Zipline Trader

Release 1.5.0

unknown

Dec 30, 2020

CONTENTS

1 Supported Data Sources 3

2 Supported Brokers 5

Index 83

Welcome to zipline-trader, the on-premise trading platform built on top of Quantopian's zipline.

Quantopian closed their services, so this project tries to be a suitable replacement.

zipline-trader is based on previous projects and work:

- zipline project.
- zipline-live project.
- zipline-live2 project.

zipline-live and zipline-live2 are past iterations of this project and this is the up to date project.

After Quantopian closed their services, this project was updated to supply a suitable and sustainable replacement for users that want to run their algorithmic trading on their own without relying on online services that may disappear one day. It is designed to be an extensible, drop-in replacement for zipline with multiple brokerage support to enable on premise trading of zipline algorithms.

I recommend using python 3.6

CHAPTER

ONE

SUPPORTED DATA SOURCES

Out of the box, zipline-trader supports Alpaca as a free data source. You could use the quantopian-quandl bundle used in old zipline versions or any other bundle you create (how to create a bundle on a later section)

CHAPTER

TWO

SUPPORTED BROKERS

Currently 2 brokers are supported:

- Alpaca
- IB

2.1 Install

Please use python 3.6 and let's avoid unnecessary issues.
Linux and Windows Installations are automatically tested. Mac OS users might have an issue with Bcolz. For now, refer to the original zipline docs if you do. Fixing this issue is in the backlog.
If you use Python for anything other than Zipline, I **strongly** recommend that you install in a `virtualenv

<https://virtualenv.readthedocs.org/en/latest>`_.

The Hitchhiker's Guide to Python provides an excellent tutorial on virtualenv.

For now I support only installation through github. You could do that in one of these ways:

2.1.1 Installing with git clone

- git clone https://github.com/shlomikushchi/zipline-trader.git
- <create/activate a virtual env> - optional but recommended
- pip install -e .

The last step will install this project from source, giving you the ability to debug zipline-trader's code.

2.1.2 Installing using pip directly from github

You can install it with ability to debug it like this:

```
pip install -e git://github.com/shlomikushchi/zipline-trader.git#egg=zipline-trader
```

To install a specific version, you could do this (installing version 1.5.0):

```
pip install -e git://github.com/shlomikushchi/zipline-trader.git@1.5.0#egg=zipline-
↪trader
```

2.1.3 Installing from pypi (coming soon)

The most known way of installing would be installing from pypi:

```
pip install zipline-trader
```

- Installing using Anaconda (Probably supported in the future)

2.1.4 Notes

Installing zipline is a bit complicated, and therefore installing zipline-trader. There are two reasons for zipline installation additional complexity:

1. Zipline ships several C extensions that require access to the CPython C API. In order to build the C extensions, `pip` needs access to the CPython header files for your Python installation.
2. Zipline depends on numpy, the core library for numerical array computing in Python. Numpy depends on having the LAPACK linear algebra routines available.

Because LAPACK and the CPython headers are non-Python dependencies, the correct way to install them varies from platform to platform. Once you've installed the necessary additional dependencies (see below for your particular platform)

GNU/Linux

On Debian-derived Linux distributions, you can acquire all the necessary binary dependencies from `apt` by running:

```
$ sudo apt-get install libatlas-base-dev python-dev gfortran pkg-config libfreetype6-
↪dev hdf5-tools
```

On recent RHEL-derived derived Linux distributions (e.g. Fedora), the following should be sufficient to acquire the necessary additional dependencies:

```
$ sudo dnf install atlas-devel gcc-c++ gcc-gfortran libgfortran python-devel redhat-
↪rpm-config hdf5
```

On Arch Linux, you can acquire the additional dependencies via `pacman`:

```
$ pacman -S lapack gcc gcc-fortran pkg-config hdf5
```

There are also AUR packages available for installing ta-lib, an optional Zipline dependency.

OSX

The version of Python shipped with OSX by default is generally out of date, and has a number of quirks because it's used directly by the operating system. For these reasons, many developers choose to install and use a separate Python installation. The Hitchhiker's Guide to Python provides an excellent guide to Installing Python on OSX, which explains how to install Python with the Homebrew manager.

Assuming you've installed Python with Homebrew, you'll also likely need the following brew packages:

```
$ brew install freetype pkg-config gcc openssl hdf5
```

2.2 Alpaca Data Bundle

Out of the box, I support Alpaca as a data source for data ingestion.
If you haven't created an account, start with that: Alpaca Signup.
Why? You could get price data for free using the Alpaca data API. Free data is hard to get.
Any other vendor could be added and you are not obligated to use that.

2.2.1 How To Use It

Please don't use the old method of zipline ingestion. It doesn't work for this bundle, and it will be abandoned. So DON'T DO THIS:

```
zipline ingest -b alpaca_api
```

DO THIS INSTEAD:
*I currently only support daily data but free minute data will soon follow.
To ingest daily data bundle using the alpaca api, you need to follow these steps:

- The bundle is defined in this file: `zipline/data/bundles/alpaca_api.py`
- There is a method called `initialize_client()`, it relies on the fact that you define your alpaca credentials in a file called `alpaca.yaml` in your root directory. it should look like this:

```
key_id: "<YOUR-KEY>"
secret: "<YOUR-SECRET>"
base_url: https://paper-api.alpaca.markets
```

- you need to define your zipline root in an environment variable (This is where the ingested data will be stored). It should be something like this:

```
ZIPLINE_ROOT=~/.zipline
```

  It means you could basically put it anywhere you want as long as you always use that as your zipline root.

  It also means that different bundles could have different locations.

- By defauilt the bundle ingests 30 days backwards, but you can change that under the `__main__` section of `zipline/data/bundles/alpaca_api.py`.

The ingestion process for daily data using Alpaca is extremely fast due to the Alpaca API allowing to query 200 equities in one api call.

Notes

- You are ready to research, backtest or paper trade using the pipeline functionality.
- You should repeat this process daily since every day you will have new price data.
- This data doesn't include Fundamental data, only price data so we'll need to handle it separately.

2.3 Research & backtesting in the Notebook environment

To run your research environment you first need to make sure jupyter is installed.
Follow the instructions in Jupyter.org

```
e.g. pip install notebook
```

Start your Jupyter server

```
jupyter notebook
```

2.3.1 Working With The Research Environment

This was one of Quantopian's strengths and now you could run it locally too.
In the next few examples we will see how to:

- Load your Alpaca (or any other) data bundle
- How to get pricing data from the bundle
- How to create and run a pipeline
- How tu run a backtest INSIDE the notebook (using python files will follow)
- How to analyze your results with pyfolio (for that you will need to install pyfolio)

2.3.2 Loading Your Data Bundle

Now that you have a jupyter notebook running you could load your previously ingested data bundle.
Follow this notebook for a usage example: Load Data Bundle.

2.3.3 Simple Pipeline

You can work with pipeline just as it was on Quantopian, and in the following example you could see hwo to create a simple pipeline and get the data: Simple Pipeline.

2.3.4 Run and analyze a backtest

Running a backtest is the way to test your ideas. You could do it inside a notebook or in your python IDE (your choice).
The advantage of using the notebook is the ability to use Pyfolio to analyze the results in a simple manner as could be seen here: Bactesting.

2.4 Backtesting

Old Zipline users know the command line tool that used to run backtests. e.g:

```
zipline --start 2014-1-1 --end 2018-1-1 -o dma.pickle
```

This is still supported but not recommended. One does not have much power when running a backtest that way.
The recommended way is to run inside a python file, preferably using an IDE so you could debug your code with breakpoints and memory view.
I will show you exactly how to do so, providing a template that you could just copy and develop your code in.

2.4.1 Important notes before we start

You of course need to have everything already installed, so go to the install part if you don't
You need to have an ingested data bundle. You could use the Alpaca Data Bundle or user your own.
You need to have some understanding on how a zipline algo is used. (Beginner Tutorial)
You need to set the `ZIPLINE_ROOT` env variable to point to your ingested data bundle.

2.4.2 Algo Template

This next code snippet is a simple algorithm that is self contained. You could copy that into a python file and just execute it

```
import pytz
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
import pandas_datareader.data as yahoo_reader

from zipline.utils.calendars import get_calendar
from zipline.api import order_target, symbol
```

(continues on next page)

(continued from previous page)

```
from zipline.data import bundles
from zipline import run_algorithm


def get_benchmark(symbol=None, start=None, end=None):
    bm = yahoo_reader.DataReader(symbol,
                                 'yahoo',
                                 pd.Timestamp(start),
                                 pd.Timestamp(end))['Close']
    bm.index = bm.index.tz_localize('UTC')
    return bm.pct_change(periods=1).fillna(0)


def initialize(context):
    context.equity = symbol("AMZN")


def handle_data(context, data):
    order_target(context.equity, 100)


def before_trading_start(context, data):
    pass


def analyze(context, perf):
    ax1 = plt.subplot(211)
    perf.portfolio_value.plot(ax=ax1)
    ax2 = plt.subplot(212, sharex=ax1)
    perf.sym.plot(ax=ax2, color='r')
    plt.gcf().set_size_inches(18, 8)
    plt.legend(['Algo', 'Benchmark'])
    plt.ylabel("Returns", color='black', size=25)


if __name__ == '__main__':
    bundle_name = 'alpaca_api'
    bundle_data = bundles.load(bundle_name)

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

    start = pd.Timestamp(datetime(2020, 1, 1, tzinfo=pytz.UTC))
    end = pd.Timestamp(datetime(2020, 11, 1, tzinfo=pytz.UTC))

    r = run_algorithm(start=start,
                      end=end,
                      initialize=initialize,
                      capital_base=100000,
                      handle_data=handle_data,
                      benchmark_returns=get_benchmark(symbol="SPY",
                                                      start=start.date().isoformat(),
                                                      end=end.date().isoformat()),
                      bundle='alpaca_api',
                      broker=None,
                      state_filename="./demo.state",
                      trading_calendar=trading_calendar,
```

(continues on next page)

(continued from previous page)

```
                    before_trading_start=before_trading_start,
                    #                    analyze=analyze,
                    data_frequency='daily'
                    )
    fig, axes = plt.subplots(1, 1, figsize=(16, 5), sharex=True)
    r.algorithm_period_return.plot(color='blue')
    r.benchmark_period_return.plot(color='red')

    plt.legend(['Algo', 'Benchmark'])
    plt.ylabel("Returns", color='black', size=20)
    plt.show()
```

2.5 Going Live

Old zipline-live/zipline-live2 users know the command line tool that used to go live. e.g:

```
zipline run -f ~/zipline-algos/demo.py --state-file ~/zipline-algos/demo.state --
↪realtime-bar-target ~/zipline-algos/realtime-bars/ --broker ib --broker-uri␣
↪localhost:7496:1232 --bundle quantopian-quandl --data-frequency minute
```

You could still do that, but you shouldn't.
The recommended way is to run inside a python file, preferably using an IDE so you could debug your code with breakpoints and memory view.
I will show you exactly how to do so, providing a template that you could just copy and develop your code in. It will be the exact same example I provided in the backtest section with a different call to `run_algorithm` which will connect to a broker (you could use IB or Alpaca for now)

2.5.1 Important notes before we start

You of course need to have everything already installed, so go to the install part if you don't
You need to have an ingested data bundle. You could use the Alpaca Data Bundle or user your own.
You need to have some understanding on how a zipline algo is used. (Beginner Tutorial)
You need to set the `ZIPLINE_ROOT` env variable to point to your ingested data bundle.
You need to store the alpaca credentials in a file called `alpaca.yaml` in your root directory. it should look like this:

```
key_id: "<YOUR-KEY>"
secret: "<YOUR-SECRET>"
base_url: https://paper-api.alpaca.markets
```

2.5.2 Algo Template

This next code snippet is a simple algorithm that is self contained. You could copy that into a python file and just execute it. It will connect to the Alpaca broker.

```
import os
import yaml
import pytz
import pandas as pd
from datetime import datetime
import pandas_datareader.data as yahoo_reader

from zipline.utils.calendars import get_calendar
from zipline.api import order_target, symbol
from zipline.data import bundles
from zipline import run_algorithm
from zipline.gens.brokers.alpaca_broker import ALPACABroker


def get_benchmark(symbol=None, start=None, end=None):
    bm = yahoo_reader.DataReader(symbol,
                                 'yahoo',
                                 pd.Timestamp(start),
                                 pd.Timestamp(end))['Close']
    bm.index = bm.index.tz_localize('UTC')
    return bm.pct_change(periods=1).fillna(0)


def initialize(context):
    pass


def handle_data(context, data):
    order_target(context.equity, 100)


def before_trading_start(context, data):
    context.equity = symbol("AMZN")


if __name__ == '__main__':
    bundle_name = 'alpaca_api'
    bundle_data = bundles.load(bundle_name)

    with open("alpaca.yaml", mode='r') as f:
        o = yaml.safe_load(f)
        os.environ["APCA_API_KEY_ID"] = o["key_id"]
        os.environ["APCA_API_SECRET_KEY"] = o["secret"]
        os.environ["APCA_API_BASE_URL"] = o["base_url"]
    broker = ALPACABroker()

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

    start = pd.Timestamp(datetime(2020, 1, 1, tzinfo=pytz.UTC))
    end = pd.Timestamp.utcnow()
```

(continues on next page)

(continued from previous page)

```
    run_algorithm(start=start,
                  end=end,
                  initialize=initialize,
                  handle_data=handle_data,
                  capital_base=100000,
                  benchmark_returns=get_benchmark(symbol="SPY",
                                                  start=start.date().isoformat(),
                                                  end=end.date().isoformat()),
                  bundle='alpaca_api',
                  broker=broker,
                  state_filename="./demo.state",
                  trading_calendar=trading_calendar,
                  before_trading_start=before_trading_start,
                  data_frequency='daily'
                  )
```

2.6 Troubleshooting/FAQ

In this section I will put issues/questions users faced, so it might help new users too.

2.6.1 Dos And Donts

Right Way to Ingest Data

Old Zipline users are used to do this with the `zipline` cli. For now, avoid it.
Don't use this to ingest:

```
zipline ingest -b alpaca_api
```

I changed the way we ingest new data bundles. please refer to the Alpaca Data Bundle and read iut again.

Alpaca.yaml file issues

When using the Alpaca bundle, you must pass credentials to the Alpaca servers. It can't be avoided.
The easiest way to do this is by using a local file. The format is not important, I chose `yaml`. It's simple.
Make sure you put the alpaca credentials in the right place
First make sure the name of the file is `alpaca.yaml`. (avoid mistakes like using `.yml` postfix)
Put it in the right location - your root **python** folder. not inside the zipline-trader folder.

SQLite file doesn't exist.

If you happen to get this error when trying to work with data you just downloaded:

```
ValueError: SQLite file '/home/ubuntu/.zipline/data/alpaca_api/2020-12-07T02;06;17.
↪365878/assets-7.sqlite' doesn't exist.
```

It means you didn't define the `ZIPLINE_ROOT` correctly. You need to make sure this environment variable is defined for every python code you execute.

Mac OS

Currently we have issues installing on Mac OS due to usage of Bcolz. I will be resolved eventually. If anyone from the community want to resolve this for everyone else - you are welcome.
In the meantime - use a linux docker container to bypass that.

2.7 Original Zipline Docs

asdsd

2.7.1 Zipline Beginner Tutorial

Basics

Zipline is an open-source algorithmic trading simulator written in Python.

The source can be found at: https://github.com/quantopian/zipline

Some benefits include:

- Realistic: slippage, transaction costs, order delays.
- Stream-based: Process each event individually, avoids look-ahead bias.
- Batteries included: Common transforms (moving average) as well as common risk calculations (Sharpe).
- Developed and continuously updated by Quantopian which provides an easy-to-use web-interface to Zipline, 10 years of minute-resolution historical US stock data, and live-trading capabilities. This tutorial is directed at users wishing to use Zipline without using Quantopian. If you instead want to get started on Quantopian, see here.

This tutorial assumes that you have zipline correctly installed, see the installation instructions if you haven't set up zipline yet.

Every `zipline` algorithm consists of two functions you have to define:

- `initialize(context)`
- `handle_data(context, data)`

Before the start of the algorithm, `zipline` calls the `initialize()` function and passes in a `context` variable. `context` is a persistent namespace for you to store variables you need to access from one algorithm iteration to the next.

After the algorithm has been initialized, `zipline` calls the `handle_data()` function once for each event. At every call, it passes the same `context` variable and an event-frame called `data` containing the current trading bar with open, high, low, and close (OHLC) prices as well as volume for each stock in your universe. For more information on these functions, see the relevant part of the Quantopian docs.

My First Algorithm

Let's take a look at a very simple algorithm from the `examples` directory, `buyapple.py`:

```
from zipline.examples import buyapple
buyapple??
```

```
from zipline.api import order, record, symbol


def initialize(context):
    pass


def handle_data(context, data):
    order(symbol('AAPL'), 10)
    record(AAPL=data.current(symbol('AAPL'), 'price'))
```

As you can see, we first have to import some functions we would like to use. All functions commonly used in your algorithm can be found in `zipline.api`. Here we are using `order()` which takes two arguments: a security object, and a number specifying how many stocks you would like to order (if negative, `order()` will sell/short stocks). In this case we want to order 10 shares of Apple at each iteration. For more documentation on `order()`, see the Quantopian docs.

Finally, the `record()` function allows you to save the value of a variable at each iteration. You provide it with a name for the variable together with the variable itself: `varname=var`. After the algorithm finished running you will have access to each variable value you tracked with `record()` under the name you provided (we will see this further below). You also see how we can access the current price data of the AAPL stock in the `data` event frame (for more information see here).

