7

Event-Based Backtesting Factor Portfolios with Zipline Reloaded

Zipline Reloaded is an event-driven backtesting framework that processes market events sequentially, allowing for more realistic modeling of order execution and slippage. Unlike vector-based frameworks, it accounts for the temporal sequence of market events, making it suitable for complex strategies that involve conditional orders or asset interactions. While generally slower than vector-based approaches, event-based backtesting frameworks tend to better simulate market dynamics making them helpful for path-dependent strategies requiring intricate order logic, state management, and risk management.

Zipline Reloaded is well suited for backtesting large universes and complex portfolio construction techniques. The Pipeline API is designed for high-efficiency computation of factors among thousands of securities. We'll use Zipline Reloaded to backtest portfolio factor strategies, the results of which can be analyzed with other tools in the Zipline Reloaded ecosystem..

In this chapter, we present the following recipes:

- Backtesting a momentum factor strategy with Zipline Reloaded
- Exploring a mean reversion strategy with Zipline Reloaded

Technical Requirements

We installed Zipline Reloaded in *Chapter 5*, *Build Alpha Factors for Stock Portfolios*. In case you missed it, follow along with the instructions here. The steps to install Zipline Reloaded differ depending on your operating system.

For Windows, Unix/Linux, and Mac Intel users

If you're running on an Intel x86 chip, you can use **conda**:

```
conda install -c conda-forge zipline-reloaded pyfolio-reloaded alphalens-reloaded -y
```

For Mac M1/M2 users

If you have a Mac with an M1 or M2 chip, you need to install some dependencies first. The easiest way is to use Homebrew (https://brew.sh).

Install the dependencies with Homebrew:

```
brew install freetype pkg-config gcc openssl hdf5 ta-lib
```

Install the Python dependencies with **conda**:

```
conda install -c conda-forge pytables h5py -y
```

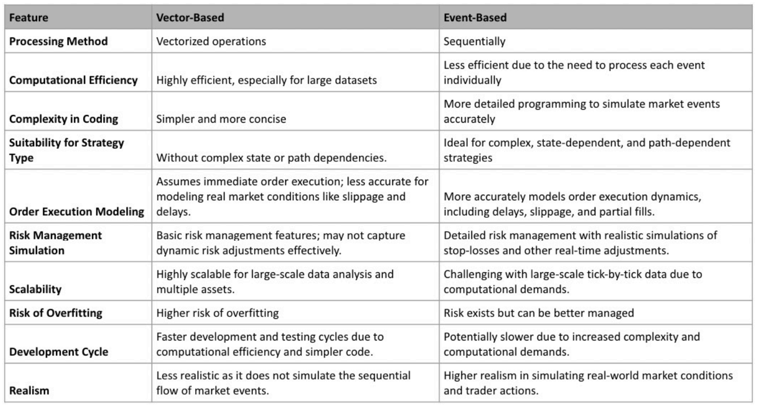
Install the Zipline Reloaded ecosystem:

```
pip install zipline-reloaded pyfolio-reloaded alphalens-reloaded
```

In this example, we'll use the free data bundle provided by Nasdaq Data Link. This dataset has market price data on 3,000 stocks through 2018.

Backtesting a momentum factor strategy with Zipline Reloaded

Before we begin, it's important to understand the difference between vector-based backtesting frameworks and event-based backtesting frameworks. In ***Chapter 6***, *Vector-Based Backtesting with VectorBT*, we touched on the differences between vector-based and event-based backtesting. Here is a more detailed assessment:



Feature	Vector-Based	Event-Based
Processing Method	Vectorized operations	Sequentially
Computational Efficiency	Highly efficient, especially for large datasets	Less efficient due to the need to process each event individually
Complexity in Coding	Simpler and more concise	More detailed programming to simulate market events accurately
Suitability for Strategy Type	Without complex state or path dependencies.	Ideal for complex, state-dependent, and path-dependent strategies
Order Execution Modeling	Assumes immediate order execution; less accurate for modeling real market conditions like slippage and delays.	More accurately models order execution dynamics, including delays, slippage, and partial fills.
Risk Management Simulation	Basic risk management features; may not capture dynamic risk adjustments effectively.	Detailed risk management with realistic simulations of stop-losses and other real-time adjustments.
Scalability	Highly scalable for large-scale data analysis and multiple assets.	Challenging with large-scale tick-by-tick data due to computational demands.
Risk of Overfitting	Higher risk of overfitting	Risk exists but can be better managed
Development Cycle	Faster development and testing cycles due to computational efficiency and simpler code.	Potentially slower due to increased complexity and computational demands.
Realism	Less realistic as it does not simulate the sequential flow of market events.	Higher realism in simulating real-world market conditions and trader actions.

Figure 8.1: Comparison of vector-based and event-based backtesting frameworks

In ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*, we defined a custom factor that computes a momentum score for US equities over a 252-day window. The factor's value is determined by comparing the 252-day relative price change against the 22-day relative price change. This difference is then standardized by dividing it by the standard deviation of the 126-day returns, effectively scaling the momentum score by the asset's recent volatility. The result provides a normalized momentum score for each asset, capturing both long-term and short-term price movements in relation to its recent volatility.

In this recipe, we'll incorporate the factor into the Zipline Reloaded backtesting framework and inspect the performance of the strategy.

