Zipline Pipeline

Introduction

On any given trading day, the entire universe of stocks consists of thousands of securities. Usually, you will not be interested in investing in all the stocks in the entire universe, but rather, you will likely select only a subset of these to invest. For example, you may only want to invest in stocks that have a 10-day average closing price of $10.00 or less. Or you may only want to invest in the top 500 securities ranked by some factor.

In order to avoid spending a lot of time doing data wrangling to select only the securities you are interested in, people often use **pipelines**. In general, a pipeline is a placeholder for a series of data operations used to filter and rank data according to some factor or factors.

In this notebook, you will learn how to work with the **Zipline Pipeline**. Zipline is an open-source algorithmic trading simulator developed by *Quantopian*. We will learn how to use the Zipline Pipeline to filter stock data according to factors.

Install Packages

Loading Data with Zipline

Before we build our pipeline with Zipline, we will first see how we can load the stock data we are going to use into Zipline. Zipline uses **Data Bundles** to make it easy to use different data sources. A data bundle is a collection of pricing data, adjustment data, and an asset database. Zipline employs data bundles to preload data used to run backtests and store data for future runs. Zipline comes with a few data bundles by default but it also has the ability to ingest new bundles. The first step to using a data bundle is to ingest the data. Zipline's ingestion process will start by downloading the data or by loading data files from your local machine. It will then pass the data to a set of writer objects that converts the original data to Zipline's internal format (`bcolz` for pricing data, and `SQLite` for split/merger/dividend data) that hs been optimized for speed. This new data is written to a standard location that Zipline can find. By default, the new data is written to a subdirectory of `ZIPLINE_ROOT/data/<bundle>`, where `<bundle>` is the name given to the bundle ingested and the subdirectory is named with the current date. This allows Zipline to look at older data and run backtests on older copies of the data. Running a backtest with an old ingestion makes it easier to reproduce backtest results later.

In this notebook, we will be using stock data from **Quotemedia**. In the Udacity Workspace you will find that the stock data from Quotemedia has already been ingested into Zipline. Therefore, in the code below we will use Zipline's `bundles.load()` function to load our previously ingested stock data from Quotemedia. In order to use the `bundles.load()` function we first need to do a couple of things. First, we need to specify the name of the bundle previously ingested. In this case, the name of the Quotemedia data bundle is `eod-quotemedia`:

In [2]:
```
# Specify the bundle name
bundle_name = 'eod-quotemedia'
```

Second, we need to register the data bundle and its ingest function with Zipline, using the `bundles.register()` function. The ingest function is responsible for loading the data into memory and passing it to a set of writer objects provided by Zipline to convert the data to Zipline's internal format. Since the original Quotemedia data was contained in `.csv` files, we will use the `csvdir_equities()` function to generate the ingest function for our Quotemedia data bundle. In addition, since Quotemedia's `.csv` files contained daily stock data, we will set the time frame for our ingest function, to `daily`.

In [5]:
```
from zipline.data import bundles
from zipline.data.bundles.csvdir import csvdir_equities

# Create an ingest function
ingest_func = csvdir_equities(['daily'], bundle_name)

# Register the data bundle and its ingest function
bundles.register(bundle_name, ingest_func);
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:8: UserWarning: Overwriting bundle with name 'eod-quotemedi
a'
```

Once our data bundle and ingest function are registered, we can load our data using the `bundles.load()` function. Since this function loads our previously ingested data, we need to set `ZIPLINE_ROOT` to the path of the most recent ingested data. The most recent data is located in the `cwd/../../data/project_4_eod/` directory, where `cwd` is the current working directory. We will specify this location using the `os.environ[]` command.

In [6]:
```
import os

# Set environment variable 'ZIPLINE_ROOT' to the path where the most recent data is located
os.environ['ZIPLINE_ROOT'] = os.path.join(os.getcwd(), '..', '..', 'data', 'project_4_eod')

# Load the data bundle
bundle_data = bundles.load(bundle_name)
```

Building an Empty Pipeline

Once we have loaded our data, we can start building our Zipline pipeline. We begin by creating an empty Pipeline object using Zipline's `Pipeline` class. A Pipeline object represents a collection of named expressions to be compiled and executed by a Pipeline Engine. The `Pipeline(columns=None, screen=None)` class takes two optional parameters, `columns` and `screen`. The `columns` parameter is a dictionary used to indicate the intial columns to use, and the `screen` parameter is used to setup a screen to exclude unwanted data.

In the code below we will create a `screen` for our pipeline using Zipline's built-in `.AverageDollarVolume()` class. We will use the `.AverageDollarVolume()` class to produce a 60-day Average Dollar Volume of closing prices for every stock in our universe. We then use the `.top(10)` attribute to specify that we want to filter down our universe each day to just the top 10 assets. Therefore, this screen will act as a filter to exclude data from our stock universe each day. The average dollar volume is a good first pass filter to avoid illiquid assets.

In [7]:
```
from zipline.pipeline import Pipeline
from zipline.pipeline.factors import AverageDollarVolume

# Create a screen for our Pipeline
universe = AverageDollarVolume(window_length = 60).top(10)

# Create an empty Pipeline with the given screen
pipeline = Pipeline(screen = universe)
```

In the code above we have named our Pipeline object `pipeline` so that we can identify it later when we make computations. Remember a Pipeline is an object that represents computations we would like to perform every day. A freshly-constructed pipeline, like the one we just created, is empty. This means it doesn't yet know how to compute anything, and it won't produce any values if we ask for its outputs. In the sections below, we will see how to provide our Pipeline with expressions to compute.

