Mastering pandas for Finance

This book will teach you to use Python and the Python Data Analysis Library (pandas) to solve real-world financial problems.

Starting with a focus on pandas data structures, you will learn to load and manipulate time-series financial data and then calculate common financial measures, leading into more advanced derivations using fixed- and moving-windows. This leads into correlating time-series data to both index and social data to build simple trading algorithms. From there, you will learn about more complex trading algorithms and implement them using open source back-testing tools. Then, you will examine the calculation of the value of options and Value at Risk. This then leads into the modeling of portfolios and calculation of optimal portfolios based upon risk. All concepts will be demonstrated continuously through progressive examples using interactive Python and IPython Notebook.

By the end of the book, you will be familiar with applying pandas to many financial problems, giving you the knowledge needed to leverage pandas in the real world of finance.

Who this book is written for

If you are interested in quantitative finance, financial modeling, and trading, or simply want to learn how Python and pandas can be applied to finance, then this book is ideal for you. Some knowledge of Python and pandas is assumed. Interest in financial concepts is helpful, but no prior knowledge is expected.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 'Algorithmic Trading'
- A synopsis of the book’s content
- More information on Mastering pandas for Finance

About the Author

Michael Heydt is an independent consultant, educator, and trainer with nearly 30 years of professional software development experience, during which time, he focused on Agile software design and implementation using advanced technologies in multiple verticals, including media, finance, energy, and healthcare. He holds an MS degree in mathematics and computer science from Drexel University and an executive master's of technology management degree from the University of Pennsylvania's School of Engineering and Wharton Business School. His studies and research have focused on technology management, software engineering, entrepreneurship, information retrieval, data sciences, and computational finance.

Since 2005, he has specialized in building energy and financial trading systems for major investment banks on Wall Street and for several global energy-trading companies, utilizing .NET, C#, WPF, TPL, DataFlow, Python, R, Mono, iOS, and Android. His current interests include creating seamless applications using desktop, mobile, and wearable technologies, which utilize high-concurrency, high-availability, and real-time data analytics; augmented and virtual reality; cloud services; messaging; computer vision; natural user interfaces; and software-defined networks. He is the author of numerous technology articles, papers, and books. He is a frequent speaker at .NET user groups and various mobile and cloud conferences, and he regularly delivers webinars and conducts training courses on emerging and advanced technologies. To know more about Michael, visit his website at http://bseamless.com/.
Mastering pandas for Finance

*Mastering pandas for Finance* will teach you how to use Python and pandas to model and solve real-world financial problems using pandas, Python, and several open source tools that assist in various financial tasks, such as option pricing and algorithmic trading.

This book brings together various diverse concepts related to finance in an attempt to provide a unified reference to discover and learn several important concepts in finance and explains how to implement them using a core of Python and pandas that provides a unified experience across the different models and tools.

You will start by learning about the facilities provided by pandas to model financial information, specifically time-series data, and to use its built-in capabilities to manipulate time-series data, group and derive aggregate results, and calculate common financial measurements, such as percentage changes, correlation of time-series, various moving window operations, and key data visualizations for finance.

After establishing a strong foundation from which to use pandas to model financial time-series data, the book turns its attention to using pandas as a tool to model the data that is required as a base for performing other financial calculations. The book will cover diverse areas in which pandas can assist, including the correlations of Google trends with stock movements, creating algorithmic trading systems, and calculating options payoffs, prices, and behaviors. The book also shows how to model portfolios and their risk and to optimize them for specific risk/return tolerances.

What This Book Covers

*Chapter 1, Getting Started with pandas Using Wakari.io*, walks you through using Wakari.io, an online collaborative data analytics platform, that utilizes Python, IPython Notebook, and pandas. We will start with a brief overview of Wakari.io and step through how to upgrade the default Python environment and install all of the tools used throughout this text. At the end, you will have a fully functional financial analytics platform supporting all of the examples we will cover.

*Chapter 2, Introducing the Series and DataFrame*, teaches you about the core pandas data structures—the Series and the DataFrame. You will learn how a Series expands on the functionality of the NumPy array to provide much richer representation and manipulation of sequences of data through the use of high-performance indices. You will then learn about the pandas DataFrame and how to use it to model two-dimensional tabular data.
Chapter 3, *Reshaping, Reorganizing, and Aggregating*, focuses on how to use pandas to group data, enabling you to perform aggregate operations on grouped data to assist with deriving analytic results. You will learn to reorganize, group, and aggregate stock data and to use grouped data to calculate simple risk measurements.

Chapter 4, *Time-series*, explains how to use pandas to represent sequences of pricing data that are indexed by the progression of time. You will learn how pandas represents date and time as well as concepts such as periods, frequencies, time zones, and calendars. The focus then shifts to learning how to model time-series data with pandas and to perform various operations such as shifting, lagging, resampling, and moving window operations.

Chapter 5, *Time-series Stock Data*, leads you through retrieving and performing various financial calculations using historical stock quotes obtained from Yahoo! Finance. You will learn to retrieve quotes, perform various calculations, such as percentage changes, cumulative returns, moving averages, and volatility, and finish with demonstrations of several analysis techniques including return distribution, correlation, and least squares analysis.

Chapter 6, *Trading Using Google Trends*, demonstrates how to form correlations between index data and trends in searches on Google. You will learn how to gather index data from Quandl along with trend data from Google and then how to correlate this time-series data and use that information to generate trade signals, which will be used to calculate the effectiveness of the trading strategy as compared to the actual market performance.

Chapter 7, *Algorithmic Trading*, introduces you to the concepts of algorithmic trading through demonstrations of several trading strategies, including simple moving averages, exponentially weighted averages, crossovers, and pairs-trading. You will then learn to implement these strategies with pandas data structures and to use Zipline, an open source back-testing tool, to simulate trading behavior on historical data.