Getting ready

We assume you ingested the free **quandl** data bundle and it's still available on your local machine. We also assume you still have your Nasdaq API key in the environment variables. If not, run the following code after the imports:

```
from zipline.data.bundles.core import load
os.environ["QUANDL_API_KEY"] = "YOUR_API_KEY"
bundle_data = load("quandl", os.environ, None)
```

The free data is limited to about 3,000 US equities and the data collection stops in 2018. If you'd like data coverage for nearly 20,000 US equities updated daily, you can consider the premium data service offered by Nasdaq.

The instructions to set up this premium data service are outlined in detail in the article *How to ingest premium market data with Zipline Reloaded*, which you can find at this URL: https://www.pyquantnews.com/free-python-resources/how-to-ingest-premium-market-data-with-zipline-reloaded.

If you use this premium data, replace **quandl** with **quotemedia** in the **run_algorithm** function that follows.

How to do it...

In this recipe, we introduce several novel features of the Zipline Reloaded backtesting framework that require their own imports — namely, the date and time rules, the Pipeline API, custom factors, and commission and slippage models:

1. Start by importing the libraries we'll need for the backtest:

```
import pandas as pd
import numpy as np
from zipline import run_algorithm
from zipline.pipeline import Pipeline
from zipline.pipeline.data import USEquityPricing
from zipline.pipeline.factors import AverageDollarVolume, CustomFactor, Returns
from zipline.api import (
    attach_pipeline,
    calendars,
    pipeline_output,
    date_rules,
    time_rules,
    set_commission,
    set_slippage,
    record,
    order_target_percent,
    get_open_orders,
    get_datetime,
    schedule_function
)
import pandas_datareader as web
```

2. Next, define the number of long and short stocks we want in our portfolio:

```
N_LONGS = N_SHORTS = 50
```

3. Use the same custom momentum factor we defined in the previous chapter:

```
class MomentumFactor(CustomFactor):
    inputs = [USEquityPricing.close,
        Returns(window_length=126)]
    window_length = 252
    def compute(self, today, assets, out, prices, returns):
        out[:] = (
            (prices[-21] -prices[-252]) / prices[-252]
            - (prices[-1] - prices[-21]) / prices[-21]
        ) / np.nanstd(returns, axis=0)
```

4. Use the same pipeline we defined in the previous chapter:

```
def make_pipeline():
    momentum = MomentumFactor()
    dollar_volume = AverageDollarVolume(
        window_length=30)
        return Pipeline(
            columns={
                "factor": momentum,
                "longs": momentum.top(N_LONGS),
                "shorts": momentum.bottom(N_SHORTS),
                "ranking": momentum.rank(),
            },
            screen=dollar_volume.top(100),
        )
```

5. Zipline Reloaded is an event-driven backtesting framework that allows us to “hook” into different events, including an event that fires before trading starts. We use this hook to “install” our factor pipeline:

```
def before_trading_start(context, data):
    context.factor_data = pipeline_output(
        "factor_pipeline")
```

6. Next, we define the **`initialize`** function, which is run when the back-test starts:

```
def initialize(context):
    attach_pipeline(make_pipeline(),
        "factor_pipeline")
    schedule_function(
        rebalance,
        date_rules.week_start(),
        time_rules.market_open(),
        calendar=calendars.US_EQUITIES,
    )
```

7. Now, define a function that contains the logic to rebalance our portfolio. Here we buy the top **`N_LONGS`** stocks with the highest ranking factor and short the bottom **`N_SHORTS`** stocks with the lowest ranking factor:

```
def rebalance(context, data):
    factor_data = context.factor_data
    record(factor_data=factor_data.ranking)
    assets = factor_data.index
    record(prices=data.current(assets, "price"))
    longs = assets[factor_data.longs]
    shorts = assets[factor_data.shorts]
    divest = set(
        context.portfolio.positions.keys()) - set(
            longs.union(shorts))
    exec_trades(
        data,
        assets=divest,
        target_percent=0
    )
    exec_trades(
        data,
        assets=longs,
        target_percent=1 / N_LONGS
    )
    exec_trades(
        data,
        assets=shorts,
        target_percent=-1 / N_SHORTS
    )
```

8. We abstract away the order execution in an **`exec_trades`** function, which loops through the provided assets and executes the orders:

```
def exec_trades(data, assets, target_percent):
    for asset in assets:
        if data.can_trade(
            asset) and not get_open_orders(asset):
                order_target_percent(asset,
                    target_percent)
```