Factors and Filters

The `.AverageDollarVolume()` class used above is an example of a factor. In this section we will take a look at two types of computations that can be expressed in a pipeline: **Factors** and **Filters**. In general, factors and filters represent functions that produce a value from an asset in a moment in time, but are distinguished by the types of values they produce. Let's start by looking at factors.

Factors

In general, a **Factor** is a function from an asset at a particular moment of time to a numerical value. A simple example of a factor is the most recent price of a security. Given a security and a specific moment in time, the most recent price is a number. Another example is the 10-day average trading volume of a security. Factors are most commonly used to assign values to securities which can then be combined with filters or other factors. The fact that you can combine multiple factors makes it easy for you to form new custom factors that can be as complex as you like. For example, constructing a Factor that computes the average of two other Factors can be simply illustrated usingthe pseudocode below:

```
f1 = factor1(...)
f2 = factor2(...)
average = (f1 + f2) / 2.0
```

Filters

In general, a **Filter** is a function from an asset at a particular moment in time to a boolean value (True of False). An example of a filter is a function indicating whether a security's price is below $5. Given a security and a specific moment in time, this evaluates to either **True** or **False**. Filters are most commonly used for selecting sets of securities to include or exclude from your stock universe. Filters are usually applied using comparison operators, such as <, <=, !=, ==, >, >=.

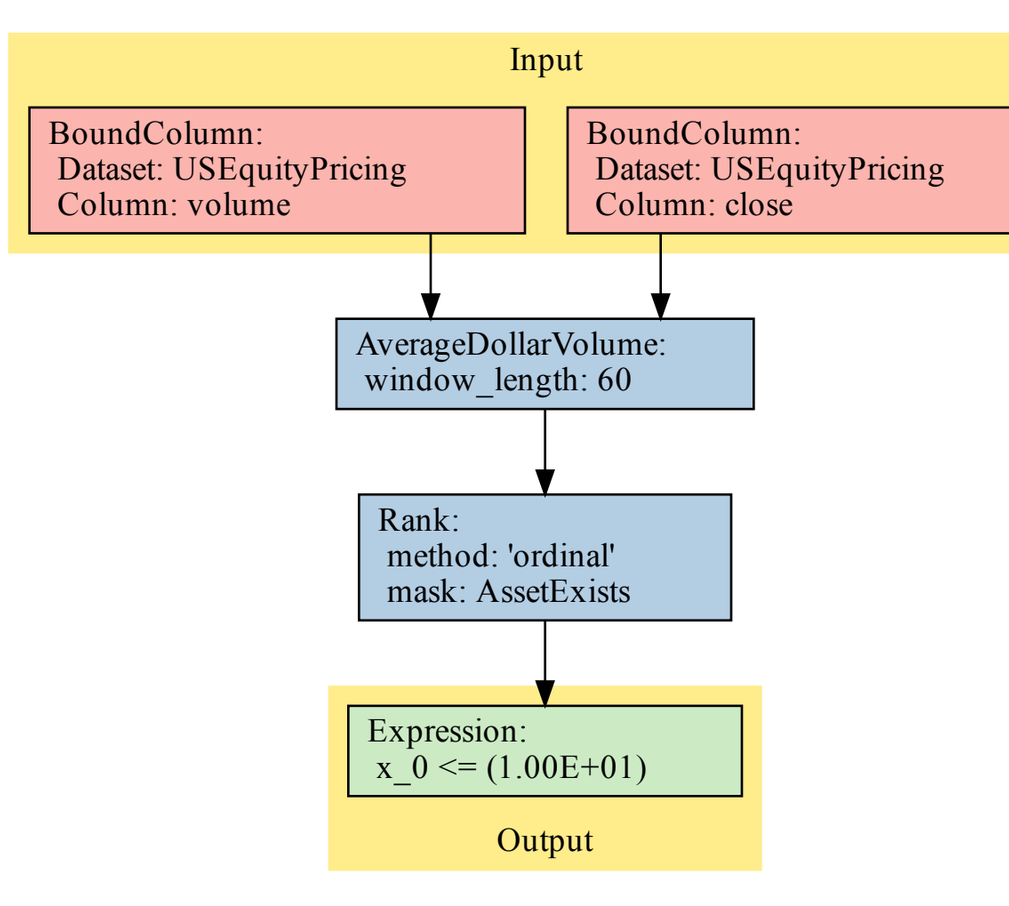
Viewing the Pipeline as a Diagram

Zipline's Pipeline class comes with the attribute `.show_graph()` that allows you to render the Pipeline as a Directed Acyclic Graph (DAG). This graph is specified using the DOT language and consequently we need a DOT graph layout program to view the rendered image. In the code below, we will use the Graphviz pakage to render the graph produced by the `.show_graph()` attribute. Graphviz is an open-source package for drawing graphs specified in DOT language scripts.

In [8]:
```
import graphviz

# Render the pipeline as a DAG
pipeline.show_graph()
```

Out[8]:



Right now, our pipeline is empty and it only contains a screen. Therefore, when we rendered our `pipeline`, we only see the diagram of our `screen`:

```
AverageDollarVolume(window_length = 60).top(10)
```

By default, the `.AverageDollarVolume()` class uses the `USEquityPricing` dataset, containing daily trading prices and volumes, to compute the average dollar volume:

```
average_dollar_volume = np.nansum(close_price * volume, axis=0) / len(close_price)
```

The top of the diagram reflects the fact that the `.AverageDollarVolume()` class gets its inputs (closing price and volume) from the `USEquityPricing` dataset. The bottom of the diagram shows that the output is determined by the expression `x_0 <= 10`. This expression reflects the fact that we used `.top(10)` as a filter in our `screen`. We refer to each box in the diagram as a Term.