Running the Algorithm

To now test this algorithm on financial data, `zipline` provides three interfaces: A command-line interface, `IPython Notebook` magic, and `run_algorithm()`.

Ingesting Data

If you haven't ingested the data, you'll need a Quandl API key to ingest the default bundle. Then run:

```
$ QUANDL_API_KEY=<yourkey> zipline ingest [-b <bundle>]
```

where `<bundle>` is the name of the bundle to ingest, defaulting to `quandl`.

you can check out the *ingesting data* section for more detail.

Command Line Interface

After you installed zipline you should be able to execute the following from your command line (e.g. `cmd.exe` on Windows, or the Terminal app on OSX):

```
$ zipline run --help
```

```
Usage: zipline run [OPTIONS]

Run a backtest for the given algorithm.

Options:
 -f, --algofile FILENAME         The file that contains the algorithm to run.
 -t, --algotext TEXT             The algorithm script to run.
 -D, --define TEXT               Define a name to be bound in the namespace
                                 before executing the algotext. For example
                                 '-Dname=value'. The value may be any python
                                 expression. These are evaluated in order so
                                 they may refer to previously defined names.
 --data-frequency [daily|minute]
                                 The data frequency of the simulation.
                                 [default: daily]
 --capital-base FLOAT            The starting capital for the simulation.
                                 [default: 10000000.0]
 -b, --bundle BUNDLE-NAME        The data bundle to use for the simulation.
                                 [default: quandl]
 --bundle-timestamp TIMESTAMP    The date to lookup data on or before.
                                 [default: <current-time>]
 -s, --start DATE                The start date of the simulation.
 -e, --end DATE                  The end date of the simulation.
 -o, --output FILENAME           The location to write the perf data. If this
                                 is '-' the perf will be written to stdout.
                                 [default: -]
 --trading-calendar TRADING-CALENDAR
                                 The calendar you want to use e.g. LSE. NYSE
                                 is the default.
 --print-algo / --no-print-algo  Print the algorithm to stdout.
 --benchmark-file                The csv file that contains the benchmark
                                 returns (date, returns columns)
 --benchmark-symbol              The instrument's symbol to be used as
                                 a benchmark.
                                 (should exist in the ingested bundle)
 --benchmark-sid                 The sid of the instrument to be used as a
                                 benchmark.
                                 (should exist in the ingested bundle)
 --no-benchmark                  This flag is used to set the benchmark to
                                 zero. Alpha, beta and benchmark metrics
                                 are not calculated
 --help                          Show this message and exit.
```

As you can see there are a couple of flags that specify where to find your algorithm (`-f`) as well as parameters specifying which data to use, defaulting to `quandl`. There are also arguments for the date range to run the algorithm over (`--start` and `--end`).To use a benchmark, you need to choose one of the benchmark options listed before. You can always use the option (`--no-benchmark`) that uses zero returns as a benchmark (alpha, beta and benchmark metrics are not calculated in this case). Finally, you'll want to save the performance metrics of your algorithm so that you can analyze how it performed. This is done via the `--output` flag and will cause it to write the performance `DataFrame` in the pickle Python file format. Note that you can also define a configuration file with these parameters

that you can then conveniently pass to the `-c` option so that you don't have to supply the command line args all the time (see the .conf files in the examples directory).

Thus, to execute our algorithm from above and save the results to `buyapple_out.pickle`, we call `zipline run` as follows:

```
zipline run -f ../zipline/examples/buyapple.py --start 2016-1-1 --end 2018-1-1 -o␣
↪buyapple_out.pickle --no-benchmark
```

```
AAPL
[2018-01-03 04:30:51.843465] INFO: Performance: Simulated 503 trading days out of 503.
[2018-01-03 04:30:51.843598] INFO: Performance: first open: 2016-01-04 14:31:00+00:00
[2018-01-03 04:30:51.843672] INFO: Performance: last close: 2017-12-29 21:00:00+00:00
```

`run` first calls the `initialize()` function, and then streams the historical stock price day-by-day through `handle_data()`. After each call to `handle_data()` we instruct `zipline` to order 10 stocks of AAPL. After the call of the `order()` function, `zipline` enters the ordered stock and amount in the order book. After the `handle_data()` function has finished, `zipline` looks for any open orders and tries to fill them. If the trading volume is high enough for this stock, the order is executed after adding the commission and applying the slippage model which models the influence of your order on the stock price, so your algorithm will be charged more than just the stock price * 10. (Note, that you can also change the commission and slippage model that `zipline` uses, see the Quantopian docs for more information).

Let's take a quick look at the performance `DataFrame`. For this, we use `pandas` from inside the IPython Notebook and print the first ten rows. Note that `zipline` makes heavy usage of `pandas`, especially for data input and outputting so it's worth spending some time to learn it.

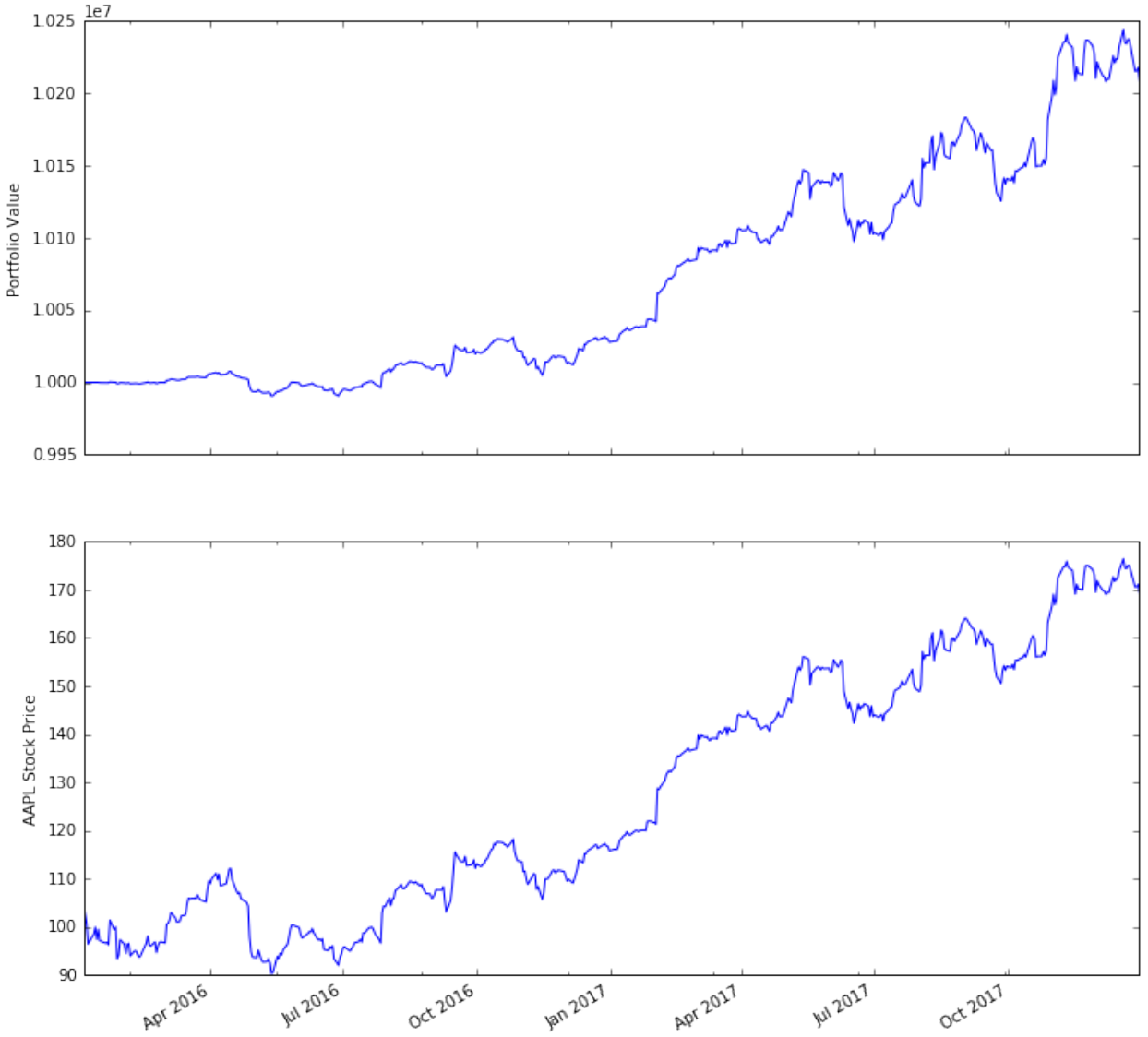
```
import pandas as pd
perf = pd.read_pickle('buyapple_out.pickle') # read in perf DataFrame
perf.head()
```

As you can see, there is a row for each trading day, starting on the first business day of 2016. In the columns you can find various information about the state of your algorithm. The very first column `AAPL` was placed there by the `record()` function mentioned earlier and allows us to plot the price of apple. For example, we could easily examine now how our portfolio value changed over time compared to the AAPL stock price.

```
%pylab inline
figsize(12, 12)
import matplotlib.pyplot as plt

ax1 = plt.subplot(211)
perf.portfolio_value.plot(ax=ax1)
ax1.set_ylabel('Portfolio Value')
ax2 = plt.subplot(212, sharex=ax1)
perf.AAPL.plot(ax=ax2)
ax2.set_ylabel('AAPL Stock Price')
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
<matplotlib.text.Text at 0x10c48c198>
```



As you can see, our algorithm performance as assessed by the `portfolio_value` closely matches that of the AAPL stock price. This is not surprising as our algorithm only bought AAPL every chance it got.

IPython Notebook

The IPython Notebook is a very powerful browser-based interface to a Python interpreter (this tutorial was written in it). As it is already the de-facto interface for most quantitative researchers `zipline` provides an easy way to run your algorithm inside the Notebook without requiring you to use the CLI.

To use it you have to write your algorithm in a cell and let `zipline` know that it is supposed to run this algorithm. This is done via the `%%zipline` IPython magic command that is available after you `import zipline` from within the IPython Notebook. This magic takes the same arguments as the command line interface described above. Thus to run the algorithm from above with the same parameters we just have to execute the following cell after importing `zipline` to register the magic.

```
%load_ext zipline
```

```
%%zipline --start 2016-1-1 --end 2018-1-1
from zipline.api import symbol, order, record

def initialize(context):
    pass

def handle_data(context, data):
    order(symbol('AAPL'), 10)
    record(AAPL=data[symbol('AAPL')].price)
```

Note that we did not have to specify an input file as above since the magic will use the contents of the cell and look for your algorithm functions there. Also, instead of defining an output file we are specifying a variable name with `-o` that will be created in the name space and contain the performance `DataFrame` we looked at above.

```
_.head()
```

Access to Previous Prices Using `history`

Working example: Dual Moving Average Cross-Over

The Dual Moving Average (DMA) is a classic momentum strategy. It's probably not used by any serious trader anymore but is still very instructive. The basic idea is that we compute two rolling or moving averages (mavg) – one with a longer window that is supposed to capture long-term trends and one shorter window that is supposed to capture short-term trends. Once the short-mavg crosses the long-mavg from below we assume that the stock price has upwards momentum and long the stock. If the short-mavg crosses from above we exit the positions as we assume the stock to go down further.

As we need to have access to previous prices to implement this strategy we need a new concept: History

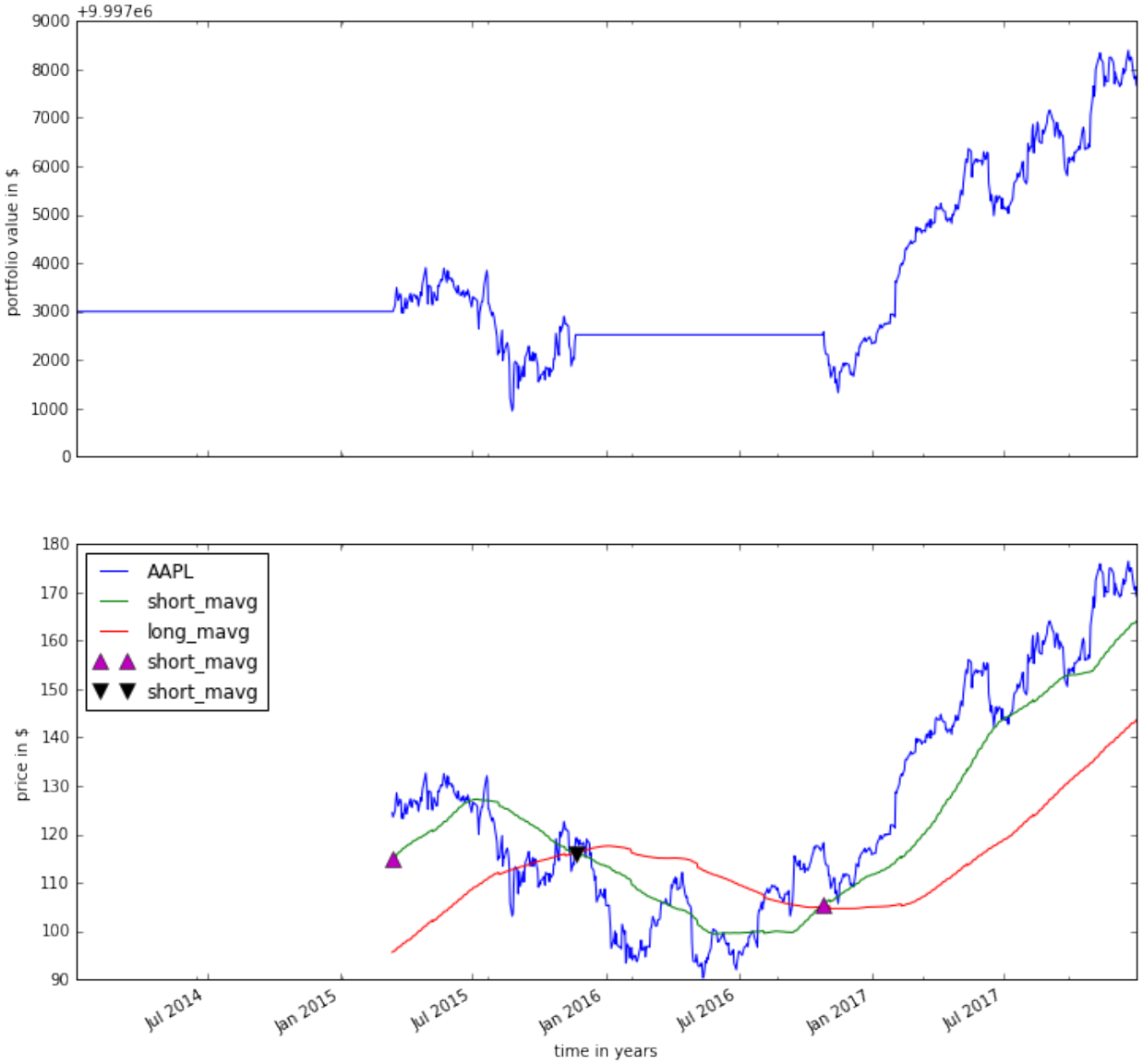
`data.history()` is a convenience function that keeps a rolling window of data for you. The first argument is the number of bars you want to collect, the second argument is the unit (either `'1d'` or `'1m'`, but note that you need to have minute-level data for using `1m`). For a more detailed description of `history()`'s features, see the Quantopian docs. Let's look at the strategy which should make this clear:

```
%%zipline --start 2014-1-1 --end 2018-1-1 -o dma.pickle


from zipline.api import order_target, record, symbol
import matplotlib.pyplot as plt

def initialize(context):
    context.i = 0
    context.asset = symbol('AAPL')


def handle_data(context, data):
    # Skip first 300 days to get full windows
    context.i += 1
    if context.i < 300:
        return

    # Compute averages
    # data.history() has to be called with the same params
    # from above and returns a pandas dataframe.
    short_mavg = data.history(context.asset, 'price', bar_count=100, frequency="1d").
↪mean()
```

(continues on next page)

(continued from previous page)

```
    long_mavg = data.history(context.asset, 'price', bar_count=300, frequency="1d").
↪mean()

    # Trading logic
    if short_mavg > long_mavg:
        # order_target orders as many shares as needed to
        # achieve the desired number of shares.
        order_target(context.asset, 100)
    elif short_mavg < long_mavg:
        order_target(context.asset, 0)

    # Save values for later inspection
    record(AAPL=data.current(context.asset, 'price'),
           short_mavg=short_mavg,
           long_mavg=long_mavg)


def analyze(context, perf):
    fig = plt.figure()
    ax1 = fig.add_subplot(211)
    perf.portfolio_value.plot(ax=ax1)
    ax1.set_ylabel('portfolio value in $')

    ax2 = fig.add_subplot(212)
    perf['AAPL'].plot(ax=ax2)
    perf[['short_mavg', 'long_mavg']].plot(ax=ax2)

    perf_trans = perf.ix[[t != [] for t in perf.transactions]]
    buys = perf_trans.ix[[t[0]['amount'] > 0 for t in perf_trans.transactions]]
    sells = perf_trans.ix[
        [t[0]['amount'] < 0 for t in perf_trans.transactions]]
    ax2.plot(buys.index, perf.short_mavg.ix[buys.index],
             '^', markersize=10, color='m')
    ax2.plot(sells.index, perf.short_mavg.ix[sells.index],
             'v', markersize=10, color='k')
    ax2.set_ylabel('price in $')
    plt.legend(loc=0)
    plt.show()
```



Here we are explicitly defining an `analyze()` function that gets automatically called once the backtest is done (this is not possible on Quantopian currently).