Chapter 8, *Working with Options*, teaches you to model and evaluate options. You will first learn briefly about options, how they function, and how to calculate their payoffs. You will then load options data from Yahoo! Finance into pandas data structures and examine various options attributes, such as implied volatility and volatility smiles and smirks. We then examine the pricing of options with Black-Scholes using Mibian and finish with an overview of Greeks and how to calculate them using Mibian.

Chapter 9, *Portfolios and Risk*, will teach you how to model portfolios of multiple stocks using pandas. You will learn about the concepts of Modern Portfolio Theory and how to apply those theories with pandas and Python to calculate the risk and returns of a portfolio, assign different weights to different instruments in a portfolio, derive the Sharpe ratio, calculate efficient frontiers and value at risk, and optimize portfolio instrument allocation.
Algorithmic Trading

In this chapter, we will examine how to use pandas and a library known as Zipline to develop automated trading algorithms. Zipline (http://www.zipline.io/) is a Python-based algorithmic trading library. It provides event-driven approximations of live-trading systems. It is currently used in production as the trading engine that powers Quantopian (https://www.quantopian.com/), a free, community-centered platform for collaborating on the development of trading algorithms with a web browser.

We previously simulated trading based on a historical review of social and stock data, but these examples were naive in that they glossed over many facets of real trading, such as transaction fees, commissions, and slippage, among many others. Zipline provides robust capabilities to include these factors in the trading model.

Zipline also provides a facility referred to as backtesting. Backtesting is the ability to run an algorithm on historical data to determine the effectiveness of the decisions made on actual market data. This can be used to vet the algorithm and compare it to others in an effort to determine the best trading decisions for your situation.

We will examine three specific and fundamental trading algorithms: simple crossover, dual moving average crossover, and pairs trade. We will first look at how these algorithms operate and make decisions, and then we will actually implement these using Zipline and execute and analyze them on historical data.

This chapter will cover the following topics in detail:

- The process of algorithmic trading
- Momentum and mean-reversion strategies
- Moving averages and their significance in automated decision making
- Simple and exponentially weighted moving averages
- Common algorithms used in algorithmic trading
Crossovers, including simple and dual moving average crossovers
• Pairs trading strategies
• Implementing dual moving crossover and pairs trading algorithms in Zipline

Notebook setup
The Notebook and examples will all require the following code to execute and format output. Later in the chapter, we will import the Zipline package but only after first discussing how to install it in your Python environment:

```python
In [1]:
   import pandas as pd
   import pandas.io.data as web
   import numpy as np
   from datetime import datetime
   import matplotlib.pyplot as plt
   %matplotlib inline

   pd.set_option('display.notebook_repr_html', False)
   pd.set_option('display.max_columns', 8)
   pd.set_option('display.max_rows', 10)
   pd.set_option('display.width', 78)
   pd.set_option('precision', 6)
```

The process of algorithmic trading
Algorithmic trading is the use of an automated system to execute trades in a market. These trades are executed in a predetermined manner using one or more algorithms and without human interaction. In this chapter, we will examine several common trading algorithms, along with tools that you can use in combination with pandas to determine the effectiveness of your trading algorithms.

Financial markets move in cycles. Proper identification of the movement of the market can lead to opportunities for profit by making appropriate and timely buys or sells of financial instruments. There are two broad categories for predicting movement in the market, which we will examine in this chapter: momentum strategies and mean-reversion strategies.
Momentum strategies
In momentum trading, trading focuses on stocks that are moving in a specific direction on high volume, measuring the rate of change in price changes. It is typically measured by continuously computing price differences at fixed time intervals. Momentum is a useful indicator of the strength or weakness of the price although it is typically more useful during rising markets as they occur more frequently than falling markets; therefore, momentum-based prediction gives better results in a rising market.

Mean-reversion strategies
Mean reversion is a theory in trading that prices and returns will eventually move back towards the mean of the stock or of another historical average, such as the growth of the economy or an industry average. When the market price is below the average price, a stock is considered attractive for purchase as it is expected that the price will rise and, hence, a profit can be made by buying and holding the stock as it rises and then selling at its peak. If the current market price is above the mean, the expectation is the price will fall and there is potential for profit in shorting the stock.

Moving averages
Whether using a momentum or mean-reversion strategy for trading, the analyses will, in one form or another, utilize moving averages of the closing price of stocks. We have seen these before when we looked at calculating a rolling mean. We will now examine several different forms of rolling means and cover several concepts that are important to use in order to make trading decisions based upon how one or more means move over time:

- Simple moving average
- Exponential moving average

Simple moving average
A moving average is a technical analysis technique that smooths price data by calculating a constantly updated average price. This average is taken over a specific period of time, ranging from minutes, to days, weeks, and months. The period selected depends on the type of movement of interest, such as making a decision on short-term, medium-term, or long-term investment.

Moving averages give us a means to relate the price data to determine a trend indicator. A moving average does not predict price direction but instead gives us a means of determining the direction of the price with a lag, which is the size of the window.
Algorithmic Trading

In financial markets, a moving average can be considered support in a rising market and resistance in a falling market.

For more info on support and resistance, visit http://www.investopedia.com/articles/technical/061801.asp.