Datasets and Dataloaders

One of the features of Zipline's Pipeline is that it separates the actual source of the stock data from the abstract description of that dataset. Therefore, Zipline employs **DataSets** and **Loaders** for those datasets. `DataSets` are just abstract collections of sentinel values describing the columns/types for a particular dataset. While a `loader` is an object which, given a request for a particular chunk of a dataset, can actually get the requested data. For example, the loader used for the `USEquityPricing` dataset, is the `USEquityPricingLoader` class. The `USEquityPricingLoader` class will delegate the loading of baselines and adjustments to lower-level subsystems that know how to get the pricing data in the default formats used by Zipline (`bcolz` for pricing data, and `SQLite` for split/merger/dividend data). As we saw in the beginning of this notebook, data bundles automatically convert the stock data into `bcolz` and `SQLite` formats. It is important to note that the `USEquityPricingLoader` class can also be used to load daily OHLCV data from other datasets, not just from the `USEquityPricing` dataset. Simliarly, it is also possible to write different loaders for the same dataset and use those instead of the default loader. Zipline contains lots of other loaders to allow you to load data from different datasets.

In the code below, we will use `USEquityPricingLoader(BcolzDailyBarWriter, SQLiteAdjustmentWriter)` to create a loader from a `bcolz` equity pricing directory and a `SQLite` adjustments path. Both the `BcolzDailyBarWriter` and `SQLiteAdjustmentWriter` determine the path of the pricing and adjustment data. Since we will be using the Quotemedia data bundle, we will use the `bundle_data.equity_daily_bar_reader` and the `bundle_data.adjustment_reader` as our `BcolzDailyBarWriter` and `SQLiteAdjustmentWriter`, respectively.

In [9]:
```
from zipline.pipeline.loaders import USEquityPricingLoader

# Set the dataloader
pricing_loader = USEquityPricingLoader(bundle_data.equity_daily_bar_reader, bundle_data.adjustment_reader)
```

Pipeline Engine

Zipline employs computation engines for executing Pipelines. In the code below we will use Zipline's `SimplePipelineEngine()` class as the engine to execute our pipeline. The `SimplePipelineEngine(get_loader, calendar, asset_finder)` class associates the chosen data loader with the corresponding dataset and a trading calendar. The `get_loader` parameter must be a callable function that is given a loadable term and returns a `PipelineLoader` to use to retrieve the raw data for that term in the pipeline. In our case, we will be using the `pricing_loader` defined above, we therefore, create a function called `choose_loader` that returns our `pricing_loader`. The function also checks that the data that is being requested corresponds to OHLCV data, otherwise it retunrs an error. The `calendar` parameter must be a `DatetimeIndex` array of dates to consider as trading days when computing a range between a fixed `start_date` and `end_date`. In our case, we will be using the same trading days as those used in the NYSE. We will use Zipline's `get_calendar('NYSE')` function to retrieve the trading days used by the NYSE. We then use the `.all_sessions` attribute to get the `DatetimeIndex` from our `trading_calendar` and pass it to the `calendar` parameter. Finally, the `asset_finder` parameter determines which assets are in the top-level universe of our stock data at any point in time. Since we are using the Quotemedia data bundle, we set this parameter to the `bundle_data.asset_finder`.

In [10]:
```
from zipline.utils.calendars import get_calendar
from zipline.pipeline.data import USEquityPricing
from zipline.pipeline.engine import SimplePipelineEngine

# Define the function for the get_loader parameter
def choose_loader(column):
    if column not in USEquityPricing.columns:
        raise Exception('Column not in USEquityPricing')
    return pricing_loader

# Set the trading calendar
trading_calendar = get_calendar('NYSE')

# Create a Pipeline engine
engine = SimplePipelineEngine(get_loader = choose_loader,
                              calendar = trading_calendar.all_sessions,
                              asset_finder = bundle_data.asset_finder)
```

Running a Pipeline

Once we have chosen our engine we are ready to run or execute our pipeline. We can run our pipeline by using the `.run_pipeline()` attribute of the `SimplePipelineEngine` class. In particular, the `SimplePipelineEngine.run_pipeline(pipeline, start_date, end_date)` implements the following algorithm for executing pipelines:

1. Build a dependency graph of all terms in the `pipeline`. In this step, the graph is sorted topologically to determine the order in which we can compute the terms.

1. Ask our AssetFinder for a "lifetimes matrix", which should contain, for each date between `start_date` and `end_date`, a boolean value for each known asset indicating whether the asset existed on that date.

1. Compute each term in the dependency order determined in step 1, caching the results in a a dictionary so that they can be fed into future terms.

1. For each date, determine the number of assets passing the `pipeline` screen. The sum, N, of all these values is the total number of rows in our output Pandas Dataframe, so we pre-allocate an output array of length N for each factor in terms.

1. Fill in the arrays allocated in step 4 by copying computed values from our output cache into the corresponding rows.

1. Stick the values computed in step 5 into a Pandas DataFrame and return it.

In the code below, we run our pipeline for a single day, so our `start_date` and `end_date` will be the same. We then print some information about our `pipeline_output`.