Although it might not be directly apparent, the power of `history()` (pun intended) can not be under-estimated as most algorithms make use of prior market developments in one form or another. You could easily devise a strategy that trains a classifier with scikit-learn which tries to predict future market movements based on past prices (note, that most of the `scikit-learn` functions require `numpy.ndarray`s rather than `pandas.DataFrame`s, so you can simply pass the underlying `ndarray` of a `DataFrame` via `.values`).

We also used the `order_target()` function above. This and other functions like it can make order management and portfolio rebalancing much easier. See the Quantopian documentation on order functions for more details.

Conclusions

We hope that this tutorial gave you a little insight into the architecture, API, and features of `zipline`. For next steps, check out some of the examples.

Feel free to ask questions on our mailing list, report problems on our GitHub issue tracker, get involved, and checkout Quantopian.

2.7.2 Data Bundles

A data bundle is a collection of pricing data, adjustment data, and an asset database. Bundles allow us to preload all of the data we will need to run backtests and store the data for future runs.

Discovering Available Bundles

Zipline comes with a few bundles by default as well as the ability to register new bundles. To see which bundles we have available, we may run the `bundles` command, for example:

```
$ zipline bundles
my-custom-bundle 2016-05-05 20:35:19.809398
my-custom-bundle 2016-05-05 20:34:53.654082
my-custom-bundle 2016-05-05 20:34:48.401767
quandl <no ingestions>
quantopian-quandl 2016-05-05 20:06:40.894956
```

The output here shows that there are 3 bundles available:

- `my-custom-bundle` (added by the user)
- `quandl` (provided by zipline, though deprecated)
- `quantopian-quandl` (provided by zipline, the default bundle)

The dates and times next to the name show the times when the data for this bundle was ingested. We have run three different ingestions for `my-custom-bundle`. We have never ingested any data for the `quandl` bundle so it just shows `<no ingestions>` instead. Finally, there is only one ingestion for `quantopian-quandl`.

Ingesting Data

The first step to using a data bundle is to ingest the data. The ingestion process will invoke some custom bundle command and then write the data to a standard location that zipline can find. By default the location where ingested data will be written is `$ZIPLINE_ROOT/data/<bundle>` where by default `ZIPLINE_ROOT=~/.zipline`. The ingestion step may take some time as it could involve downloading and processing a lot of data. To ingest a bundle, run:

```
$ zipline ingest [-b <bundle>]
```

where `<bundle>` is the name of the bundle to ingest, defaulting to `quantopian-quandl`.

Old Data

When the `ingest` command is used it will write the new data to a subdirectory of `$ZIPLINE_ROOT/data/<bundle>` which is named with the current date. This makes it possible to look at older data or even run backtests with the older copies. Running a backtest with an old ingestion makes it easier to reproduce backtest results later.

One drawback of saving all of the data by default is that the data directory may grow quite large even if you do not want to use the data. As shown earlier, we can list all of the ingestions with the *bundles command*. To solve the problem of leaking old data there is another command: `clean`, which will clear data bundles based on some time constraints.

For example:

```
# clean everything older than <date>
$ zipline clean [-b <bundle>] --before <date>

# clean everything newer than <date>
$ zipline clean [-b <bundle>] --after <date>

# keep everything in the range of [before, after] and delete the rest
$ zipline clean [-b <bundle>] --before <date> --after <after>

# clean all but the last <int> runs
$ zipline clean [-b <bundle>] --keep-last <int>
```

Running Backtests with Data Bundles

Now that the data has been ingested we can use it to run backtests with the `run` command. The bundle to use can be specified with the `--bundle` option like:

```
$ zipline run --bundle <bundle> --algofile algo.py ...
```

We may also specify the date to use to look up the bundle data with the `--bundle-timestamp` option. Setting the `--bundle-timestamp` will cause `run` to use the most recent bundle ingestion that is less than or equal to the `bundle-timestamp`. This is how we can run backtests with older data. `bundle-timestamp` uses a less-than-or-equal-to relationship so that we can specify the date that we ran an old backtest and get the same data that would have been available to us on that date. The `bundle-timestamp` defaults to the current day to use the most recent data.

Default Data Bundles

Quandl WIKI Bundle

By default zipline comes with the `quantopian-quandl` data bundle which uses quandl's WIKI dataset. The quandl data bundle includes daily pricing data, splits, cash dividends, and asset metadata. Quantopian has ingested the data from quandl and rebundled it to make ingestion much faster. To ingest the `quantopian-quandl` data bundle, run either of the following commands:

```
$ zipline ingest -b quantopian-quandl
$ zipline ingest
```

Either command should only take a few seconds to download the data.

Note: Quandl has discontinued this dataset. The dataset is no longer updating, but is reasonable for trying out Zipline without setting up your own dataset.

Writing a New Bundle

Data bundles exist to make it easy to use different data sources with zipline. To add a new bundle, one must implement an `ingest` 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. The ingest function may work by downloading data from a remote location like the `quandl` bundle or it may just load files that are already on the machine. The function is provided with writers that will write the data to the correct location transactionally. If an ingestion fails part way through the bundle will not be written in an incomplete state.

The signature of the ingest function should be:

```
ingest(environ,
       asset_db_writer,
       minute_bar_writer,
       daily_bar_writer,
       adjustment_writer,
       calendar,
       start_session,
       end_session,
       cache,
       show_progress,
       output_dir)
```

environ

`environ` is a mapping representing the environment variables to use. This is where any custom arguments needed for the ingestion should be passed, for example: the `quandl` bundle uses the environment to pass the API key and the download retry attempt count.

asset_db_writer

`asset_db_writer` is an instance of `AssetDBWriter`. This is the writer for the asset metadata which provides the asset lifetimes and the symbol to asset id (sid) mapping. This may also contain the asset name, exchange and a few other columns. To write data, invoke `write()` with dataframes for the various pieces of metadata. More information about the format of the data exists in the docs for write.

minute_bar_writer

`minute_bar_writer` is an instance of `BcolzMinuteBarWriter`. This writer is used to convert data to zipline's internal bcolz format to later be read by a `BcolzMinuteBarReader`. If minute data is provided, users should call `write()` with an iterable of (sid, dataframe) tuples. The `show_progress` argument should also be forwarded to this method. If the data source does not provide minute level data, then there is no need to call the write method. It is also acceptable to pass an empty iterator to `write()` to signal that there is no minutely data.

Note: The data passed to `write()` may be a lazy iterator or generator to avoid loading all of the minute data into memory at a single time. A given sid may also appear multiple times in the data as long as the dates are strictly increasing.

---

daily_bar_writer

`daily_bar_writer` is an instance of `BcolzDailyBarWriter`. This writer is used to convert data into zipline's internal bcolz format to later be read by a `BcolzDailyBarReader`. If daily data is provided, users should call `write()` with an iterable of (sid dataframe) tuples. The `show_progress` argument should also be forwarded to this method. If the data source does not provide daily data, then there is no need to call the write method. It is also acceptable to pass an empty iterable to `write()` to signal that there is no daily data. If no daily data is provided but minute data is provided, a daily rollup will happen to service daily history requests.

Note: Like the `minute_bar_writer`, the data passed to `write()` may be a lazy iterable or generator to avoid loading all of the data into memory at once. Unlike the `minute_bar_writer`, a sid may only appear once in the data iterable.

---

adjustment_writer

`adjustment_writer` is an instance of `SQLiteAdjustmentWriter`. This writer is used to store splits, mergers, dividends, and stock dividends. The data should be provided as dataframes and passed to `write()`. Each of these fields are optional, but the writer can accept as much of the data as you have.

calendar

`calendar` is an instance of `zipline.utils.calendars.TradingCalendar`. The calendar is provided to help some bundles generate queries for the days needed.

start_session

`start_session` is a `pandas.Timestamp` object indicating the first day that the bundle should load data for.

end_session

`end_session` is a `pandas.Timestamp` object indicating the last day that the bundle should load data for.

cache

`cache` is an instance of `dataframe_cache`. This object is a mapping from strings to dataframes. This object is provided in case an ingestion crashes part way through. The idea is that the ingest function should check the cache for raw data, if it doesn't exist in the cache, it should acquire it and then store it in the cache. Then it can parse and write the data. The cache will be cleared only after a successful load, this prevents the ingest function from needing to re-download all the data if there is some bug in the parsing. If it is very fast to get the data, for example if it is coming from another local file, then there is no need to use this cache.

show_progress

`show_progress` is a boolean indicating that the user would like to receive feedback about the ingest function's progress fetching and writing the data. Some examples for where to show how many files you have downloaded out of the total needed, or how far into some data conversion the ingest function is. One tool that may help with implementing `show_progress` for a loop is `maybe_show_progress`. This argument should always be forwarded to `minute_bar_writer.write` and `daily_bar_writer.write`.

output_dir

`output_dir` is a string representing the file path where all the data will be written. `output_dir` will be some subdirectory of `$ZIPLINE_ROOT` and will contain the time of the start of the current ingestion. This can be used to directly move resources here if for some reason your ingest function can produce it's own outputs without the writers. For example, the `quantopian:quandl` bundle uses this to directly untar the bundle into the `output_dir`.

Ingesting Data from .csv Files

Zipline provides a bundle called `csvdir`, which allows users to ingest data from `.csv` files. The format of the files should be in OHLCV format, with dates, dividends, and splits. A sample is provided below. There are other samples for testing purposes in `zipline/tests/resources/csvdir_samples`.

```
date,open,high,low,close,volume,dividend,split
2012-01-03,58.485714,58.92857,58.42857,58.747143,75555200,0.0,1.0
2012-01-04,58.57143,59.240002,58.468571,59.062859,65005500,0.0,1.0
2012-01-05,59.278572,59.792858,58.952858,59.718571,67817400,0.0,1.0
2012-01-06,59.967144,60.392857,59.888573,60.342857,79573200,0.0,1.0
```

(continues on next page)

(continued from previous page)

```
2012-01-09,60.785713,61.107143,60.192856,60.247143,98506100,0.0,1.0
2012-01-10,60.844284,60.857143,60.214287,60.462856,64549100,0.0,1.0
2012-01-11,60.382858,60.407143,59.901428,60.364285,53771200,0.0,1.0
```

Once you have your data in the correct format, you can edit your `extension.py` file in `~/.zipline/extension.py` and import the csvdir bundle, along with `pandas`.

```
import pandas as pd

from zipline.data.bundles import register
from zipline.data.bundles.csvdir import csvdir_equities
```

We'll then want to specify the start and end sessions of our bundle data:

```
start_session = pd.Timestamp('2016-1-1', tz='utc')
end_session = pd.Timestamp('2018-1-1', tz='utc')
```

And then we can `register()` our bundle, and pass the location of the directory in which our `.csv` files exist:

```
register(
    'custom-csvdir-bundle',
    csvdir_equities(
        ['daily'],
        '/path/to/your/csvs',
    ),
    calendar_name='NYSE', # US equities
    start_session=start_session,
    end_session=end_session
)
```

To finally ingest our data, we can run:

```
$ zipline ingest -b custom-csvdir-bundle
Loading custom pricing data:   [############------------------------]   33% | FAKE:␣
↪sid 0
Loading custom pricing data:   [########################------------]   66% | FAKE1:␣
↪sid 1
Loading custom pricing data:   [####################################]  100% | FAKE2:␣
↪sid 2
Loading custom pricing data:   [####################################]  100%
Merging daily equity files:  [####################################]

# optionally, we can pass the location of our csvs via the command line
$ CSVDIR=/path/to/your/csvs zipline ingest -b custom-csvdir-bundle
```

If you would like to use equities that are not in the NYSE calendar, or the existing zipline calendars, you can look at the `Trading Calendar Tutorial` to build a custom trading calendar that you can then pass the name of to `register()`.

2.7.3 Trading Calendars

What is a Trading Calendar?

A trading calendar represents the timing information of a single market exchange. The timing information is made up of two parts: sessions, and opens/closes. This is represented by the Zipline `TradingCalendar` class, and is used as the parent class for all new `TradingCalendar` s.

A session represents a contiguous set of minutes, and has a label that is midnight UTC. It is important to note that a session label should not be considered a specific point in time, and that midnight UTC is just being used for convenience.

For an average day of the New York Stock Exchange, the market opens at 9:30AM and closes at 4PM. Trading sessions can change depending on the exchange, day of the year, etc.

Why Should You Care About Trading Calendars?

Let's say you want to buy a share of some equity on Tuesday, and then sell it on Saturday. If the exchange in which you're trading that equity is not open on Saturday, then in reality it would not be possible to trade that equity at that time, and you would have to wait until some other number of days past Saturday. Since you wouldn't be able to place the trade in reality, it would also be unreasonable for your backtest to place a trade on Saturday.

In order for you to backtest your strategy, the dates in that are accounted for in your data bundle and the dates in your `TradingCalendar` should match up; if the dates don't match up, then you you're going to see some errors along the way. This holds for both minutely and daily data.

The TradingCalendar Class

The `TradingCalendar` class has many properties we should be thinking about if we were to build our own `TradingCalendar` for an exchange. These include properties such as:

- Name of the Exchange
- Timezone
- Open Time
- Close Time
- Regular & Ad hoc Holidays
- Special Opens & Closes

And several others. If you'd like to see all of the properties and methods available to you through the `TradingCalendar` API, please take a look at the API Reference

Now we'll take a look at the London Stock Exchange Calendar `LSEExchangeCalendar` as an example below:

```
class LSEExchangeCalendar(TradingCalendar):
  """
  Exchange calendar for the London Stock Exchange

  Open Time: 8:00 AM, GMT
  Close Time: 4:30 PM, GMT

  Regularly-Observed Holidays:
    - New Years Day (observed on first business day on/after)
    - Good Friday
```

(continues on next page)

(continued from previous page)

```
    - Easter Monday
    - Early May Bank Holiday (first Monday in May)
    - Spring Bank Holiday (last Monday in May)
    - Summer Bank Holiday (last Monday in May)
    - Christmas Day
    - Dec. 27th (if Christmas is on a weekend)
    - Boxing Day
    - Dec. 28th (if Boxing Day is on a weekend)
  """

  @property
  def name(self):
    return "LSE"

  @property
  def tz(self):
    return timezone('Europe/London')

  @property
  def open_time(self):
    return time(8, 1)

  @property
  def close_time(self):
    return time(16, 30)

  @property
  def regular_holidays(self):
    return HolidayCalendar([
      LSENewYearsDay,
      GoodFriday,
      EasterMonday,
      MayBank,
      SpringBank,
      SummerBank,
      Christmas,
      WeekendChristmas,
      BoxingDay,
      WeekendBoxingDay
    ])
```

You can create the `Holiday` objects mentioned in `def regular_holidays(self)` through the `pandas <https://pandas.pydata.org/pandas-docs/stable/>`__ module, ``pandas.tseries.holiday.Holiday``, and also take a look at the LSEExchangeCalendar code as an example, or take a look at the code snippet below.

```
from pandas.tseries.holiday import (
    Holiday,
    DateOffset,
    MO
)

SomeSpecialDay = Holiday(
    "Some Special Day",
    month=1,
    day=9,
```

(continues on next page)

(continued from previous page)

```
    offset=DateOffSet(weekday=MO(-1))
)
```

Building a Custom Trading Calendar

Now we'll build our own custom trading calendar. This calendar will be used for trading assets that can be traded on a 24/7 exchange calendar. This means that it will be open on Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday, and the exchange will open at 12AM and close at 11:59PM. The timezone which we'll use is UTC.

First we'll start off by importing some modules that will be useful to us.

```
# for setting our open and close times
from datetime import time
# for setting our start and end sessions
import pandas as pd
# for setting which days of the week we trade on
from pandas.tseries.offsets import CustomBusinessDay
# for setting our timezone
from pytz import timezone

# for creating and registering our calendar
from trading_calendars import register_calendar, TradingCalendar
from zipline.utils.memoize import lazyval
```

And now we'll actually build this calendar, which we'll call `TFSExchangeCalendar`:

```
class TFSExchangeCalendar(TradingCalendar):
  """
  An exchange calendar for trading assets 24/7.

  Open Time: 12AM, UTC
  Close Time: 11:59PM, UTC
  """

  @property
  def name(self):
    """
    The name of the exchange, which Zipline will look for
    when we run our algorithm and pass TFS to
    the --trading-calendar CLI flag.
    """
    return "TFS"

  @property
  def tz(self):
    """
    The timezone in which we'll be running our algorithm.
    """
    return timezone("UTC")

  @property
  def open_time(self):
    """
    The time in which our exchange will open each day.
    """
```

(continues on next page)

(continued from previous page)

```
    return time(0, 0)

  @property
  def close_time(self):
    """
    The time in which our exchange will close each day.
    """
    return time(23, 59)

  @lazyval
  def day(self):
    """
    The days on which our exchange will be open.
    """
    weekmask = "Mon Tue Wed Thu Fri Sat Sun"
    return CustomBusinessDay(
      weekmask=weekmask
    )
```

Conclusions

In order for you to run your algorithm with this calendar, you'll need have a data bundle in which your assets have dates that run through all days of the week. You can read about how to make your own data bundle in the Writing a New Bundle documentation, or use the csvdir bundle for creating a bundle from CSV files.

2.7.4 Risk and Performance Metrics

The risk and performance metrics are summarizing values calculated by Zipline when running a simulation. These metrics can be about the performance of an algorithm, like returns or cash flow, or the riskiness of an algorithm, like volatility or beta. Metrics may be reported minutely, daily, or once at the end of a simulation. A single metric may choose to report at multiple time-scales where appropriate.

Metrics Sets

Zipline groups risk and performance metrics into collections called "metrics sets". A single metrics set defines all of the metrics to track during a single backtest. A metrics set may contain metrics that report at different time scales. The default metrics set will compute a host of metrics, such as algorithm returns, volatility, Sharpe ratio, and beta.

