Learning pandas

This learner’s guide will help you understand how to use the features of pandas for interactive data manipulation and analysis.

This book is your ideal guide to learning about pandas, all the way from installing it to creating one- and two-dimensional indexed data structures, indexing and slicing-and-dicing that data to derive results, loading data from local and Internet-based resources, and finally creating effective visualizations to form quick insights. You start with an overview of pandas and NumPy and then dive into the details of pandas, covering pandas’ Series and DataFrame objects, before ending with a quick review of using pandas for several problems in finance.

With the knowledge you gain from this book, you will be able to quickly begin your journey into the exciting world of data science and analysis.

Who this book is written for

If you are a Python programmer who wants to get started with performing data analysis using pandas and Python, this is the book for you. Some experience with statistical analysis would be helpful but is not mandatory.

What you will learn from this book

- Install pandas on Windows, Mac, and Linux using the Anaconda Python distribution
- Learn how pandas builds on NumPy to implement flexible indexed data
- Adopt pandas’ Series and DataFrame objects to represent one- and two-dimensional data constructs
- Index, slice, and transform data to derive meaning from information
- Load data from files, databases, and web services
- Manipulate dates, times, and time series data
- Group, aggregate, and summarize data
- Visualize techniques for pandas and statistical data

Get to grips with pandas—a versatile and high-performance Python library for data manipulation, analysis, and discovery.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'A Tour of pandas'
- A synopsis of the book’s content
- More information on Learning pandas

About the Author

**Michael Heydt** is an independent consultant, educator, and trainer with nearly 30 years of professional software development experience, during which 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 been specializing 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, 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 (*Instant Lucene .NET*, *Learning pandas*). He is a frequent speaker at .NET users' groups and various mobile and cloud conferences, and he regularly delivers webinars on advanced technologies.
Learning pandas

This book is about learning to use pandas, an open source library for Python, which was created to enable Python to easily manipulate and perform powerful statistical and mathematical analyses on tabular and multidimensional datasets. The design of pandas and its power combined with the familiarity of Python have created explosive growth in its usage over the last several years, particularly among financial firms as well as those simply looking for practical tools for statistical and data analysis.

While there exist many excellent examples of using pandas to solve many domain-specific problems, it can be difficult to find a cohesive set of examples in a form that allows one to effectively learn and apply the features of pandas. The information required to learn practical skills in using pandas is distributed across many websites, slide shares, and videos, and is generally not in a form that gives an integrated guide to all of the features with practical examples in an easy-to-understand and applicable fashion.

This book is therefore intended to be a go-to reference for learning pandas. It will take you all the way from installation, through to creating one- and two-dimensional indexed data structures, to grouping data and slicing-and-dicing them, with common analyses used to demonstrate derivation of useful results. This will include the loading and saving of data from resources that are local and Internet-based and creating effective data visualizations that provide instant ability to visually realize insights into the meaning previously hidden within complex data.

What This Book Covers

Chapter 1, A Tour of pandas, is a hands-on introduction to the key features of pandas. It will give you a broad overview of the types of data tasks that can be performed with pandas. This chapter will set the groundwork for learning as all concepts introduced in this chapter will be expanded upon in subsequent chapters.

Chapter 2, Installing pandas, will show you how to install Anaconda Python and pandas on Windows, OS X, and Linux. This chapter also covers using the conda package manager to upgrade pandas and its dependent libraries to the most recent version.

Chapter 3, NumPy for pandas, will introduce you to concepts in NumPy, particularly NumPy arrays, which are core for understanding the pandas Series and DataFrame objects.
Chapter 4, *The pandas Series Object*, covers the pandas `Series` object and how it expands upon the functionality of the NumPy array to provide richer representation and manipulation of sequences of data through the use of high-performance indexes.

Chapter 5, *The pandas DataFrame Object*, introduces the primary data structure of pandas, the `DataFrame` object, and how it forms a two-dimensional representation of tabular data by aligning multiple `Series` objects along a common index to provide seamless access and manipulation across elements in multiple series that are related by a common index label.

Chapter 6, *Accessing Data*, shows how data can be loaded and saved from external sources into both `Series` and `DataFrame` objects. You will learn how to access data from multiple sources such as files, HTTP servers, database systems, and web services, as well as how to process data in CSV, HTML, and JSON formats.

Chapter 7, *Tidying Up Your Data*, instructs you on how to use the various tools provided by pandas for managing dirty and missing data.

