Lesson 3: The Basics of Fundamentals

Introduction
Fundamentals with Morningstar is finally here. Almost thirteen years of historical corporate data (given that you’re reading this in 2015) dating all the way back to 2002. Need Apple’s quarter three earnings in 2006? We have it. Need Microsoft’s company headquarters in 2008? We have it. Need Netflix’s IPO date? We have it. Best of all, it comes in a Point-In-Time format, meaning that we give the data to you as it was at that point in time. So enough of the introductions, lets get started.

What you’re going to learn:
● Basic Concepts:
  ○ Understanding before_trading_start
  ○ Querying for fundamental data
● Advanced Concepts:
  ○ The limits behind how much data you can obtain
● Algorithm walkthrough:
  ○ Piotroski Score by Quantopian
  ○ Growth ranking for healthy stocks by Naoki Nagai
  ○ Identifying uptrending volatile small caps by Richard Prokopyshen

Resources:
● More Quantopian Fundamental Algorithms: Contest Winners
● Our Documentation
● Press Release

Basic Concepts

Understanding before_trading_start
The inclusion of fundamentals into Quantopian means that there’s an additional method that your algorithm needs: before_trading_start. So while before there was initialize and handle_data, we now have initialize, before_trading_start, and handle_data. And as the name suggests, before_trading_starts is run at the beginning of each trading day before the market open.
`before_trading_start` always, and requires, `context` as the parameters. And because `context` is passed as a parameter, that also means that you can access anything that `context` contains. So if you were to set a attribute in `initialize`, you could access that same attribute in `before_trading_start`. So in the following code snippet, the `stocks_to_buy` attribute is directly accessible in `before_trading_start`.

```python
def initialize(context):
    context.stocks_to_buy = 5
    context.stock = sid(24)

def before_trading_start(context):
    print context.stock

def handle_data(context, data):
    order(sid(24), 50)
```

To Summarize:
- In order to use fundamentals, you need a new method called `before_trading_start` which takes `context` as a variable.

**Querying for Fundamental Data**

Obtaining fundamental data really revolves around two functions: `get_fundamentals()` and `update_universe()`. The really interesting function is `get_fundamentals` which returns a pandas DataFrame with securities as the columns and the fundamental values as the indexes.

Using it is actually quite simple. In your `before_trading_starts` function, as soon as you start writing out `get_fundamentals` it will autocomplete and fill in a couple of values for you.

```python
def initialize(context):
    context.stocks_to_buy = 5
    context.stock = sid(24)

def before_trading_start(context):
    get_fundamentals(
        query(  
            # put your query in here
        ).filter(  
            # put your filters here
        ).order_by(  
            # sort your query
        ).limit(100)
    )

def handle_data(context, data):
    order(sid(24), 50)
```
You’ll notice that there are four parts: `query`, `filter`, `order_by`, and `limit`. So let’s break each of these down:

- **query**: This is where you specify exactly what data that you want back. If you want PE ratio you’ll start typing out `fundamentals.[ ]` and an autocomplete box will popup for you to type out PE ratio. Try it out! you should get something like:

```python
def before_trading_start(context):
    context.fundamental_df = get_fundamentals(
        query(
            fundamentals.
        ),
        filter(
            fundamentals.
        ),
        order_by(
            fundamentals.
        ),
        limit(100)
    )
```

- **filter**: This is where you can start narrowing down your choices. Let’s say you’re constructing a value investing portfolio and are only looking at securities with a PE ratio less than 5. You would do something like:

```python
def before_trading_start(context):
    context.fundamental_df = get_fundamentals(
        query(
            fundamentals.
        ),
        filter(
            fundamentals.
        ),
        order_by(
            fundamentals.
        ),
        limit(100)
    )
```

In this example, I’m filtering for where securities have a PE ratio greater than 5 and a market cap > 100000. You can have as many filters as you want.