Selecting the Metrics Set

When running a simulation, the user may select the metrics set to report. How you select the metrics set depends on the interface being used to run the algorithm.

Command Line and IPython Magic

When running with the command line or IPython magic interfaces, the metrics set may be selected by passing the `--metrics-set` argument. For example:

```
$ zipline run algorithm.py -s 2014-01-01 -e 2014-02-01 --metrics-set my-metrics-set
```

run_algorithm

When running through the `run_algorithm()` interface, the metrics set may be passed with the `metrics_set` argument. This may either be the name of a registered metrics set, or a set of metric object. For example:

```
run_algorithm(..., metrics_set='my-metrics-set')
run_algorithm(..., metrics_set={MyMetric(), MyOtherMetric(), ...})
```

Running Without Metrics

Computing risk and performance metrics is not free, and contributes to the total runtime of a backtest. When actively developing an algorithm, it is often helpful to skip these computations to speed up the debugging cycle. To disable the calculation and reporting of all metrics, users may select the built-in metrics set `none`. For example:

```
$ zipline run algorithm.py -s 2014-01-01 -e 2014-02-01 --metrics-set none
```

Defining New Metrics

A metric is any object that implements some subset of the following methods:

- `start_of_simulation`
- `end_of_simulation`
- `start_of_session`
- `end_of_session`
- `end_of_bar`

These functions will be called at the time indicated by their name, at which point the metric object may collect any needed information and optionally report a computed value. If a metric does not need to do any processing at one of these times, it may omit a definition for the given method.

A metric should be reusable, meaning that a single instance of a metric class should be able to be used across multiple backtests. Metrics do not need to support multiple simulations at once, meaning that internal caches and data are consistent between `start_of_simulation` and `end_of_simulation`.

start_of_simulation

The `start_of_simulation` method should be thought of as a per-simulation constructor. This method should initialize any caches needed for the duration of a single simulation.

The `start_of_simulation` method should have the following signature:

```
def start_of_simulation(self,
                        ledger,
                        emission_rate,
                        trading_calendar,
                        sessions,
                        benchmark_source):
    ...
```

`ledger` is an instance of `Ledger` which is maintaining the simulation's state. This may be used to lookup the algorithm's starting portfolio values.

`emission_rate` is a string representing the smallest frequency at which metrics should be reported. `emission_rate` will be either `minute` or `daily`. When `emission_rate` is `daily`, `end_of_bar` will not be called at all.

`trading_calendar` is an instance of `TradingCalendar` which is the trading calendar being used by the simulation.

`sessions` is a `pandas.DatetimeIndex` which is holds the session labels, in sorted order, that the simulation will execute.

`benchmark_source` is an instance of `BenchmarkSource` which is the interface to the returns of the benchmark specified by `set_benchmark()`.

end_of_simulation

The `end_of_simulation` method should have the following signature:

```
def end_of_simulation(self,
                      packet,
                      ledger,
                      trading_calendar,
                      sessions,
                      data_portal,
                      benchmark_source):
    ...
```

`ledger` is an instance of `Ledger` which is maintaining the simulation's state. This may be used to lookup the algorithm's final portfolio values.

`packet` is a dictionary to write the end of simulation values for the given metric into.

`trading_calendar` is an instance of `TradingCalendar` which is the trading calendar being used by the simulation.

`sessions` is a `pandas.DatetimeIndex` which is holds the session labels, in sorted order, that the simulation has executed.

`data_portal` is an instance of `DataPortal` which is the metric's interface to pricing data.

`benchmark_source` is an instance of `BenchmarkSource` which is the interface to the returns of the benchmark specified by `set_benchmark()`.

start_of_session

The start_of_session method may see a slightly different view of the ledger or data_portal than the previous end_of_session if the price of any futures owned move between trading sessions or if a capital change occurs.

The start_of_session method should have the following signature:

```
def start_of_session(self,
                     ledger,
                     session_label,
                     data_portal):
    ...
```

ledger is an instance of Ledger which is maintaining the simulation's state. This may be used to lookup the algorithm's current portfolio values.

session_label is a Timestamp which is the label of the session which is about to run.

data_portal is an instance of DataPortal which is the metric's interface to pricing data.

end_of_session

The end_of_session method should have the following signature:

```
def end_of_session(self,
                   packet,
                   ledger,
                   session_label,
                   session_ix,
                   data_portal):
```

packet is a dictionary to write the end of session values. This dictionary contains two sub-dictionaries: daily_perf and cumulative_perf. When applicable, the daily_perf should hold the current day's value, and cumulative_perf should hold a cumulative value for the entire simulation up to the current time.

ledger is an instance of Ledger which is maintaining the simulation's state. This may be used to lookup the algorithm's current portfolio values.

session_label is a Timestamp which is the label of the session which is has just completed.

session_ix is an int which is the index of the current trading session being run. This is provided to allow for efficient access to the daily returns through ledger.daily_returns_array[:session_ix + 1].

data_portal is an instance of DataPortal which is the metric's interface to pricing data

end_of_bar

Note: end_of_bar is only called when emission_mode is minute.

The end_of_bar method should have the following signature:

```
def end_of_bar(self,
               packet,
               ledger,
```

(continues on next page)

(continued from previous page)

```
                dt,
                session_ix,
                data_portal):
```

`packet` is a dictionary to write the end of session values. This dictionary contains two sub-dictionaries: `minute_perf` and `cumulative_perf`. When applicable, the `minute_perf` should hold the current partial day's value, and `cumulative_perf` should hold a cumulative value for the entire simulation up to the current time.

`ledger` is an instance of `Ledger` which is maintaining the simulation's state. This may be used to lookup the algorithm's current portfolio values.

`dt` is a `Timestamp` which is the label of bar that has just completed.

`session_ix` is an `int` which is the index of the current trading session being run. This is provided to allow for efficient access to the daily returns through `ledger.daily_returns_array[:session_ix + 1]`.

`data_portal` is an instance of `DataPortal` which is the metric's interface to pricing data.

Defining New Metrics Sets

Users may use `zipline.finance.metrics.register()` to register a new metrics set. This may be used to decorate a function taking no arguments which returns a new set of metric object instances. For example:

```
from zipline.finance import metrics

@metrics.register('my-metrics-set')
def my_metrics_set():
    return {MyMetric(), MyOtherMetric(), ...}
```

This may be embedded in the user's `extension.py`.

The reason that a metrics set is defined as a function which produces a set, instead of just a set, is that users may want to fetch external data or resources to construct their metrics. By putting this behind a callable, users do not need to fetch the resources when the metrics set is not being used.

2.7.5 Development Guidelines

This page is intended for developers of Zipline, people who want to contribute to the Zipline codebase or documentation, or people who want to install from source and make local changes to their copy of Zipline.

All contributions, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome. We track issues on GitHub and also have a mailing list where you can ask questions.

Creating a Development Environment

First, you'll need to clone Zipline by running:

```
$ git clone git@github.com:your-github-username/zipline.git
```

Then check out to a new branch where you can make your changes:

```
$ git checkout -b some-short-descriptive-name
```

If you don't already have them, you'll need some C library dependencies. You can follow the install guide to get the appropriate dependencies.

Once you've created and activated a virtual environment, run the `etc/dev-install` script to install all development dependencies in their required order:

```
$ python3 -m venv venv
$ source venv/bin/activate
$ etc/dev-install
```

Or, using virtualenvwrapper:

```
$ mkvirtualenv zipline
$ etc/dev-install
```

After installation, you should be able to use the `zipline` command line interface from your virtualenv:

```
$ zipline --help
```

To finish, make sure *tests* pass.

If you get an error running nosetests after setting up a fresh virtualenv, please try running

```
# where zipline is the name of your virtualenv
$ deactivate zipline
$ workon zipline
```

During development, you can rebuild the C extensions by running:

```
$ python setup.py build_ext --inplace
```

Development with Docker

If you want to work with zipline using a Docker container, you'll need to build the `Dockerfile` in the Zipline root directory, and then build `Dockerfile-dev`. Instructions for building both containers can be found in `Dockerfile` and `Dockerfile-dev`, respectively.

Style Guide & Running Tests

We use flake8 for checking style requirements and nosetests to run Zipline tests. Our continuous integration tools will run these commands.

Before submitting patches or pull requests, please ensure that your changes pass when running:

```
$ flake8 zipline tests
```

In order to run tests locally, you'll need TA-lib, which you can install on Linux by running:

```
$ wget http://prdownloads.sourceforge.net/ta-lib/ta-lib-0.4.0-src.tar.gz
$ tar -xvzf ta-lib-0.4.0-src.tar.gz
$ cd ta-lib/
$ ./configure --prefix=/usr
$ make
$ sudo make install
```

And for `TA-lib` on OS X you can just run:

```
$ brew install ta-lib
```

Then run `pip install` TA-lib:

```
$ pip install -r ./etc/requirements_talib.in -c ./etc/requirements_locked.txt
```

You should now be free to run tests:

```
$ nosetests
```

Continuous Integration

We use Travis CI for Linux-64 bit builds and AppVeyor for Windows-64 bit builds.

Note: We do not currently have CI for OSX-64 bit builds. 32-bit builds may work but are not included in our integration tests.

Packaging

To learn about how we build Zipline conda packages, you can read this section in our release process notes.

Updating dependencies

If you update the zipline codebase so that it now depends on a new version of a library, then you should update the lower bound on that dependency in `etc/requirements.in` (or `etc/requirements_dev.in` as appropriate). We use pip-compile to find mutually compatible versions of dependencies for the `etc/requirements_locked.txt` lockfile used in our CI environments.

When you update a dependency in an `.in` file, you need to re-run the `pip-compile` command included in the header of the lockfile; otherwise the lockfile will not meet the constraints specified to pip by zipline at install time (via `etc/requirements.in` via `setup.py`).

If the zipline codebase can still support an old version of a dependency, but you want to update to a newer version of that library in our CI environments, then only the lockfile needs updating. To update the lockfile without bumping the lower bound, re-run the `pip-compile` command included in the header of the lockfile with the addition of the `--upgrade-package` or `-P` flag, e.g.

```
$ pip-compile --output-file=etc/reqs.txt etc/reqs.in ... -P six==1.13.0 -P "click>4.0.
↪0"
```

As you can see above, you can include multiple such constraints in a single invocation of `pip-compile`.

Contributing to the Docs

If you'd like to contribute to the documentation on zipline.io, you can navigate to `docs/source/` where each reStructuredText (`.rst`) file is a separate section there. To add a section, create a new file called `some-descriptive-name.rst` and add `some-descriptive-name` to `appendix.rst`. To edit a section, simply open up one of the existing files, make your changes, and save them.

We use Sphinx to generate documentation for Zipline, which you will need to install by running:

```
$ pip install -r ./etc/requirements_docs.in -c ./etc/requirements_locked.txt
```

If you would like to use Anaconda, please follow the installation guide to create and activate an environment, and then run the command above.

To build and view the docs locally, run:

```
# assuming you're in the Zipline root directory
$ cd docs
$ make html
$ {BROWSER} build/html/index.html
```

Commit messages

Standard prefixes to start a commit message:

```
BLD: change related to building Zipline
BUG: bug fix
DEP: deprecate something, or remove a deprecated object
DEV: development tool or utility
DOC: documentation
ENH: enhancement
MAINT: maintenance commit (refactoring, typos, etc)
REV: revert an earlier commit
STY: style fix (whitespace, PEP8, flake8, etc)
TST: addition or modification of tests
REL: related to releasing Zipline
PERF: performance enhancements
```

Some commit style guidelines:

Commit lines should be no longer than 72 characters. The first line of the commit should include one of the above prefixes. There should be an empty line between the commit subject and the body of the commit. In general, the message should be in the imperative tense. Best practice is to include not only what the change is, but why the change was made.

Example:

```
MAINT: Remove unused calculations of max_leverage, et al.

In the performance period the max_leverage, max_capital_used,
cumulative_capital_used were calculated but not used.

At least one of those calculations, max_leverage, was causing a
divide by zero error.

Instead of papering over that error, the entire calculation was
a bit suspect so removing, with possibility of adding it back in
```

(continues on next page)

(continued from previous page)

```
later with handling the case (or raising appropriate errors) when
the algorithm has little cash on hand.
```

Formatting Docstrings

When adding or editing docstrings for classes, functions, etc, we use numpy as the canonical reference.

Updating the Whatsnew

We have a set of whatsnew files that are used for documenting changes that have occurred between different versions of Zipline. Once you've made a change to Zipline, in your Pull Request, please update the most recent `whatsnew` file with a comment about what you changed. You can find examples in previous `whatsnew` files.

2.7.6 API Reference

Running a Backtest

Algorithm API

The following methods are available for use in the `initialize`, `handle_data`, and `before_trading_start` API functions.

In all listed functions, the `self` argument is implicitly the currently-executing `TradingAlgorithm` instance.

Data Object

Scheduling Functions

Orders

Order Cancellation Policies

Assets

Trading Controls

Zipline provides trading controls to help ensure that the algorithm is performing as expected. The functions help protect the algorithm from certian bugs that could cause undesirable behavior when trading with real money.

Simulation Parameters

Commission Models

Slippage Models

Pipeline

For more information, see *Pipeline API*

Miscellaneous

Blotters

Pipeline API

Built-in Factors

Built-in Filters

Pipeline Engine

Data Loaders

Asset Metadata

Trading Calendar API

Data API

Writers

Readers

Bundles

`zipline.data.bundles.`**`bundles`**
The bundles that have been registered as a mapping from bundle name to bundle data. This mapping is immutable and may only be updated through `register()` or `unregister()`.

Risk Metrics

Algorithm State

Built-in Metrics

Metrics Sets

zipline.data.finance.metrics.**metrics_sets**
The metrics sets that have been registered as a mapping from metrics set name to load function. This mapping is immutable and may only be updated through `register()` or `unregister()`.

Utilities

Caching

Command Line

2.7.7 Release Process

Note: This page is intended for developers of zipline.

Updating the Release Notes

When we are ready to ship a new release of zipline, edit the *Release Notes* page. We will have been maintaining a `whatsnew` file while working on the release with the new version. First, find that file in: `docs/source/whatsnew/<version>.txt`. It will be the highest version number. Edit the release date field to be today's date in the format:

```
<month> <day>, <year>
```

for example, November 6, 2015. Remove the active development warning from the `whatsnew`, since it will no longer be pending release. Update the title of the release from "Development" to "Release x.x.x" and update the underline of the title to match the title's width.

If you are renaming the release at this point, you'll need to git mv the file and also update releases.rst to reference the renamed file.

To build and view the docs locally, run:

```
$ cd docs
$ make html
$ {BROWSER} build/html/index.html
```

Updating the Python stub files

PyCharm and other linters and type checkers can use Python stub files for type hinting. For example, we generate stub files for the `api` namespace, since that namespace is populated at import time by decorators on TradingAlgorithm methods. Those functions are therefore hidden from static analysis tools, but we can generate static files to make them available. Under **Python 3**, run the following to generate any stub files:

```
$ python etc/gen_type_stubs.py
```

Note: In order to make stub consumers aware of the classes referred to in the stub, the stub file should import those classes. However, since `... import *` and `... import ... as ...` in a stub file will export those imports, we import the names explicitly. For the stub for `zipline.api`, this is done in a header string in the `gen_type_stubs.py` script mentioned above. If new classes are added as parameters or return types of `zipline.api` functions, then new imports should be added to that header.

Updating the `__version__`

We use versioneer to manage the `__version__` and `setup.py` version. This means that we pull this information from our version control's tags to ensure that they stay in sync and to have very fine grained version strings for development installs.

To upgrade the version use the git tag command like:

```
$ git tag <major>.<minor>.<micro>
$ git push && git push --tags
```

This will push the the code and the tag information.

Next, click the "Draft a new release" button on the zipline releases page. For the new release, choose the tag you just pushed, and publish the release.

Uploading PyPI packages

`sdist`

To build the `sdist` (source distribution) run:

```
$ python setup.py sdist
```

from the zipline root. This will create a gzipped tarball that includes all the python, cython, and miscellaneous files needed to install zipline. To test that the source dist worked correctly, `cd` into an empty directory, create a new virtualenv and then run:

```
$ pip install <zipline-root>/dist/zipline-<major>.<minor>.<micro>.tar.gz
$ python -c 'import zipline;print(zipline.__version__)'
```

This should print the version we are expecting to release.

Note: It is very important to both `cd` into a clean directory and make a clean virtualenv. Changing directories ensures that we have included all the needed files in the manifest. Using a clean virtualenv ensures that we have listed all the required packages.

Now that we have tested the package locally, it should be tested using the test PyPI server.

```
$ pip install twine
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/zipline-<version-
↪number>.tar.gz
```

Twine will prompt you for a username and password, which you should have access to if you're authorized to push Zipline releases.

Note: If the package version has been taken: locally update your setup.py to override the version with a new number. Do not use the next version, just append a `.<nano>` section to the current version. PyPI prevents the same package version from appearing twice, so we need to work around this when debugging packaging problems on the test server.

Warning: Do not commit the temporary version change.

This will upload zipline to the pypi test server. To test installing from pypi, create a new virtualenv, `cd` into a clean directory and then run:

```
$ pip install --extra-index-url https://test.pypi.org/simple zipline
$ python -c 'import zipline;print(zipline.__version__)'
```

This should pull the package you just uploaded and then print the version number.

Now that we have tested locally and on PyPI test, it is time to upload to PyPI:

```
$ twine upload dist/zipline-<version-number>.tar.gz
```

bdist

Because zipline now supports multiple versions of numpy, we're not building binary wheels, since they are not tagged with the version of numpy with which they were compiled.