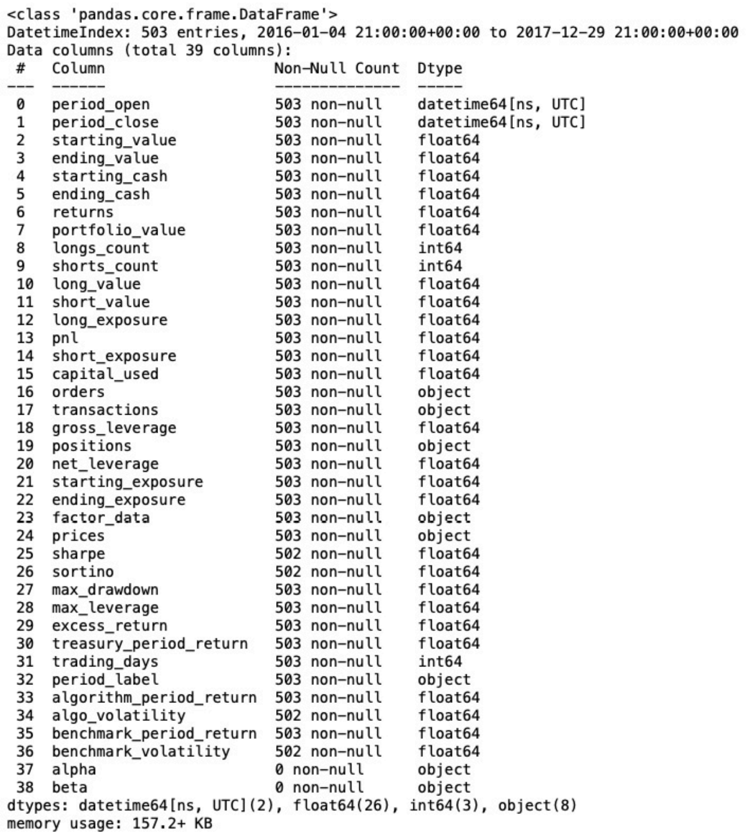
9. Finally, we run the backtest using the **`run_algorithm`** function:

```
start = pd.Timestamp("2016")
end = pd.Timestamp("2018")
perf = run_algorithm(
    start=start,
    end=end,
    initialize=initialize,
    before_trading_start=before_trading_start,
    capital_base=100_000,
    bundle="quandl",
)
```

The output is a DataFrame that contains trading, risk, and performance statistics for each day in the backtest:

```
perf.info()
```

It's this DataFrame that we use to analyze the characteristics of the backtest:



```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 503 entries, 2016-01-04 21:00:00+00:00 to 2017-12-29 21:00:00+00:00
Data columns (total 39 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   period_open              503 non-null    datetime64[ns, UTC]
 1   period_close             503 non-null    datetime64[ns, UTC]
 2   starting_value           503 non-null    float64
 3   ending_value             503 non-null    float64
 4   starting_cash            503 non-null    float64
 5   ending_cash              503 non-null    float64
 6   returns                  503 non-null    float64
 7   portfolio_value          503 non-null    float64
 8   longs_count              503 non-null    int64
 9   shorts_count             503 non-null    int64
 10  long_value               503 non-null    float64
 11  short_value              503 non-null    float64
 12  long_exposure            503 non-null    float64
 13  pnl                      503 non-null    float64
 14  short_exposure           503 non-null    float64
 15  capital_used             503 non-null    float64
 16  orders                   503 non-null    object
 17  transactions             503 non-null    object
 18  gross_leverage           503 non-null    float64
 19  positions                503 non-null    object
 20  net_leverage             503 non-null    float64
 21  starting_exposure        503 non-null    float64
 22  ending_exposure          503 non-null    float64
 23  factor_data              503 non-null    object
 24  prices                   503 non-null    object
 25  sharpe                   502 non-null    float64
 26  sortino                  502 non-null    float64
 27  max_drawdown             503 non-null    float64
 28  max_leverage             503 non-null    float64
 29  excess_return            503 non-null    float64
 30  treasury_period_return   503 non-null    float64
 31  trading_days             503 non-null    int64
 32  period_label             503 non-null    object
 33  algorithm_period_return  503 non-null    float64
 34  algo_volatility          502 non-null    float64
 35  benchmark_period_return  503 non-null    float64
 36  benchmark_volatility     502 non-null    float64
 37  alpha                    0 non-null      object
 38  beta                     0 non-null      object
dtypes: datetime64[ns, UTC](2), float64(26), int64(3), object(8)
memory usage: 157.2+ KB
```

Figure 8.2: Output DataFrame from our Zipline Reloaded backtest

How it works…

We dove into the inner workings of the Pipeline API and custom factors in *Chapter 5*, *Build Alpha Factors for Stock Portfolios*, so here, we'll focus on the remaining code to run the backtest. The **`before_trading_start`** function is invoked prior to the beginning of each trading day. It takes two parameters:

- **`context`**: A persistent namespace that allows the storage of variables between function calls. It retains its values across multiple days and can be used to store and manage data that the algorithm needs over time.
- **`data`**: An object that provides access to current and historical pricing and volume data, among other things.

Before every trading day, we fetch the output of a defined pipeline named **`factor_pipeline`**. The **`pipeline_output`** function retrieves the computed results of the named pipeline for the current day. These results are then stored in the **`context`** namespace under the **`factor_data`** key. This allows the algorithm to access and use the pipeline's output in subsequent functions or during the trading day.

IMPORTANT NOTE

The **`context`** *object in the Zipline Reloaded backtesting framework serves as a persistent namespace, allowing algorithms to store and manage variables across multiple function calls and trading sessions. It retains its values throughout the algorithm's execution, acting as a central repository* *for data, counters, flags, and other essential information that the algorithm requires for its operations and decision-making processes.*

The **initialize** function is used to set up initial configurations and operations that the algorithm will use throughout its execution. Here, the function first attaches a data pipeline, created by the **make_pipeline** method, and names it **factor_pipeline**. Subsequently, the **schedule_function** method is used to schedule the **rebalance** function to run at the start of each week at the market's opening time, using the US equities trading calendar. This ensures that the algorithm will regularly adjust its portfolio based on the logic defined in the **rebalance** function following the exchange's open days.

The **rebalance** function adjusts the portfolio based on specific criteria. Initially, the function retrieves the factor data stored in the **context** object and adds the rank to the output DataFrame using the **record** function. It then fetches the current prices of the assets under consideration and adds them to the output DataFrame as well. The assets are then categorized into three groups: **longs** (assets the algorithm intends to buy), **shorts** (assets it plans to sell short), and **divest** (current portfolio holdings in neither the **longs** nor **shorts** list, indicating they should be sold off).

