INSTANT

Short | Fast | Focused

Data Intensive Apps with pandas How-to

Manipulate, visualize, and analyze your data with pandas

Trent Hauck

[PACKT] PUBLISHING

Instant Data Intensive Apps with pandas How-to

Manipulate, visualize, and analyze your data with pandas

Trent Hauck

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant Data Intensive Apps with pandas How-to

First published: May 2013

Production Reference: 1170513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-558-3

`www.packtpub.com`

Credits

Author
Trent Hauck

Reviewer
Dan Mantyla

Acquisition Editor
Akram Hussain

Commissioning Editor
Ameya Sawant

Technical Editors
Dheera Meril Paul
Vrinda Nitesh Bhosale

Copy Editor
Alfida Paiva

Project Coordinator
Siddhant Shetty

Proofreaders
Maria Gould
Amy Guest

Graphics
Ronak Dhruv

Production Coordinator
Aditi Gajjar

Cover Work
Aditi Gajjar
Prachali Bhiwandker

Cover Image
Sheetal Aute

About the Author

Trent Hauck is a graduate from the University of Kansas. He holds a Bachelor's in Accounting and a Master's in Finance. Early in his career he worked in Finance and Insurance, but has transitioned to Marketing and Analytics. Working with data and finding tools for efficient use has been a theme throughout.

I'd like to thank my friends and family for their support and patience.

About the Reviewer

Dan Mantyla is a HPC Systems Programmer for Atipa Technologies and has been using Python since 2009. As a utility infielder of all things, software development and with a BSCS from the University of Kansas, Dan has done everything from contributing to open source web frameworks to designing advanced cluster management architectures in Python. He contributes to `HPCwire.com` and lives in a beautiful Lawrence, KS, home of Linux New Media and birthplace of Python Django.

Table of Contents

Preface

Python has long been considered one of the main "glue" languages for programmers and data analysts due to its easy-to-learn syntax and a vast array of libraries to extend functionality. In the last two or so years Python has become a language that isn't used for just the glue, but the analysis itself. A big reason for that is pandas. This book covers pandas and other libraries by example. The book is sectioned into recipes which start off with the basics, but through progressively useful (and sometimes difficult) recipes the reader will get a good feel for pandas.

What this book covers

Working with files covers basic DataFrame creation as well as working with files. Very rarely is data generated from within, so knowing how to work with files is important to get off the ground.

Slicing pandas objects explains how to slice the DataFrame. After following the first recipe a DataFrame can be created, but the next step is understanding how to slice the DataFrame. Thankfully, it isn't much different from a native Python array.

Subsetting data covers how to select just the data that is of interest, quite often during data analysis. Cohort analysis is a good consequence of being able to subset data.

Working with dates will cover the beginning of how to carry out date manipulation. Date math sucks. pandas makes it suck a little less.

Modifying data with functions teaches how to use functions to modify data. Not only is the analyst often given too much data, but also the data often needs some work and applying functions to that data can allow for easier use.

Combining datasets discusses how to take multiple datasets and combine them into one, which is very similar to using SQL to join datasets.

Using indexes to manipulate objects demonstrates the use of indexes for data manipulation. Index in pandas allows for easy manipulation of data. One way to think about Indexes is that if data is made up of metrics and dimensions, indexes are the metadata used to describe those metrics and dimensions.

Getting data from the Web explains how to get data from the Web. A hidden gem in pandas is its ability to get data from all over. This brief recipe will show off this feature.

Combining pandas with scikit-learn explores how to integrate pandas with scikit-learn, a library for Machine Learning. One of the great things about pandas is its ability to work with other libraries in the PyData ecosystem.

Integrating pandas with statistics packages will discuss integration of pandas, using StatsModels a module for classical statistics and econometric analysis.

Using Flask for the backend uses Flask. It's a micro framework that allows for quickly building a web backend.

Visualizing pandas objects will discuss graphing and charting with pandas. Often large amounts of data can't be understood just by looking at the raw numbers.

Reporting with pandas objects puts it all together. Using Flask as the backend, the reader will build a basic reporting app to display a subset of a DataFrame.

What you need for this book

Obviously pandas is required, but in addition to pandas the following Python packages are required:

- Numpy
- SciKit Learn
- StatsModels
- Flask
- matplotlib
- IPython (not technically required, but highly recommended)
- Dependencies aren't mentioned; therefore, it is a good idea to use pip to install these packages which will alert unmet dependencies.

Who this book is for

This book is for the people who interact with tabular datasets and who like Python, and would like for Python to be a bigger part of how they interact with data. The readers should be comfortable with Python, but no pandas experience is required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create an incredibly simple `DataFrame` to start."

A block of code is set as follows:

```
> import pandas as pd #standard convention throughout the book
> import numpy as np
> my_df = pd.DataFrame([1,2,3])
> my_df
   0
0  1
1  2
2  3
```

Any command-line input or output is written as follows:

```
>import matplotlib.pyplot as plt
>plt.plot(close)
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Data-intensive Apps with pandas How-to

Welcome to *Instant Data Intensive Apps with pandas How-to*. This book is a collection of recipes with the intention of transforming the reader from being a novice with just a simple idea of what pandas is, to a highly efficient user of the library. This book is not meant to be a comprehensive look through all the nooks of pandas, but is meant to get the reader up speed in the most common tasks.

Working with files (Simple)

In this recipe we'll introduce the pandas **DataFrame** by doing some quick exercises, then move onto one of the most fundamental parts of data analysis; getting data in and out of files.

Getting ready

Most of the rest of the book is working with data once it's in a pandas data structure, but this recipe is about those structures themselves and getting data in and out of them. Open your interpreter, preferably IPython.

How to do it...

1. Create an incredibly simple DataFrame to start with. A DataFrame can handle lists, NumPy arrays, dicts of strings, and more.

```
> import pandas as pd
 #standard convention throughout the book
> import numpy as np
> my_df = pd.DataFrame([1,2,3])
> my_df
   0
0  1
1  2
2  3
```

2. The first example is too simple, and isn't useful. Add some column headers and index for more information about the DataFrame.

```
> cols = ['A', 'B']
> idx = pd.Index(list('name'), name='a')
> data = np.random.normal(10, 1, (4, 2))
> df = pd.DataFrame(data, columns=cols, index=idx)
df
           A          B
a
n   9.945858  10.607128
a  10.742073   8.968044
m  10.178861   7.293450
e  10.251922  10.657038

#a single column is a series
> df.A
a
n     9.945858
a    10.742073
m    10.178861
e    10.251922
Create a Panel by passing a dictionary of DataFrames to the
constructor.
# multiple DataFrames is a panel
> pan = pd.Panel({'df1': df, 'df2': df})
> pan
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: df1 to df2
Major_axis axis: n to e
Minor_axis axis: A to B
```

3. There are many ways to do I/O with pandas; in this step we will write the DataFrame out to several mediums.

```
#write df to different file types
> df.to_csv('df.csv')
> df.to_latex('df.tex') #useful with Pweave
> df.to_excel('df.xlsx') #requires extra packages
> df.to_html('df.html')
> df.to_string()
'           A          B\na                      \na   9.945858
10.607128\nb  10.742073   8.968044\nc  10.178861   7.293450\nd
10.251922  10.657038'


#read df from the files, output methods aren't symmetric
#often there's an intermediate step


> pd.read_csv('df.csv')


#back and forth with json
#json isn't officially supported, the reasons why are beyond #the
scope of this book
> with open('df.json', 'w') as f:
  json.dump(df.to_dict(), f)


> with open('df.json') as f:
  df_json = json.load(f)
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

How it works...