Documentation

To update zipline.io, checkout the latest master and run:

```
python <zipline_root>/docs/deploy.py
```

This will build the documentation, checkout a fresh copy of the `gh-pages` git branch, and copy the built docs into the zipline root.

Note: The docs should always be built with **Python 3**. Many of our api functions are wrapped by preprocessing functions which accept *args and **kwargs. In Python 3, sphinx will respect the `__wrapped__` attribute and display the correct arguments.

Now, using our browser of choice, view the `index.html` page and verify that the docs look correct.

Once we are happy, push the updated docs to the GitHub `gh-pages` branch.

```
$ git add .
$ git commit -m "DOC: update zipline.io"
$ git push origin gh-pages
```

zipline.io will update in a few moments.

Uploading conda packages

Travis and AppVeyor build zipline conda packages for us (for Linux/OSX and Windows respectively). Once they have built and uploaded to anaconda.org the zipline packages for the release commit to master, we should move those packages from the "ci" label to the "main" label. We should also do this for any packages we uploaded for zipline's dependencies. You can do this from the anaconda.org web interface. This is also a good time to remove all the old "ci" packages from anaconda.

To build the conda packages for zipline locally, run:

```
$ python etc/conda_build_matrix.py
```

If all of the builds succeed, then this will not print anything and exit with `EXIT_SUCCESS`. If there are build issues, we must address them and decide what to do.

Once all of the builds in the matrix pass, we can upload them to anaconda with:

```
$ python etc/conda_build_matrix.py --upload
```

If you would like to test this command by uploading to a different user, this may be specified with the `--user` flag.

Next Commit

Push a new commit post-release that adds the `whatsnew` for the next release, which should be titled according to a micro version increment. If that next release turns out to be a major/minor version increment, the file can be renamed when that's decided. You can use `docs/source/whatsnew/skeleton.txt` as a template for the new file.

Include the `whatsnew` file in `docs/source/releases.rst`. New releases should appear at the top. The syntax for this is:

```
.. include:: whatsnew/<version>.txt
```

2.7.8 Release Notes

Release 1.4.1

Release 1.4.1

Date October 5, 2020

This release includes a small number of bug fixes, documentation improvements, and build/dependency enhancements.

Conda packages for zipline and its dependencies are now available for python 3.6 on the 'conda-forge' Anaconda channel. They're also available on the 'Quantopian' channel, but we'll stop updating those eventually.

Bug Fixes

- Fix for calling `run_algorithm` without `benchmark_returns` (#2762)

Maintenance and Refactorings

- Support for empyrical 0.5.3 (#2526)
- Removed dependence on contextlib2 in py3 environments (#2757)
- Update default bundle to 'quantopian-quandl' at more entrypoints (#2763)

Build

- CI with newer statsmodels and scipy (#2739)
- GitHub Actions CI on linux and macos (#2743, #2767)
- Added conda packaging for zipline and its dependencies to conda-forge (#2665)

Documentation

- Various documentation improvements (#2763, #2771, #2772, #2776, #2780)

Release 1.4.0

Release 1.4.0

Date July 22, 2020

Highlights

Removed Implicit Dependency on Benchmarks and Treasury Returns

Previously, Zipline implicitly fetched these required inputs from third party API sources if they were not provided by users: treasury data from the US Federal Reserve's API, and benchmarks from IEX. This meant that simulations required an internet connection and stable APIs for these data sources, neither of which were guaranteed for many users.

We removed the dependency on treasury curves, since they weren't actually being used anymore. And we replaced the implicit downloading of benchmark returns with explicit options:

```
--benchmark-file                The csv file that contains the benchmark
                                returns (date, returns columns)
--benchmark-symbol              The instrument's symbol to be used as
                                a benchmark.
                                (should exist in the ingested bundle)
--benchmark-sid                 The sid of the instrument to be used as a
                                benchmark.
                                (should exist in the ingested bundle)
--no-benchmark                  This flag is used to set the benchmark to
```

(continues on next page)

(continued from previous page)

```
                                zero. Alpha, beta and benchmark metrics
                                are not calculated
```

(#2627, #2642)

New Built In Factors

- `PercentChange`: Calculates the percent change over the given `window_length`. Note: Percent change is calculated as `(new - old) / abs(old)`. (#2506)
- `PeerCount`: Gives the number of occurrences of each distinct category in a classifier. (#2509)
- `ConstantMixin`: A mixin for creating a Pipeline term with a constant value. (#2697)
- `if_else()`: Allows users to create expressions that conditionally draw from the outputs of one of two terms. (#2697)
- `fillna()`: Allows users to fill missing data with either a constant value, or values from another term. (#2697)
- `clip()`: Allows users to constrain a factor's values to a given range. (#2708)
- `mean()`, `stddev()`, `max()`, `min()`, `median()`, `sum()`, `notnull_count()`: Summarize data across the entire domain into a scalar factor. (#2697)

Enhancements

- Added International Pipelines (#2262)
- Added DataSetFamily (née MultiDimensionalDataSet) - a shorthand for creating a collection of regular DataSets that share the same columns. (#2402)
- Added `get_column()` for looking up columns by name (#2210)
- Added `CheckWindowsClassifier` that allows us to test lookback windows of categorical and string columns using Pipeline. (#2458)
- Added `PipelineHooks` which is now used to display Pipline progress bars (#2467)
- `BoundColumn` comparisons will now result in an error. This prevents writing `EquityPricing.volume > 1000` (silently returning bad data) insteads of `EquityPricing.volume.latest > 1000`. (#2537)
- Added currency conversion support to Pipeline. (#2586)
- Added `--benchmark-file` and `--benchmark-symbol` command line arguments to make it easier to provide benchmark data. (#2642)
- Added support for Python 3.6 (#2643)
- Added `mask` argument to Factor.peer_count. (#2676)
- Added `if_else()` and `fillna()` for allowing conditional logic in Pipelines. (#2691)
- Added daily summary methods to Factor for collecting summary statistics for the entire universe. (#2697)
- Added `clip()` method for clipping values to a range. (#2708)
- Added support for Pipeline term arithmetic with more than 32 terms. (#2727)

Bug Fixes

- Fixed support for non unique sid->exchange mappings. (#2289)
- Fixed crash on dividend warning. (#2323)
- Fixed `week_start` when Monday precedes the New Year. (#2394)
- Ensured correct dtypes when unpacking empty dataframes. (#2444)
- Fixed a bug where a Pipeline term with `window_length=0` would not copy the input before calling `compute()` which could cause incorrect results if the input was reused in the Pipeline. (#2723)

Performance

- Added `HDF5DailyBarWriter`, which writes daily pricing in a new format as an HDF5 file. Each OHLCV field is stored as a 2D array in a chunked HDF5 dataset, with a row per sid and a column per day. The file also supports multiple countries. Added `HDF5DailyBarReader`, which implements the BarReader interface and can read files written by HDF5DailyBarWriter. (#2295)
- Vectorized dividend ratio calculation (#2298)
- Improved performance of the `RollingPearson` and `RollingPearsonOfReturns` pipeline factors. (#2071)

Maintenance and Refactorings

- Made `parameter_space()` reset instance fixtures between runs (#2433)
- Removed unused treasury curves data handling. (#2626)

Miscellaneous

International Pipelines

Pipeline now supports international data.

Pipeline is a tool that allows you to define computations over a universe of assets and a period of time. In the past, you could only run pipelines on the US equity market. Now, you can now specify a domain over which a pipeline should be computed. The name "domain" refers to the mathematical concept of the "domain of a function", which is the set of potential inputs to a function. In the context of Pipeline, the domain specifies the set of assets and a corresponding trading calendar over which the expressions of a pipeline should be computed.

For example, the following pipeline returns the latest close price and volume for all Canadian equities, every day.

```
pipe = Pipeline(
    columns={
        'price': EquityPricing.close.latest,
        'volume': EquityPricing.volume.latest,
        'mcap': factset.Fundamentals.mkt_val.latest,
    },
    domain=CA_EQUITIES,
)
```

Another challenge related to currencies is the fact that some exchanges don't require stocks to be listed in local currency. For example, the London Stock Exchange only has about 75% of its listings denominated in GBP*. The other 25% are primarily listed in EUR or USD. This can make it hard to make cross sectional comparisons.

To solve this problem, most people rely on currency conversions to bring price-based fields into the same currency. Pipeline columns now support an `fx` method for specifying what currency the data should be viewed as. This method is only available on terms which are "currency-aware", for example open or close, but not on terms that do not care about currency like volume.

Currently, there is no way to load international data into a bundle. We are working on ways to make it easy to get international data into Zipline.

(#2265, #2262, and many others)

The domains that Zipline currently supports for running pipelines (using the latest trading-calendars package) are the following:

- Argentina
- Australia
- Austria
- Belgium
- Brazil
- Canada
- Chile
- China
- Czech Republic
- Colombia
- Czechia
- Finland
- France
- Germany
- Greece
- Hong Kong
- Hungary
- India
- Indonesia
- Ireland
- Italy
- Japan
- Malaysia
- Mexico
- Netherlands
- New Zealand

- Norway
- Pakistan
- Peru
- Philippines
- Poland
- Portugal
- Russia
- Singapore
- Spain
- Sweden
- Taiwan
- Thailand
- Turkey
- United Kingdom
- United States
- South Africa
- South Korea
- Switzerland

(#2301, #2333, #2338, #2355, #2369, #2550, #2552, #2559)

DataSetFamily

Dataset families are used to represent data where the unique identifier for a row requires more than just asset and date coordinates. A `DataSetFamily` can also be thought of as a collection of `DataSet` objects, each of which has the same columns, domain, and ndim.

`DataSetFamily` objects are defined with one or more `Column` objects, plus one additional field: `extra_dims`.

The `extra_dims` field defines coordinates other than asset and date that must be fixed to produce a logical timeseries. The column objects determine columns that will be shared by slices of the family.

`extra_dims` are represented as an ordered dictionary where the keys are the dimension name, and the values are a set of unique values along that dimension.

To work with a `DataSetFamily` in a pipeline expression, one must choose a specific value for each of the extra dimensions using the `slice()` method. For example, given a `DataSetFamily`:

```
class SomeDataSet(DataSetFamily):
    extra_dims = [
        ('dimension_0', {'a', 'b', 'c'}),
        ('dimension_1', {'d', 'e', 'f'}),
    ]

    column_0 = Column(float)
    column_1 = Column(bool)
```

This dataset might represent a table with the following columns:

```
sid :: int64
asof_date :: datetime64[ns]
timestamp :: datetime64[ns]
dimension_0 :: str
dimension_1 :: str
column_0 :: float64
column_1 :: bool
```

Here we see the implicit `sid`, `asof_date` and `timestamp` columns as well as the extra dimensions columns.

This `DataSetFamily` can be converted to a regular `DataSet` with:

```
DataSetSlice = SomeDataSet.slice(dimension_0='a', dimension_1='e')
```

This sliced dataset represents the rows from the higher dimensional dataset where `(dimension_0 == 'a') & (dimension_1 == 'e')`.

(#2402, #2452, #2456)

Release 1.3.0

Release 1.3.0

Date July 16, 2018

This release includes several enhancements and performance improvements along with a small number of bug fixes. We recommend that all users upgrade to this version.

Note: This will likely be the last minor release in the Zipline 1.x series. The release next will be Zipline 2.0, which will include a number of small breaking changes required to support international equities.

Highlights

Support for Newer Numpy/Pandas Versions

Zipline has historically been very conservative when updating versions of numpy, pandas, and other "PyData" ecosystem packages. This conservatism is primarily due to the fact that Zipline is used as the backtesting engine for Quantopian, which means that updating package versions risks breaking a large installed codebase. Of course, many Zipline users don't have the backwards compatibility requirements that Quantopian has, and they'd like to be able to use the latest and greatest package versions.

As part of this release, we're now building and testing Zipline with two package configurations:

- "Stable", using numpy version 1.11 and pandas version 0.18.1.
- "Latest", using numpy version 1.14 and pandas version 0.22.0.

Other combinations of numpy and pandas **may** work, but these package sets will be built and tested during our normal development cycle.

Moving forward, our goal is to continue to maintain support for two sets of packages at any given time. The "stable" package set will change relatively infrequently, and will contain the versions of numpy and pandas supported on

Quantopian. The "latest" package set will change regularly, and will contain recently-released versions of numpy and pandas.

Our hope with these changes is to strike a balance between stability and novelty without taking on too great a maintenance burden by supporting every possible combination of packages. (#2194)

Standalone `trading_calendars` Module

One of the most popular features of Zipline is its collection of trading calendars, which provide information about holidays and trading hours of various markets. As part of this release, Zipline's calendar-related functionality has been moved to a separate trading-calendars package, allowing users that only needed access to the calendars to use them without taking on the rest of Zipline's dependencies.

For backwards compability, Zipline will continue to re-export calendar-related functions. For example, `zipline.get_calendar()` still exists, but is now an alias for `trading_calendars.get_calendar`. Users that depend on this functionality are encouraged to update their imports to the new locations in `trading_calendars`. (#2219)

Custom Blotters

This release adds experimental support for running Zipline with user-defined subclasses of `Blotter`. The primary motivation for this change is to make it easier to run live algorithms from the Zipline CLI.

There are two primary ways to configure a custom blotter:

1. You can pass an instance of `Blotter` as the `blotter` parameter to `zipline.run_algorithm()`. (This functionality had existed previously, but wasn't well-documented.)
2. You can register a named **factory** for a blotter in your `extension.py` and pass the name on the command line via the `--blotter` flag.

An example usage of **(2)** might look like this:

Listing 1: ~/.zipline/extension.py

```
from zipline.extensions import register
from zipline.finance.blotter import Blotter, SimulationBlotter
from zipline.finance.cancel_policy import EODCancel

@register(Blotter, 'my-blotter')
def my_blotter():
    """Create a SimulationBlotter with a non-default cancel policy.
    """
    return SimulationBlotter(cancel_policy=EODCancel())
```

To use this factory when running zipline from the command line, we would invoke zipline like this:

```
$ zipline run --blotter my-blotter <...other-args...>
```

As part of this change, the `Blotter` class has been converted to an abstract base class. The default blotter used in simulations is now named `zipline.finance.blotter.SimulationBlotter`.

(#2210, #2251)

Custom Command-Line Arguments

This release adds support for passing custom arguments to the `zipline` command-line interface. Custom command-line arguments are passed via the `-x` flag followed by a `key=value` pair. Arguments passed this way can be accessed from Python code (e.g., an algorithm or an extension) via attributes of `zipline.extension_args`. For example, if zipline is invoked like this:

```
$ zipline -x argle=bargle run ...
```

then the result of `zipline.extension_args.argle` would be the string `"bargle"`.

Custom arguments can be grouped into namespaces by including `.` characters in keys. For example, if zipline is invoked like this:

```
$ zipline -x argle.bargle=foo
```

then `zipline.extension_args.argle` will contain an object with a `bargle` attribute containing the string `"foo"`. Keys can contain multiple dots to create nested namespaces. (#2210)

Enhancements

- Added support for pandas 0.22 and numpy 1.14. See above for details. (#2194)
- Moved `zipline.utils.calendars` into a separately-installable trading-calendars package. (#2219)
- Added support for specifying custom string arguments with the `-x` flag. See above for details. (#2210)

Experimental Features

Warning: Experimental features are subject to change.

- Added support for registering custom subclass of `zipline.finance.blotter.Blotter`. See above for details. (#2210, #2251)

Bug Fixes

- Fixed a bug in `zipline.pipeline.Factor.winsorize()` where NaN values were incorrectly included in value counts when determining cutoff thresholds for winsorization. (#2138)
- Fixed a crash in `zipline.pipeline.Factor.top()` with a count of 1 and no groupby. (#2218)
- Fixed a bug where calling `data.history` with a negative lookback would fetch prices from the future. (#2164)
- Fixed a bug where `StopOrder`, `zipline.finance.execution.LimitOrder`, and `zipline.finance.execution.StopLimitOrder` prices were being rounded to the nearest penny regardless of asset tick size. Prices are now rounded based on the `tick_size` attribute of the asset being ordered. (#2211)

Performance

- Improved performance when fetching minutely prices for assets that trade regularly. (#2108)
- Improved performance when fetching minutely prices for many assets by tuning cache sizes. (#2110)

Maintenance and Refactorings

- Refactored large parts of the Zipline test suite to make it easier to change the signature of `zipline.algorithm.TradingAlgorithm`. (#2169, #2168, #2165, #2171)

Build

- Added support for running travis builds with pandas 0.18 and 0.22. (#2194)
- Added OSX builds to the travis build matrix. (#2244)

Release 1.2.0

Release 1.2.0

Date April 4, 2018

Highlights

Extensible Risk and Performance Metrics (#2081)

The risk and performance metrics are summarizing values calculated by Zipline when running a simulation, for example: returns or Sharpe ratio. 1.1.2 introduces a new API for registering custom risk and performance metrics defined by the user. We have also made it possible to run a backtest without computing any metrics to improve the feedback cycle when debugging an algorithm.

For more information, see *Risk and Performance Metrics*.

Docs, Trading Calendars, and Benchmarks

Zipline now defaults to using the `quandl` bundle, which you'll need an API Key for, and can find information about in the Data Bundles documentation.

We've added many Tutorial & Documentations updates, including information on how to create your own `TradingCalendar`, pass it to your algorithm via the Zipline CLI, and how to use custom csv data using the `csvdir` bundle.

Zipline is no longer being tested and packaged for Python 3.4.

Zipline now requests data for SPY, the default benchmark used for Zipline backtests, using the IEX Trading API, and no longer uses `pandas-datareader`. You can run a backtest up to 5 years from the current day using this data.