Chapter 8, *Combining and Reshaping Data*, covers various techniques for combining, splitting, joining, and merging data located in multiple pandas objects, and then demonstrates how to reshape data using concepts such as pivots, stacking, and melting.

Chapter 9, *Grouping and Aggregating Data*, focuses on how to use pandas to group data to enable you to perform aggregate operations on grouped data to assist in deriving analytic results.

Chapter 10, *Time-series Data*, will instruct you on how to use pandas to represent sequences of information that is indexed by the progression of time. This chapter will first cover how pandas represents dates and time, as well as concepts such as periods, frequencies, time zones, and calendars. The focus then shifts to time-series data and various operations such as shifting, lagging, resampling, and moving window operations.

Chapter 11, *Visualization*, dives into the integration of pandas with matplotlib to visualize pandas data. This chapter will demonstrate how to represent and present many common statistical and financial data visualizations, including bar charts, histograms, scatter plots, area plots, density plots, and heat maps.

Chapter 12, *Applications to Finance*, brings together everything learned through the previous chapters with practical examples of using pandas to obtain, manipulate, analyze, and visualize stock data.
In this chapter, we will take a look at **pandas**, which is an open source Python-based data analysis library. It provides high-performance and easy-to-use data structures and data analysis tools built with the Python programming language. The pandas library brings many of the good things from R, specifically the `DataFrame` objects and R packages such as `plyr` and `reshape2`, and places them in a single library that you can use in your Python applications.

The development of pandas was begun in 2008 by Wes McKinney when he worked at AQR Capital Management. It was opened sourced in 2009 and is currently supported and actively developed by various organizations and contributors. It was initially designed with finance in mind, specifically with its ability around time series data manipulation, but emphasizes the data manipulation part of the equation leaving statistical, financial, and other types of analyses to other Python libraries.

In this chapter, we will take a brief tour of pandas and some of the associated tools such as IPython notebooks. You will be introduced to a variety of concepts in pandas for data organization and manipulation in an effort to form both a base understanding and a frame of reference for deeper coverage in later sections of this book. By the end of this chapter, you will have a good understanding of the fundamentals of pandas and even be able to perform basic data manipulations. Also, you will be ready to continue with later portions of this book for more detailed understanding.

This chapter will introduce you to:

- pandas and why it is important
- IPython and IPython Notebooks
- Referencing pandas in your application
- The `Series` and `DataFrame` objects of pandas
- How to load data from files and the Web
- The simplicity of visualizing pandas data
pandas and why it is important

pandas is a library containing high-level data structures and tools that have been created to assist a Python programmer to perform powerful data manipulations, and discover information in that data in a simple and fast way.

The simple and effective data analysis requires the ability to index, retrieve, tidy, reshape, combine, slice, and perform various analyses on both single and multidimensional data, including heterogeneous typed data that is automatically aligned along index labels. To enable these capabilities, pandas provides the following features (and many more not explicitly mentioned here):

- High performance array and table structures for representation of homogenous and heterogeneous data sets: the Series and DataFrame objects
- Flexible reshaping of data structure, allowing the ability to insert and delete both rows and columns of tabular data
- Hierarchical indexing of data along multiple axes (both rows and columns), allowing multiple labels per data item
- Labeling of series and tabular data to facilitate indexing and automatic alignment of data
- Ability to easily identify and fix missing data, both in floating point and as non-floating point formats
- Powerful grouping capabilities and a functionality to perform split-apply-combine operations on series and tabular data
- Simple conversion from ragged and differently indexed data of both NumPy and Python data structures to pandas objects
- Smart label-based slicing and subsetting of data sets, including intuitive and flexible merging, and joining of data with SQL-like constructs
- Extensive I/O facilities to load and save data from multiple formats including CSV, Excel, relational and non-relational databases, HDF5 format, and JSON
Explicit support for time series-specific functionality, providing functionality for date range generation, moving window statistics, time shifting, lagging, and so on

Built-in support to retrieve and automatically parse data from various web-based data sources such as Yahoo!, Google Finance, the World Bank, and several others

For those desiring to get into data analysis and the emerging field of data science, pandas offers an excellent means for a Python programmer (or just an enthusiast) to learn data manipulation. For those just learning or coming from a statistical language like R, pandas can offer an excellent introduction to Python as a programming language.

pandas itself is not a data science toolkit. It does provide some statistical methods as a matter of convenience, but to draw conclusions from data, it leans upon other packages in the Python ecosystem, such as SciPy, NumPy, scikit-learn, and upon graphics libraries such as matplotlib and ggvis for data visualization. This is actually the strength of pandas over other languages such as R, as pandas applications are able to leverage an extensive network of robust Python frameworks already built and tested elsewhere.