Most of the file input and output in pandas is the orchestration behind the scenes of formatting the value outputs, and then writing those values to a file. There are many options for formatting file output. The `to_csv` method takes many parameters. Some of the more common parameters are as follows:

- `sep`: It specifies the value to separate with, in the output file
- `index`: It is a Boolean that decides whether or not to print the index
- `na_rep`: It specifies what to substitute for the `na` values

The following snippet writes the DataFrame `df` and writes it to a file called `file.tsv`, and it's formatted according to the parameters passed to the method.

```
> df.to_csv('file.tsv', sep='\t', index=False, na_rep='NULL')
```

There's more...

In addition to standard file input and output functionalities, pandas has several built-in niceties.

Parsing dates at file read time

Using Panda's sophisticated date parser, a CSV can read and parse dates at the same time, as shown in the following command line:

```
> df = pd.read_csv('dates.csv', date_parser=True, parse_dates='YYYY-MM-
DD', index_col=0)
```

Besides the parsing capabilities, pandas also has a very handy `date_range` function, which returns a range of dates determined by the inputs. For example, it's very easy to get the months of 2012 in a series. This is shown in the following command line:

```
> pd.date_range('2012-01-01', '2012-12-31', freq='M')
```

Accessing data from a public source

pandas can also read CSV data from the Web, assuming `http://www.example.com/data.csv` is the URL. Take a look at the following example:

```
> df = pd.read_csv(url)
```

Slicing pandas objects (Simple)

In this recipe we'll walk through some basic functionalities about slicing pandas objects. If you're familiar with array slicing, this will be very familiar to you, but with a few idiosyncrasies for pandas.

Getting ready

Open up your interpreter, and execute the following steps:

How to do it...

1. Create a simple DataFrame to explore the different slicing abilities of pandas.

   ```
   > dim = (10, 3)

   > df = pd.DataFrame(np.random.normal(0, 1, dim), columns ['one',
   'two', 'three'])
   ```

2. Select the first two rows of the column named `'one'`.

   ```
   > df['one'][:2]

   0   -0.492156

   1   -0.476418

   Name: one
   ```

3. Pass an array of column names instead of `'one'`.

   ```
   > df[['one', 'two']][:2]

           one        two

   0 -0.492156  1.978798

   -0.476418 -0.225360
   ```

4. Use a negative index to navigate backwards through the DataFrame.

   ```
   > df[['one', 'two']][-3:-2]

          one        two

   7 -0.392156  1.478198
   ```

5. Select every fifth row from the DataFrame `df`.

   ```
   > df[::5]

         one        two      three

   0  0.317379 -0.551568 -1.617768

   5 -0.171340  2.025818  0.206053
   ```

6. Use the `head` and `tail` functions to easily select the top and bottom of the DataFrame.

   ```
   > df.head(2)

          one        two

   0 -0.492156  1.978798

   1 -0.476418 -0.225360
   ```

How it works...

At some level, pandas objects behave similar to NumPy arrays; they are after all abstractions built on top of them. However, because we have more metadata about the data structures we can use that to our advantage.

After the initial pandas object is created, simple slicing occurs according to the following structure:

```
> df[column names][rows]
```

Here `column names` is a string (or an array, if multiple columns) and `rows` is the number of rows that we wish to use.

There's more...

The methods that have already been described are very useful at a higher level, but there are more granular operations available.

Direct index access

The `.ix` command is an advanced method for selecting and slicing a DataFrame. Taking the sample from the preceding example, `df.ix[1:3 ,['one', 'two']] = 10` will not only select the specified subset of the data, but also set its value equal to `10`. The `.xs` command has a more explicit interface for working with indexes.

Resetting the index

Often, the index of the DataFrame becomes out of alignment when slicing data. In pandas, the easiest way to reset an index is with the `reset_index()` method of the DataFrame object.

Subsetting data (Simple)

In this recipe we'll select parts of a DataFrame based on elements within the DataFrame. For example, select data only if it's greater than zero.

Getting ready

Open up your interpreter and follow along with the interpreter's session in the following *How to do it...* section.

How to do it...

1. Create a sample DataFrame `df` to manipulate.

```
> d = {'Cost': np.random.normal(100, 5, 100),
     'Profit': np.random.normal(50, 5, 100),
     'CatA': np.random.choice(['a', 'b', 'c'], 100),
     'CatB': np.random.choice(['e', 'f', 'g'], 100)}

> df = pd.DataFrame(d)
```

2. Pass a Boolean value to the DataFrame to select only those rows that evaluate to `True`.

```
> df[df.CatA == 'a'][:5]
   CatA CatB        Cost     Profit
0     a    g  102.716045  48.585048
1     a    g  103.342873  44.285223
3     a    f  100.563783  52.609880
4     a    f   92.057118  53.030021
10    a    f   97.494169  44.849129

> mask = np.logical_and(df.CatA=='a', df.CatB=='e')
> df[mask][:5]
CatA CatB        Cost     Profit
15    a    e  103.532647  44.625927
20    a    e  101.780083  48.415357
57    a    e   95.795958  46.033187
76    a    e   96.831145  53.876999
77    a    e  100.991794  44.308626

> a_e = ['a', 'e']
> CatA_a_e = df[df.CatA.isin(a_e)]
> only_a_e = CatA_a_e[CatA_a_e.CatB.isin(a_e)]
> only_a_e[:5]
      CatA CatB        Cost     Profit
15    a    e  103.532647  44.625927
20    a    e  101.780083  48.415357
57    a    e   95.795958  46.033187
76    a    e   96.831145  53.876999
77    a    e  100.991794  44.308626
```

How it works...

Subsetting is an integral part of data analysis, and is very simple to do in pandas. A nice pattern for creating subsets is to create mask arrays, which are arrays of Booleans, and then passing those into the DataFrame. For example, executing `df[df.CatA.isin(a_e)]` in the interpreter will return an array of the same length as `df.CatA`.

There's more...

There are more powerful pandas functions that make selecting subsets of DataFrames more concise.

The where and mask commands

In addition to masking data based on arrays, pandas contains many helper functions to subset and assign values to data. The two main ones are `df.where()` and `df.mask()`, which are complements. The `df.CatA.where(df.CatA == 'a')` function doesn't automatically remove data like in the previous examples; it creates a copy of `df.CatA` with `NaN` in places where `df.CatA` doesn't equals to `'a'`.

Substituting with the where command

Taking the `where` command one step further, it's easy to replace the `NaN` element in one step. For example, `df.CatA.where(df.CatA == 'a', 'e')` will substitute `NaN` where `df.CatA` is equal to `'a'`; however, because there is the second argument, the `NaN` will be replaced by `'e'`.

Working with dates (Medium)

In this recipe we'll talk about working with dates in pandas. Because pandas was initially written with financial time series, it has a lot of out of the box date functionalities.

Getting ready

Open up your interpreter and follow the command progression in the following section. Difficult financial analysis was the mother of pandas creation; therefore, it has many efficient and easy ways for dealing with dates.

How to do it...