To demonstrate this, take a look at the closing price of MSFT for 2014 related to its 7-day, 30-day, and 120-day rolling means during the same period:

In [2]:
   msft = web.DataReader("MSFT", "yahoo",
   datetime(2000, 1, 1),
   datetime(2014, 12, 31))

msft[:5]

Out[2]:

<table>
<thead>
<tr>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
<th>Adj Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>117.38</td>
<td>118.62</td>
<td>112.00</td>
<td>116.56</td>
<td>53228400</td>
<td>41.77</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>113.56</td>
<td>117.12</td>
<td>112.25</td>
<td>112.62</td>
<td>54119000</td>
<td>40.36</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>111.12</td>
<td>116.38</td>
<td>109.38</td>
<td>113.81</td>
<td>64059600</td>
<td>40.78</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>112.19</td>
<td>113.88</td>
<td>108.38</td>
<td>110.00</td>
<td>54976600</td>
<td>39.41</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>108.62</td>
<td>112.25</td>
<td>107.31</td>
<td>111.44</td>
<td>62013600</td>
<td>39.93</td>
</tr>
</tbody>
</table>

Now, we can calculate the rolling means using pd.rolling_mean():

In [3]:
   msft['MA7'] = pd.rolling_mean(msft['Adj Close'], 7)
   msft['MA30'] = pd.rolling_mean(msft['Adj Close'], 30)
   msft['MA90'] = pd.rolling_mean(msft['Adj Close'], 90)
   msft['MA120'] = pd.rolling_mean(msft['Adj Close'], 120)
Then, we plot the price versus various rolling means to see this concept of support:

In [4]:
msft['2014'][['Adj Close', 'MAM7',
               'MA30', 'MA120']].plot(figsize=(12,8));

The price of MSFT had a progressive rise over 2014, and the 120-day rolling mean has functioned as a floor/support, where the price bounces off this floor as it approaches it. The longer the window of the rolling mean, the lower and smoother the floor will be in an uptrending market.
Algorithmic Trading

Contrast this with the price of the stock in 2002, when it had a steady decrease in value:

In [5]:

```
msft['2014'][['Adj Close', 'MA7',
              'MA30', 'MA120']].plot(figsize=(12,8));
```

In this situation, the 120-day moving average functions as a ceiling for about 9 months. This ceiling is referred to as resistance as it tends to push prices down as they rise up towards this ceiling.

The price does not always respect the moving average. In both of these cases, the prices have crossed over the moving average, and, at times, it has reversed its movement slightly before or just after crossing the average.
In general, though, if the price is above a particular moving average, then it can be said that the trend for that stock is up relative to that average and when the price is below a particular moving average, the trend is down.

The means of calculating the moving average used in the previous example is considered a simple moving average (SMA). The example demonstrated calculated the 7, 30, and 120 SMA values.

While valuable and used to form the basis of other technical analyses, simple moving averages have several drawbacks. They are listed as follows:

- The shorter the window used, the more the noise in the signal feeds into the result
- Even though it uses actual data, it is lagging behind it by the size of the window
- It never reaches the peaks or valleys of the actual data as it is smoothing the data
- It does not tell you anything about the future
- The average calculated at the end of the window can be significantly skewed by the values earlier in the window that are significantly skewed from the mean

To help address some of these concerns, it is common to instead use an exponentially weighted moving average.

**Exponentially weighted moving average**

Exponential moving averages reduce the lag and effect of exceptional values early in a window by applying more weight to recent prices. The amount of weighting applied to the most recent price depends on the number of periods in the moving average and how the exponential function is formulated.

In general, the weighted moving average is calculated using the following formula:

$$y_t = \frac{\sum_{i=0}^{t} w_i x_{t-i}}{\sum_{i=0}^{t} w_i}$$

In the preceding formula, $x_t$ is the input and $y_t$ is the result.
The EW functions in pandas support two variants of exponential weights: The default, `adjust=True`, uses the following weights:

$$w_i = (1 - \alpha)^i w_i = (1 - \alpha)^i$$

When `adjust=False` is specified, moving averages are calculated using the following formula:

$$y_0 = x_0$$

The preceding formula is followed by this formula:

$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t$$

This is equivalent to using weights:

$$w_i = \begin{cases} \alpha(1 - \alpha)^i & \text{if } i < t \\ (1 - \alpha)^i & \text{if } i = t. \end{cases}$$

However, instead of dealing with these formulas as described, pandas takes a slightly different approach to specifying the weighting. Instead of specifying an alpha between 0 and 1, pandas attempts to make the process less abstract by letting you specify alpha in terms of either span, center of mass, or half life:

$$\alpha = \begin{cases} \frac{2}{s + 1} & \text{s = span} \\ \frac{1}{1 + c} & \text{c = center of mass} \\ 1 - \exp \frac{\log s}{h} & \text{h = half life} \end{cases}$$

One must specify precisely one of the three values to the `pd.ewma()` function at which point pandas will use the corresponding formulation for alpha.
As an example, a span of 10 corresponds to what is commonly referred to as a 10-day exponentially weighted moving average. The following command demonstrates the calculation of the percentage weights that will be used for each data point in a 10-span EWMA (alpha=0.18181818):

In [6]:
    periods = 10
    alpha = 2.0/(periods +1)
    factors = (1-alpha) ** np.arange(1, 11)
    sum_factors = factors.sum()
    weights = factors/sum_factors
    weights

Out[6]:
    array([ 0.21005616,  0.17186413,  0.14061611,  0.11504954,
            0.09413145, 0.07701664,  0.06301361,  0.05155659,  0.04218267,
            0.03451309])

The most recent value is weighted at 21 percent of the result, and this decreases by a factor (1-alpha) across all the points, and the total of these weights is equal to 1.0.

The center of mass option specifies the point where half of the number of weights would be on each side of the center of mass. In the case of a 10-period span, the center of mass is 5.5. Data points 1, 2, 3, 4, and 5 are on one side, and 6, 7, 8, 9, and 10 are on the other. The actual weight is not taken into account — just the number of items.

The half-life specification specifies the period of time for the percentage of the weighting factor to become half of its value. For the 10-period span, the half-life value is 3.454152. The first weight is 0.21, and we would expect that to reduce to 0.105 just under halfway between points 4 and 5 (1+3.454152=4.454152). These values are 0.115 and 0.094, and 0.105 is indeed between the two.
The following example demonstrates how the exponential weighted moving average differs from a normal moving average. It calculates both kinds of averages for a 90-day window and plots the results:

```
In [7]:
    span = 90
    msft_ewma = msft[['Adj Close']].copy()
    msft_ewma['MA90'] = pd.rolling_mean(msft_ewma, span)
    msft_ewma['EWMA90'] = pd.ewma(msft_ewma['Adj Close'],
                                   span=span)
    msft_ewma['2014'].plot(figsize=(12, 8));
```

The exponential moving averages exhibit less lag, and, therefore, are more sensitive to recent prices and price changes. Since more recent values are favored, they will turn before simple moving averages, facilitating decision making on changes in momentum.