In this book, we will look at how to use pandas for data manipulation, with a specific focus on gathering, cleaning, and manipulation of various forms of data using pandas. Detailed specifics of data science, finance, econometrics, social network analysis, Python, and IPython are left as reference. You can refer to some other excellent books on these topics already available at https://www.packtpub.com/.

**pandas and IPython Notebooks**

A popular means of using pandas is through the use of IPython Notebooks. IPython Notebooks provide a web-based interactive computational environment, allowing the combination of code, text, mathematics, plots, and right media into a web-based document. IPython Notebooks run in a browser and contain Python code that is run in a local or server-side Python session that the notebooks communicate with using WebSockets. Notebooks can also contain markup code and rich media content, and can be converted to other formats such as PDF, HTML, and slide shows.
The following is an example of an IPython Notebook from the IPython website (http://ipython.org/notebook.html) that demonstrates the rich capabilities of notebooks:

### Simple spectral analysis

An illustration of the Discrete Fourier Transform

\[
X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi}{N} in} \quad k = 0, \ldots, N - 1
\]

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a dataset using SciPy’s audio file support:

```python
In [1]: from scipy.io import wavfile
   ...: rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib’s built-in `spectrogram` routine:

```python
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
   ...: ax1.plot(x); ax1.set_title('Raw audio signal')
   ...: ax2.spectrogram(x); ax2.set_title('Spectrogram');
```

IPython Notebooks are not strictly required for using pandas and can be installed into your development environment independently or alongside of pandas. During the course of this book, we will install pandas and an IPython Notebook server. You will be able to perform code examples in the text directly in an IPython console interpreter, and the examples will be packaged as notebooks that can be run with a local notebook server. Additionally, the workbooks will be available online for easy and immediate access at https://wakari.io/sharing/bundle/LearningPandas/LearningPandas_Index.
To learn more about IPython Notebooks, visit the notebooks site at http://ipython.org/ipython-doc/dev/notebook/, and for more in-depth coverage, refer to another book, Learning IPython for Interactive Computing and Data Visualization, Cyrille Rossant, Packt Publishing.

Referencing pandas in the application

All pandas programs and examples in this book will always start by importing pandas (and NumPy) into the Python environment. There is a common convention used in many publications (web and print) of importing pandas and NumPy, which will also be used throughout this book. All workbooks and examples for chapters will start with code similar to the following to initialize the pandas library within Python.

In [1]:
    # import numpy and pandas, and DataFrame / Series
    import numpy as np
    import pandas as pd
    from pandas import DataFrame, Series

    # Set some pandas options
    pd.set_option('display.notebook_repr_html', False)
    pd.set_option('display.max_columns', 10)
    pd.set_option('display.max_rows', 10)

    # And some items for matplotlib
    %matplotlib inline
    import matplotlib.pyplot as plt
    pd.options.display.mpl_style = 'default'

NumPy and pandas go hand-in-hand, as much of pandas is built on NumPy. It is, therefore, very convenient to import NumPy and put it in a np. namespace. Likewise, pandas is imported and referenced with a pd. prefix. Since DataFrame and Series objects of pandas are used so frequently, the third line then imports the Series and DataFrame objects into the global namespace so that we can use them without a pd. prefix.
A Tour of pandas

The three `pd.set_options()` method calls set up some defaults for IPython Notebooks and console output from pandas. These specify how wide and high any output will be, and how many columns it will contain. They can be used to modify the output of IPython and pandas to fit your personal needs to display results. The options set here are convenient for formatting the output of the examples to the constraints of the text.

**Primary pandas objects**

A programmer of pandas will spend most of their time using two primary objects provided by the pandas framework: `Series` and `DataFrame`. The `DataFrame` objects will be the overall workhorse of pandas and the most frequently used as they provide the means to manipulate tabular and heterogeneous data.

**The pandas Series object**

The base data structure of pandas is the `Series` object, which is designed to operate similar to a NumPy array but also adds index capabilities. A simple way to create a `Series` object is by initializing a `Series` object with a Python array or Python list.