1. Let's examine the `date_range` functionality within pandas.

```
> Y2K = pd.date_range('2000-01-01', '2000-12-31')
> Y2K
```

```
class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-12-31 00:00:00]
Length: 366, Freq: D, Timezone: None

#it is very easy to create date range of a different frequency
> Y2K_hourly = pd.date_range('2000-01-01', '2000-12-31', freq='H')
> Y2K_hourly
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-12-31 00:00:00]
Length: 8761, Freq: H, Timezone: None
```

2. Create a time series and slice it by passing a range of dates to Series.

```
> Y2K_temp = pd.Series(np.random.normal(75, 10, len(Y2K)),
index=Y2K)
> Y2K_temp.head()
2000-01-01    77.425233
2000-01-02    67.949946
2000-01-03    74.079854
2000-01-04    83.048726
2000-01-05    88.435598
Freq: D

> Y2K_temp['2000-01-01':'2000-01-02']
2000-01-01    77.425233
2000-01-02    67.949946
Freq: D

> from datetime import date
> Y2K_temp[date(2000, 1, 1):date(2000, 1, 2)]
2000-01-01    77.425233
2000-01-02    67.949946
Freq: D

#pandas has functionality to move into and out-of date scopes
> Y2K_temp.resample('H', fill_method='pad')[:1]
2000-01-01 01:00:00    77.425233
Freq: H
```

How it works...

The `date_range` function is defined by dates and frequencies. See the following section for the various frequency designations. The easiest way is to define a start date, end date, and frequency, but there are other ways as well. You can also change the frequency, or resample to a smaller or larger time interval.

There's more...

pandas adds a lot more functionalities to handle dates. These are mostly convenient methods because working with dates is a necessary evil of data analysis.

Alternative date range specification

Time series in pandas don't have to be defined by a start and end date. In pandas, it is possible to represent the time of the Series as an interval of dates with a common period between data points. For example, if we want to create a Series just like Y2K, we can do so as follows:

```
pd.date_range(start='2012-01-01', periods=366, freq='D')
```

Upsampling and downsampling Series

pandas offers the ability to move up and down the granularity of a time series. For example, given a Series of random numbers `s` for all the days in 2012, calculating the sum for each month is done by the following formula:

```
s.resample('M', how='sum')
```

In the preceding example, the `'M'` variable specifies that we're upsampling to month. Downsampling is also done in a similar way; however, pandas provides functionalities for handling the disaggregation in a convenient way.

Modifying data with functions (Simple)

In this recipe we'll walk through the process of applying a function to a DataFrame. This is a simple but very important part of data analysis. Rarely, if ever, will a data in raw form be sufficient for data analysis. Often, that data needs to be transformed into some other form, and to do that you'll need to apply functions to pandas objects.

Getting ready

Open up your interpreter, and type the following commands successively.

How to do it...

1. Create a simple `Series` of simulated `open` and `close` for a year.

```
> data = {'Open': np.random.normal(100, 5, 366),
          'Close': np.random.normal(100, 5, 366)}

> df = pd.DataFrame(data)

> df
<class 'pandas.core.frame.DataFrame'>
Int64Index: 366 entries, 0 to 365
Data columns:
Close    366  non-null values
Open     366  non-null values
dtypes: float64(2)
```

2. Apply element-wise functions.

```
> df.apply(np.mean, axis=1).head(3)
0    103.313391
1     99.668034
2     97.875755

#passing a lambda is a common pattern
> df.apply(lambda x: (x['Open'] - x['Close']), axis=1).head(3)
0     6.519618
1     8.407379
2    16.838463

#define a more complex function
> def percent_change(x):
    return (x['Open'] - x['Close']) / x['Open']
> df.apply(percent_change, axis=1).head(3)
0    0.061175
1    0.080940
2    0.158413

#change axis, axis = 0 is default
```

```
> df.apply(np.mean, axis=0)
Close    99.739967
Open     99.631989
```

3. Define a standalone function that takes two arguments. One is the element itself, and another argument.

```
> def greater_than_x(element, x):
    return element > x

> df.Open.apply(greater_than_x, args=(100,)).head(3)
0     False
1     True
2     False
Name: Open

#This can be used as in conjunction with subset capabilities
> mask = df.Open.apply(greater_than_x, args=(100,))

> df.Open[mask].head()
1    100.713672
3    105.492173
4    100.171148
6    104.810547
7    110.539181
Name: Open

#It's also possible to do a rolling apply, this applys #aggregate
functions over a certain number of rows
#For instance we can get the five day moving average
> pd.rolling_apply(df.Close, 5, np.mean)

#There are actually a several built-in rolling functions
> pd.rolling_corr(df.Close, df.Open, 5)[:5]
0         NaN
1         NaN
2         NaN
3       NaN
4    0.137234  #why are the first 4 NaN
```

How it works...

pandas sits on top of NumPy; thus pandas takes advantage of the broadcasting capabilities inherent within NumPy. For example, execute the following script to see the differences in NumPy:

```
> a = [1, 2, 3]
> a * 2
[1, 2, 3, 1, 2, 3]
> b = np.array(a)
> b * 2
[2, 4, 6]
```

Understanding the underlying NumPy structure is beyond the scope, but is extremely helpful in the long run.

There's more...

pandas makes additional use of the `apply` function in place of the `for loop` function. Quite often it's necessary to do more complex operations on an entire column(s) of a DataFrame, but broadcasting or looping won't cut it.

Other apply options

There are other `apply` functions in the family. For example, the `applymap` function operates in a slightly different manner than the `apply` function. The `applymap` function operates on a single value and returns a single value, whereas the `apply` function takes an array-like data structure as an input.

Alternative solutions

Functions can also be applied iteratively; however, this tends to make the functions slow and leads to unnecessarily verbose code.

```
> for x in df.Open:
  some_function(x)
vs.
> df.Open.apply(some_function)
```

Combining datasets (Medium)

Given that we have several different types of DataFrames, how can we best join them into one DataFrame for additional use? We'll also talk about merging and appending them in the following *There's more...* section.

Getting ready

Open up your interpreter and type the following given commands successively. Very rarely will an analyst receive data in a single flat file. Quite often, data will need to be either appended to the bottom of the DataFrame or attached to the side. For example, if a set of data comes directly from a normalized database, the analyst will need to combine them by joining them using Primary and Foreign Keys.

How to do it...

1. Create two basic DataFrames `df1` and `df2`.

```
> rng = pd.date_range('2000-01-01', '2000-01-05')

> tickers = pd.DataFrame(['MSFT', 'AAPL'], columns= ['Ticker'])

> df1 = pd.DataFrame({'TickerID': [0]*5,
  'Price': np.random.normal(100, 10, 5)}, index=rng)

> df2 = pd.DataFrame({'TickerID': [1]*5,
  'Price': np.random.normal(100, 10, 5)}, index=rng)
```

2. The `concat` method is similar to the `union` command in SQL.

```
#select the first and last value from concat
> pd.concat([df1, df2]).ix[[0, -1]]
                Price  TickerID
2000-01-01  76.937336         0
2000-01-05  89.070339         1
```

3. Merge the two DataFrames into a single DataFrame.

```
> pd.merge(df1, df2, left_index=True, right_index=True)
               Price_x  TickerID_x     Price_y  TickerID_y
2000-01-01   76.937336           0   79.707124           1
```

```
2000-01-02  127.788659          0  107.694863          1
2000-01-03   99.990745          0  106.456367          1
2000-01-04  112.125489          0   81.928378          1
2000-01-05  101.941148          0   89.070339          1

> pd.merge(df1, tickers, right_index=True, left_on='TickerID')
       Price      TickerID   Ticker
2000-01-01   76.937336         1   MSFT
2000-01-02  127.788659         1   MSFT
2000-01-03   99.990745         1   MSFT
2000-01-04  112.125489         1   MSFT
2000-01-05  101.941148         1   MSFT
```

How it works...

