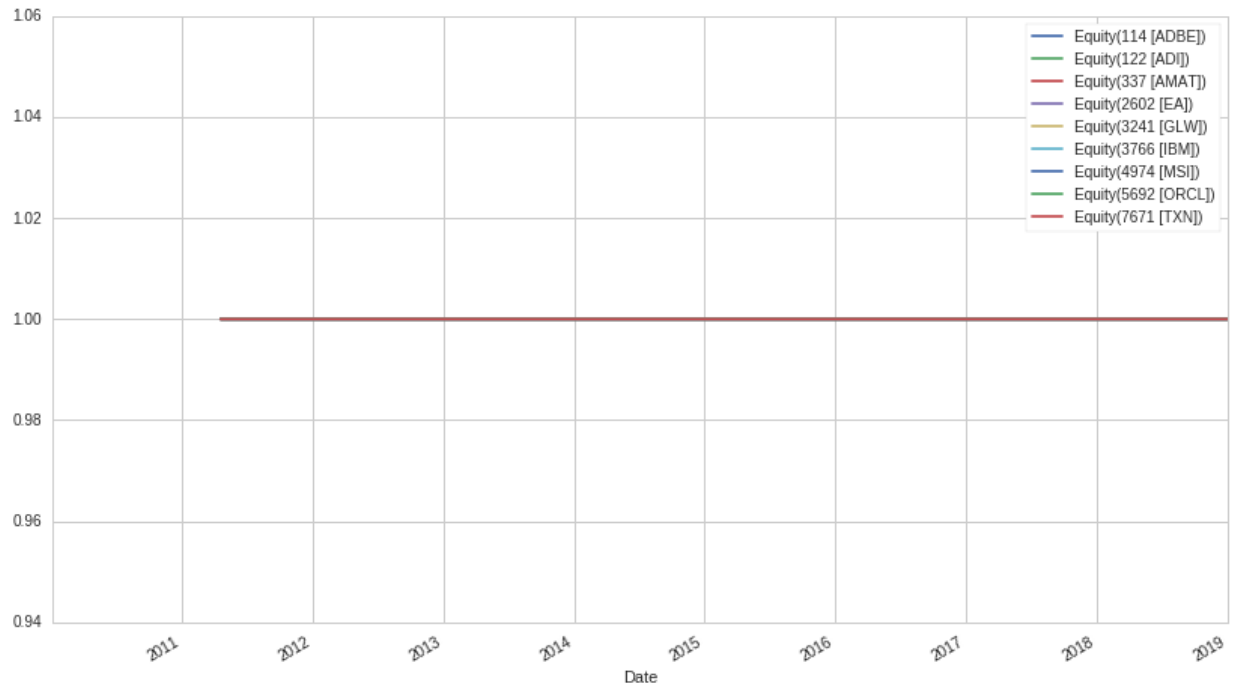
Out[56]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd3f5566320>
```



So now all the entries in signal_sample2 contain the average earnings of the first company(ies) to report for each quarter. Now we simply have to turn it into a buy/sell signal by dividing by the absolute value of the df:

In [57]:
```
signal_sample3 = signal_sample2 / abs(signal_sample2)
signal_sample3.plot()
```

Out[57]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd3f528a320>
```



It appears that the tech sector only reports positive surprises in this range. We will see if other sectors do the same, if so, it may be a problem.