Comparatively, a simple moving average represents a truer average of prices for the entire time period. Therefore, a simple moving average may be better suited to identify the support or resistance level.
Technical analysis techniques

We will now cover two categories of technical analysis techniques, which utilize moving averages in different ways to be able to determine trends in market movements and hence give us the information needed to make potentially profitable transactions. We will examine how this works in this section, and in the upcoming section on Zipline, we will see how to implement these strategies in pandas and Zipline.

Crossovers

A crossover is the most basic type of signal for trading. The simplest form of a crossover is when the price of an asset moves from one side of a moving average to the other. This crossover represents a change in momentum and can be used as a point of making the decision to enter or exit the market.

The following command exemplifies several crossovers in the Microsoft data:

```
In [8]:
    msft['2002-1':'2002-9'][['Adj Close',
                              'MA30']].plot(figsize=(12,8));
```
As an example, the cross occurring on July 09, 2002, is a signal of the beginning of a downtrend and would likely be used to close out any existing long positions. Conversely, a close above a moving average, as shown around August 13, may suggest the beginning of a new uptrend and a signal to go short on the stock.

A second type of crossover, referred to as a dual moving average crossover, occurs when a short-term average crosses a long-term average. This signal is used to identify that momentum is shifting in the direction of the short-term average. A buy signal is generated when the short-term average crosses the long-term average and rises above it, while a sell signal is triggered by a short-term average crossing long-term average and falling below it.

To demonstrate this, the following command shows MSFT for January 2002 through June 2002. There is one crossover of the 30- and 90-day moving averages with the 30-day crossing moving from above to below the 90-day average. This is a significant signal of the downswing of the stock during upcoming intervals:

In [9]:
```python
msft['2002-1':'2002-6'][['Adj Close', 'MA30', 'MA90']].plot(figsize=(12,8));
```
Pairs trading

Pairs trading is a strategy that implements a statistical arbitrage and convergence. The basic idea is that, as we have seen, prices tend to move back to the mean. If two stocks can be identified that have a relatively high correlation, then the change in the difference in price between the two stocks can be used to signal trading events if one of the two moves out of correlation with the other.

If the change in the spread between the two stocks exceeds a certain level (their correlation has decreased), then the higher-priced stock can be considered to be in a short position and should be sold as it is assumed that the spread will decrease as the higher-priced stock returns to the mean (decreases in price as the correlation returns to a higher level). Likewise, the lower-priced stock is in a long position, and it is assumed that the price will rise as the correlation returns to normal levels.

This strategy relies on the two stocks being correlated as temporary reductions in correlation by one stock making either a positive or negative move. This is based upon the effects on one of the stocks that outside of shared market forces. This difference can be used to our advantage in an arbitrage by selling and buying equal amounts of each stock and profiting as the two prices move back into correlation. Of course, if the two stocks move into a truly different level of correlation, then this might be a losing situation.

Coca-Cola (KO) and Pepsi (PEP) are a canonical example of pairs-trading as they are both in the same market segment and are both likely to be affected by the same market events, such as the price of the common ingredients.
Algorithmic Trading

As an example, the following screenshot shows the price of Pepsi and Coca-Cola from January 1997 through June 1998 (we will revisit this series of data later when we implement pairs trading):

These prices are generally highly correlated during this period, but there is a marked change in correlation that starts in August 1997 and seems to take until the end of the year to move back into alignment. This is a situation where pairs trading can give profits if identified and executed properly.
Algo trading with Zipline

Zipline is a very powerful tool with many options, most of which we will not be able to investigate in this book. It makes creating trading algorithms and their simulation on historical data very easy (but there is still some creativity required).

Zipline provides several operational models. One allows the execution of Python script files via the command line. We will exclusively use a model where we include Zipline into our pandas application and request it to run our algorithms.

To do this, we will need to implement our algorithms and instruct Zipline on how to run them. This is actually a very simple process, and we will walk through implementing three algorithms of increasing complexity: buy apple, dual moving average crossover, and pairs trade.

The algorithms that we will implement have been discussed earlier: the dual moving average crossover and the pairs trading mean-reversion algorithm. We will, however, start with a very simple algorithm, buy apple, which will be used to demonstrate the overall process of how to create an algorithm as well as to show many of the things that Zipline handles automatically.

The three examples we will examine are available as part of this distribution, but we will examine them in detail. They have been modified to work exclusively within an IPython environment using pandas and to implement several of the constructs inherent in the examples in a manner that is better for understanding in the context of this book.

Algorithm – buy apple

Trading algorithms in Zipline are implemented in several manners. The technique we will use is creating a subclass of Zipline, that is, the TradingAlgorithm class and run the simulation within IPython with the Zipline engine.

The tracing is implemented as a static variable and the initialize method is called by Zipline as a static method to set up trading simulation. Also, initialize is called by Zipline prior to the completion of the call to super(), so to enable tracing, the member must be initialized before the call to super().

[ 181 ]
The following is a simple algorithm for trading AAPL that is provided with the Zipline examples, albeit modified to be in a class, and run in IPython. Then, print some additional diagnostic code to trace how the process is executing in more detail:

```
In [11]:
class BuyApple(zp.TradingAlgorithm):
    trace=False

    def __init__(self, trace=False):
        BuyApple.trace = trace
        super(BuyApple, self).__init__()

    def initialize(context):
        if BuyApple.trace: print("---> initialize")
        if BuyApple.trace: print(context)
        if BuyApple.trace: print("<--- initialize")

    def handle_data(self, context):
        if BuyApple.trace: print("---> handle_data")
        if BuyApple.trace: print(context)
        self.order("AAPL", 1)
        if BuyApple.trace: print("<-- handle_data")
```

Trading simulation starts with the call to the static .initialize() method. This is your opportunity to initialize the trading simulation. In this sample, we do not perform any initialization other than printing the context for examination.