In [11]:
```
import pandas as pd

# Set the start and end dates
start_date = pd.Timestamp('2016-01-05', tz = 'utc')
end_date = pd.Timestamp('2016-01-05', tz = 'utc')

# Run our pipeline for the given start and end dates
pipeline_output = engine.run_pipeline(pipeline, start_date, end_date)

# We print information about the pipeline output
print('The pipeline output has type:', type(pipeline_output), '\n')

# We print whether the pipeline output is a MultiIndex Dataframe
print('Is the pipeline output a MultiIndex Dataframe:', isinstance(pipeline_output.index, pd.core.index.MultiIndex), '\n')

# If the pipeline output is a MultiIndex Dataframe we print the two levels of the index
if isinstance(pipeline_output.index, pd.core.index.MultiIndex):

    # We print the index level 0
    print('Index Level 0:\n\n', pipeline_output.index.get_level_values(0), '\n')

    # We print the index level 1
    print('Index Level 1:\n\n', pipeline_output.index.get_level_values(1), '\n')
```

```
The pipeline output has type: <class 'pandas.core.frame.DataFrame'>

Is the pipeline output a MultiIndex Dataframe: True

Index Level 0:

 DatetimeIndex(['2016-01-05', '2016-01-05', '2016-01-05', '2016-01-05',
               '2016-01-05', '2016-01-05', '2016-01-05', '2016-01-05',
               '2016-01-05', '2016-01-05'],
              dtype='datetime64[ns, UTC]', freq=None)

Index Level 1:

 Index([   Equity(3 [AAPL]),    Equity(19 [AGN]),   Equity(38 [AMZN]),
          Equity(59 [BAC]),    Equity(173 [FB]),    Equity(192 [GE]),
        Equity(198 [GOOG]), Equity(199 [GOOGL]),  Equity(312 [MSFT]),
        Equity(323 [NFLX])],
      dtype='object')
```

We can see above that the return value of `.run_pipeline()` is a `MultiIndex` Pandas DataFrame containing a row for each asset that passed our pipeline's screen. We can also see that the 0th level of the index contains the date and the 1st level of the index contains the tickers. In general, the returned Pandas DataFrame will also contain a column for each factor and filter we add to the pipeline using `Pipeline.add()`. At this point we haven't added any factors or filters to our pipeline, consequently, the Pandas Dataframe will have no columns. In the following sections we will see how to add factors and filters to our pipeline.

Get Tickers

We saw in the previous section, that the tickers of the stocks that passed our pipeline's screen are contained in the 1st level of the index. Therefore, we can use the Pandas `.get_level_values(1).values.tolist()` method to get the tickers of those stocks and save them to a list.

In [12]:
```
# Get the values in index level 1 and save them to a list
universe_tickers = pipeline_output.index.get_level_values(1).values.tolist()

# Display the tickers
universe_tickers
```

Out[12]:
```
[Equity(3 [AAPL]),
 Equity(19 [AGN]),
 Equity(38 [AMZN]),
 Equity(59 [BAC]),
 Equity(173 [FB]),
 Equity(192 [GE]),
 Equity(198 [GOOG]),
 Equity(199 [GOOGL]),
 Equity(312 [MSFT]),
 Equity(323 [NFLX])]
```

Get Data

Now that we have the tickers for the stocks that passed our pipeline's screen, we can get the historical stock data for those tickers from our data bundle. In order to get the historical data we need to use Zipline's `DataPortal` class. A `DataPortal` is an interface to all of the data that a Zipline simulation needs. In the code below, we will create a `DataPortal` and `get_pricing` function to get historical stock prices for our tickers.

We have already seen most of the parameters used below when we create the `DataPortal`, so we won't explain them again here. The only new parameter is `first_trading_day`. The `first_trading_day` parameter is a `pd.Timestamp` indicating the first trading day for the simulation. We will set the first trading day to the first trading day in the data bundle. For more information on the `DataPortal` class see the Zipline documentation

In [13]:
```
from zipline.data.data_portal import DataPortal

# Create a data portal
data_portal = DataPortal(bundle_data.asset_finder,
                         trading_calendar = trading_calendar,
                         first_trading_day = bundle_data.equity_daily_bar_reader.first_trading_day,
                         equity_daily_reader = bundle_data.equity_daily_bar_reader,
                         adjustment_reader = bundle_data.adjustment_reader)
```

Now that we have created a `data_portal` we will create a helper function, `get_pricing`, that gets the historical data from the `data_portal` for a given set of `start_date` and `end_date`. The `get_pricing` function takes various parameters:

```
def get_pricing(data_portal, trading_calendar, assets, start_date, end_date, field='close')
```

The first two parameters, `data_portal` and `trading_calendar`, have already been defined above. The third paramter, `assets`, is a list of tickers. In our case we will use the tickers from the output of our pipeline, namely, `universe_tickers`. The fourth and fifth parameters are strings specifying the `start_date` and `end_date`. The function converts these two strings into Timestamps with a Custom Business Day frequency. The last parameter, `field`, is a string used to indicate which field to return. In our case we want to get the closing price, so we set `field='close'`.

The function returns the historical stock price data using the `.get_history_window()` attribute of the `DataPortal` class. This attribute returns a Pandas Dataframe containing the requested history window with the data fully adjusted. The `bar_count` parameter is an integer indicating the number of days to return. The number of days determines the number of rows of the returned dataframe. Both the `frequency` and `data_frequency` parameters are strings that indicate the frequency of the data to query, *i.e.* whether the data is in `daily` or `minute` intervals.