- **order_by and limit**: Finally, this last part lets you define how you want to order your securities. E.g. Okay you have all securities with PE > 5 and market cap > 100000, but let’s say you wanted to get the top 10 securities with the greatest market cap. You would do something like:
And so the above would find you the 10 securities with the biggest market cap (by default the order by is set to ascending so you’ll want to set it to descending if you the biggest) with a pe ratio > 5. And it will also give you the PE ratios of each security.

So to reiterate the `get_fundamentals` call consists of `query`, `filter`, `order_by`, and `limit`. And once you have that, you’ve just completed the `get_fundamentals` part. However, you’ll notice that I have the results pointing to context. fundamental_df which is a new attribute on `context` that I’m defining here. This let’s me access `context.fundamental_df` throughout my algorithm which is necessary if I wanted to see the PE ratio of a security (in this example).

There’s still one last part. It’s great that we have the fundamental data, but it’s meaningless unless we set our universe of securities (otherwise our algorithm won’t run!). We make that really easy for you. Remember that the columns of `context.fundamental_df` are the securities. So we can just pass that into `update_universe` which takes a list of stocks and sets that as our universe for the day like so:

That sets your universe to the list of securities you’ve just obtained from `get_fundamentals` so now let’s move onto how we can incorporate that into our algorithms.

**Using the data in your algorithm**

If you remember from the last section, we set the results from `get_fundamentals` to an attribute of context called `context.fundamental_df`. So now, in somewhere like `handle_data` or something that you’ve created for your `schedule_function` parameter, you can begin accessing the data like so:

It’s really that simple!
Advanced Concepts

Universe Size

Your universe at any time is limited to 200 securities. This means you HAVE to have either a .limit(200) present or a filter strong enough to reduce your query to 200 securities. I highly suggest having a .limit(200) in your get_fundamentals call :).

Integration with Fetcher

It’s very possible to combine your algorithm with ‘fetch_csv’ but the details are hairy. I’ll give you a general skeleton of how you should do it and you can email slee@quantopian.com until we figure out a better way to do it. Basically, if you’re trying to find the fundamental data from a list of securities in a CSV, you have to put everything a day later than it is now. That’s because the first time FETCHER ever gets called is AFTER the first time ‘get_fundamentals’ ever gets called.

If that’s confusing, that’s because it IS, but here’s a quick mental framework to work with it: Whatever you’re thinking of doing, just shift the dates back by 1 day (so backwards) and you’ll get the expected results.

We’re working on better ways to incorporate the two, but until then, these are the constraints that exist for this integration.

Algorithm Walkthrough

I’m going to walk you through three algorithms: Piotroski Score by Quantopian, Fundamental Growth Stocks by Naoki Nagai, and Identifying Volatile Small Caps by Richard Prokopyshen. Now the best way to approach this section is to identify the general process of what these algorithms are doing.

The first algorithm, Piotroski Score is something written by us over here at Quantopian and it’s designed to give you all a good framework for dealing and working with multiple factors as well doing some analysis on year-to-year, historical fundamental data.

- You’ll notice that from the second user-shared algorithm by Naoki is a pure fundamentals algorithm. By that I mean almost all of it’s buying decisions (e.g. which stocks to purchase) are decided off of the ‘get_fundamentals’ query.
- Richard’s algorithm is very different. It incorporates ‘get_fundamentals’ as a launching pad for deciding a big basket of which stocks to analyze, and a number of statistical filters follow in handle_data before he narrows in on a solid buy choice. Neither is better than the other, but reflect the styles of each author. Let’s go through them now.

Note: For the sake of brevity, I’ll be skipping the initialize sections of each algorithm and will only be focusing on the ordering and fundamentals logic.
Piotroski Score by Quantopian

The Piotroski score was developed by Joseph Piotroski, a professor at the University of Chicago. ([http://www.grahaminvestor.com/articles/quantitative-tools/the-piotroski-score/](http://www.grahaminvestor.com/articles/quantitative-tools/the-piotroski-score/)). It essentially picks a number of securities to buy and short dependent on the strength of the securities’ balance sheet. The implementation of this algorithm, at first, seems quite simple. You’re getting a number of different fundamental factors and then choosing stocks whether or not they fit that criteria. However, the trickiest part is comparing current Point-In-Time fundamental data to the past year’s data. I’m going to walk you through how to do this and you’re going to learn a lot.