We implement a function called **exec_trades** to abstract away the order execution. For each asset in the list, the function first checks whether the asset is tradable and that there are no open orders for it using the **can_trade** and **get_open_orders** methods. If both conditions are met, the function places an order for the asset to adjust its position to the desired target percentage using the **order_target_percent** method.

Finally, we call **run_algorithm** to start the backtest, taking in the start and end dates, the **initialize** and **before_trading_start** functions to set up and prepare for each trading day, a starting capital of $100,000, and specifying the data bundle as **quandl**. The results of the backtest are stored in the **perf** variable for further analysis.

There's more…

The output of a Zipline Reloaded backtest provides a comprehensive overview of an algorithm's performance. It includes a time series of key metrics, such as portfolio value, returns, and specific asset positions. Additionally, it captures risk metrics, transaction logs, and other diagnostic data that helps in evaluating the strategy's robustness and potential pitfalls.

The output can be used with the risk and performance libraries pyfolio and alphalens, which we dive into in the next two chapters. For now, we'll inspect a few of the key outputs:

```
perf.portfolio_value.plot(title="Cumulative returns")
```

The result is a plot showing the cumulative equity of your algorithm.

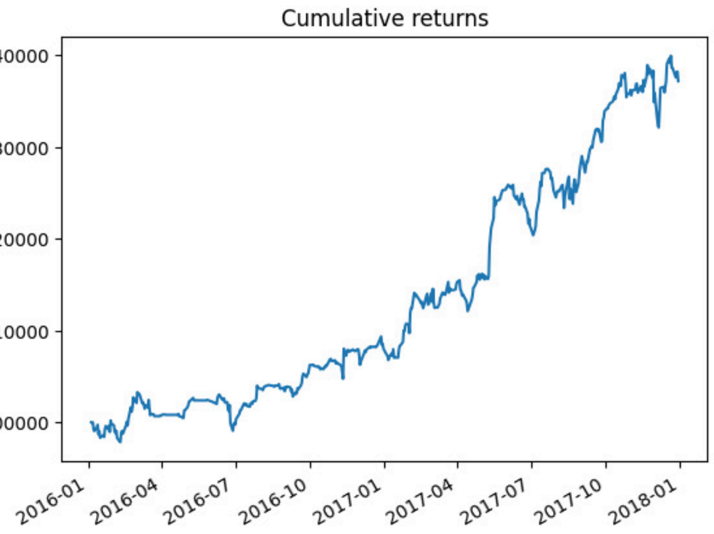


Figure 8.3: Cumulative equity of the algorithm

Create a histogram of daily returns:

```
perf.returns.hist(bins=50)
```

The result is a histogram showing the frequency of daily returns across 50 bins:

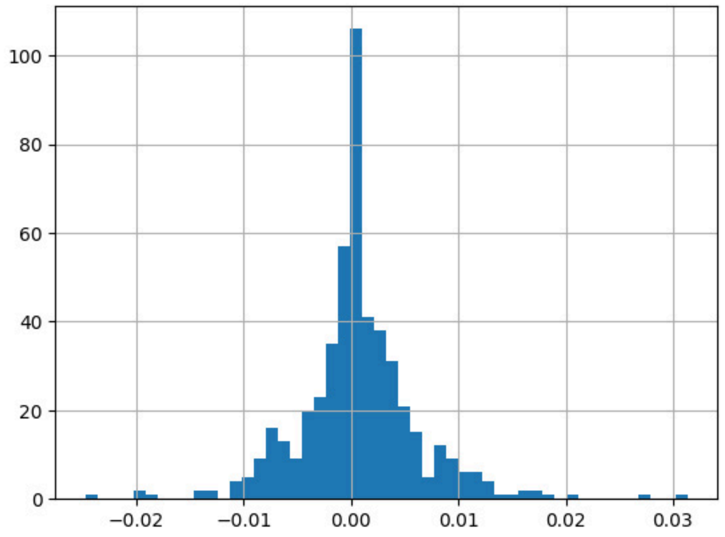


Figure 8.4: Histogram of daily portfolio returns

Plot the rolling Sharpe ratio for the algorithm:

```
perf.sharpe.plot()
```

The result is a plot visualizing the rolling Sharpe ratio of the algorithm:

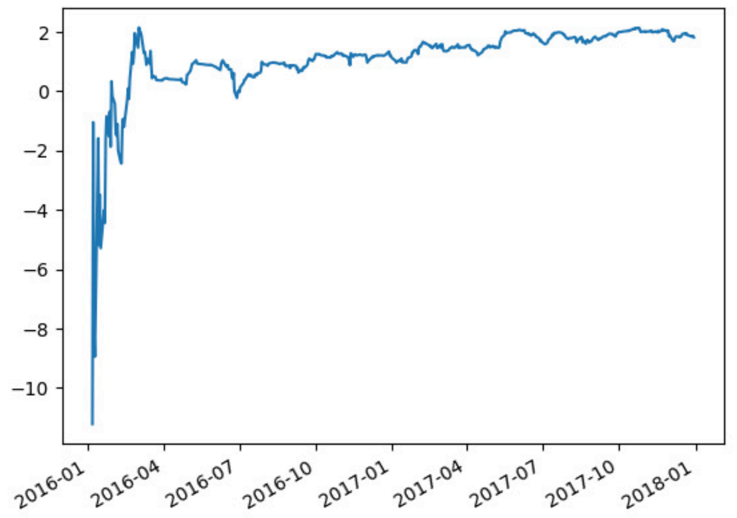


Figure 8.5: Plot of the rolling Sharpe ratio of the algorithm

> *HINT*

> *The output is a pandas DataFrame, which means all the data manipulation methods we've learned thus far apply.*