In [14]:
```
def get_pricing(data_portal, trading_calendar, assets, start_date, end_date, field='close'):

    # Set the given start and end dates to Timestamps. The frequency string C is used to
    # indicate that a CustomBusinessDay DateOffset is used
    end_dt = pd.Timestamp(end_date, tz='UTC', freq='C')
    start_dt = pd.Timestamp(start_date, tz='UTC', freq='C')

    # Get the locations of the start and end dates
    end_loc = trading_calendar.closes.index.get_loc(end_dt)
    start_loc = trading_calendar.closes.index.get_loc(start_dt)

    # return the historical data for the given window
    return data_portal.get_history_window(assets=assets, end_dt=end_dt, bar_count=end_loc - start_loc,
                                          frequency='1d',
                                          field=field,
                                          data_frequency='daily')

# Get the historical data for the given window
historical_data = get_pricing(data_portal, trading_calendar, universe_tickers,
                              start_date='2011-01-05', end_date='2016-01-05')
# Display the historical data
historical_data
```

Out[14]:

	Equity(3 [AAPL])	Equity(19 [AGN])	Equity(38 [AMZN])	Equity(59 [BAC])	Equity(173 [FB])	Equity(192 [GE])	Equity(198 [GOOG])	Equity(199 [GOOGL])	Equity(312 [MSFT])	Equity(323 [NFLX])
2011-01-06 00:00:00+00:00	42.399	50.648	185.86	13.456	NaN	14.472	NaN	307.699	23.695	25.427
2011-01-07 00:00:00+00:00	42.702	51.440	185.49	13.279	NaN	14.371	NaN	309.174	23.515	25.614
2011-01-10 00:00:00+00:00	43.507	52.251	184.68	13.419	NaN	14.433	NaN	308.055	23.202	26.840
2011-01-11 00:00:00+00:00	43.404	51.577	184.34	13.689	NaN	14.527	NaN	308.958	23.112	26.664
2011-01-12 00:00:00+00:00	43.757	50.912	184.08	13.968	NaN	14.558	NaN	309.390	23.473	26.985
2011-01-13 00:00:00+00:00	43.917	51.137	185.53	13.763	NaN	14.503	NaN	309.299	23.177	27.355
2011-01-14 00:00:00+00:00	44.273	50.707	188.75	14.211	NaN	14.673	NaN	313.056	23.268	27.354
2011-01-18 00:00:00+00:00	43.278	52.192	191.25	13.978	NaN	14.503	NaN	320.805	23.564	27.668
2011-01-19 00:00:00+00:00	43.048	51.264	186.87	13.391	NaN	14.293	NaN	316.853	23.408	27.267
2011-01-20 00:00:00+00:00	42.265	51.684	181.96	13.549	NaN	14.371	NaN	314.355	23.311	26.428
2011-01-21 00:00:00+00:00	41.508	52.877	177.42	13.279	NaN	15.392	NaN	306.862	23.038	26.012
2011-01-24 00:00:00+00:00	42.871	53.336	176.85	12.971	NaN	15.626	NaN	306.486	23.334	26.280
2011-01-25 00:00:00+00:00	43.373	53.336	176.70	12.701	NaN	15.580	NaN	310.914	23.391	26.677
2011-01-26 00:00:00+00:00	43.685	54.636	175.39	12.627	NaN	15.533	NaN	309.204	23.663	26.147
2011-01-27 00:00:00+00:00	43.603	53.502	184.45	12.738	NaN	15.813	NaN	309.349	23.737	30.124
2011-01-28 00:00:00+00:00	42.700	52.339	171.14	12.673	NaN	15.751	NaN	301.425	22.816	31.140
2011-01-31 00:00:00+00:00	43.109	53.297	169.64	12.794	NaN	15.704	NaN	301.109	22.795	30.582
2011-02-01 00:00:00+00:00	43.834	53.737	172.11	13.335	NaN	16.219	NaN	306.466	23.015	30.414
2011-02-02 00:00:00+00:00	43.744	53.698	173.53	13.270	NaN	16.149	NaN	306.947	22.972	30.180
2011-02-03 00:00:00+00:00	43.632	54.539	173.71	13.447	NaN	16.180	NaN	306.019	22.733	30.212
2011-02-04 00:00:00+00:00	44.021	54.177	175.93	13.316	NaN	16.032	NaN	306.435	22.832	31.438
2011-02-07 00:00:00+00:00	44.705	54.529	176.43	13.670	NaN	16.274	NaN	308.098	23.184	31.145
2011-02-08 00:00:00+00:00	45.127	53.981	183.06	13.614	NaN	16.593	NaN	310.147	23.253	31.089
2011-02-09 00:00:00+00:00	45.503	54.568	185.30	13.642	NaN	16.617	NaN	309.204	22.997	31.755