The algorithm itself works like this:

- Look for the nine different fundamental factors that Piotroski specifies. An example of this would be something like “Positive ROA.” If the security fulfills that criteria, it earns a point.
- Repeat the process for the nine different fundamental factors.
- Buy the securities with the most amount of points
- Short the securities with the least amount of points

The Hard Part

The trickiest part about this will be when we start comparing year-over-year data points. You may have already noticed that Fundamentals provides point-in-time data, which means it gives you the data for that company at that point in time for the backtest. So that means if you’re algorithm is currently at January 2nd, 2014, Fundamentals will return you the EPS for January 2nd, 2014. So that means we’ll have to find some way to compare current ROA versus last year’s ROA (a factor in the Piotroski Score).

How I’m going to do that

I’m going to store all my fundamental data in a Pandas Panel which is pretty much exactly how ‘data’ (remember when you pass in handle_data(context, data)) is structured.

The basics are going to be the same, you can index it by ‘date’ and query for each security like so fundamental_history[date][stock][ROA]. So now we can compare the current EPS and such like so:

```
data[stock][‘ROA’] > fundamental_history[last_year][stock][‘ROA’]
```
The criteria:

- Profitability
  - Positive ROA
  - Positive Operating Cash Flow
  - Higher ROA in current year versus last year
  - Cash flow from operations > ROA of current year

- Leverage
  - Current ratio of long term debt < last year's ratio of long term debt
  - Current year's current_ratio > last year's current_ratio
  - No new shares issued this year

- Operating Efficiency
  - Higher gross margin compared to previous year
  - Higher asset turnover ratio compared to previous year

The data query:

```python
#: Reference fundamentals in a shorter variable
f = fundamentals

#: Query for the data that we need from fundamentals
fundamental_df = get_fundamentals(query(
    f.valuation.market_cap,
    f.operation_ratios.roa,
    f.cash_flow_statement.operating_cash_flow,
    f.cash_flow_statement.cash_flow_from_continuing_operating_activities,
    f.operation_ratios.long_term_debt_equity_ratio,
    f.operation_ratios.current_ratio,
    f.valuation.shares_outstanding,
    f.operation_ratios.gross_margin,
    f.operation_ratios.assets_turnover,
)
    .order_by(fundamentals.valuation.market_cap.desc())
    .limit(40)
))

#: Set our fundamentals into a context variable
context.fundamental_df = fundamental_df

#: Update our universe with the values
update_universe(fundamental_df.columns.values)
```

Guide:

- Up to this part, it's simple enough. I'm simply going over the nine different balance sheet items specified in the Piotroski score
- But remember that we can only have 200 securities within the universe, so I'm taking it a step further and am only looking at the top 40 securities with the biggest market cap.
Saving historical fundamental data

```python
# Only run every 25 trading days
if context.days % 25 == 0:
    # Insert a new dataframe into our dictionary
    context.fundamental_dict[get_datetime()] = context.fundamental_df

    # If it's greater than the first trading day
    if context.days > 0:
        context.fundamental_data = pd.Panel(context.fundamental_dict)
        scores = get_piotroski_scores(context.fundamental_data, get_datetime())

    # Only rebalance when we have enough data
    if scores != None:
        rebalance(context, data, scores)
```

- First thing that I do is add the current Fundamental DataFrame (which I get from before_trading_starts) and map that to the current date:
  - "context.fundamental_dict[get_datetime()] = context.fundamental_df"
- Next, I take that dictionary, which stores a rolling accumulation of ‘date’:'Fundamental data' and create a Pandas panel out of it.
- Finally, I store that data in ‘context.fundamental_data’ which is a rolling accumulation of all historical fundamental data since the beginning of the backtest.