Enhancements

- Grow minute file cache to 1550 by default (#1906)
- Change default commission to .001 (#1946)
- Enable the ability to compute multiple pipelines (#1974)
- Allow users to switch between calendars (#1800)
- New filter `NoMissingValues` (#1969)
- Fail better on `AssetFinder(nonexistent_path)` (#2000)
- Implement csvdir bundle (#1860)
- Update quandl_bundle to use Quandl API v3 (#1990)
- Add `FixedBasisPointsSlippage` slippage model (#2047)
- Create MinLeverage control (#2064)

Experimental Features

Warning: Experimental features are subject to change.

None

Bug Fixes

- `history` calls with a frequency of `1d` now work when using a Panel as the minute data source. (#1920)
- Check contract exists when using futures daily bar reader (#1892)
- `NoDataBeforeDate` edge cases (#1894)
- Fix frame column validation in Python 2.7.5 (#1954)
- Fix daily history for minute panel data backtest (#1920)
- `get_last_traded_dt` expects a trading day (#2087)
- Daily Adjustment perspective fix (#2089)

Performance

- Change algorithm account validation from happening every minute in `handle_data` to only occurring once at the end of each day (#1884)
- Blaze core loader performance improvements (#1866)
- Add a new factor that just computes beta (#2021)
- Reduces memory footprint of Quandl WIKI Prices bundle (#2053)

Maintenance and Refactorings

- Add `CachedObject.expired()` (#1881)
- Set `RollingLinearRegressionOfReturns` factor to be window_safe (#1902)
- Set `RSI` factor to be window_safe (#1904)
- Updates for better docs generation (#1890)
- Remove and zero out unused treasury curves (#1910)
- Networkx 2 changes the behavior of out_degree (#1996)
- Pass calendars to `DataPortal` (#2026)
- Remove old Yahoo code (#2032)
- Sync and fill benchmarks through latest trading day (#2044)
- Provides better error message when QUANDL_API_KEY is missing (#2078)
- Improve the error message for misaligned dates in Pipeline engine (#2131)

Build

- Update the conda tools we're using to fix our packaging (#1942)
- Upgrade empyrical to 0.3.2 (#1983)
- Update conda tooling and remove Python 3.4 builds (#2009)
- Upgrade empyrical to 0.3.3 (#2014)
- Upgrade empyrical to 0.3.4 (#2098)
- Upgrade empyrical to 0.4.2 (#2125)

Documentation

- Include `MACDSignal` in zipline.io documentation (#1828)
- Remove mentions of Yahoo from the Beginner Tutorial (#1845)
- Add contributing & questions section to the README (#1889)
- Add info about using a conda envs for installs (#1922)
- Fix Beginner Tutorial link (#1932)
- Add clean docs (#1943)
- Add distinct warnings for benchmark and treasury fetchers (#1971)
- Add CONTRIBUTING.rst (#2033)
- Add tutorial on creating a custom `TradingCalendar` (#2035)
- Docs & tutorial updates for ingesting, beginners, and csvdir (#2073)
- Documented the new risk and performance metrics API (#2081).
- Fixed a typo in the description of `--bundle-timestamp` (#2123)

Miscellaneous

None

Release 1.1.1

Release 1.1.1

Date July 5, 2017

Highlights

Zipline now has broad support for futures, in addition to equities. It's also being tested and packaged for Python 3.5.

We also saw breaking changes occur from Yahoo changing their API endpoint, thus preventing users from downloading benchmark data needed for backtests. Since that change, we have swapped out Yahoo-related benchmarking code with references to Google Finance and have removed all deprecated Yahoo code, including the usage of custom Yahoo bundles.

Enhancements

- Adds a property for BarData to know about current session's minutes (#1713)
- Adds a better error message for non-existent root symbols (#1715:)
- Adds `StaticSids` Pipeline Filter (#1717)
- Allows `zipline.data.data_portal.DataPortal.get_spot_value` to accept multiple assets (#1719)
- Adds `ContinuousFuture` to `lookup_generic` (#1718)
- Adds CFE Adhoc Holidays to `exchange_calendar_cfe` (#1698)
- Allows overriding of order amount rounding (#1722)
- Makes continuous future adjustment style an argument (#1726)
- Adds preliminary support for Futures slippage and commission models (#1738)
- Fix a bug in cost basis calculation and change all mentions of `sid` to `asset` (#1757)
- Add slippage and commission models for futures (#1748)
- Use Python 3.5 in our Dockerfile (#1806)
- Allow pipelines to be run in chunks (#1811)
- Adds *get_range* to *BenchmarkSource* (#1815)
- Adds support for relabeling classifiers in Pipeline (#1833)

Experimental Features

Warning: Experimental features are subject to change.

None

Bug Fixes

- Fixes a floating point division issue in `zipline.data.minute_bars` by using integer divison instead (#1683)
- Sorts data in `zipline.pipeline.loaders.blaze.core` on `asof_date` to resolve timestamp conflicts (#1710)
- Swapped out Yahoo for Google Finance benchmark data (#1812)
- Gold and silver futures contracts only traded during certain months (#1779)
- Fixes bug in TradingCalendar initialization when we use tzaware datetimes (#1802)
- Fixes precision issues on futures prices when rounding (#1788)

Performance

- Avoid repeated recursive calls when getting forward-filled close price (#1735)

Maintenance and Refactorings

- Adds linter recommendations to adjustments module (#1712)
- Clears up naming and logic in resample close (#1728)
- Use March quarterly cycle for several continuous futures (#1762)
- Use better repr for Transaction objects (#1746)
- Shorten repr for Asset objects (#1786)
- Removes usage of empyrical's information ratio (#1854)

Build

- Adds Python 3.5 packages (#1701)
- Swap conda-build arguments so we don't build packages on every CI build (#1813)

Documentation

- Adds Zipline Development Guidelines, for people to read about how to contribute to zipline (#1820)
- Show exchange as required for equities (#1731)
- Updates the Zipline Beginner Tutorial notebook (#1707)
- Includes PipelineEngine, pipeline Term, Factors, and other pipeline things to docs (#1826)

Miscellaneous

- Use csv market data with `run_algorithm` so we don't try to download data for tests (#1793)
- Updates Dockerfile to use Python 3.5

Release 1.1.0

Release 1.1.0

Date March 10, 2017

This release is meant to provide zipline support for pandas 0.18, as well as several bug fixes, API changes, and many performance changes.

Enhancements

- Makes the minute bar read catch NoDataOnDate exceptions if dates are not in the calendar. Before, the minute bar reader was forward filling, but now it returns nan for OHLC and 0 for V. (#1488)
- Adds `truncate` method to `BcolzMinuteBarWriter` (#1499)
- Bumps up to pandas 0.18.1 and numpy 1.11.1 (#1339)
- Adds an earnings estimates quarter loader for Pipeline (#1396)
- Creates a restricted list manager that takes in information about restricted sids and stores in memory upon instantiation (#1487)
- Adds `last_available{session, minute}` args to `DataPortal` (#1528)
- Adds `SpecificAssets` filter (#1530)
- Adds the ability for an algorithm to request the current contract for a future chain (#1529)
- Adds `chain` field to current and supporting methods in `DataPortal` and `OrderedContracts` (#1538)
- Adds history for continuous futures (#1539)
- Adds adjusted history for continuous future (#1548)
- Adds roll style which takes the volume of a future contract into account, specifically for continuous futures (#1556)
- Adds better error message when calling Zipline API functions outside of a running simulation (#1593)
- Adds `MACDSignal()`, `MovingAverageConvergenceDivergenceSignal()`, and `AnnualizedVolatility()` as built-in factors. (#1588)
- Allows running pipelines with custom date chunks in `attach_pipeline` (#1617)

- Adds `order_batch` to the trade blotter (#1596)
- Adds vectorized lookup_symbol (#1627)
- Solidifies equality comparisons for SlippageModel classes (#1657)
- Adds a factor for winsorized results (#1696)

Bug Fixes

- Changes str to string_types to avoid errors when type checking unicode and not str type. (#1315)
- Algorithms default to quantopian-quandl bundle when no data source is specified (#1479) (#1374)
- Catches all missing data exceptions when computing dividend ratios (#1507)
- Creates adjustments based on ordered assets instead of a set. Before, adjustments were created for estimates based on where assets happened to fall in a set rather than using ordered assets (#1547)
- Fixes blaze pipeline queries for when users query for the `asof_date` column (#1608)
- Datetimes should be converted in utc. DataFrames being returned were creating US/Eastern timestamps out of the ints, potentially changing the date returned to be the date before (#1635)
- Fixes default inputs for `IchimokuKinkoHyo` factor (#1638)

Performance

- Removes invocations of `get_calendar('NYSE')` which cuts down zipline import time and makes the CLI more responsive and use less memory. (#1471)
- Refcounts and releases pipeline terms when they are no longer needed (#1484)
- Saves up to 75% of calls to minute_to_session_label (#1492)
- Speeds up counting of number of minutes across contiguous session (#1497)
- Removes/defers calls to get_loc on large indices (#1504) (#1503)
- Replaces `get_loc` calls in `calc_dividend_ratios` with `get_indexer` (#1510)
- Speeds up minute to session sampling (#1549)
- Adds some micro optimizations in `data.current` (#1561)
- Adds optimization for initial workspace for pipelines (#1521)
- More memory savings (#1599)

Maintenance and Refactorings

- Updates leveraged ETF list (#747) (#1434)
- Adds additional fields to `__getitem__` for Order class (#1483)
- Adds `BarReader` base class for minute and session readers (#1486)
- Removes `future_chain` API method, to be replaced by `data.current_chain` (#1502)
- Puts zipline back on blaze master (#1505)
- Adds Tini and sets version range for numpy, pandas, and scipy in Dockerfile (#1514)

- Deprecates `set_do_not_order_list` (#1487)
- Uses `Timedelta` instead of `DateOffset` (#1487)
- Update and pin more dev requirements (#1642)

Build

- Adds binary dependency on numpy for empyrical
- Removes old numpy/pandas versions from Travis (#1339)
- Updates appveyor.yml for new numpy and pandas (#1339)
- Downgrades to scipy 0.17 (#1339)
- Bumps empyrical to 0.2.2

Documentation

- Updated example notebook for latest zipline cell magic
- Adds ANACONDA_TOKEN directions (#1589)

Miscellaneous

- Changed the short-opt for `--before` in the `zipline clean` entrypoint. The new argument is `-e`. The old argument, `-b`, conflicted with the `--bundle` short-opt (#1625).

Release 1.0.2

Release 1.0.2

Date September 8, 2016

Enhancements

- Adds forward fill checkpoint tables for the blaze core loader. This allow the loader to more efficiently forward fill the data by capping the lower date it must search for when querying data. The checkpoints should have novel deltas applied (#1276).
- Updated VagrantFile to include all dev requirements and use a newer image (#1310).
- Allow correlations and regressions to be computed between two 2D factors by doing computations asset-wise (#1307).
- Filters have been made window_safe by default. Now they can be passed in as arguments to other Filters, Factors and Classifiers (#1338).
- Added an optional `groupby` parameter to `rank()`, `top()`, and `bottom()`. (#1349).
- Added new pipeline filters, `All` and `Any`, which takes another filter and returns True if an asset produced a True for any/all days in the previous `window_length` days (#1358).
- Added new pipeline filter `AtLeastN`, which takes another filter and an int N and returns True if an asset produced a True on N or more days in the previous `window_length` days (#1367).

- Use external library empyrical for risk calculations. Empyrical unifies risk metric calculations between pyfolio and zipline. Empyrical adds custom annualization options for returns of custom frequencies. (#855)
- Add Aroon factor. (#1258)
- Add fast stochastic oscillator factor. (#1255)
- Add a Dockerfile. (#1254)
- New trading calendar which supports sessions which span across midnights, e.g. 24 hour 6:01PM-6:00PM sessions for futures trading. *zipline.utils.tradingcalendar* is now deprecated. (#1138) (#1312)
- Allow slicing a single column out of a Factor/Filter/Classifier. (#1267)
- Provide Ichimoku Cloud factor (#1263)
- Allow default parameters on Pipeline terms. (#1263)
- Provide rate of change percentage factor. (#1324)
- Provide linear weighted moving average factor. (#1325)
- Add `NotNullFilter`. (#1345)
- Allow capital changes to be defined by a target value. (#1337)
- Add `TrueRange` factor. (#1348)
- Add point in time lookups to `assets.db`. (#1361)
- Make `can_trade` aware of the asset's exchange . (#1346)
- Add `downsample` method to all computable terms. (#1394)
- Add *QuantopianUSFuturesCalendar*. (#1414)
- Enable publishing of old `assets.db` versions. (#1430)
- Enable `schedule_function` for Futures trading calendar. (#1442)
- Disallow regressions of length 1. (#1466)

Experimental

- Add support for comingled Future and Equity history windows, and enable other Future data access via data portal. (#1435) (#1432)

Bug Fixes

- Changes `AverageDollarVolume` built-in factor to treat missing close or volume values as 0. Previously, NaNs were simply discarded before averaging, giving the remaining values too much weight (#1309).
- Remove risk-free rate from sharpe ratio calculation. The ratio is now the average of risk adjusted returns over violatility of adjusted returns. (#853)
- Sortino ratio will return calculation instead of np.nan when required returns are equal to zero. The ratio now returns the average of risk adjusted returns over downside risk. Fixed mislabeled API by converting *mar* to *downside_risk*. (#747)
- Downside risk now returns the square root of the mean of downside difference squares. (#747)
- Information ratio updated to return mean of risk adjusted returns over standard deviation of risk adjusted returns. (#1322)

- Alpha and sharpe ratio are now annualized. (#1322)
- Fix units during reading and writing of daily bar `first_trading_day` attribute. (#1245)
- Optional dispatch modules, when missing, no longer cause a *NameError*. (#1246)
- Treat `schedule_function` argument as a time rule when a time rule, but no date rule is supplied. (#1221)
- Protect against boundary conditions at beginning and end trading day in schedule function. (#1226)
- Apply adjustments to previous day when using history with a frequency of *1d*. (#1256)
- Fail fast on invalid pipeline columns, instead of attempting to access the nonexistent column. (#1280)
- Fix `AverageDollarVolume` NaN handling. (#1309)

Performance

- Performance improvements to blaze core loader. (#1227)
- Allow concurrent blaze queries. (#1323)
- Prevent missing leading bcolz minute data from doing repeated unnecessary lookups. (#1451)
- Cache future chain lookups. (#1455)

Maintenance and Refactorings

- Removed remaining mentions of `add_history`. (#1287)

Documentation

Testing

- Add test fixture which sources daily pricing data from minute pricing data fixtures. (#1243)

Data Format Changes

- `BcolzDailyBarReader` and `BcolzDailyBarWriter` use trading calendar instance, instead of trading days serialized to `JSON`. (#1330)
- Change format of `assets.db` to support point in time lookups. (#1361)
- Change `BcolzMinuteBarReader``and ``BcolzMinuteBarWriter` to support varying tick sizes. (#1428)

Release 1.0.1

Release 1.0.1

Date May 27, 2016

This is a minor bug-fix release from 1.0.0 and includes a small number of bug fixes and documentation improvements.

Enhancements

- Added support for user-defined commission models. See the `zipline.finance.commission.CommissionModel` class for more details on implementing a commision model. (#1213)
- Added support for non-float columns to Blaze-backed Pipeline datasets (#1201).
- Added `zipline.pipeline.slice.Slice`, a new pipeline term designed to extract a single column from another term. Slices can be created by indexing into a term, keyed by asset. (#1267)

Bug Fixes

- Fixed a bug where Pipeline loaders were not properly initialized by `zipline.run_algorithm()`. This also affected invocations of `zipline run` from the CLI.
- Fixed a bug that caused the `%%zipline` IPython cell magic to fail (533233fae43c7ff74abfb0044f046978817cb4e4).
- Fixed a bug in the `PerTrade` commission model where commissions were incorrectly applied to each partial-fill of an order rather than on the order itself, resulting in algorithms being charged too much in commissions when placing large orders.

  `PerTrade` now correctly applies commissions on a per-order basis (#1213).
- Attribute accesses on `CustomFactors` defining multiple outputs will now correctly return an output slice when the output is also the name of a `Factor` method (#1214).
- Replaced deprecated usage of `pandas.io.data` with `pandas_datareader` (#1218).
- Fixed an issue where `.pyi` stub files for `zipline.api` were accidentally excluded from the PyPI source distribution. Conda users should be unaffected (#1230).

Documentation

- Added a new example, `zipline.examples.momentum_pipeline`, which exercises the Pipeline API (#1230).

Release 1.0.0

Release 1.0.0

Date May 19, 2016

Highlights

Zipline 1.0 Rewrite (#1105)

We have rewritten a lot of Zipline and its basic concepts in order to improve runtime performance. At the same time, we've introduced several new APIs.

At a high level, earlier versions of Zipline simulations pulled from a multiplexed stream of data sources, which were merged via heapq. This stream was fed to the main simulation loop, driving the clock forward. This strong dependency on reading all the data made it difficult to optimize simulation performance because there was no connection between the amount of data we fetched and the amount of data actually used by the algorithm.

Now, we only fetch data when the algorithm needs it. A new class, `DataPortal`, dispatches data requests to various data sources and returns the requested values. This makes the runtime of a simulation scale much more closely with the complexity of the algorithm, rather than with the number of assets provided by the data sources.

Instead of the data stream driving the clock, now simulations iterate through a pre-calculated set of day or minute timestamps. The timestamps are emitted by `MinuteSimulationClock` and `DailySimulationClock`, and consumed by the main loop in `transform()`.