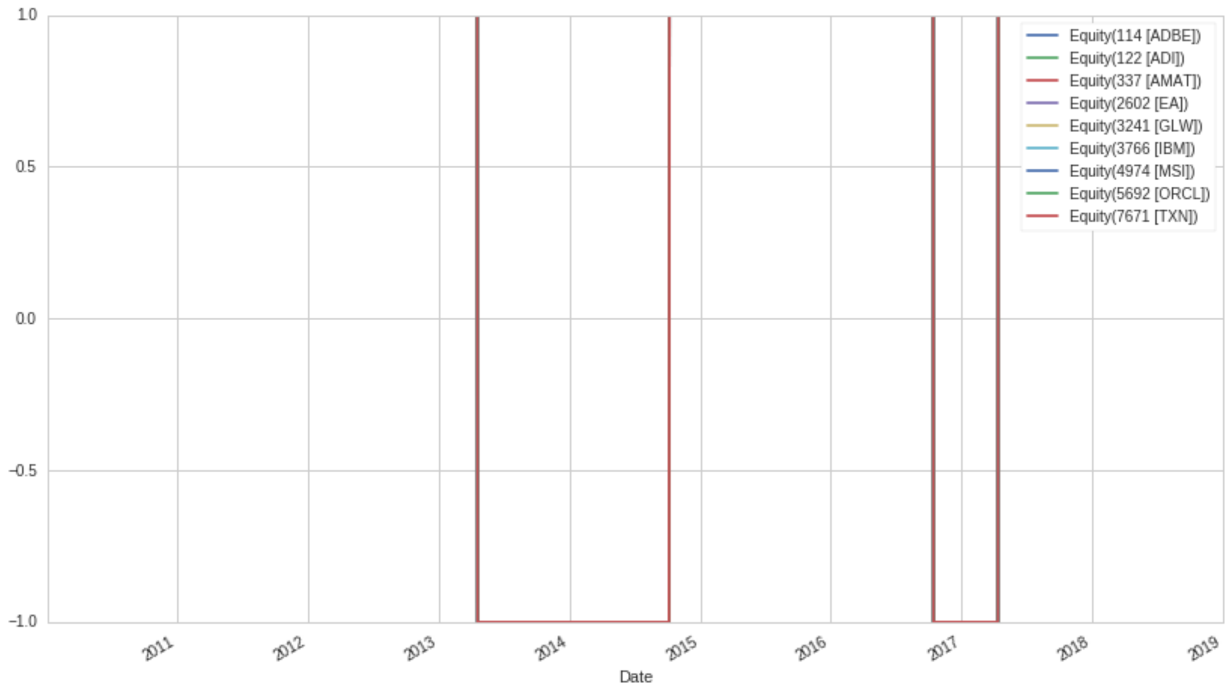
(Part 5 correction) I realize now that in cell containing:

signal_sample2 = (signal_sample.T * signal_sample.mean(axis=1)).T

Need to be changed to signal_sample.T * signal_sample.mean(axis=1) to an absolute value in order to preserve the sign of the earnings surprise. I'll redo the same idea here:

```
In [60]: signal_sample = first_report["Technology"]

         # Multiply this by the ABS(dataframe)
         signal_sample2 = (signal_sample.T * abs(signal_sample.mean(axis=1))).T

         # Change zeroes to NA
         signal_sample2 = signal_sample2.replace(0, np.nan)

         signal_sample2 = signal_sample2.dropna(how="all", axis=1).ffill(axis=1
         ).bfill(axis=1).ffill()

         signal_sample3 = signal_sample2 / abs(signal_sample2)

         signal_sample3.plot()
```

```
Out[60]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3f5115cf8>
```



Now we can see that the chart bounces between buy and sell signals. This makes a lot more sense.

Another issue that I noticed is that the dropna() removes any stock that didn't report. Since we just want this to be a buy signal for EVERY stock in the sector, we will get rid of the dropna and the backfill and forward fills will take care of the rest.