Calculating our scores

```python
def profit_logic(current_data, old_data, sid):
    # Define our profitability logic here

    # Positive ROA
    positive_roa = current_data[sid]["roa"] > 0
    # Positive Operating Cash Flow
    positive_ocf = current_data[sid]["operating_cash_flow"] > 0
    # Current ROA > Last Year ROA
    current_last_roa = current_data[sid]["roa"] > old_data[sid]["roa"]
    # Cash flow from operations > ROA
    cash_flow_roa = current_data[sid]["cash_flow_from_continuing_operating_activities"] > 
                   current_data[sid]["roa"]

    return int(positive_roa) + int(positive_ocf) + int(current_last_roa) + int(cash_flow_roa)
```

- I have three functions: ‘profit_logic’, ‘operating_logic’, and ‘leverage_logic’. Each of these take in the current day’s fundamental data, the last year’s fundamental data.
- They compare the different factors in the fundamental data and accumulate the number of points towards the end.
• Finally, we have this function called `get_piotroski_scores` which returns a dictionary of all the stocks in our universe and their respective scores.

• We then use that dictionary to walk it through a rebalance function, which simply weights our portfolio equally in the stocks with the highest scores and short positions in the lowest scores.

```
def rebalance(context, data, scores):
    # This method takes in the scores found by get_piotroski_scores and orders our portfolio accordingly
    num_long = [stock for stock in scores if scores[stock] >= 8]
    num_short = [stock for stock in scores if scores[stock] <= 2]

    # Stocks to long
    for stock in num_long:
        if stock in data:
            log.info("Going long on stock %s with score %s" % (stock.symbol, scores[stock]))
            order_target_percent(stock, 1.0/len(num_long))

    # Stocks to short
    for stock in num_short:
        if stock in data:
            log.info("Going short on stock %s with score %s" % (stock.symbol, scores[stock]))
            order_target_percent(stock, -1.0/len(num_short))

    # Exit any positions we might have
    for stock in context.portfolio.positions:
        if stock in data and stock not in num_long and stock not in num_short:
            log.info("Exiting our positions on %s" % stock.symbol)
            order_target_percent(stock, 0)

    record(number_long=len(num_long))
    record(number_short=len(num_short))
```

Summary:

• The algorithm queries for nine different fundamental criteria
• It assigns a point for each criteria that a security can meet
• We make use the Pandas Panel to create an easily indexable history object that we can use to compare year-over-year data
Basics:

I love this algorithm. It's simple. It's elegant. There are few if any hangups in its decision making power. It simply finds the top hundred highest growth securities. And here's a summary of what he does:

- Breaking down the logic of this algorithm: Naoki is literally using Morningstar's `revenue growth` field to filter for securities and selecting the top 100 companies with the highest revenue growth.
- He includes a minimum threshold of 10% of return on invested capital, and makes sure that these companies have a basic filter of market cap greater than 0 and shares outstanding is not an empty field.
- He includes a line for `[stock for stock in fundamental_df]` but at the same time, if you wanted to, you could simply skip all of that and have update_universe use `fundamental_df.columns.values` directly.
Ordering Logic:

Naoki does a lot of really smart, but simple things here that not a lot of people do but let’s walk through them step-by-step so you know what I mean:

- **Avoiding leverage:** He creates a system to track all the cash that he currently has in his portfolio. The average retail investor probably doesn’t want to dip into leverage, so you should take that into consideration when creating your algorithm.

- **Exiting current positions:** He makes sure to exit all of the current positions that he currently owns AND doesn’t forget to add that to his current cash balance. (*This means that he’s accounting for slippage*).

**IMPORTANT SideNote:** Remember that all orders are executed, at the fastest, one bar AFTER they’re placed. So if you place an order for 100 shares of AAPL at 9:31 AM, they will be executed, at the fastest, at 9:32 AM (if there’s enough volume and low market impact). This means that `context.portfolio.positions` won’t update until the next bar. You won’t believe how many questions we get on this. It’ll save you a ton of trouble if you JUST remember this one thing!