We've retired the `data[sid(N)]` and `history` APIs, replacing them with several methods on the `BarData` object: `current()`, `history()`, `can_trade()`, and `is_stale()`. Old APIs will continue to work for now, but will issue deprecation warnings.

You can now pass in an adjustments source to the `DataPortal`, and we will apply adjustments to the pricing data when looking backwards at data. Prices and volumes for execution and presented to the algorithm in data.current are the as-traded value of the asset.

New Entry Points (#1173 and #1178)

In order to make it easier to use zipline we have updated the entry points for a backtest. The three supported ways to run a backtest are now:

1. `zipline.run_algo()`
2. `$ zipline run`
3. `%zipline` (IPython magic)

Data Bundles (#1173 and #1178)

1.0.0 introduces data bundles. Data bundles are groups of data that should be preloaded and used to run backtests later. This allows users to not need to specify which tickers they are interested in each time they run an algorithm. This also allows us to cache the data between runs.

By default, the `quantopian-quandl` bundle will be used which pulls data from Quantopian's mirror of the quandl WIKI dataset. New bundles may be registered with `zipline.data.bundles.register()` like:

```
@zipline.data.bundles.register('my-new-bundle')
def my_new_bundle_ingest(environ,
                         asset_db_writer,
                         minute_bar_writer,
                         daily_bar_writer,
                         adjustment_writer,
                         calendar,
                         cache,
```

(continues on next page)

(continued from previous page)

```
                        show_progress):
    ...
```

This function should retrieve the data it needs and then use the writers that have been passed to write that data to disc in a location that zipline can find later.

This data can be used in backtests by passing the name as the `-b / --bundle` argument to `$ zipline run` or as the `bundle` argument to `zipline.run_algorithm()`.

For more information see *Data Bundles* for more information.

String Support in Pipeline (#1174)

Added support for string data in Pipeline. `zipline.pipeline.data.Column` now accepts `object` as a dtype, which signifies that loaders for that column should emit windowed iterators over the experimental new `LabelArray` class.

Several new `Classifier` methods have also been added for constructing `Filter` instances based on string operations. The new methods are:

- `element_of()`
- `startswith()`
- `endswith()`
- `has_substring()`
- `matches()`

`element_of` is defined for all classifiers. The remaining methods are only defined for string-dtype classifiers.

Enhancements

- Made the data loading classes have more consistent interfaces. This includes the equity bar writers, adjustment writer, and asset db writer. The new interface is that the resource to be written to is passed at construction time and the data to write is provided later to the *write* method as dataframes or some iterator of dataframes. This model allows us to pass these writer objects around as a resource for other classes and functions to consume (#1109 and #1149).
- Added masking to `zipline.pipeline.CustomFactor`. Custom factors can now be passed a Filter upon instantiation. This tells the factor to only compute over stocks for which the filter returns True, rather than always computing over the entire universe of stocks. (#1095)
- Added `zipline.utils.cache.ExpiringCache`. A cache which wraps entries in a `zipline.utils.cache.CachedObject`, which manages expiration of entries based on the *dt* supplied to the *get* method. (#1130)
- Implemented `zipline.pipeline.factors.RecarrayField`, a new pipeline term designed to be the output type of a CustomFactor with multiple outputs. (#1119)
- Added optional *outputs* parameter to `zipline.pipeline.CustomFactor`. Custom factors are now capable of computing and returning multiple outputs, each of which are themselves a Factor. (#1119)
- Added support for string-dtype pipeline columns. Loaders for thse columns should produce instances of `zipline.lib.labelarray.LabelArray` when traversed. `latest()` on string columns produces a string-dtype `zipline.pipeline.Classifier`. (#1174)

- Added several methods for converting Classifiers into Filters.

  The new methods are: - `element_of()` - `startswith()` - `endswith()` - `has_substring()` - `matches()`

  `element_of` is defined for all classifiers. The remaining methods are only defined for strings. (#1174)

- Added `BollingerBands` factor. This factor implements the Bollinger Bands technical indicator: https://en.wikipedia.org/wiki/Bollinger_Bands (#1199).
- Fetcher has been moved from Quantopian internal code into Zipline (#1105).
- Added new built-in factors, `RollingPearsonOfReturns`, `RollingSpearmanOfReturns` and `RollingLinearRegressionOfReturns` (#1154)

Experimental Features

Warning: Experimental features are subject to change.

- Added a new `zipline.lib.labelarray.LabelArray` class for efficiently representing and computing on string data with numpy. This class is conceptually similar to `pandas.Categorical`, in that it represents string arrays as arrays of indices into a (smaller) array of unique string values. (#1174)

Bug Fixes

None

Performance

None

Maintenance and Refactorings

None

Build

None

Documentation

- Updated documentation for the API methods (#1188).
- Updated release process to mention that docs should be built with python 3 (#1188).

Miscellaneous

- Zipline now provides a stub file for the `zipline.api` module. This module is normally dynamically created so the stub file provides some static information for utilities that can consume it, for example PyCharm (#1208).

Release 0.9.0

Release 0.9.0

Date March 29, 2016

Highlights

- Added classifiers and normalization methods to pipeline, along with new datasets and factors.
- Added support for Windows with continuous integration on AppVeyor.

Enhancements

- Added new datasets `CashBuybackAuthorizations` and `ShareBuybackAuthorizations` for use in the Pipeline API. These datasets provide an abstract interface for adding cash and share buyback authorizations data, respectively, to a new algorithm. pandas-based reference implementations for these datasets can be found in `zipline.pipeline.loaders.buyback_auth`, and experimental blaze-based implementations can be found in `zipline.pipeline.loaders.blaze.buyback_auth`. (#1022).
- Added new datasets `DividendsByExDate`, `DividendsByPayDate`, and `DividendsByAnnouncementDate` for use in the Pipeline API. These datasets provide an abstract interface for adding dividends data organized by ex date, pay date, and announcement date, respectively, to a new algorithm. pandas-based reference implementations for these datasets can be found in `zipline.pipeline.loaders.dividends`, and experimental blaze-based implementations can be found in `zipline.pipeline.loaders.blaze.dividends`. (#1093).
- Added new built-in factors, `zipline.pipeline.factors.BusinessDaysSinceCashBuybackAuth` and `zipline.pipeline.factors.BusinessDaysSinceShareBuybackAuth`. These factors use the new `CashBuybackAuthorizations` and `ShareBuybackAuthorizations` datasets, respectively. (#1022).
- Added new built-in factors, `zipline.pipeline.factors.BusinessDaysSinceDividendAnnouncement`, `zipline.pipeline.factors.BusinessDaysUntilNextExDate`, and `zipline.pipeline.factors.BusinessDaysSincePreviousExDate`. These factors use the new `DividendsByAnnouncementDate` and ``DividendsByExDate` datasets, respectively. (#1093).
- Implemented `zipline.pipeline.Classifier`, a new core pipeline API term representing grouping keys. Classifiers are primarily used by passing them as the `groupby` parameter to factor normalization methods. (#1046)
- Added factor normalization methods: `zipline.pipeline.Factor.demean()` and `zipline.pipeline.Factor.zscore()`. (#1046)
- Added `zipline.pipeline.Factor.quantiles()`, a method for computing a Classifier from a Factor by partitioning into equally-sized buckets. Also added helpers for common quantile sizes (`zipline.pipeline.Factor.quartiles()`, `zipline.pipeline.Factor.quartiles()`, and `zipline.pipeline.Factor.deciles()`) (#1075).

Experimental Features

Warning: Experimental features are subject to change.

None

Bug Fixes

- Fixed a bug where merging two numerical expressions failed given too many inputs. This caused running a pipeline to fail when combining more than ten factors or filters. (#1072)

Performance

None

Maintenance and Refactorings

None

Build

- Added AppVeyor for continuous integration on Windows. Added conda build of zipline and its dependencies to AppVeyor and Travis builds, which upload their results to anaconda.org labeled with "ci". (#981)

Documentation

None

Miscellaneous

- Adds `ZiplineTestCase` which provides hooks to consume test fixtures. Fixtures are things like: `WithAssetFinder` which will make `self.asset_finder` available to your test with some mock data (#1042).

Release 0.8.4

Release 0.8.4

Date February 24, 2016

Highlights

- Added a new `EarningsCalendar` dataset for use in the Pipeline API. (#905).
- `AssetFinder` speedups (#830 and #817).
- Improved support for non-float dtypes in Pipeline. Most notably, we now support `datetime64` and `int64` dtypes for `Factor`, and `BoundColumn.latest` now returns a proper `Filter` object when the column is of dtype `bool`.
- Zipline now supports `numpy` 1.10, `pandas` 0.17, and `scipy` 0.16 (#969).
- Batch transforms have been deprecated and will be removed in a future release. Using `history` is recommended as an alternative.

Enhancements

- Adds a way for users to provide a context manager to use when executing the scheduled functions (including `handle_data`). This context manager will be passed the `BarData` object for the bar and will be used for the duration of all of the functions scheduled to run. This can be passed to `TradingAlgorithm` by the keyword argument `create_event_context` (#828).
- Added support for `zipline.pipeline.factors.Factor` instances with `datetime64[ns]` dtypes. (#905)
- Added a new `EarningsCalendar` dataset for use in the Pipeline API. This dataset provides an abstract interface for adding earnings announcement data to a new algorithm. A pandas-based reference implementation for this dataset can be found in `zipline.pipeline.loaders.earnings`, and an experimental blaze-based implementation can be found in `zipline.pipeline.loaders.blaze.earnings`. (#905).
- Added new built-in factors, `zipline.pipeline.factors.BusinessDaysUntilNextEarnings` and `zipline.pipeline.factors.BusinessDaysSincePreviousEarnings`. These factors use the new `EarningsCalendar` dataset. (#905).
- Added `isnan()`, `notnan()` and `isfinite()` methods to `zipline.pipeline.factors.Factor` (#861).
- Added `zipline.pipeline.factors.Returns`, a built-in factor which calculates the percent change in close price over the given window_length. (#884).
- Added a new built-in factor: `AverageDollarVolume`. (#927).
- Added `ExponentialWeightedMovingAverage` and `ExponentialWeightedMovingStdDev` factors. (#910).
- Allow `DataSet` classes to be subclassed where subclasses inherit all of the columns from the parent. These columns will be new sentinels so you can register them a custom loader (#924).
- Added `coerce()` to coerce inputs from one type into another before passing them to the function (#948).
- Added `optionally()` to wrap other preprocessor functions to explicitly allow `None` (#947).
- Added `ensure_timezone()` to allow string arguments to get converted into `datetime.tzinfo` objects. This also allows `tzinfo` objects to be passed directly (#947).
- Added two optional arguments, `data_query_time` and `data_query_tz` to `BlazeLoader` and `BlazeEarningsCalendarLoader`. These arguments allow the user to specify some cutoff time for data when loading from the resource. For example, if I want to simulate executing my `before_trading_start` function at `8:45 US/Eastern` then I could pass `datetime.time(8, 45)` and `'US/Eastern'` to the loader. This means that data that is timestamped on or after `8:45` will not seen on that day in the simulation. The data will be made available on the next day (#947).

- `BoundColumn.latest` now returns a `Filter` for columns of dtype `bool` (#962).
- Added support for `Factor` instances with `int64` dtype. `Column` now requires a `missing_value` when dtype is integral. (#962)
- It is also now possible to specify custom `missing_value` values for `float`, `datetime`, and `bool` Pipeline terms. (#962)
- Added auto-close support for equities. Any positions held in an equity that reaches its `auto_close_date` will be liquidated for cash according to the equity's last sale price. Furthermore, any open orders for that equity will be canceled. Both futures and equities are now auto-closed on the morning of their `auto_close_date`, immediately prior to `before_trading_start`. (#982)

Experimental Features

Warning: Experimental features are subject to change.

- Added support for parameterized `Factor` subclasses. Factors may specify `params` as a class-level attribute containing a tuple of parameter names. These values are then accepted by the constructor and forwarded by name to the factor's `compute` function. This API is experimental, and may change in future releases.

Bug Fixes

- Fixes an issue that would cause the daily/minutely method caching to change the `len` of a `SIDData` object. This would cause us to think that the object was not empty even when it was (#826).
- Fixes an error raised in calculating beta when benchmark data were sparse. Instead `numpy.nan` is returned (#859).
- Fixed an issue pickling `sentinel()` objects (#872).
- Fixed spurious warnings on first download of treasury data (:issue *922*).
- Corrected the error messages for `set_commission()` and `set_slippage()` when used outside of the `initialize` function. These errors called the functions `override_*` instead of `set_*`. This also renamed the exception types raised from `OverrideSlippagePostInit` and `OverrideCommissionPostInit` to `SetSlippagePostInit` and `SetCommissionPostInit` (#923).
- Fixed an issue in the CLI that would cause assets to be added twice. This would map the same symbol to two different sids (#942).
- Fixed an issue where the `PerformancePeriod` incorrectly reported the total_positions_value when creating a `Account` (#950).
- Fixed issues around KeyErrors coming from history and BarData on 32-bit python, where Assets did not compare properly with int64s (#959).
- Fixed a bug where boolean operators were not properly implemented on `Filter` (#991).
- Installation of zipline no longer downgrades numpy to 1.9.2 silently and unconditionally (#969).

Performance

- Speeds up `lookup_symbol()` by adding an extension, `AssetFinderCachedEquities`, that loads equities into dictionaries and then directs `lookup_symbol()` to these dictionaries to find matching equities (#830).
- Improved performance of `lookup_symbol()` by performing batched queries. (#817).

Maintenance and Refactorings

- Asset databases now contain version information to ensure compatibility with current Zipline version (#815).
- Upgrade `requests` version to 2.9.1 (2ee40db)
- Upgrade `logbook` version to 0.12.5 (11465d9).
- Upgrade `Cython` version to 0.23.4 (5f49fa2).

Build

- Makes zipline install requirements more flexible (#825).
- Use `versioneer` to manage the project `__version__` and setup.py version (#829).
- Fixed coveralls integration on travis build (#840).
- Fixed conda build, which now uses git source as its source and reads requirements using setup.py, instead of copying them and letting them get out of sync (#937).
- Require `setuptools` > 18.0 (#951).

Documentation

- Document the release process for developers (#835).
- Added reference docs for the Pipeline API. (#864).
- Added reference docs for Asset Metadata APIs. (#864).
- Generated documentation now includes links to source code for many classes and functions. (#864).
- Added platform-specific documentation describing how to find binary dependencies. (#883).

Miscellaneous

- Added a `show_graph()` method to render a Pipeline as an image (#836).
- Adds `subtest()` decorator for creating subtests without `nose_parameterized.expand()` which bloats the test output (#833).
- Limits timer report in test output to 15 longest tests (#838).
- Treasury and benchmark downloads will now wait up to an hour to download again if data returned from a remote source does not extend to the date expected. (#841).
- Added a tool to downgrade the assets db to previous versions (#941).

Release 0.8.3

Release 0.8.3

Date November 6, 2015

Note: We advanced the version to `0.8.3` to fix a source distribution issue with pypi. There are no code changes in this version.

Release 0.8.0

Release 0.8.0

Date November 6, 2015

Highlights

- New documentation system with a new website at zipline.io
- Major performance enhancements.
- Dynamic history.
- New user defined method: `before_trading_start`.
- New api function: `schedule_function()`.
- New api function: `get_environment()`.
- New api function: `set_max_leverage()`.
- New api function: `set_do_not_order_list()`.
- Pipeline API.
- Support for trading futures.

Enhancements

- Account object: Adds an account object to context to track information about the trading account. Example:

```
context.account.settled_cash
```

  Returns the settled cash value that is stored on the account object. This value is updated accordingly as the algorithm is run (#396).

- `HistoryContainer` can now grow dynamically. Calls to `history()` will now be able to increase the size or change the shape of the history container to be able to service the call. `add_history()` now acts as a preformance hint to pre-allocate sufficient space in the container. This change is backwards compatible with `history`, all existing algorithms should continue to work as intended (#412).
- Simple transforms ported from quantopian and use history. `SIDData` now has methods for:
  - `stddev`
  - `mavg`
  - `vwap`

- returns

These methods, except for returns, accept a number of days. If you are running with minute data, then this will calculate the number of minutes in those days, accounting for early closes and the current time and apply the transform over the set of minutes. returns takes no parameters and will return the daily returns of the given asset. Example:

```
data[security].stddev(3)
```

(#429).

- New fields in Performance Period. Performance Period has new fields accessible in return value of to_dict: - gross leverage - net leverage - short exposure - long exposure - shorts count - longs count (#464).
- Allow order_percent() to work with various market values (by Jeremiah Lowin).

  Currently, order_percent() and order_target_percent() both operate as a percentage of self.portfolio.portfolio_value. This PR lets them operate as percentages of other important MVs. Also adds context.get_market_value(), which enables this functionality. For example:

```
# this is how it works today (and this still works)
# put 50% of my portfolio in AAPL
order_percent('AAPL', 0.5)
# note that if this were a fully invested portfolio, it would become 150% levered.

# take half of my available cash and buy AAPL
order_percent('AAPL', 0.5, percent_of='cash')

# rebalance my short position, as a percentage of my current short
book_target_percent('MSFT', 0.1, percent_of='shorts')

# rebalance within a custom group of stocks
tech_stocks = ('AAPL', 'MSFT', 'GOOGL')
tech_filter = lambda p: p.sid in tech_stocks
for stock in tech_stocks:
    order_target_percent(stock, 1/3, percent_of_fn=tech_filter)
```

  (#477).
- Command line option to for printing algo to stdout (by Andrea D'Amore) (#545).
- New user defined function before_trading_start. This function can be overridden by the user to be called once before the market opens every day (#389).
- New api function schedule_function(). This function allows the user to schedule a function to be called based on more complicated rules about the date and time. For example, call the function 15 minutes before market close respecting early closes (#411).
- New api function set_do_not_order_list(). This function accepts a list of assets and adds a trading guard that prevents the algorithm from trading them. Adds a list point in time list of leveraged ETFs that people may want to mark as 'do not trade' (#478).
- Adds a class for representing securities. order() and other order functions now require an instance of Security instead of an int or string (#520).
- Generalize the Security class to Asset. This is in preperation of adding support for other asset types (#535).
- New api function get_environment(). This function by default returns the string 'zipline'. This is used so that algorithms can have different behavior on Quantopian and local zipline (#384).