See also

Zipline Reloaded offers robust documentation for using its extensive features. You can find them here:

- Zipline Reloaded documentation: **https://zipline.ml4trading.io**
- An article that describes ingesting the premium US equities data: **https://www.pyquantnews.com/free-python-resources/how-to-ingest-premium-market-data-with-zipline-reloaded**

Exploring a mean reversion strategy with Zipline Reloaded

Mean reversion strategies are based on the financial principle that asset prices and returns eventually revert to their long-term mean or average level after periods of divergence or deviation. These strategies operate on the assumption that prices will bounce back to a historical mean or some form of equilibrium after moving away from it, either due to overreaction or other short-term factors. Mean reversion suggests that assets are subject to inherent and stable equilibriums. When prices deviate significantly from these equilibriums, due to factors such as emotional trading, news, or events, they are likely to revert back over time. Traders and algorithms identify assets that have deviated significantly from their historical average price or some other benchmark. This deviation can be measured using various metrics, such as z-scores, Bollinger Bands, or percentage deviations. In this recipe, we use the z-score.

In this recipe, we'll use the Zipline Reloaded factor framework to build a portfolio. The strategy buys the top oversold stocks and sells the top overbought stocks.

Getting ready

Most of this recipe will be the same as the recipe demonstrating the momentum factor, with some notable differences:

- We've updated the factor to measure mean reversion and rank our universe based on the top and bottom mean-reverting assets
- We include commission and slippage rules to enhance the realism of the backtest
- We add simple logging to provide feedback during the execution of the algorithm and to demonstrate the available portfolio attributes
- We download benchmark price data and include returns for comparison against our algorithm
- We include a custom function that is executed when the backtest is complete
- We generate an image of the pipeline that shows how the logic filters the stocks in the universe

We assume you have the libraries imported and will skip that step.

How to do it…

We will expand on the previous recipe by adding complexity to our analysis:

1. Set the number of longs and shorts and the lookback periods:

```
N_LONGS = N_SHORTS = 50
MONTH = 21
YEAR = 12 * MONTH
```

2. Create the mean reversion factor:

```
class MeanReversion(CustomFactor):
    inputs = [Returns(window_length=MONTH)]
    window_length = YEAR
    def compute(self, today, assets, out,
        monthly_returns):
            df = pd.DataFrame(monthly_returns)
            out[:] = df.iloc[-1].sub(
                df.mean()).div(df.std())
```

3. Implement the function that returns the pipeline using the factor:

```
def make_pipeline():
    mean_reversion = MeanReversion()
    dollar_volume = AverageDollarVolume(
        window_length=30)
    return Pipeline(
        columns={
            "longs": mean_reversion.bottom(N_LONGS),
            "shorts": mean_reversion.top(N_SHORTS),
            "ranking": mean_reversion.rank(
                ascending=False),
        },
        screen=dollar_volume.top(100),
    )
```

4. Implement the function that hooks into the event that fires before trading starts:

```
def before_trading_start(context, data):
    context.factor_data = pipeline_output(
        "factor_pipeline")
```

5. Implement the function that is invoked when the backtest begins. Note the addition of commission and slippage models:

```
def initialize(context):
    attach_pipeline(make_pipeline(),
        "factor_pipeline")
    schedule_function(
        rebalance,
        date_rules.week_start(),
        time_rules.market_open(),
        calendar=calendars.US_EQUITIES,
    )
    set_commission(
        us_equities=commission.PerShare(
            cost=0.00075, min_trade_cost=0.01
        )
    )
    set_slippage(
        us_equities=slippage.VolumeShareSlippage(
            volume_limit=0.0025, price_impact=0.01
        )
    )
```

6. Add a **print** statement to the same **rebalance** function we created in the last recipe. This **print** statement provides feedback as the algorithm is running:

```
def rebalance(context, data):
    factor_data = context.factor_data
    record(factor_data=factor_data.ranking)
```

7. In the next section of the **rebalance** function, extract the symbols from the **factor_data** DataFrame and record the asset prices:

```
    assets = factor_data.index
    record(prices=data.current(assets, "price"))
```

8. Now we identify the assets to go long, to go short, and to divest from the portfolio:

```
    longs = assets[factor_data.longs]
    shorts = assets[factor_data.shorts]
    divest = set(
        context.portfolio.positions.keys()) - set(
            longs.union(shorts))
```

9. Finally, we print output to the user and call the **exec_trades** function to execute our desired orders:

```
    print(
        f"{get_datetime().date()} | Longs {len(longs)} | Shorts | {len(shorts)} | {context.port
    )
    exec_trades(
        data,
        assets=divest,
        target_percent=0
    )
    exec_trades(
        data,
        assets=longs,
        target_percent=1 / N_LONGS
    )
    exec_trades(
        data,
        assets=shorts,
        target_percent=-1 / N_SHORTS
    )
```