If the reader is familiar with R's functionalities, then he/she can see that joining data in pandas is not much different than in R. We'll cover more on indexes later, but thinking of the default index as a Primary Key, or the combination of hierarchical index as a Composite Key, elucidates the joining process.

There's more...

There are many options that can be supplied to the `merge` and `join` methods to modify the DataFrames' behaviour.

Merge and join details

The `merge` (and `join`) method uses a `how` parameter, which is a string of the `join` database. The possible values are `'left'`, `'right'`, `'outer'`, and `'inner'`.

Specifying outputs in join

The `join` function (not the previously mentioned one) is easy to use to join DataFrames.

```
> f1.join(df2, lsuffix=".1", rsuffix=".2")
```

Use suffixes to disambiguate columns if the DataFrames have similar column names. `join` defaults to joining of indexes, but the `on` parameter can be used to specify a column. For example:

```
a_df.join(another_df, on='columnA')
```

Concatenation

One way to join the datasets is to just stack them on top of each other. This is similar to a `union` command in SQL. Given two DataFrames, `One` and `Two`, a concatenation is done in the following way:

```
pd.concat([One, Two])
```

A list can also be used. Although it will be awkward for two DataFrames, it makes much more sense in the event of 50 DataFrames.

Using indexes to manipulate objects (Medium)

Indexes are not advanced because they're difficult, but if we want to be an expert with pandas it is important that we use them well. We will discuss hierarchical indexes in the following *There's more...* section.

Getting ready

A good understanding of indexes in pandas is crucial to quickly move the data around. From a business intelligence perspective, they create a distinction similar to that of metrics and dimensions in an OLAP cube. To illustrate this point, this recipe walks through getting stock data out of pandas, combining it, then reindexing it for easy chomping.

How to do it...

1. Use the `DataReader` object to transfer stock price information into a DataFrame and to explore the basic axis of Panel.

```
> from pandas.i git push -u origin master
 o.data import DataReader
> tickers = ['gs', 'ibm', 'f', 'ba', 'axp']
> dfs = {}
> for ticker in tickers:
        dfs[ticker] = DataReader(ticker, "yahoo", '2006-01-01')

# a yet undiscussed data structure, in the same way the a
# DataFrame is a collection of Series, a Panel is a collection of
# DataFrames
> pan = pd.Panel(dfs)
> pan
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 5 (items) x 1764 (major_axis) x 6 (minor_axis)
Items axis: axp to ibm
Major_axis axis: 2006-01-03 00:00:00 to 2013-01-04 00:00:00
Minor_axis axis: Open to Adj Close


> pan.items
Index([axp, ba, f, gs, ibm], dtype=object)


> pan.minor_axis
Index([Open, High, Low, Close, Volume, Adj Close], dtype=object)


> pan.major_axis
<class 'pandas.tseries.index.DatetimeIndex'>
[2006-01-03 00:00:00, ..., 2013-01-04 00:00:00]
Length: 1764, Freq: None, Timezone: None
```

2. Use the axis selectors to easily compute different sets of summary statistics.

```
> pan.minor_xs('Open').mean()
axp     46.227466
ba      70.746451
f        9.135794
gs     151.655091
ibm    129.570969


# major axis is sliceable as well
> day_slice = pan.major_axis[1]
> pan.major_xs(day_slice)[['gs', 'ba']]
                  ba          gs
Open           70.08      127.35
High           71.27      128.91
Low            69.86      126.38
Close          71.17      127.09
Volume    3165000.00  4861600.00
Adj Close      60.43      118.12
Convert the Panel to a DataFrame.
> dfs = []
> for df in pan:
    idx = pan.major_axis
    idx = pd.MultiIndex.from_tuples(zip([df]*len(idx), idx))
    idx.names = ['ticker', 'timestamp']
    dfs.append(pd.DataFrame(pan[df].values, index=idx,
columns=pan.minor_axis))


> df = pd.concat(dfs)
> df
Data columns:
Open         8820  non-null values
High         8820  non-null values
Low          8820  non-null values
Close        8820  non-null values
Volume       8820  non-null values
Adj Close    8820  non-null values
dtypes: float64(6)
```

3. Perform the analogous operations as in the preceding examples on the newly created DataFrame.

```
# selecting from a MultiIndex isn't much different than the Panel
# (output muted)
> df.ix['gs':'ibm']
> df['Open']
```

How it works...

The previous example was certainly contrived, but when indexing and statistical techniques are incorporated, the power of pandas begins to come through. Statistics will be covered in an upcoming recipe.

pandas' indexes by themselves can be thought of as descriptors of a certain point in the DataFrame. When `ticker` and `timestamp` are the only indexes in a DataFrame, then the point is individualized by the `ticker`, `timestamp`, and `column name`. After the point is individualized, it's more convenient for aggregation and analysis.

There's more...

Indexes show up all over the place in pandas so it's worthwhile to see some other use cases as well.

Advanced header indexes

Hierarchical indexing isn't limited to rows. Headers can also be represented by `MultiIndex`, as shown in the following command line:

```
> header_top = ['Price', 'Price', 'Price', 'Price', 'Volume', 'Price']
> df.columns = pd.MultiIndex.from_tuples(zip(header_top, df.columns)
```

Performing aggregate operations with indexes

As a prelude to the following sections, we'll do a single `groupby` function here since they work with indexes so well.

```
> df.groupby(level=['tickers', 'day'])['Volume'].mean()
```

This answers the question for each `ticker` and for each `day` (not date), that is, what was the mean volume over the life of the data.

Getting data from the Web (Simple)

In this recipe we'll talk about getting data from other random places. Part of pandas' great functionality is that it can get data from several sources.

Getting ready

The `read_csv` method will get more attention in this recipe. It can read in data from a URL if properly formatted, such as being formatted like a CSV file. Also there is an example of using a built-in function to pull stock data.

How to do it...

1. Locate a URL with an accessible dataset.

   ```
   > url = 'http://s3.amazonaws.com/trenthauck-public/book_data.csv'
   ```

2. Pass this `url` to the `read_csv` method.

   ```
   #use this df to practice splitting, there's a mix of categorical
   #and numeric data
   > df = pd.read_csv(url, df = pd.read_csv)
   ```

How it works...

pandas has a lot of ways to get general data sets. Within the I/O package, there are ways to get data from Google Analytics, the World Bank, and single stock OHLC data. Because pandas can contain many tabular data sets, it's very easy to use it as an ETL tool, and it probably doesn't get as much exposure in that area as it deserves.

There's more...

This book will begin to change now. Most of the effort required to move data around should now be understood. Those skills are only tools now. That's the beauty of pandas; a lot of the functionalities for slicing all tabular data into less than 20 pages. That's not to say what's been presented is comprehensive, but these simple techniques are a large portion of general pandas usage.