The implementation of the actual trading is handled in the override of the handle_data method. This method will be called for each day of the trading simulation. It is your opportunity to analyze the state of the simulation provided by the context and make any trading actions you desire. In this example, we will buy one share of AAPL regardless of how AAPL is performing.

The trading simulation can be started by instantiating an instance of BuyApple() and calling that object's .run method, thereby passing the base data for the simulation, which we will retrieve from Zipline's own method for accessing data from Yahoo! Finance:

```
In [12]:
    import zipline.utils.factory as zpf
    data = zpf.load_from_yahoo(stocks=['AAPL'],
```
Our first simulation will purposely use only one week of historical data so that we can easily keep the output to a nominal size that will help us to easily examine the results of the simulation:

In [13]:
    result = BuyApple().run(data['2000-01-03':'2000-01-07'])
    ---> initialize
    BuyApple(
        capital_base=100000.0
        sim_params=
        SimulationParameters(
            period_start=2006-01-01 00:00:00+00:00,
            period_end=2006-12-31 00:00:00+00:00,
Algorithmic Trading

capital_base=100000.0,
data_frequency=daily,
emission_rate=daily,
first_open=2006-01-03 14:31:00+00:00,
last_close=2006-12-29 21:00:00+00:00),
initialized=False,
slippage=VolumeShareSlippage(
  volume_limit=0.25,
  price_impact=0.1),
commission=PerShare(cost=0.03, min trade cost=None),
blotter=Blotter(
  transact_partial=(VolumeShareSlippage(
    volume_limit=0.25,
    price_impact=0.1), PerShare(cost=0.03, min trade cost=None)),
  open_orders=defaultdict(<type 'list'>, {}),
  orders={},
  new_orders=[],
  current_dt=None),
recorded_vars={})

<--- initialize
--- handle_data
BarData({'AAPL': SIDData({'volume': 1000, 'sid': 'AAPL',
'source_id': 'DataFrameSource-fc37c5097c557f0d46d6713256f4ea3',
'dt': Timestamp('2000-01-03 00:00:00+0000', tz='UTC'),
'type': 4, 'price': 111.94})})

<-- handle_data
--- handle_data
[2015-04-16 21:53] INFO: Performance: last close: 2000-01-07 21:00:00+00:00

BarData({'AAPL': SIDData({'price': 102.5, 'volume': 1000, 'sid': 'AAPL',
'source_id': 'DataFrameSource-fc37c5097c557f0d46d6713256f4ea3',
'dt': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
'type': 4})})
BarData({'AAPL': SIDData({'price': 104.0, 'volume': 1000, 'sid': 'AAPL', 'source_id': 'DataFrameSource-fc37c5097c557f0d46d6713256f4eea3', 'dt': Timestamp('2000-01-05 00:00:00+0000', tz='UTC'), 'type': 4})})

BarData({'AAPL': SIDData({'price': 95.0, 'volume': 1000, 'sid': 'AAPL', 'source_id': 'DataFrameSource-fc37c5097c557f0d46d6713256f4eea3', 'dt': Timestamp('2000-01-05 00:00:00+0000', tz='UTC'), 'type': 4})})

BarData({'AAPL': SIDData({'price': 99.5, 'volume': 1000, 'sid': 'AAPL', 'source_id': 'DataFrameSource-fc37c5097c557f0d46d6713256f4eea3', 'dt': Timestamp('2000-01-05 00:00:00+0000', tz='UTC'), 'type': 4})})

The context in the `initialize` method shows us some parameters that the simulation will use during its execution. The context also shows that we start with a base capitalization of 100000.0. There will be a commission of $0.03 assessed for each share purchased.

The context is also printed for each day of trading. The output shows us that Zipline passes the price data for each day of AAPL. We do not utilize this information in this simulation and blindly purchase one share of AAPL.

The result of the simulation is assigned to the result variable, which we can analyze for detailed results of the simulation on each day of trading. This is a DataFrame where each column represents a particular measurement during the simulation, and each row represents the values of those variables on each day of trading during the simulation.

We can examine a number of the variables to demonstrate what Zipline was doing during the processing. The orders variable contains a list of all orders made during the day. The following command gets the orders for the first day of the simulation:

```
In [14]:
    result.iloc[0].orders
```

```
Out[14]:
[{'amount': 1,
```
This tells us that Zipline placed an order in the market for one share of AAPL on 2000-01-03. The order filled the value 0, which means that this trade has not yet been executed in the market.

On the second day of trading, Zipline reports that two orders were made:

In [15]:
   result.iloc[1].orders

Out[15]:
[{'amount': 1,
  'commission': 0.03,
  'created': Timestamp('2000-01-03 00:00:00+0000', tz='UTC'),
  'dt': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
  'filled': 1,
  'id': 'dccb19f416104f259a7f0bff726136a2',
  'limit': None,
  'limit_reached': False,
  'sid': 'AAPL',
  'status': 1,
  'stop': None,
  'stop_reached': False},
{'amount': 1,
  'commission': None,
  'created': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
  'dt': Timestamp('2000-01-04 00:00:00+0000', tz='UTC'),
  'filled': 0,
  'id': '1ec23ea51fd7429fa97b9f29a66bf66a'}]
The first order listed has the same ID as the order from day one. This tells us that this represents that same order, and we can see this from the filled key, which is now 1 and from the fact that this order has been filled in the market.

The second order is a new order, which represents our request on the second day of trading, which will be reported as filled at the start of day two.