10. Implement the same **`exec_trades`** function as in the previous recipe:

```
def exec_trades(data, assets, target_percent):
    for asset in assets:
        if data.can_trade(
            asset) and not get_open_orders(asset):
            order_target_percent(
                asset, target_percent)
```

11. The **`analyze`** function is run after the backtest is complete. We have access to the **`context`** object and the output of the backtest in the **`perf`** DataFrame. This is useful to run reports or event trigger alerts if certain thresholds are passed. In this example, we simply plot the portfolio value:

```
def analyze(context, perf):
    perf.portfolio_value.plot()
```

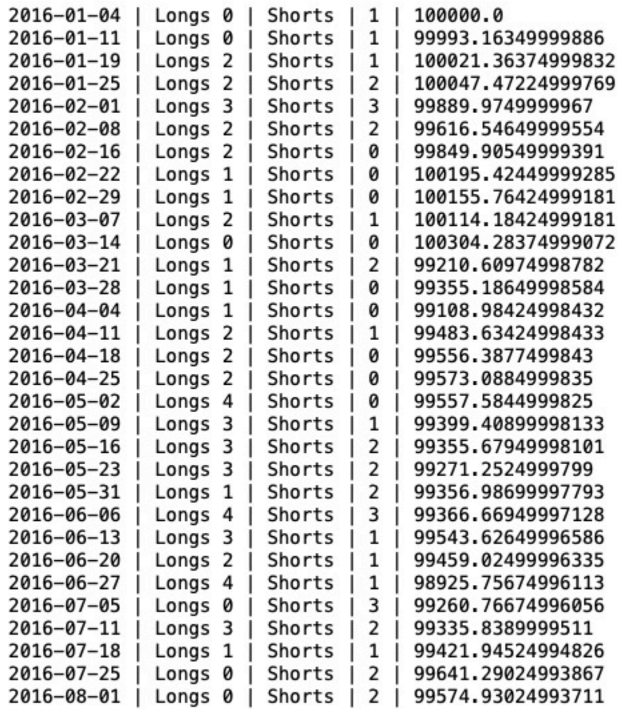
12. Use **`pandas_datareader`** to compute the daily returns of a benchmark. In this case, we use the S&P 500 index:

```
start = pd.Timestamp(“2016”)
end = pd.Timestamp(“2018”)
sp500 = web.DataReader('SP500', 'fred', start,
    end).SP500
benchmark_returns = sp500.pct_change()
```

13. Finally, run the backtest and cache the output:

```
perf = run_algorithm(
    start=start,
    end=end,
    initialize=initialize,
    analyze=analyze,
    benchmark_returns=benchmark_returns,
    before_trading_start=before_trading_start,
    capital_base=100_000,
    bundle="quandl"
)
perf.to_pickle("mean_reversion.pickle")
```

While the backtest is running, you'll see output that resembles the following:



```
2016-01-04 | Longs 0 | Shorts | 1 | 100000.0
2016-01-11 | Longs 0 | Shorts | 1 | 99993.16349999886
2016-01-19 | Longs 2 | Shorts | 1 | 100021.36374999832
2016-01-25 | Longs 2 | Shorts | 2 | 100047.47224999769
2016-02-01 | Longs 3 | Shorts | 3 | 99889.9749999967
2016-02-08 | Longs 2 | Shorts | 2 | 99616.54649999554
2016-02-16 | Longs 2 | Shorts | 0 | 99849.90549999391
2016-02-22 | Longs 1 | Shorts | 0 | 100195.42449999285
2016-02-29 | Longs 1 | Shorts | 0 | 100155.76424999181
2016-03-07 | Longs 2 | Shorts | 1 | 100114.18424999181
2016-03-14 | Longs 0 | Shorts | 0 | 100304.28374999072
2016-03-21 | Longs 1 | Shorts | 2 | 99210.60974998782
2016-03-28 | Longs 1 | Shorts | 0 | 99355.18649998584
2016-04-04 | Longs 1 | Shorts | 0 | 99108.98424998432
2016-04-11 | Longs 2 | Shorts | 1 | 99483.63424998433
2016-04-18 | Longs 2 | Shorts | 0 | 99556.3877499843
2016-04-25 | Longs 2 | Shorts | 0 | 99573.0884999835
2016-05-02 | Longs 4 | Shorts | 0 | 99557.5844999825
2016-05-09 | Longs 3 | Shorts | 1 | 99399.40899998133
2016-05-16 | Longs 3 | Shorts | 2 | 99355.67949998101
2016-05-23 | Longs 3 | Shorts | 2 | 99271.2524999799
2016-05-31 | Longs 1 | Shorts | 2 | 99356.98699997793
2016-06-06 | Longs 4 | Shorts | 3 | 99366.66949997128
2016-06-13 | Longs 3 | Shorts | 1 | 99543.62649996586
2016-06-20 | Longs 2 | Shorts | 1 | 99459.02499996335
2016-06-27 | Longs 4 | Shorts | 1 | 98925.75674996113
2016-07-05 | Longs 0 | Shorts | 3 | 99260.76674996056
2016-07-11 | Longs 3 | Shorts | 2 | 99335.8389999511
2016-07-18 | Longs 1 | Shorts | 1 | 99421.94524994826
2016-07-25 | Longs 0 | Shorts | 2 | 99641.29024993867
2016-08-01 | Longs 0 | Shorts | 2 | 99574.93024993711
```