- **Checking for data and thinly traded stocks:**
  - `if stock in data` - This is one of the most important things next to slippage that you can remember. This will check if there’s currently trading activity for that security. So if you’re getting a KeyError, this is probably why!
  - `if cash > price * numshares and numshares < data[stock].volume * 0.2` - Only trade if our cash balance is greater than the amount we’re going to buy
(accounting for leverage) and only trade if the amount that we want to buy is less than 20% of all current trading volume for that security. This is brilliant! This accounts for thinly traded securities, reduces the amount of market impact, and increases the probability that we enter into positions where future liquidity is most likely good.

- **Reducing cash by amount ordered:** Lastly, Naoki reduces his starting cash balance by the amount of securities we just ordered. He has to do this because his context variables won’t be updated until his orders are actually executed, which won’t be until at least the next bar (see the SideNote).

**Summary:**
Naoki writes a simple but elegant algorithm that uses fundamental’s revenue growth value as it’s criteria for purchasing stocks. Then, he moves onto order on these decisions with a number of great trading guards in order to insure that his algorithm is entering in at the right conditions.

Next, we’re going to look at what an algorithm that incorporates a bit of statistical analysis along with fundamentals looks like.
Identifying Uptrending Volatile Small Caps by Richard Prokopyshen

This next algorithm is a bit more sophisticated in that it incorporates both a bit of fundamentals and statistical analysis to filter down securities into a buyable list. The idea of the algorithm is to trade on volatile small cap stocks. So to explain the basic structure:

1. Find small cap stocks
2. Trade on the most volatile ones that are uptrending

Richard explains the concept of the algorithm himself: “The premise of the algorithm is to use the Morningstar fundamentals data to screen for volatile small cap stocks (market cap<$2B) that are trending up, and have both positive free cash flow as well as basic eps, and are trading in a specific range. This combination of stock screening coupled with recent price activity analysis would be much more difficult to perform on other trading platforms, if even possible.

The algorithm also showcases many of the trading features possible on Quantopian, such as monitoring cash balance to limit purchases to what can be afforded, setting a trading commission model, using trailing stop orders and profit taking limit orders to improve performance while taking prudent risk management actions. The algo stays cash positive at all times as shown in the custom signals portion of the trading summary.”

```python
def before_trading_start(context):
    num_stocks = 100
    dffund = get_fundamentals(
        query(
            fundamentals.valuation.shares_outstanding,
            fundamentals.valuation.market_cap
        ).filter(fundamentals.valuation.market_cap != None)
        .filter(fundamentals.valuation.market_cap < 2000000000)
        .filter(fundamentals.valuation.shares_outstanding != None)
        .filter(fundamentals cash_flow_statement.free_cash_flow > 0)
        .filter(fundamentals earnings_report.basic_eps > 0)
        .order_by(fundamentals.valuation.market_cap.desc())
        .limit(num_stocks)
    )

    context.MyList=[]
    for S in dffund:
        if (S['market_cap'] in dffund) and (S['shares_outstanding'] in dffund):
            # Approximate price - will test in handle_data before we buy
            ShareP = dffund['market_cap']/dffund['shares_outstanding']
            if (ShareP < 25) and (ShareP > 2):
                context.MyList.append(S)

    # Update our universe
    update_universe(context.MyList)
```
Basics:
- Richard looks for the small cap stocks that have a market cap of less than 2000000000 and with a share price between $2.00 and $25.00
- He does the same filtering and ordering logic that Naoki was able to do but also performs a bit more post-analysis in order to get a list of securities to put into `update_universe`
- Unlike Naoki however, notice that Richard doesn’t put the DataFrame from `get_fundamentals` into any real `context` variable but rather, is exclusively looking for a list of securities to update his universe with.

Ordering Logic:
- After Richard get’s his universe updated on a daily basis with a few criteria from `before_trading_start`, he begins to look through each of those securities
  - He looks through to find which securities are trending upwards by finding a linear regression and detecting where `SlopeM` is > 0
If he finds one, he adds that to a list where `TrendList` is greater than 0

- Much like before, both Richard and Naoki go to great lengths to insure that they’re keeping appropriate cash limits before entering into more positions
  - Notice that Richard beautifully determines the order size by looking at his current portfolio value and comparing that against the maximum amount of holdings he’s set at `context.MaxHoldings`