During the simulation, Zipline keeps track of the amount of cash we have (capital) at the start and end of the day. As we purchase stocks, our cash is reduced. Starting and ending cash is represented by the `starting_cash` and `ending_cash` variables of the result.

Zipline also accumulates the total value of the purchases of stock during the simulation. This value is represented in each trading period using the `ending_value` variable of the result.

The following command shows us the running values for `ending_cash` and `ending_value`, along with `ending_value`:

```
In [16]:
    result[['starting_cash', 'ending_cash', 'ending_value']]
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Starting Cash</th>
<th>Ending Cash</th>
<th>Ending Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>100000.0000</td>
<td>100000.0000</td>
<td>0.0</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>99897.46999</td>
<td>99793.43998</td>
<td>208.0</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>99793.43998</td>
<td>99698.40997</td>
<td>285.0</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>99698.40997</td>
<td>99598.87996</td>
<td>398.0</td>
</tr>
</tbody>
</table>

Ending cash represents the amount of cash (capital) that we have to invest at the end of the given day. We made an order on day one for one share of the apple, but since the transaction did not execute until the next day, we still have our starting seed at the end of the day. But on day two, this will execute at the value reported at the close of day one, which is 111.94. Hence, our `ending_cash` is reduced by 111.94 for one share and also deducted is the $0.03 for the commission resulting in 9987.47.
Algorithmic Trading

At the end of day two, our ending_value, that is, our position in the market, is 102.5 as we have accumulated one share of AAPL, and it closed at 102.5 on day two.

We did not print starting_cash and starting_value as this will always be equal to our initial capitalization of 100000.0 and a portfolio value of 0.0 as we have not yet bought any securities.

While investing, we would be interested in the overall value of our portfolio, which, in this case, would be the value of our on-hand cash + our position in the market. This can be easily calculated:

In [17]:
   
   pvalue = result.ending_cash + result.ending_value
   pvalue

Out[17]:
   2000-01-03 21:00:00    100000.00000
   2000-01-04 21:00:00     99999.96999
   2000-01-05 21:00:00    100001.43998
   2000-01-06 21:00:00     99983.40997
   2000-01-07 21:00:00     99996.87996
   dtype: float64

There is also a convenient shorthand to retrieve this result:

In [18]:
   
   result.portfolio_value

Out[18]:
   2000-01-03 21:00:00    100000.00000
   2000-01-04 21:00:00     99999.96999
   2000-01-05 21:00:00    100001.43998
   2000-01-06 21:00:00     99983.40997
   2000-01-07 21:00:00     99996.87996
   Name: portfolio_value, dtype: float64

In a similar vein, we can also calculate the daily returns on our investment using .pct_change():

In [19]:
   
   result.portfolio_value.pct_change()
This is actually a column of the results from the simulation, so we do not need to actually calculate it:

In [20]:
result['returns']

Out[20]:
2000-01-03 21:00:00            NaN
2000-01-04 21:00:00   -3.00103e-07
2000-01-05 21:00:00    1.46999e-05
2000-01-06 21:00:00   -1.80297e-04
2000-01-07 21:00:00    1.34722e-04
Name: portfolio_value, dtype: float64

Using this small trading interval, we have seen what type of calculations Zipline performs during each period. Now, let's run this simulation over a longer period of time to see how it performs. The following command runs the simulation across the entire year 2000:

In [21]:
result_for_2000 = BuyApple().run(data['2000'])

Out[21]:
[2015-02-15 05:05] INFO: Performance: first open: 2000-01-03 14:31:00+00:00
[2015-02-15 05:05] INFO: Performance: last close: 2000-12-29 21:00:00+00:00

The following command shows us our cash on hand and the value of our investments throughout the simulation:

In [22]:
result_for_2000[['ending_cash', 'ending_value']]
Out[22]:

<table>
<thead>
<tr>
<th>Date</th>
<th>ending_cash</th>
<th>ending_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-03</td>
<td>100000.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>99897.47</td>
<td>102.50</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>99793.44</td>
<td>208.00</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>99698.41</td>
<td>285.00</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>99598.88</td>
<td>398.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000-12-22</td>
<td>82082.91</td>
<td>3705.00</td>
</tr>
<tr>
<td>2000-12-26</td>
<td>82068.19</td>
<td>3643.12</td>
</tr>
<tr>
<td>2000-12-27</td>
<td>82053.35</td>
<td>3687.69</td>
</tr>
<tr>
<td>2000-12-28</td>
<td>82038.51</td>
<td>3702.50</td>
</tr>
<tr>
<td>2000-12-29</td>
<td>82023.60</td>
<td>3734.88</td>
</tr>
</tbody>
</table>

[252 rows x 2 columns]

The following command visualizes our overall portfolio value during the year 2000:

In [23]:

```
result_for_2000.portfolio_value.plot(figsize=(12,8));
```
Our strategy has lost us money over the year 2000. AAPL generally trended downward during the year, and simply buying every day is a losing strategy.

The following command runs the simulation over 5 years:

In [24]:
result = BuyApple().run(data['2000':'2004']).portfolio_value
result.plot(figsize=(12,8));

INFO: Performance: Simulated 1256 trading days out of 1256.
INFO: Performance: first open: 2000-01-03 14:31:00+00:00
INFO: Performance: last close: 2004-12-31 21:00:00+00:00

Hanging in with this strategy over several more years has paid off as AAPL had a marked upswing in value starting in mid-2013.
Algorithm – dual moving average crossover

We now analyze a dual moving average crossover strategy. This algorithm will buy apple once its short moving average crosses its long moving average. This will indicate upward momentum and a buy situation. It will then begin selling shares once the averages cross again, which will represent downward momentum.

We will load data for AAPL for 1990 through 2014, but we will only use the data from 1990 through 2001 in the simulation:

In [25]:
    
    sub_data = data['1990':'2002-01-01']
    sub_data.plot();

The following class implements a double moving average crossover where investments will be made whenever the short moving average moves across the long moving average. We will trade only at the cross, not continuously buying or selling until the next cross. If trending down, we will sell all of our stock. If trending up, we buy as many shares as possible up to 100. The strategy will record our buys and sells in extra data returned from the simulation:

In [26]:

---
class DualMovingAverage(zp.TradingAlgorithm):
    def initialize(context):
        # we need to track two moving averages, so we will set
        # these up in the context the .add_transform method
        # informs Zipline to execute a transform on every day
        # of trading

        # the following will set up a MovingAverage transform,
        # named short_mavg, accessing the .price field of the
        # data, and a length of 100 days
        context.add_transform(zp.transforms.MovingAverage,
                              'short_mavg', ['price'],
                              window_length=100)

        # and the following is a 400 day MovingAverage
        context.add_transform(zp.transforms.MovingAverage,
                              'long_mavg', ['price'],
                              window_length=400)

        # this is a flag we will use to track the state of
        # whether or not we have made our first trade when the
        # means cross. We use it to identify the single event
        # and to prevent further action until the next cross
        context.invested = False

    def handle_data(self, data):
        # access the results of the transforms
        short_mavg = data['AAPL'].short_mavg['price']
        long_mavg = data['AAPL'].long_mavg['price']

        # these flags will record if we decided to buy or sell
        buy = False
        sell = False

        # check if we have crossed
        if short_mavg > long_mavg and not self.invested:
            # short moved across the long, trending up
            # buy up to 100 shares
            self.order_target('AAPL', 100)
            # this will prevent further investment until
# the next cross
self.invested = True
buy = True  # records that we did a buy
elif short_mavg < long_mavg and self.invested:
    # short move across the long, trending down
    # sell it all!
    self.order_target('AAPL', -100)
    # prevents further sales until the next cross
    self.invested = False
    sell = True  # and note that we did sell

# add extra data to the results of the simulation to
# give the short and long ma on the interval, and if
# we decided to buy or sell
self.record(short_mavg=short_mavg,
            long_mavg=long_mavg,
            buy=buy,
            sell=sell)

We can now execute this algorithm by passing it data from 1990 through 2001, as shown here:

In [27]:
    results = DualMovingAverage().run(sub_data)

                     14:31:00+00:00
                     21:00:00+00:00

To analyze the results of the simulation, we can use the following function that creates several charts that show the short/long means relative to price, the value of the portfolio, and the points at which we made buys and sells:

In [28]:
    def analyze(data, perf):
        fig = plt.figure()
        ax1 = fig.add_subplot(211, ylabel='Price in $')
        data['AAPL'].plot(ax=ax1, color='r', lw=2.)
        perf[['short_mavg', 'long_mavg']].plot(ax=ax1, lw=2.)
        ax1.plot(perf.ix[perf.buy].index, perf.short_mavg[perf.buy],

        ax1.plot(perf.ix[perf.sell].index, perf.short_mavg[perf.sell],}
Using this function, we can plot the decisions made and the resulting portfolio value as trades are executed:

In [29]:
    analyze(sub_data, results)
Algorithmic Trading

The crossover points are noted on the graphs using triangles. Upward-pointing red triangles identify buys and downward-pointing black triangles identify sells. Portfolio value stays level after a sell as we are completely divested from the market until we make another purchase.

Algorithm – pairs trade

To demonstrate a pairs trade algorithm, we will create one such algorithm and run data for Pepsi and Coca-Cola through the simulation. Since these two stocks are in the same market segment, their prices tend to follow each other based on common influences in the market.

If there is an increase in the delta between the two stocks, a trader can potentially make money by buying the stock that stayed the same and selling the increasing stock. The assumption is that the two stocks will revert to a common spread on the mean. Hence, if the stock that stayed normal increases to close the gap, then the buy will result in increased value. If the rising stock reverts, then the sell will create profit. If both happen, then even better.

To start with, we will need to gather data for Coke and Pepsi:

```
In [30]:
    data = zpf.load_from_yahoo(stocks=['PEP', 'KO'],
                        indexes={},
                        start=datetime(1997, 1, 1),
                        end=datetime(1998, 6, 1),
                        adjusted=True)

    data.plot(figsize=(12,8));
    PEP
    KO
```
Analyzing the chart, we can see that the two stocks tend to follow along the same trend line, but that there is a point where Coke takes a drop relative to Pepsi (August 1997 through December 1997). It then tends to follow the same path although with a wider spread during 1998 than in early 1997.
We can dive deeper into this information to see what we can do with pairs trading. In this algorithm, we will examine how the spread between the two stocks change. Therefore, we need to calculate the spread:

```python
In [31]:
    data['PriceDelta'] = data.PEP - data.KO
    data['1997':].PriceDelta.plot(figsize=(12,8))
    plt.ylabel('Spread')
    plt.axhline(data.Spread.mean());
```

Using this information, we can make a decision to buy one stock and sell the other if the spread exceeds a particular size. In the algorithm we implement, we will normalize the spread data on a 100-day window and use that to calculate the z-score on each particular day.

If the z-score is > 2, then we will want to buy PEP and sell KO as the spread increases over our threshold with PEP taking the higher price. If the z-score is < -2, then we want to buy KO and sell PEP, as PEP takes the lower price as the spread increases. Additionally, if the absolute value of the z-score < 0.5, then we will sell off any holdings we have in either stock to limit our exposure as we consider the spread to be fairly stable and we can divest.
One calculation that we will need to perform during the simulation is calculating the regression of the two series prices. This will then be used to calculate the z-score of the spread at each interval. To do this, the following function is created:

```python
In [32]:
    @zp.transforms.batch_transform
def ols_transform(data, ticker1, ticker2):
p0 = data.price[ticker1]
p1 = sm.add_constant(data.price[ticker2], prepend=True)
slope, intercept = sm.OLS(p0, p1).fit().params
return slope, intercept
```

You may wonder what the `@zp.transforms.batch_transform` code does. At each iteration of the simulation, Zipline will only give us the data representing the current price. Passing the data from `handle_data` to this function would only pass the current day's data. This decorator will tell Zipline to pass all of the historical data instead of the current day's data. This makes this very simple as, otherwise, we would need to manage multiple windows of data manually in our code.

The actual algorithm is then implemented using a 100-day window where we will execute on the spread when the z-score is > 2.0 or < -2.0. If the absolute value of the z-score is < 0.5, then we will empty our position in the market to limit exposure:

```python
In [33]:
class Pairtrade(zp.TradingAlgorithm):
    def initialize(self, window_length=100):
        self.spreads=[]
        self.invested=False
        self.window_length=window_length
        self.ols_transform= \
            ols_transform(refresh_period=self.window_length,
                          window_length=self.window_length)

    def handle_data(self, data):
        # calculate the regression, will be None until 100 samples
        params=self.ols_transform.handle_data(data, 'PEP', 'KO')
        if params:
            intercept, slope=params
            zscore=self.compute_zscore(data, slope, intercept)
            self.record(zscore=zscore)
            self.place_orders(data, zscore)
```
def compute_zscore(self, data, slope, intercept):
    # calculate the spread
    spread=(data['PEP'].price-(slope*data['KO'].price+
    intercept))
    self.spreads.append(spread)  # record for z-score calc
    self.record(spread = spread)

    spread_wind=self.spreads[-self.window_length:]
    zscore=(spread - np.mean(spread_wind))/np.std(spread_wind)
    return zscore

def place_orders(self, data, zscore):
    if zscore>=2.0 and not self.invested:
        # buy the spread, buying PEP and selling KO
        self.order('PEP', int(100/data['PEP'].price))
        self.order('KO', -int(100/data['KO'].price))
        self.invested=True
        self.record(action="PK")
    elif zscore<-2.0 and not self.invested:
        # buy the spread, buying KO and selling PEP
        self.order('PEP', -int(100 / data['PEP'].price))
        self.order('KO', int(100 / data['KO'].price))
        self.invested = True
        self.record(action='KP')
    elif abs(zscore)<.5 and self.invested:
        # minimize exposure
        ko_amount=self.portfolio.positions['KO'].amount
        self.order('KO', -1*ko_amount)
        pep_amount=self.portfolio.positions['PEP'].amount
        self.order('PEP', -1*pep_amount)
        self.invested=False
        self.record(action='DE')
    else:
        # take no action
        self.record(action='noop')
Then, we can run the algorithm with the following command:

```
In [34]:
    perf = Pairtrade().run(data['1997':])
    [2015-02-16 01:54] INFO: Performance: first open: 1997-01-02 14:31:00+00:00
    [2015-02-16 01:54] INFO: Performance: last close: 1998-06-01 20:00:00+00:00
```

During the simulation of the algorithm, we recorded any transactions made, which can be accessed using the action column of the result DataFrame:

```
In [35]:
    selection = ((perf.action=='PK') | (perf.action=='KP') | (perf.action=='DE'))
    actions = perf[selection][['action']]
    actions
```

```
Out[35]:
1997-07-16 20:00:00    KP
1997-07-22 20:00:00    DE
1997-08-05 20:00:00    PK
1997-10-15 20:00:00    DE
1998-03-09 21:00:00    PK
1998-04-28 20:00:00    DE
```

Our algorithm made six transactions. We can examine these transactions by visualizing the prices, spreads, z-scores, and portfolio values relative to when we made transactions (represented by vertical lines):

```
In [36]:
    ax1 = plt.subplot(411)
    data[['PEP', 'KO']].plot(ax=ax1)
    plt.ylabel('Price')
    data.Spread.plot(ax=ax2)
    plt.ylabel('Spread')
    ax3 = plt.subplot(413)
    perf['1997':].zscore.plot()
```
ax3.axhline(2, color='k')
ax3.axhline(-2, color='k')
plt.ylabel('Z-score')

ax4 = plt.subplot(414)
perf['1997':].portfolio_value.plot()
plt.ylabel('Protfolio Value')

for ax in [ax1, ax2, ax3, ax4]:
    for d in actions.index[actions.action=='PK']:
        ax.axvline(d, color='g')
    for d in actions.index[actions.action=='KP']:
        ax.axvline(d, color='c')
    for d in actions.index[actions.action=='DE']:
        ax.axvline(d, color='r')

plt.gcf().set_size_inches(16, 12)
The first event is on 1997-07-16 when the algorithm saw the spread become less than -2, and, therefore, triggered a sale of KO and a buy of PEP. This quickly turned around and moved to a z-score of 0.19 on 1997-07-22, triggering a divesting of our position. During this time, even though we played the spread, we still lost because a reversion happened very quickly.

On 1997-08-05, the z-score moved above 2.0 to 2.12985 and triggered a purchase of KO and a sale of PEP. The z-score stayed around 2.0 until 1997-10-15 when it dropped to -0.1482 and, therefore, we divested. Between those two dates, since the spread stayed fairly consistent around 2.0, our playing of the spread made us consistent returns as we can see with the portfolio value increasing steadily over that period.

On 1998-03-09, a similar trend was identified, and again, we bought KO and sold PEP. Unfortunately the spread started to minimize and we lost a little during this period.

**Summary**

In this chapter, we took an adventure into learning the fundamentals of algorithmic trading using pandas and Zipline. We started with a little theory to set a framework for understanding how the algorithms would be implemented. From there, we implemented three different trading algorithms using Zipline and dived into the decisions made and their impact on the portfolios as the transactions were executed. Finally, we established a fundamental knowledge of how to simulate markets and make automated trading decisions.
Where to buy this book

You can buy Mastering pandas for Finance from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.