The Next stage

Going forward, we will focus on answering questions about a larger set of stock data, foundational questions such as the variance of closing prices to the more complex questions such as if the stock price of IBM and of MSFT are similar.

```
> from pandas.io.data import DataReader
> gs = DataReader('gs', 'yahoo', '2006-01-01')
> gs.describe()[['Open', 'High']] #the simplest groupby
         Open         High
count  1764.000000  1764.000000
mean    151.655091   153.851729
std      38.977143    39.109340
min      54.000000    54.540000
25%     120.492500   122.955000
50%     152.695000   154.535000
75%     176.562500   178.805000
max     243.550000   250.700000
```

The previous command grouped the stock price by each column then described it with aggregate statistics. It is also easy to apply a particular function as shown in the following example:

```
> gs['year'] = gs.index.year
#will be described later in detail
#muted
> gs.groupby(['year'])['Open'].apply(lambda x: np.mean(x))
```

Combining pandas with scikit-learn (Advanced)

In this recipe we'll walk through an example of using pandas with scikit-learn, which is a Python library for machine learning.

Getting ready

pandas can do so much heavy lifting—in terms of data movement and basic data analysis and in terms of elementary statistics and data munging—that you may not need additional tools. But eventually, to do sophisticated analysis we integrate pandas with scikit-learn (discussed in this recipe) and StatsModels (discussed in the next recipe).

In addition to the previous stats packages, this recipe will introduce simple plotting functions.

In order to carry out the plotting, for the rest of this book, you must have `matplotlib` installed.

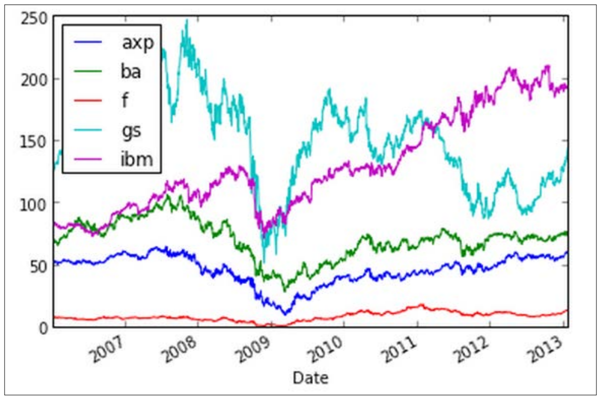
How to do it...

1. Create a DataFrame named `close` from the `pan` DataFrame mentioned in the *Using indexes to manipulate objects* recipe.

   ```
   > close = pan.minor_xs('Close')
   ```

2. The idea of our analysis (and a very contrived one) is that maybe we can predict if Ford went up on a given day in comparison with the other stocks in our data set that went up that day. First let's take a look at the data.

   ```
   > close.plot()
   ```



```
#From the graph it appears that there may be a relationship in
#daily price movements
```

3. Create a matrix for the features determined by the performance.

```
> diff = (close - close.shift(1))
> diff = diff[diff < 0].fillna(0)
> diff = diff[diff >= 0].fillna(1)
> diff.head()
        axp  ba  f  gs  ibm
Date
2006-01-03    0   0  0   0    0
2006-01-04    1   0  0   1    1
2006-01-05    0   1  0   1    0
2006-01-06    0   1  0   0    0
2006-01-09    0   1  0   0    1
```

4. After the data is in a usable form, carry out the rest of the SVM analysis.

```
> from sklearn import svm
> x = diff[['axp', 'ba', 'ibm', 'gs']]
> y = diff['f']
> obj = svm.SVC()
> ft = obj.fit(x, y)
> ft
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0,
  kernel='rbf', probability=False, shrinking=True, tol=0.001,
  verbose=False)
#so given the fit, we can then look at predictions
#all stocks up
> ft.predict([1,1,1,1])
array([ 1.])
#all stocks down
> ft.predict([0,0,0,0])
array([ 0.])
```

How it works...

This is by no means a book on machine learning; that said, let's have a quick discussion on SVM, the machine learning technique that we have used and also why pandas integrates seamlessly with it in most cases.

Support Vector Machine (**SVM**) is a machine-learning technique used to classify outputs based on features. In the preceding example, we created a `diff` matrix that holds the features, and in this case stocks such as AXP, BA, IBM, and GS, as well as an output variable `F`. From a mathematical perspective, SVM constructs a (hyper) plane between the outputs of zero and outputs of one. If the inputs we chose with `ft.predict([1,1,1,1])` fall on the side of the hyper plane where Ford goes up, the SVM predicts `1`, if it doesn't, it predicts `0`.

pandas integrates very well with scikit-learn; in the same way it integrates well with other packages associated with PyData. They all rely on the fast arrays implemented by NumPy. If you inspect a pandas object, you'll see that it essentially is a NumPy array with a masterfully crafted buffer between the low-level aspects of NumPy and the need for easy data use that pandas gives.

There's more

pandas is only one part of the scientific Python ecosystem. pandas is actually fairly new on the scene if you consider the fact that NumPy has been in the current form since 2005.

The NumPy object

It's easy to turn a pandas object into a NumPy object. The simplest way is to call the `values` method.

```
> df = pd.DataFrame(range(10), columns=['ColA'])
> type(df.values)
numpy.ndarray
```

Other tools

This is really only the start of what one can do with scikit-learn. There are many other tools available including basic regression, neural nets, and Gaussian processes. The main takeaway from this recipe should be that since all these tools are built on the same NumPy ecosystem, it is not difficult to use them together. We'll see another example of that in the next recipe.

Integrating pandas with statistics packages (Advanced)

In this recipe, we'll discuss about integrating pandas with StatsModels, a package for classical statistics in Python.

Getting ready

In this recipe, StatsModels will be used to perform a simple regression analysis. There is some overlap between the functionalities present within StatsModels and pandas; there is even some between StatsModels and the package described earlier, scikit-learn.

StatsModels is built with econometrics in mind. So, a lot of regression types are available, but in this recipe we'll do ordinary least squares, which is a very basic linear model. This analysis will use the stock closing data from the non-Ford stocks and measure its linear relationship with Ford.

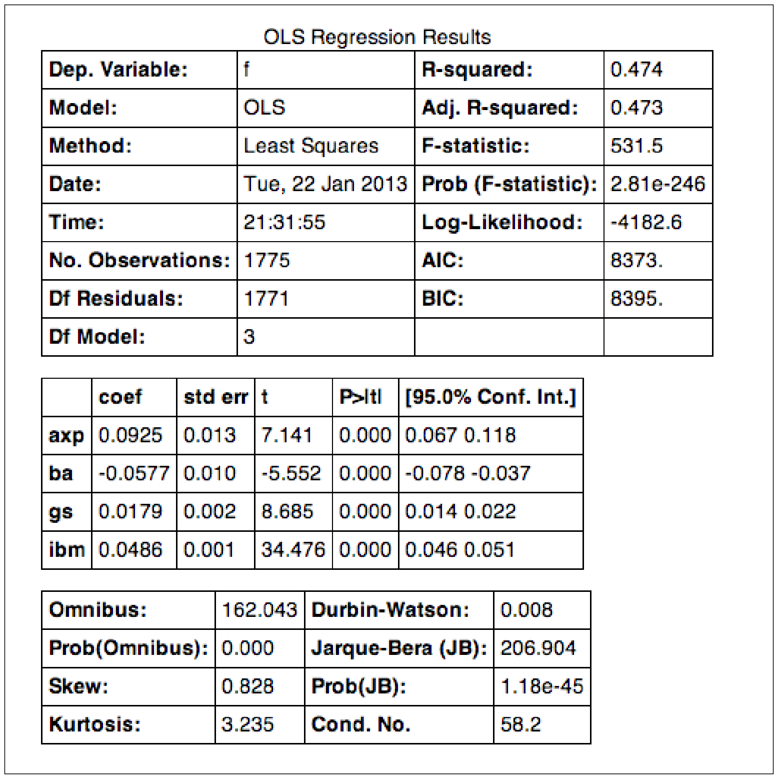
How to do it...