2011-02-10 00:00:00+00:00	45.043	55.194	186.21	13.503	NaN	16.585	NaN	309.174	22.610	31.885
2011-02-11 00:00:00+00:00	45.336	55.878	189.25	13.763	NaN	16.632	NaN	313.216	22.405	33.010
2011-02-14 00:00:00+00:00	45.632	55.428	190.42	13.875	NaN	16.765	NaN	315.047	22.388	35.364
2011-02-15 00:00:00+00:00	45.724	55.223	189.03	13.763	NaN	16.734	NaN	313.041	22.298	34.398
2011-02-16 00:00:00+00:00	46.134	55.184	186.62	13.829	NaN	16.718	NaN	313.076	22.347	33.960
2011-02-17 00:00:00+00:00	45.520	54.715	187.76	13.801	NaN	16.780	NaN	313.598	22.504	33.661
...	...	...	...	...	...	...	...	...	...	...
2015-11-20 00:00:00+00:00	113.945	305.661	668.45	16.931	107.32	28.068	756.60	777.000	51.142	123.840
2015-11-23 00:00:00+00:00	112.465	295.154	678.99	16.758	106.95	28.004	755.98	776.700	51.142	125.030
2015-11-24 00:00:00+00:00	113.544	304.653	671.15	16.758	105.74	28.068	748.28	769.630	51.198	123.310
2015-11-25 00:00:00+00:00	112.732	313.291	675.34	16.729	105.41	27.793	748.15	769.260	50.670	124.160
2015-11-27 00:00:00+00:00	112.522	312.802	673.26	16.768	105.45	27.793	750.26	771.970	50.896	125.440
2015-11-30 00:00:00+00:00	112.990	307.060	664.80	16.720	104.24	27.409	742.60	762.850	51.293	123.330
2015-12-01 00:00:00+00:00	112.073	315.472	679.06	17.084	107.12	27.619	767.04	783.790	52.114	125.370
2015-12-02 00:00:00+00:00	111.061	310.552	676.01	16.950	106.07	27.436	762.38	777.850	52.104	128.930
2015-12-03 00:00:00+00:00	110.029	306.140	666.25	16.642	104.38	27.491	752.54	768.200	51.151	126.810
2015-12-04 00:00:00+00:00	113.687	309.613	672.64	17.123	106.18	27.912	766.81	779.210	52.765	130.930
2015-12-07 00:00:00+00:00	112.971	309.623	669.83	16.873	105.61	27.802	763.25	772.990	52.670	125.360
2015-12-08 00:00:00+00:00	112.923	303.714	677.33	16.536	106.49	27.637	762.37	775.140	52.652	126.980
2015-12-09 00:00:00+00:00	110.430	299.742	664.79	16.450	104.60	27.894	751.61	762.550	51.887	124.200
2015-12-10 00:00:00+00:00	110.956	299.048	662.32	16.546	105.42	28.058	749.46	760.040	52.161	122.910
2015-12-11 00:00:00+00:00	108.100	295.497	640.15	16.094	102.12	27.701	738.87	750.420	51.019	118.910
2015-12-14 00:00:00+00:00	107.431	296.426	657.91	16.161	104.66	27.701	747.77	762.540	52.038	120.670
2015-12-15 00:00:00+00:00	105.531	298.030	658.64	16.758	104.55	27.756	743.40	760.090	52.095	118.600
2015-12-16 00:00:00+00:00	106.342	302.344	675.77	17.075	106.79	28.361	758.09	776.590	52.972	122.640
2015-12-17 00:00:00+00:00	104.088	302.775	670.65	16.642	106.22	28.177	749.43	769.830	52.567	122.510
2015-12-18 00:00:00+00:00	101.271	299.909	664.14	16.123	104.04	27.928	739.31	756.850	51.085	118.020
2015-12-21 00:00:00+00:00	102.512	303.176	664.51	16.325	104.77	28.039	747.77	760.800	51.746	116.630
2015-12-22 00:00:00+00:00	102.417	303.831	663.15	16.431	105.51	28.122	750.00	767.130	52.236	116.240
2015-12-23 00:00:00+00:00	103.735	303.812	663.70	16.681	104.63	28.546	750.31	768.510	52.680	118.160
2015-12-24 00:00:00+00:00	103.181	304.457	662.79	16.613	105.02	28.436	748.40	765.840	52.538	117.330
2015-12-28 00:00:00+00:00	102.025	302.159	675.20	16.479	105.93	28.500	762.51	782.240	52.803	117.110
2015-12-29 00:00:00+00:00	103.859	307.647	693.97	16.623	107.26	28.851	776.60	793.960	53.369	119.120
2015-12-30 00:00:00+00:00	102.503	308.204	689.07	16.402	106.22	28.639	771.00	790.300	53.142	116.710
2015-12-31 00:00:00+00:00	100.535	305.700	675.89	16.190	104.66	28.731	758.88	778.010	52.359	114.380
2016-01-04 00:00:00+00:00	100.621	300.779	636.99	15.805	102.22	28.325	741.84	759.440	51.717	109.960
2016-01-05 00:00:00+00:00	98.100	302.012	633.79	15.805	102.73	28.353	742.58	761.530	51.953	107.660