- Extends `get_environment()` to expose more of the environment to the algorithm. The function now accepts an argument that is the field to return. By default, this is `'platform'` which returns the old value of `'zipline'` but the following new fields can be requested:
  - `''arena'`: Is this live trading or backtesting?
  - `'data_frequency'`: Is this minute mode or daily mode?
  - `'start'`: Simulation start date.
  - `'end'`: Simulation end date.
  - `'capital_base'`: The starting capital for the simulation.
  - `'platform'`: The platform that the algorithm is running on.
  - `'*'`: A dictionary containing all of these fields.

  (#449).
- New api function `set_max_leveraged()`. This method adds a trading guard that prevents your algorithm from over leveraging itself (#552).

Experimental Features

Warning: Experimental features are subject to change.

- Adds new Pipeline API. The pipeline API is a high-level declarative API for representing trailing window computations on large datasets (#630).
- Adds support for futures trading (#637).
- Adds Pipeline loader for blaze expressions. This allows users to pull data from any format blaze understands and use it in the Pipeline API. (#775).

Bug Fixes

- Fix a bug where the reported returns could sharply dip for random periods of time (#378).
- Fix a bug that prevented debuggers from resolving the algorithm file (#431).
- Properly forward arguments to user defined `initialize` function (#687).
- Fix a bug that would cause treasury data to be redownloaded every backtest between midnight EST and the time when the treasury data was available (#793).
- Fix a bug that would cause the user defined `analyze` function to not be called if it was passed as a keyword argument to `TradingAlgorithm` (#819).

Performance

- Major performance enhancements to history (by Dale Jung) (#488).

Maintenance and Refactorings

- Remove simple transform code. These are available as methods of `SIDData` (#550).

Build

None

Documentation

- Switched to sphinx for the documentation (#816).

Release 0.7.0

Release 0.7.0

Date July 25, 2014

Highlights

- Command line interface to run algorithms directly.
- IPython Magic `%%zipline` that runs algorithm defined in an IPython notebook cell.
- API methods for building safeguards against runaway ordering and undesired short positions.
- New history() function to get a moving DataFrame of past market data (replaces BatchTransform).
- A new beginner tutorial.

Enhancements

- CLI: Adds a CLI and IPython magic for zipline. Example:

```
python run_algo.py -f dual_moving_avg.py --symbols AAPL --start 2011-1-1 --end␣
↪2012-1-1 -o dma.pickle
```

  Grabs the data from yahoo finance, runs the file dual_moving_avg.py (and looks for `dual_moving_avg_analyze.py` which, if found, will be executed after the algorithm has been run), and outputs the perf `DataFrame` to `dma.pickle` (#325).

- IPython magic command (at the top of an IPython notebook cell). Example:

```
%%zipline --symbols AAPL --start 2011-1-1 --end 2012-1-1 -o perf
```

  Does the same as above except instead of executing the file looks for the algorithm in the cell and instead of outputting the perf df to a file, creates a variable in the namespace called perf (#325).

- Adds Trading Controls to the algorithm API.

  The following functions are now available on `TradingAlgorithm` and for algo scripts:

  `set_max_order_size(self, sid=None, max_shares=None, max_notional=None)` Set a limit on the absolute magnitude, in shares and/or total dollar value, of any single order placed by this algorithm for a given sid. If `sid` is None, then the rule is applied to any order placed by the algorithm. Example:

```
def initialize(context):
    # Algorithm will raise an exception if we attempt to place an
    # order which would cause us to hold more than 10 shares
    # or 1000 dollars worth of sid(24).
    set_max_order_size(sid(24), max_shares=10, max_notional=1000.0)
```

  `set_max_position_size(self, sid=None, max_shares=None, max_notional=None)` - Set a limit on the absolute magnitude, in either shares or dollar value, of any position held by the algorithm for a given sid. If `sid` is None, then the rule is applied to any position held by the algorithm. Example:

```
def initialize(context):
    # Algorithm will raise an exception if we attempt to order more than
    # 10 shares or 1000 dollars worth of sid(24) in a single order.
    set_max_order_size(sid(24), max_shares=10, max_notional=1000.0)

``set_max_order_count(self, max_count)``
Set a limit on the number of orders that can be placed by the algorithm in
a single trading day.
Example:
```

```
def initialize(context):
    # Algorithm will raise an exception if more than 50 orders are placed in a␣
↪day.
    set_max_order_count(50)
```

  `set_long_only(self)` Set a rule specifying that the algorithm may not hold short positions. Example:

```
def initialize(context):
    # Algorithm will raise an exception if it attempts to place
    # an order that would cause it to hold a short position.
    set_long_only()
```

  (#329).

- Adds an `all_api_methods` classmethod on `TradingAlgorithm` that returns a list of all `TradingAlgorithm` API methods (#333).
- Expanded record() functionality for dynamic naming. The record() function can now take positional args before the kwargs. All original usage and functionality is the same, but now these extra usages will work:

```
name = 'Dynamically_Generated_String'
record( name, value, ... )
record( name, value1, 'name2', value2, name3=value3, name4=value4 )
```

  The requirements are simply that the poritional args occur only before the kwargs (#355).

- history() has been ported from Quantopian to Zipline and provides moving window of market data. history() replaces BatchTransform. It is faster, works for minute level data and has a superior interface. To use it, call `add_history()` inside of `initialize()` and then receive a pandas `DataFrame` by calling history() from inside `handle_data()`. Check out the tutorial and an example. (#345 and #357).
- history() now supports `1m` window lengths (#345).

Bug Fixes

- Fix alignment of trading days and open and closes in trading environment (#331).
- RollingPanel fix when adding/dropping new fields (#349).

Performance

None

Maintenance and Refactorings

- Removed undocumented and untested HDF5 and CSV data sources (#267).
- Refactor sim_params (#352).
- Refactoring of history (#340).

Build

- The following dependencies have been updated (zipline might work with other versions too):

```
-pytz==2013.9
+pytz==2014.4
+numpy==1.8.1
-numpy==1.8.0
+scipy==0.12.0
+patsy==0.2.1
+statsmodels==0.5.0
-six==1.5.2
+six==1.6.1
-Cython==0.20
+Cython==0.20.1
-TA-Lib==0.4.8
+--allow-external TA-Lib --allow-unverified TA-Lib TA-Lib==0.4.8
-requests==2.2.0
+requests==2.3.0
-nose==1.3.0
+nose==1.3.3
-xlrd==0.9.2
+xlrd==0.9.3
-pep8==1.4.6
+pep8==1.5.7
-pyflakes==0.7.3
-pip-tools==0.3.4
+pyflakes==0.8.1`
-scipy==0.13.2
-tornado==3.2
-pyparsing==2.0.1
-patsy==0.2.1
-statsmodels==0.4.3
+tornado==3.2.1
+pyparsing==2.0.2
-Markdown==2.3.1
+Markdown==2.4.1
```

Contributors

The following people have contributed to this release, ordered by numbers of commit:

```
38  Scott Sanderson
29  Thomas Wiecki
26  Eddie Hebert
 6  Delaney Granizo-Mackenzie
 3  David Edwards
 3  Richard Frank
 2  Jonathan Kamens
 1  Pankaj Garg
 1  Tony Lambiris
 1  fawce
```

Release 0.6.1

Release 0.6.1

Date April 23, 2014

Highlights

- Major fixes to risk calculations, see Bug Fixes section.
- Port of `history()` function, see Enhancements section
- Start of support for Quantopian algorithm script-syntax, see ENH section.
- conda package manager support, see Build section.

Enhancements

- Always process new orders i.e. on bars where `handle_data` isn't called, but there is 'clock' data e.g. a consistent benchmark, process orders.
- Empty positions are now filtered from the portfolio container. To help prevent algorithms from operating on positions that are not in the existing universe of stocks. Formerly, iterating over positions would return positions for stocks which had zero shares held. (Where an explicit check in algorithm code for `pos.amount != 0` could prevent from using a non-existent position.)
- Add trading calendar for BMF&Bovespa.
- Add beginning of algo script support.
- Starts on the path of parity with the script syntax in Quantopian's IDE on https://quantopian.com Example:

```
from datetime import datetime import pytz
from zipline import TradingAlgorithm
from zipline.utils.factory import load_from_yahoo

from zipline.api import order

def initialize(context):
    context.test = 10
```

(continues on next page)

(continued from previous page)

```
def handle_date(context, data):
    order('AAPL', 10)
    print(context.test)

if __name__ == '__main__':
    import pylab as pl
    start = datetime(2008, 1, 1, 0, 0, 0, 0, pytz.utc)
    end = datetime(2010, 1, 1, 0, 0, 0, 0, pytz.utc)
    data = load_from_yahoo(
        stocks=['AAPL'],
        indexes={},
        start=start,
        end=end)
    data = data.dropna()
    algo = TradingAlgorithm(
        initialize=initialize,
        handle_data=handle_date)
    results = algo.run(data)
    results.portfolio_value.plot()
    pl.show()
```

- Add HDF5 and CSV sources.
- Limit `handle_data` to times with market data. To prevent cases where custom data types had unaligned timestamps, only call `handle_data` when market data passes through. Custom data that comes before market data will still update the data bar. But the handling of that data will only be done when there is actionable market data.
- Extended commission PerShare method to allow a minimum cost per trade.
- Add symbol api function A `symbol()` lookup feature was added to Quantopian. By adding the same API function to zipline we can make copy&pasting of a Zipline algo to Quantopian easier.
- Add simulated random trade source. Added a new data source that emits events with certain user-specified frequency (minute or daily). This allows users to backtest and debug an algorithm in minute mode to provide a cleaner path towards Quantopian.
- Remove dependency on benchmark for trading day calendar. Instead of the benchmarks' index, the trading calendar is now used to populate the environment's trading days. Remove `extra_date` field, since unlike the benchmarks list, the trading calendar can generate future dates, so dates for current day trading do not need to be appended. Motivations:
  - The source for the open and close/early close calendar and the trading day calendar is now the same, which should help prevent potential issues due to misalignment.
  - Allows configurations where the benchmark is provided as a generator based data source to need to supply a second benchmark list just to populate dates.
- Port `history()` API method from Quantopian. Opens the core of the `history()` function that was previously only available on the Quantopian platform.

  The history method is analoguous to the `batch_transform` function/decorator, but with a hopefully more precise specification of the frequency and period of the previous bar data that is captured. Example usage:

```
from zipline.api import history, add_history

def initialize(context):
    add_history(bar_count=2, frequency='1d', field='price')
```

(continues on next page)

(continued from previous page)

```
def handle_data(context, data):
    prices = history(bar_count=2, frequency='1d', field='price')
    context.last_prices = prices
```

N.B. this version of history lacks the backfilling capability that allows the return a full DataFrame on the first bar.

Bug Fixes

- Adjust benchmark events to match market hours (#241). Previously benchmark events were emitted at 0:00 on the day the benchmark related to: in 'minute' emission mode this meant that the benchmarks were emitted before any intra-day trades were processed.
- Ensure perf stats are generated for all days When running with minutely emissions the simulator would report to the user that it simulated 'n - 1' days (where n is the number of days specified in the simulation params). Now the correct number of trading days are reported as being simulated.
- Fix repr for cumulative risk metrics. The `__repr__` for RiskMetricsCumulative was referring to an older structure of the class, causing an exception when printed. Also, now prints the last values in the metrics DataFrame.
- Prevent minute emission from crashing at end of available data. The next day calculation was causing an error when a minute emission algorithm reached the end of available data. Instead of a generic exception when available data is reached, raise and catch a named exception so that the tradesimulation loop can skip over, since the next market close is not needed at the end.
- Fix pandas indexing in trading calendar. This could alternatively be filed under Performance. Index using loc instead of the inefficient index-ing of day, then time.
- Prevent crash in vwap transform due to non-existent member. The WrongDataForTransform was referencing a `self.fields` member, which did not exist. Add a self.fields member set to `price` and `volume` and use it to iterate over during the check.
- Fix max drawdown calculation. The input into max drawdown was incorrect, causing the bad results. i.e. the `compounded_log_returns` were not values representative of the algorithms total return at a given time, though `calculate_max_drawdown` was treating the values as if they were. Instead, the `algorithm_period_returns` series is now used, which does provide the total return.
- Fix cost basis calculation. Cost basis calculation now takes direction of txn into account. Closing a long position or covering a short shouldn't affect the cost basis.
- Fix floating point error in `order()`. Where order amounts that were near an integer could accidentally be floored or ceilinged (depending on being postive or negative) to the wrong integer. e.g. an amount stored internally as -27.99999 was converted to -27 instead of -28.
- Update perf period state when positions are changed by splits. Otherwise, `self._position_amounts` will be out of sync with position.amount, etc.
- Fix misalignment of downside series calc when using exact dates. An oddity that was exposed while working on making the return series passed to the risk module more exact, the series comparison between the returns and mean returns was unbalanced, because the mean returns were not masked down to the downside data points; however, in most, if not all cases this was papered over by the call to `.valid()` which was removed in this change set.
- Check that self.logger exists before using it. `self.logger` is initialized as `None` and there is no guarantee that users have set it, so check that it exists before trying to pass messages to it.
- Prevent out of sync market closes in performance tracker. In situations where the performance tracker has been reset or patched to handle state juggling with warming up live data, the `market_close` member of the

performance tracker could end up out of sync with the current algo time as determined by the performance tracker. The symptom was dividends never triggering, because the end of day checks would not match the current time. Fix by having the tradesimulation loop be responsible, in minute/minute mode, for advancing the market close and passing that value to the performance tracker, instead of having the market close advanced by the performance tracker as well.

- Fix numerous cumulative and period risk calculations. The calculations that are expected to change are:
  - `cumulative.beta`
  - `cumulative.alpha`
  - `cumulative.information`
  - `cumulative.sharpe`
  - `period.sortino`

  How Risk Calculations Are Changing Risk Fixes for Both Period and Cumulative

  Downside Risk

  Use sample instead of population for standard deviation.

  Add a rounding factor, so that if the two values are close for a given dt, that they do not count as a downside value, which would throw off the denominator of the standard deviation of the downside diffs.

  Standard Deviation Type

  Across the board the standard deviation has been standardized to using a 'sample' calculation, whereas before cumulative risk was mostly using 'population'. Using `ddof=1` with `np.std` calculates as if the values are a sample.

  Cumulative Risk Fixes

  Beta

  Use the daily algorithm returns and benchmarks instead of annualized mean returns.

  Volatility

  Use sample instead of population with standard deviation.

  The volatility is an input to other calculations so this change affects Sharpe and Information ratio calculations.

  Information Ratio

  The benchmark returns input is changed from annualized benchmark returns to the annualized mean returns.

  Alpha

  The benchmark returns input is changed from annualized benchmark returns to the annualized mean returns.

  Period Risk Fixes

  Sortino

  Now uses the downside risk of the daily return vs. the mean algorithm returns for the minimum acceptable return instead of the treasury return.

  The above required adding the calculation of the mean algorithm returns for period risk.

  Also, uses `algorithm_period_returns` and `tresaury_period_return` as the cumulative Sortino does, instead of using algorithm returns for both inputs into the Sortino calculation.

Performance

- Removed `alias_dt` transform in favor of property on SIDData. Adding a copy of the Event's dt field as datetime via the `alias_dt` generator, so that the API was forgiving and allowed both datetime and dt on a SIDData object, was creating noticeable overhead, even on an noop algorithms. Instead of incurring the cost of copying the datetime value and assigning it to the Event object on every event that is passed through the system, add a property to SIDData which acts as an alias `datetime` to `dt`. Eventually support for `data['foo'].datetime` may be removed, and could be considered deprecated.
- Remove the drop of 'null return' from cumulative returns. The check of existence of the null return key, and the drop of said return on every single bar was adding unneeded CPU time when an algorithm was run with minute emissions. Instead, add the 0.0 return with an index of the trading day before the start date. The removal of the `null return` was mainly in place so that the period calculation was not crashing on a non-date index value; with the index as a date, the period return can also approximate volatility (even though the that volatility has high noise-to-signal strength because it uses only two values as an input.)

Maintenance and Refactorings

- Allow `sim_params` to provide data frequency for the algorithm. In the case that `data_frequency` of the algorithm is None, allow the `sim_params` to provide the `data_frequency`.

  Also, defer to the algorithms data frequency, if provided.

Build

- Added support for building and releasing via conda For those who prefer building with https://docs.conda.io/en/latest/ to compiling locally with pip. The following should install Zipline on many systems.

```
conda install -c quantopian zipline
```

Contributors

The following people have contributed to this release, ordered by numbers of commit:

```
49  Eddie Hebert
28  Thomas Wiecki
11  Richard Frank
 2  Jamie Kirkpatrick
 2  Jeremiah Lowin
 1  Colin Alexander
 1  Michael Schatzow
 1  Moises Trovo
 1  Suminda Dharmasena
```

INDEX

Z

zipline.data.bundles.bundles (*built-in variable*), 40
zipline.data.finance.metrics.metrics_sets (*built-in variable*), 41