1. The import process for StatsModels is structured a bit differently from most other packages. However, the convention for importing it as a two-letter name is similar.

   ```
   > import statsmodels.api as sm
   ```

2. Define the `x` (exogenous) and `y` (endogenous) variables. The exogenous variable is really a vector that contains the regressors for the endogenous variable.

   ```
   > x = close[['axp', 'ba', 'gs', 'ibm']]
   > y = close['f']

   > ols_model = sm.OLS(y,x)
   > fit = ols_model.fit()

   #looking at the summary in IPython Notebook
   ```



OLS Regression Results

Dep. Variable:	f	**R-squared:**	0.474
Model:	OLS	**Adj. R-squared:**	0.473
Method:	Least Squares	**F-statistic:**	531.5
Date:	Tue, 22 Jan 2013	**Prob (F-statistic):**	2.81e-246
Time:	21:31:55	**Log-Likelihood:**	-4182.6
No. Observations:	1775	**AIC:**	8373.
Df Residuals:	1771	**BIC:**	8395.
Df Model:	3		

	coef	**std err**	**t**	**P>|t|**	**[95.0% Conf. Int.]**
axp	0.0925	0.013	7.141	0.000	0.067 0.118
ba	-0.0577	0.010	-5.552	0.000	-0.078 -0.037
gs	0.0179	0.002	8.685	0.000	0.014 0.022
ibm	0.0486	0.001	34.476	0.000	0.046 0.051

Omnibus:	162.043	**Durbin-Watson:**	0.008
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	206.904
Skew:	0.828	**Prob(JB):**	1.18e-45
Kurtosis:	3.235	**Cond. No.**	58.2

3. Select the different t-values from the regression.

```
> fit.t()
```

```
array([  7.14134063,  -5.55239471,   8.68521999,
34.47609677])
```

4. Get a printout of the confidence intervals of the betas.

```
> fit.conf_int()
```

```
           0         1
axp   0.067076  0.117870
ba   -0.078078 -0.037316
gs    0.013854  0.021937
ibm   0.045798  0.051323
```

There's more

As you play more with pandas, StatsModels is a great place to go and get data sets to practice with.

```
> longly = sm.datasets.longley.load_pandas()
#copper is now an object that has the data (among other things)
> longly.endog.head()
0    60323
1    61122
2    60171
3    61187
4    63221
Name: TOTEMP
> longly.exog.head()
GNPDEFL     GNP  UNEMP  ARMED     POP  YEAR
0     83.0  234289   2356   1590  107608  1947
1     88.5  259426   2325   1456  108632  1948
2     88.2  258054   3682   1616  109773  1949
3     89.5  284599   3351   1650  110929  1950
4     96.2  328975   2099   3099  112075  1951
```

Using Flask for the backend (Advanced)

In this recipe we'll spin up an easy backend with Flask, a micro framework that is great for quickly creating a simple web application. It's also very extensible so it can handle larger projects.

Getting ready

The next step for building a data application is to convey the analysis. Displaying it on the Web is the easiest way to pass the message to others. Thankfully, for passing that message, Flask can be used as it is a very simple way of displaying a web page with which pandas can be integrated.

Please note that Flask isn't fully ported to Python 3, so it is advised to use Python 2.7.

The structure of this recipe is a bit different from the others in the way that you won't interact directly with the REPL. Each comment will be the filename, then the subsequent code will be the contents of that file.

How to do it...

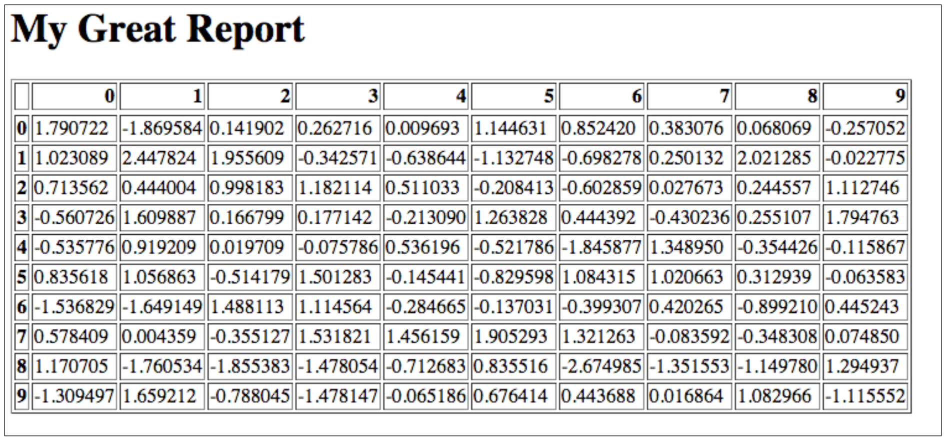
1. Create a file called `app.py` and populate it with the following text. If you do not have Flask installed, run the `pip install flask` setup.

```
#app.py
from flask import Flask, render_template

import pandas as pd
import numpy as np

app = Flask(__name__)

@app.route('/report')
def report():
    df = pd.DataFrame(np.random.randn(10,10))
    return render_template('report.html', df=df.to_html())

if __name__ == '__main__':
        app.run()
```

2. Create a folder named `templates` in the same directory where `app.py` was created. And in that folder, create a file called `report.html`. In the `report.html` file enter the following text:

```
#templates/report.html
<!DOCTYPE html>
<html>
    <body>
        <h1>My Great Report</h1>
        #This is the only fancy part… we're clearing the html
        #from the to_html() method
        {{ df|safe }}
    </body>
</html>
```

3. Run `python app.py`, then go to `localhost:5000/report` in the browser. It will then look like the following image:



My Great Report

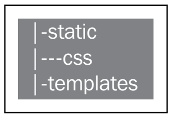
	0	1	2	3	4	5	6	7	8	9
0	1.790722	-1.869584	0.141902	0.262716	0.009693	1.144631	0.852420	0.383076	0.068069	-0.257052
1	1.023089	2.447824	1.955609	-0.342571	-0.638644	-1.132748	-0.698278	0.250132	2.021285	-0.022775
2	0.713562	0.444004	0.998183	1.182114	0.511033	-0.208413	-0.602859	0.027673	0.244557	1.112746
3	-0.560726	1.609887	0.166799	0.177142	-0.213090	1.263828	0.444392	-0.430236	0.255107	1.794763
4	-0.535776	0.919209	0.019709	-0.075786	0.536196	-0.521786	-1.845877	1.348950	-0.354426	-0.115867
5	0.835618	1.056863	-0.514179	1.501283	-0.145441	-0.829598	1.084315	1.020663	0.312939	-0.063583
6	-1.536829	-1.649149	1.488113	1.114564	-0.284665	-0.137031	-0.399307	0.420265	-0.899210	0.445243
7	0.578409	0.004359	-0.355127	1.531821	1.456159	1.905293	1.321263	-0.083592	-0.348308	0.074850
8	1.170705	-1.760534	-1.855383	-1.478054	-0.712683	0.835516	-2.674985	-1.351553	-1.149780	1.294937
9	-1.309497	1.659212	-0.788045	-1.478147	-0.065186	0.676414	0.443688	0.016864	1.082966	-1.115552

The `df.to_html()` function prints the DataFrame as an HTML table. The `{{ df|safe }}` element allows the HTML table to be escaped so that it will be displayed properly.