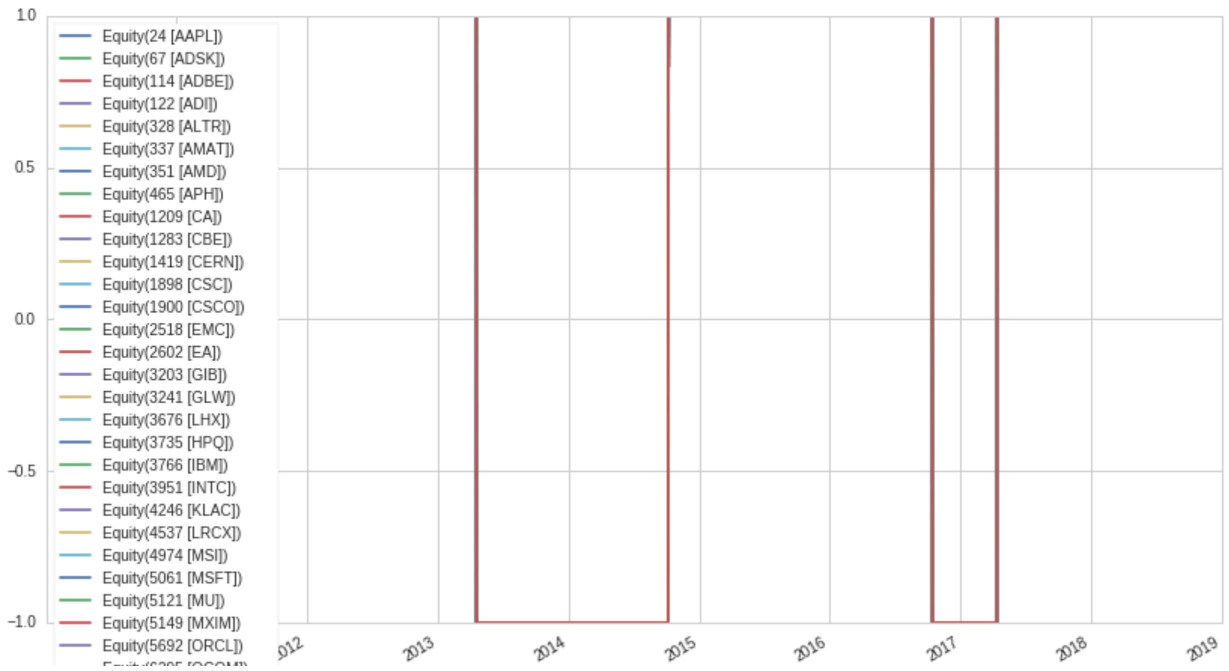
Part 5 -- Signal Generation

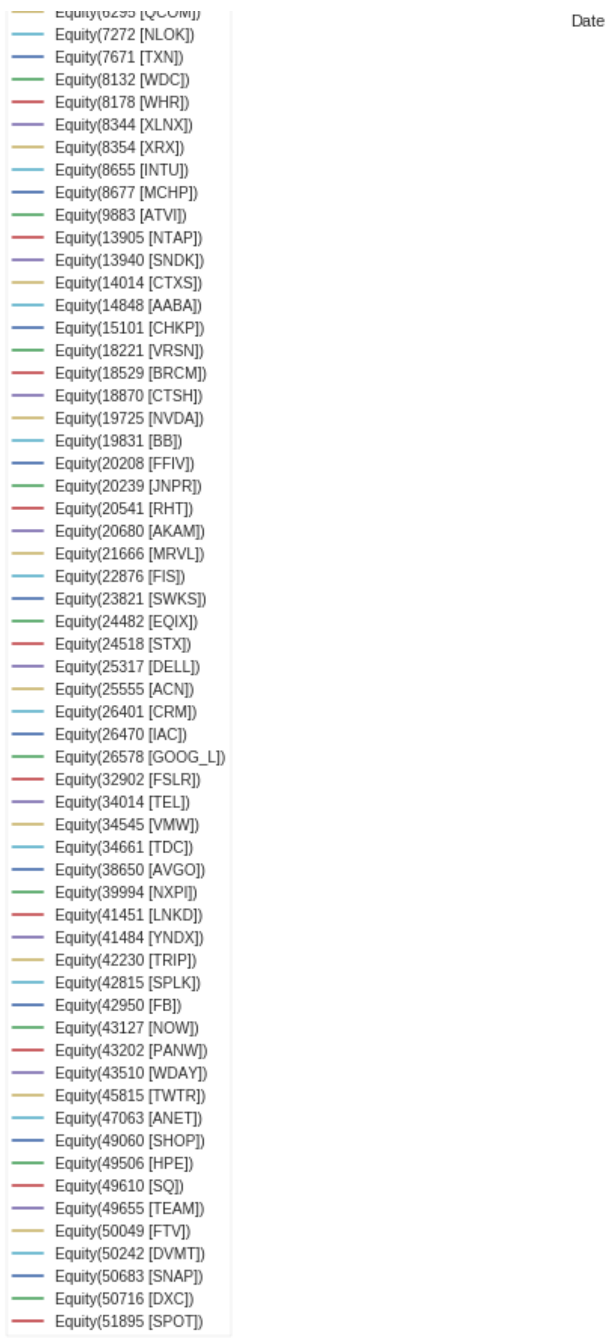
We will extend what we did in the one sector case to all the sectors to see what each of their respective buy/sell signals look like:

In [63]:
```
signal = {}

# Can literally copy and paste into the loop--probably not a good habi
t
# but for illustrative purposes we can maintain the same var names
for sector in first_report.keys():
    signal_sample = first_report[sector]

    # Multiply this by the ABS(dataframe)
    signal_sample2 = (signal_sample.T * abs(signal_sample.mean(axis=1)
)).T

    # Change zeroes to NA
    signal_sample2 = signal_sample2.replace(0, np.nan)

    signal_sample2 = signal_sample2.ffill(axis=1).bfill(axis=1).ffill(
)

    signal_sample3 = signal_sample2 / abs(signal_sample2)

    signal[sector] = signal_sample3
```

In [64]:
```
# Sanity check
signal["Technology"].plot()
```

Out[64]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd3f4e6dfd0>
```





From this point onwards, all that's left is multiplying the signal by the return of each individual stocks and then we can evaluate the performance of the strategy. I think I will leave this to be done at MIRC.

```
In [ ]:
```