1257 rows × 10 columns

Date Alignment

When pipeline returns with a date of, e.g., `2016-01-07` this includes data that would be known as of before the **market open** on `2016-01-07`. As such, if you ask for latest known values on each day, it will return the closing price from the day before and label the date `2016-01-07`. All factor values assume to be run prior to the open on the labeled day with data known before that point in time.

Adding Factors and Filters

Now that you know how build a pipeline and execute it, in this section we will see how we can add factors and filters to our pipeline. These factors and filters will determine the computations we want our pipeline to compute each day.

We can add both factors and filters to our pipeline using the `.add(column, name)` method of the `Pipeline` class. The `column` parameter represetns the factor or filter to add to the pipeline. The `name` parameter is a string that determines the name of the column in the output Pandas Dataframe for that factor of fitler. As mentioned earlier, each factor and filter will appear as a column in the output dataframe of our pipeline. Let's start by adding a factor to our pipeline.

Factors

In the code below, we will use Zipline's built-in `SimpleMovingAverage` factor to create a factor that computes the 15-day mean closing price of securities. We will then add this factor to our pipeline and use `.show_graph()` to see a diagram of our pipeline with the factor added.

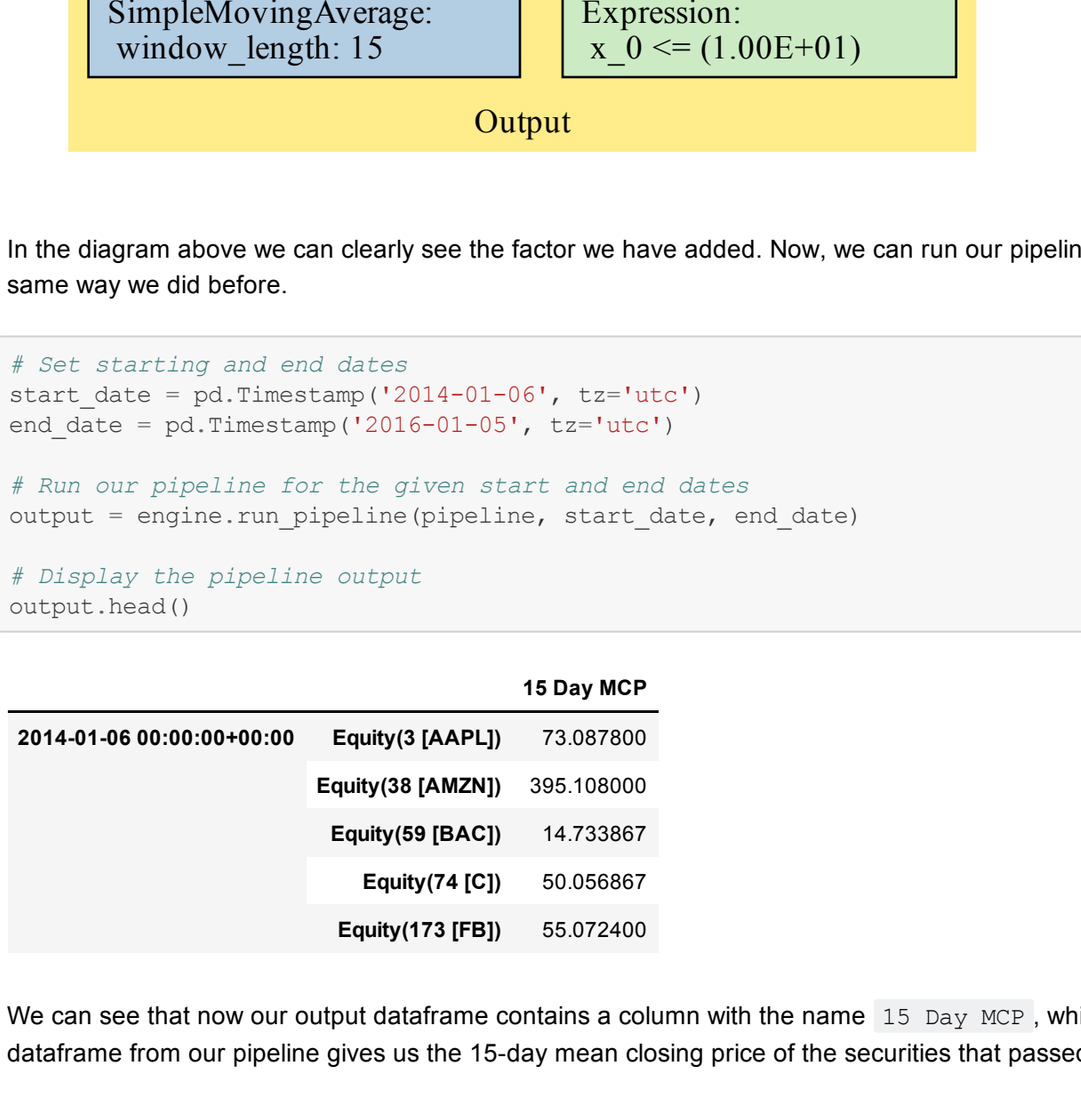
In [15]:
```
from zipline.pipeline.factors import SimpleMovingAverage

# Create a factor that computes the 15-day mean closing price of securities
mean_close_15 = SimpleMovingAverage(inputs = [USEquityPricing.close], window_length = 15)

# Add the factor to our pipeline
pipeline.add(mean_close_15, '15 Day MCP')

# Render the pipeline as a DAG
pipeline.show_graph()
```

Out[15]:



In the diagram above we can clearly see the factor we have added. Now, we can run our pipeline again and see its output. The pipeline is run in exactly the same way we did before.

In [16]:
```
# Set starting and end dates
start_date = pd.Timestamp('2014-01-06', tz='utc')
end_date = pd.Timestamp('2016-01-05', tz='utc')

# Run our pipeline for the given start and end dates
output = engine.run_pipeline(pipeline, start_date, end_date)

# Display the pipeline output
output.head()
```

Out[16]:

		15 Day MCP
2014-01-06 00:00:00+00:00	Equity(3 [AAPL])	73.087800
	Equity(38 [AMZN])	395.108000
	Equity(59 [BAC])	14.733867
	Equity(74 [C])	50.056867
	Equity(173 [FB])	55.072400

We can see that now our output dataframe contains a column with the name `15 Day MCP`, which is the name we gave to our factor before. This ouput dataframe from our pipeline gives us the 15-day mean closing price of the securities that passed our `screen`.