There's more

The `to_html()` method has many parameters dedicated to the configuration of the HTML output. For instance, combining the `to_html()` method's ability to change the class of the DataFrame, it's very easy to output tables in good-looking HTML.

To get `bootstrap css` ready, download it and then copy the `bootstrap.css` file into the `css` folder within the `static` folder. Your folder structure should look as follows:

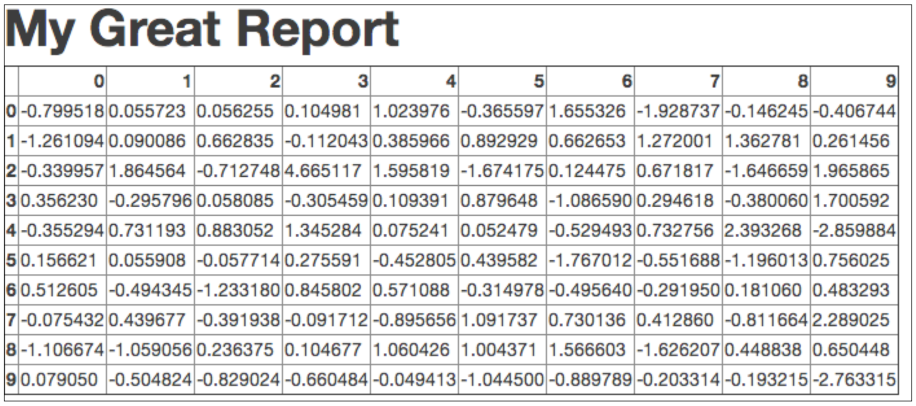


```
|-static
|---css
|-templates
```

And add this line to the head of the `report.html` document.

```
<link rel="stylesheet" type="text/css" href={{
url_for('static', filename='css/bootstrap.css') }}>
```

This will give you a table that resembles the following image:



My Great Report

	0	1	2	3	4	5	6	7	8	9
0	-0.799518	0.055723	0.056255	0.104981	1.023976	-0.365597	1.655326	-1.928737	-0.146245	-0.406744
1	-1.261094	0.090086	0.662835	-0.112043	0.385966	0.892929	0.662653	1.272001	1.362781	0.261456
2	-0.339957	1.864564	-0.712748	4.665117	1.595819	-1.674175	0.124475	0.671817	-1.646659	1.965865
3	0.356230	-0.295796	0.058085	-0.305459	0.109391	0.879648	-1.086590	0.294618	-0.380060	1.700592
4	-0.355294	0.731193	0.883052	1.345284	0.075241	0.052479	-0.529493	0.732756	2.393268	-2.859884
5	0.156621	0.055908	-0.057714	0.275591	-0.452805	0.439582	-1.767012	-0.551688	-1.196013	0.756025
6	0.512605	-0.494345	-1.233180	0.845802	0.571088	-0.314978	-0.495640	-0.291950	0.181060	0.483293
7	-0.075432	0.439677	-0.391938	-0.091712	-0.895656	1.091737	0.730136	0.412860	-0.811664	2.289025
8	-1.106674	-1.059056	0.236375	0.104677	1.060426	1.004371	1.566603	-1.626207	0.448838	0.650448
9	0.079050	-0.504824	-0.829024	-0.660484	-0.049413	-1.044500	-0.889789	-0.203314	-0.193215	-2.763315

Visualizing pandas objects (Advanced)

The plotting functionality in pandas is getting better with each release. The maintainers have a long term view as well. Right now pandas utilizes `matplotlib` to do its plotting, which has served the Python community very well, but there are a fair amount of attempts to combine pandas with JavaScript's graphing libraries.

Getting ready

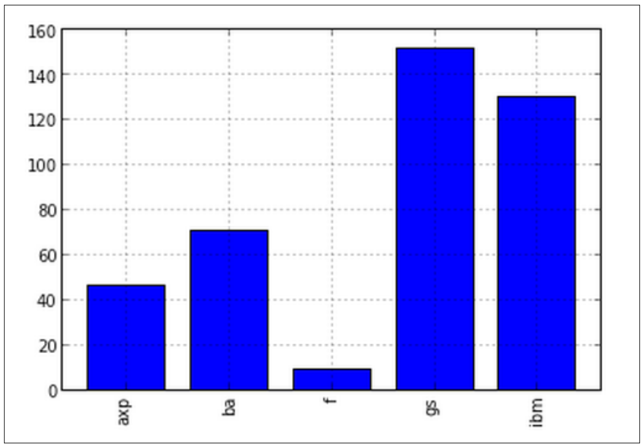
Plotting in pandas is done via an interface with `matplotlib`, as was done earlier with `close.plot()`. But it's easy to go beyond that for a variety of reasons. pandas offers not only simple diagnostic plots, but also some more advanced visualizations such as `scatter_matrix`.

pandas plotting is subjected to `matplotlib`, which facilitates the simple call above. Therefore it is also subjected to the same commands for formatting, which will be discussed later. For the plotting sections it is recommended that you use IPython Notebook.

How to do it...

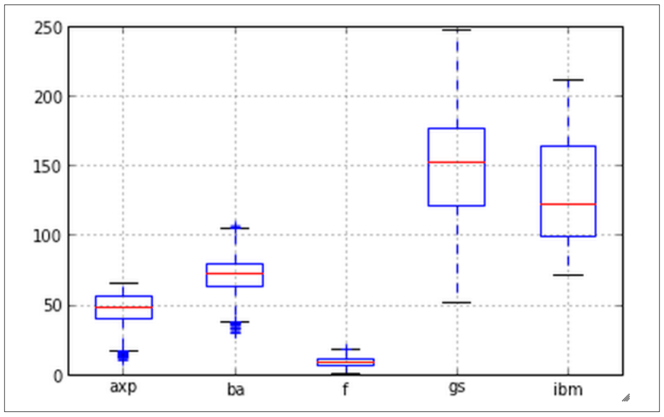
1. Using the same `df Close` method as in the prior examples, aggregate the DataFrame by `mean`, then plot it using the `.plot` method.

```
> close.mean().plot(kind='bar')
```