Figure 8.6: Logs from the algorithm run

14. When the backtest completes, the **analyze** function is invoked and the performance is plotted:

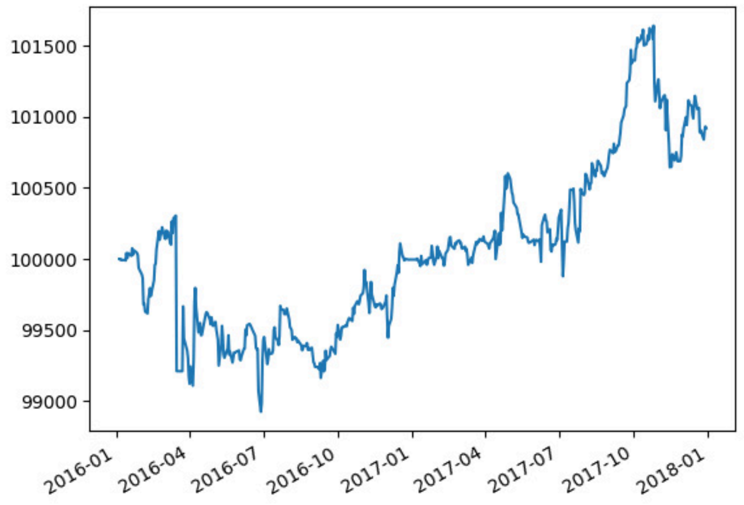


Figure 8.7: Plot automatically generated from the analyze function

IMPORTANT NOTE

The keen eye may spot what looks like a bug: why are we taking long or short positions in only a few stocks when we've set the number of longs and shorts to 50? The answer lies in the sequence in which the **Pipeline** *class processes its filters. It's important to note that we've included a screen argument tied to the* **dollar_volume** *factor. This effectively filters the stocks in the pipeline to only those with a dollar volume exceeding $100,000, and this screening occurs after the long and short selections are made. If you were to remove this screening criterion, the log would display 50 longs and 50 shorts as expected.Top of Form*

Bottom of Form

How it works…

Our custom factor uses monthly returns as its input, specified by the **Returns** class with a window length set to a constant, **MONTH**, representing a month. The overall window length for the factor is set to a constant, **YEAR**, representing a year. Within the **compute** method, the monthly returns are converted into a DataFrame. The factor's value for each asset is then computed by taking the last month's return, subtracting the mean of all monthly returns, and then dividing by the standard deviation of those returns. This results in a z-score-like metric, indicating how many standard deviations the latest month's return is from the mean, which can be used to gauge mean reversion tendencies for each asset.

The **make_pipeline** function is similar to the one we constructed in the last recipe except it uses the mean reversion factor. Within the function, an instance of the **MeanReversion** factor is created, which calculates a mean reversion score for each asset. Additionally, the **AverageDollarVolume** class is used to compute the average dollar volume over a 30-day window for liquidity screening. The core of the function is the **Pipeline** object, which is set up to produce three columns:

- **longs**: Identifies the assets with the lowest mean reversion scores (i.e., the most undervalued assets)
- **shorts**: Pinpoints the assets with the highest scores (i.e., the most overvalued assets)
- **ranking**: Provides a rank for each asset based on its mean reversion score in descending order

To ensure the pipeline focuses on liquid assets, a screen is applied that only considers the top 100 assets based on their average dollar volume.

In the **initialize** function, we attach the pipeline and schedule the rebalancing in the same way as the last recipe. However, in this implementation, we include commission and slippage models. The **set_commission** function configures the commission model for US equities to be based on a per-share cost. Specifically, the algorithm will be charged $0.00075 for each share traded, with a minimum trade cost set at $0.01, ensuring that even small trades incur a nominal fee. The **set_commission** function allows for a large range of commission models to match your actual broker's commission schedule. Following this, the **set_slippage** function establishes the slippage model for US equities using the **VolumeShareSlippage** method. This model simulates the impact of an order on the stock price based on the order's size relative to the stock's average volume. The parameters dictate that an order can consume up to 0.25% of the stock's daily volume and that each order will impact the stock price by 1%.

Finally, to compare our algorithm's return to the benchmark, we compute daily returns for the S&P 500 index.

There's more…

Our pipeline is simple: we generate a mean reversion factor, screen the top and bottom stocks for a dollar volume greater than $100,000, and return the results. Zipline Reloaded supports compound factor models where several factors are combined. In those cases, it helps to visualize what's going on. Luckily, we can generate an illustration of how the pipeline is constructed:

```
p = make_pipeline()
p.show_graph()
```

The result is a pictorial depiction of the pipeline:

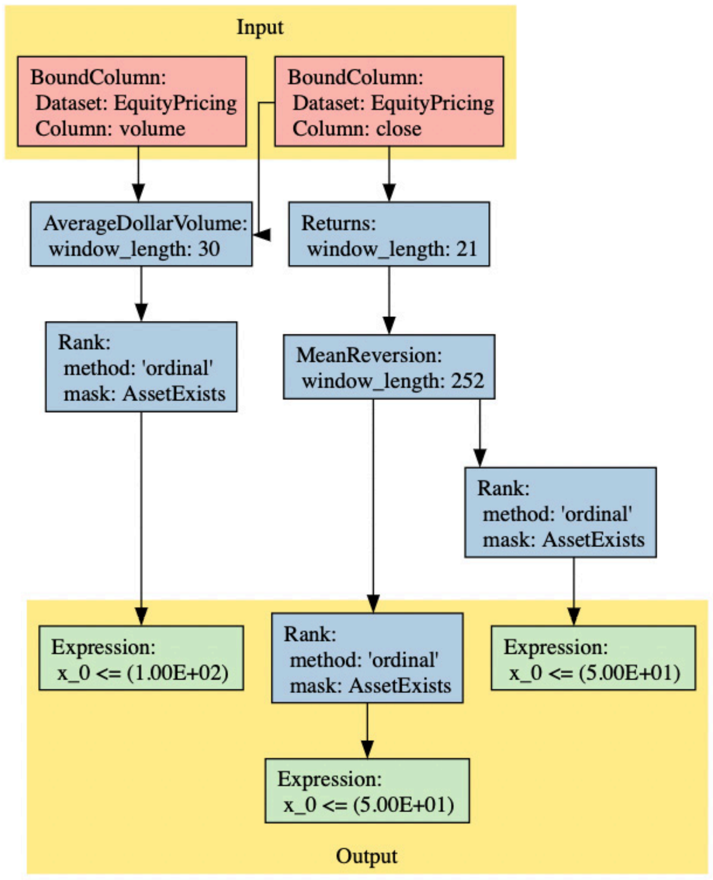

Input
BoundColumn: Dataset: EquityPricing Column: volume
BoundColumn: Dataset: EquityPricing Column: close
AverageDollarVolume: window_length: 30
Returns: window_length: 21
Rank: method: 'ordinal' mask: AssetExists
MeanReversion: window_length: 252
Rank: method: 'ordinal' mask: AssetExists
Expression: x_0 <= (1.00E+02)
Rank: method: 'ordinal' mask: AssetExists
Expression: x_0 <= (5.00E+01)
Expression: x_0 <= (5.00E+01)
Output

Figure 8.8: Graphical representation of the pipeline

By including benchmark returns, Zipline Reloaded computes the rolling alpha and beta of our portfolio against the benchmark. We learned how to hedge the beta and amplify the alpha in the last chapter. Using the output of the backtest, we have these metrics at our disposal.

Plot the rolling beta against the benchmark:

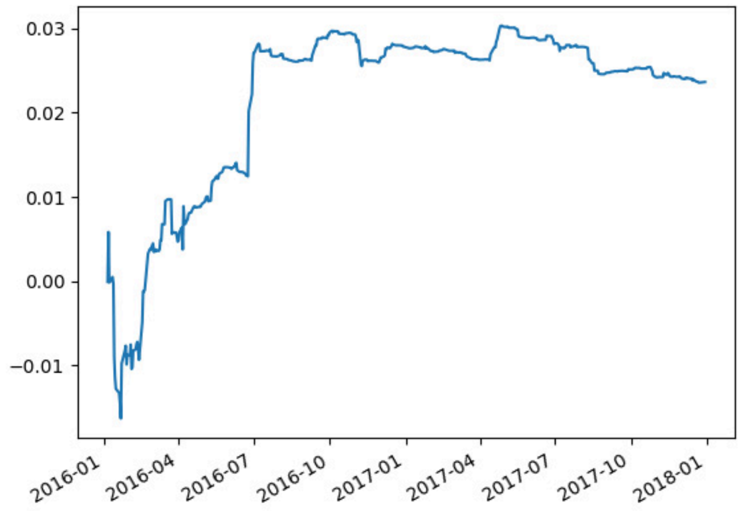


Figure 8.9: Rolling beta of our algorithm's returns against the benchmark

Plot the rolling alpha against the benchmark:

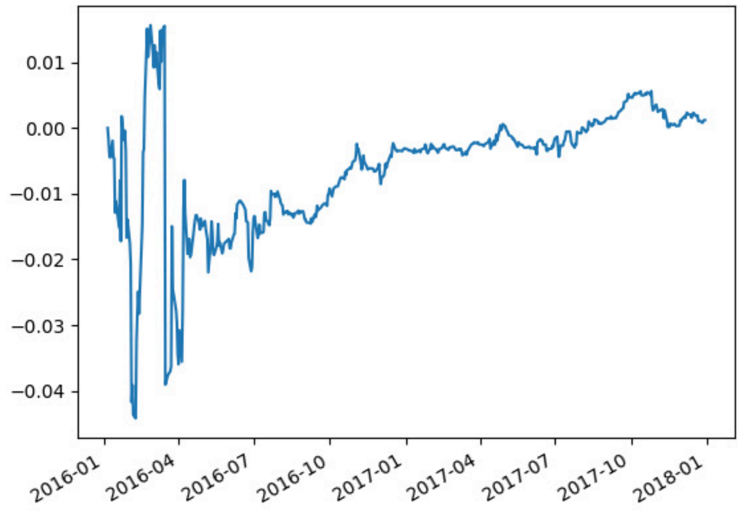


Figure 8.10: Rolling alpha of our algorithm's returns against the benchmark

See also

We went deeper into some more advanced features of Zipline Reloaded. These features aim to create a more realistic simulation of the market dynamics algorithmic traders face every day. To dive deeper, check out the documentation listed here:

- API documentation for the Pipeline API, which describes its available input parameters: **https://zipline.ml4trading.io/api-reference.html#pipeline-api**
- Different built-in slippage models available within Zipline Reloaded: **https://zipline.ml4trading.io/api-reference.html#slippage-models**
- Different built-in commission models available within Zipline Reloaded: **https://zipline.ml4trading.io/api-reference.html#commission-models**