Filters

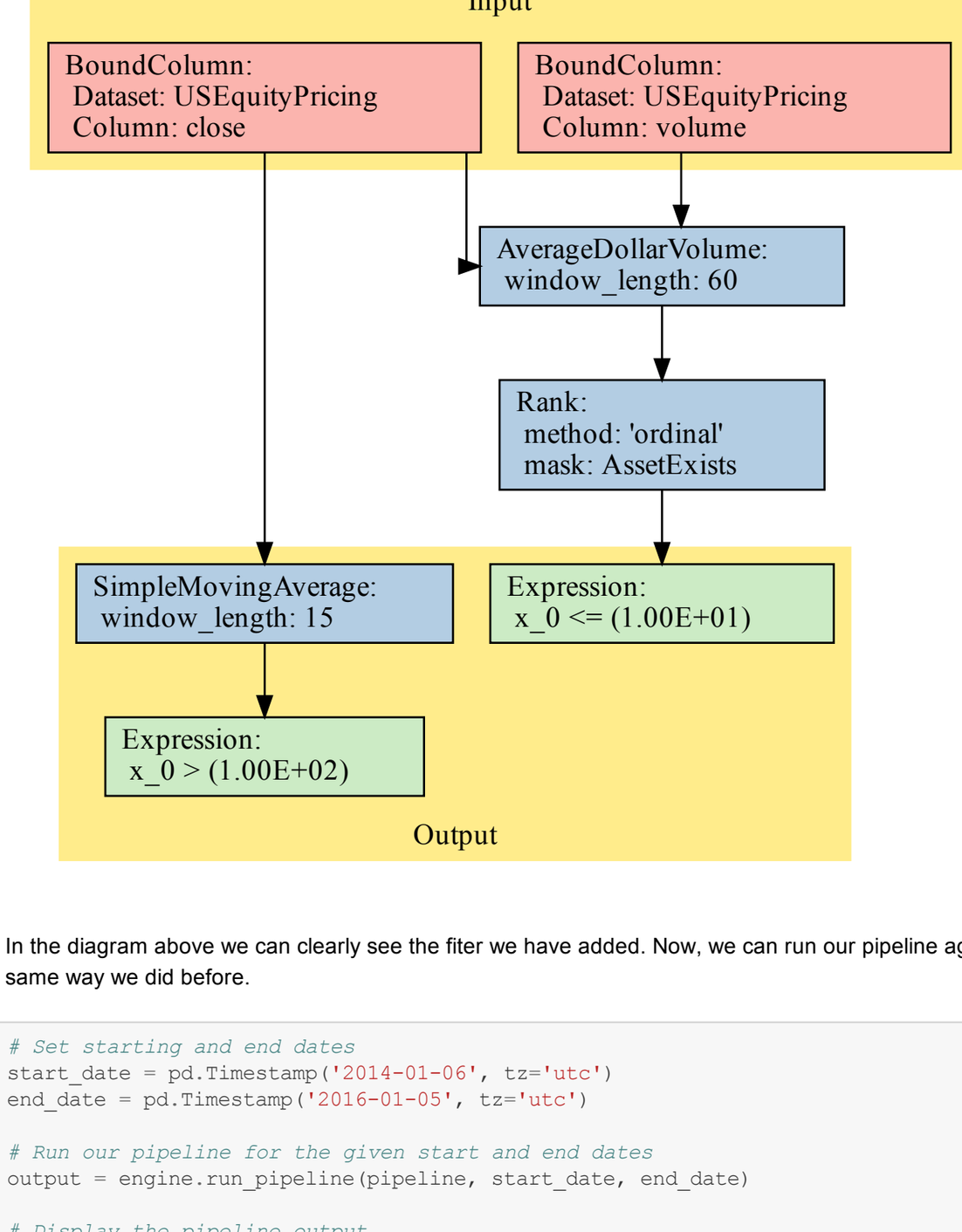
Filters are created and added to the pipeline in the same way as factors. In the code below, we create a filter that returns `True` whenever the 15-day average closing price is above $100. Remember, a filter produces a `True` or `False` value for each security every day. We will then add this filter to our pipeline and use `.show_graph()` to see a diagram of our pipeline with the filter added.

In [17]:
```
# Create a Filter that returns True whenever the 15-day average closing price is above $100
high_mean = mean_close_15 > 100

# Add the filter to our pipeline
pipeline.add(high_mean, 'High Mean')

# Render the pipeline as a DAG
pipeline.show_graph()
```

Out[17]:



In the diagram above we can clearly see the fiter we have added. Now, we can run our pipeline again and see its output. The pipeline is run in exactly the same way we did before.

In [18]:
```
# Set starting and end dates
start_date = pd.Timestamp('2014-01-06', tz='utc')
end_date = pd.Timestamp('2016-01-05', tz='utc')

# Run our pipeline for the given start and end dates
output = engine.run_pipeline(pipeline, start_date, end_date)

# Display the pipeline output
output.head()
```

Out[18]:

		15 Day MCP	High Mean
2014-01-06 00:00:00+00:00	Equity(3 [AAPL])	73.087800	False
	Equity(38 [AMZN])	395.108000	True
	Equity(59 [BAC])	14.733867	False
	Equity(74 [C])	50.056867	False
	Equity(173 [FB])	55.072400	False

We can see that now our output dataframe contains a two columns, one for the filter and one for the factor. The new column has the name `High Mean`, which is the name we gave to our filter before. Notice that the filter column only contains Boolean values, where only the securities with a 15-day average closing price above $100 have `True` values.