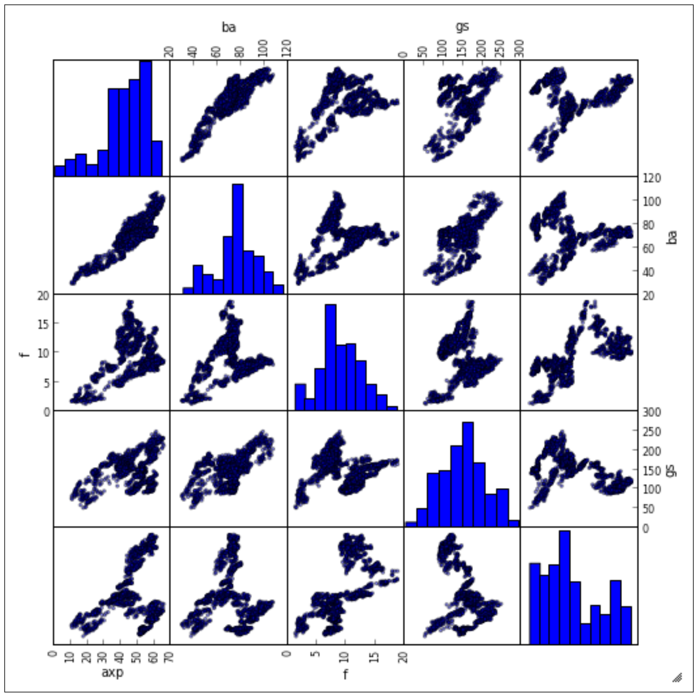
2. Create a `boxplot` method, which is another method that is directly accessible from the DataFrame object.

```
> close.boxplot()
```



3. Earlier we were trying to determine the relationship of closing prices between stocks—a scatter matrix is a good choice. In order to do more complex visualizations than a simple box plot or a bar chart, access `pandas.tools.plotting`.

```
> from pandas.tools.plotting import scatter_matrix
> scatter_matrix(close)
```



There's more

The number of ways to configure the plots of pandas objects is far too large to cover in a single recipe, but there's a huge gallery of examples available online for `matplotlib`. Many of the concepts applicable to `matplotlib` at large are also applicable to pandas plotting specifically.

Additional options for scatter_matrix

Passing the `diagonal` parameter to the `scatter_matrix` function with the `kde` string will display the kernel density estimation instead of a histogram.

```
> scatter_matrix(close, diagonal='kde')
```

Other options for producing plots

It is worth investing in learning IPython Notebook in order to share the analysis that pandas makes so easily. It's featured throughout the book, but here general plot formatting will be discussed.

First execute the following:

```
>import matplotlib.pyplot as plt
>plt.plot(close)
```

It is not by accident that the output looks very similar to the previous plot.

Because of the similarities, modifying the size of the output is quite simple.

For example, IPython has a `figsize` function via `pylab`, which allows you to set the size of the plot.

Reporting with pandas objects (Medium)

In the final recipe, we'll do an example that is a bit more complex than the one in the recipe in which we introduced Flask as a backend. Flask can often be used for dead simple APIs, so in this recipe we'll walk through filtering a DataFrame by month via a Flask route. While the data will be fed to a template, this recipe can easily be modified to `jsonify` the DataFrame and do something with an API.

Getting ready

The final step is putting all this together to present a report. There are a few ways this could be done depending on the desired level of effort, audience, and other factors.

One option for a more formal reporting could be to use the `to_latex` method to output a DataFrame to the laTeX markup—if the name wasn't enough. Combined with the `Pweave` package, writing DataFrames as nicely formatted tables can lead to an aesthetically pleasing report.

But the option pursued here is similar to what was used earlier. In fact, that application will be used as the basis for this. It will be a simplistic example of taking input from the user and reporting an `aggregate` function in a visually pleasing manner.

How to do it...

1. First we need to modify `app.py` so that it can accept user input to modify `to_html` to work with Bootstrap.

```
from flask import Flask, render_template
import pandas as pd

app = Flask(__name__)
@app.route('/report/<month>')
def report(month):
    from pandas.io.data import DataReader
    ibm = DataReader('ibm', 'yahoo', '2010-01-01', '2010-
      12-31')
    df = ibm[ibm.index.month == month]
```

```
        return render_template('report.html',
        df=df.to_html(classes='table'))
    if __name__ == '__main__':
            app.run()
```

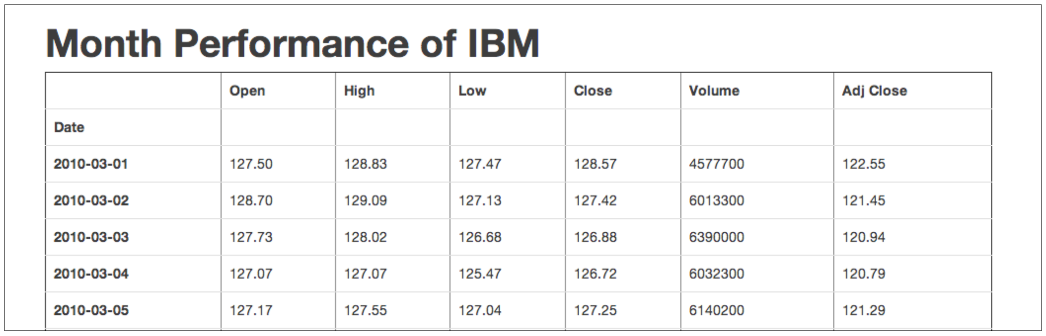
2. Next, add custom `css` to `style.css` and modify HTML to work with the CSS file. First edit the `report.html` file in the `templates` folder.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href={{
    url_for('static', filename='css/bootstrap.css') }}>
    <link rel="stylesheet" type="text/css" href={{
     url_for('static', filename='css/style.css') }}>
  </head>
  <body>
    <h1>Month Performance of IBM</h1>
    {{ df|safe }}
  </body>
</html>
```

3. Modify `style.css` to contain the following:

```
body {
  margin: auto;
  width: 960px;
    padding-top: 50px;
}
```

4. Now when you start the flask server and go to `localhost:5000/report/3`, you'll get March's performance for IBM.



Month Performance of IBM

	Open	High	Low	Close	Volume	Adj Close
Date						
2010-03-01	127.50	128.83	127.47	128.57	4577700	122.55
2010-03-02	128.70	129.09	127.13	127.42	6013300	121.45
2010-03-03	127.73	128.02	126.68	126.88	6390000	120.94
2010-03-04	127.07	127.07	125.47	126.72	6032300	120.79
2010-03-05	127.17	127.55	127.04	127.25	6140200	121.29

How it works...

Flask is a simple Python framework that can become quite powerful when used with a large number of extensions. It also serves as a very easy way to set up a basic API for another service. This is actually a good way to develop the reporting application. By separating the service that delivers data from the visualization, and mark up of the frontend it makes the code easier to maintain long-term. It's essentially the same idea behind keeping CSS out of HTML.

There's more

This book has only skimmed the surface of an ever growing toolkit, specifically pandas and PyData tools overall. Python has been a good language for a long time, and with a vibrant community, hopefully it can continue.

Next steps in Python visualization

There is a lot of desire and work being done to integrate `d3.js` with pandas to allow for easier web graphics. It's already quite simple to display a table in a nice, formatted HTML, but having graphics in the browser would be a huge win for pandas.

Also, little knowledge about JavaScript, d3, and pandas can been combined for decomposing a DataFrame into an array of arrays.

The future of pandas

Work is constantly being done on pandas. One reason for this is the fair number of contributors. Certainly the core developers do a lion's share of the work, but there is a long tail of contributors who help to slowly move it forward. Next to learn in pandas is how to take a look at the objects themselves, not simply the problems but also how to solve the abstractions. In doing so, you'll often find small things that can be fixed.