```python
In [2]:
    # create a four item DataFrame
    s = Series([1, 2, 3, 4])
    s

Out [2]:
   0  1
   1  2
   2  3
   3  4
   dtype: int64
```

This has created a pandas `Series` from the list. Notice that printing the series resulted in what appears to be two columns of data. The first column in the output is not a column of the `Series` object, but the index labels. The second column is the values of the `Series` object. Each row represents the index label and the value for that label. This `Series` was created without specifying an index, so pandas automatically creates indexes starting at zero and increasing by one.
Elements of a `Series` object can be accessed through the index using `[ ]`. This informs the `Series` which value to return given one or more index values (referred to in pandas as labels). The following code retrieves the items in the series with labels 1 and 3.

In [3]:
   # return a Series with the rows with labels 1 and 3
   s[[1, 3]]

Out [3]:
   1    2
   3    4
dtype: int64

It is important to note that the lookup here is not by zero-based positions 1 and 3 like an array, but by the values in the index.

A `Series` object can be created with a user-defined index by specifying the labels for the index using the `index` parameter.

In [4]:
   # create a series using an explicit index
   s = Series([1, 2, 3, 4],
               index = ['a', 'b', 'c', 'd'])
   s

Out [4]:
   a    1
   b    2
   c    3
   d    4
dtype: int64

Notice that the index labels in the output now have the index values that were specified in the `Series` constructor.
Data in the Series object can now be accessed by alphanumeric index labels by passing a list of the desired labels, as the following demonstrates:

In [5]:
    # look up items the series having index 'a' and 'd'
    s[['a', 'd']]

Out [5]:
    a    1
    d    4
dtype: int64

This demonstrates the previous point that the lookup is by label value and not by zero-based position.

It is still possible to refer to the elements of the Series object by their numerical position.

In [6]:
    # passing a list of integers to a Series that has
    # non-integer index labels will look up based upon
    # 0-based index like an array
    s[[1, 2]]

Out [6]:
    b    2
    c    3
dtype: int64

A Series is still smart enough to determine that you passed a list of integers and, therefore, that you want to do value lookup by zero-based position.
The `s.index` property allows direct access to the index of the `Series` object.

In [7]:
   # get only the index of the Series
   s.index

Out [7]:
   Index([u'a', u'b', u'c', u'd'], dtype='object')

The index is itself actually a pandas object. This shows us the values of the index and that the data type of each label in the index is `object`.

A common usage of a `Series` in pandas is to represent a time series that associates date/time index labels with a value. A date range can be created using the pandas method `pd.date_range()`.

In [8]:
   # create a Series who's index is a series of dates
   # between the two specified dates (inclusive)
   dates = pd.date_range('2014-07-01', '2014-07-06')
   dates

Out [8]:
   <class 'pandas.tseries.index.DatetimeIndex'>
   [2014-07-01, ..., 2014-07-06]
   Length: 6, Freq: D, Timezone: None

This has created a special index in pandas referred to as a `DatetimeIndex`, which is a pandas index that is optimized to index data with dates and times.

At this point, the index is not particularly useful without having values for each index. We can use this index to create a new `Series` object with values for each of the dates.

In [9]:
   # create a Series with values (representing temperatures)
   # for each date in the index
   temps1 = Series([80, 82, 85, 90, 83, 87],
                   index = dates)
A Tour of pandas

temps1

Out [9]:
2014-07-01    80
2014-07-02    82
2014-07-03    85
2014-07-04    90
2014-07-05    83
2014-07-06    87
Freq: D, dtype: int64

Statistical methods provided by NumPy can be applied to a pandas Series. The following returns the mean of the values in the Series.

In [10]:
   # calculate the mean of the values in the Series
   temps1.mean()

Out [10]:
   84.5

Two Series objects can be applied to each other with an arithmetic operation. The following code calculates the difference in temperature between two Series.

In [11]:
   # create a second series of values using the same index
   temps2 = Series([70, 75, 69, 83, 79, 77],
                   index = dates)
   # the following aligns the two by their index values
   # and calculates the difference at those matching labels
   temp_diffs = temps1 - temps2
   temp_diffs

Out [11]:
2014-07-01    10
2014-07-02     7
2014-07-03    16
2014-07-04     7
2014-07-05     4
The result of an arithmetic operation (+, -, /, *, ...) on two `Series` objects that are non-scalar values returns another `Series` object.

Time series data such as that shown here can also be accessed via the index or by an offset into the `Series` object.

In [12]:
   # lookup a value by date using the index
   temp_diffs['2014-07-03']

Out [12]:
   16

In [13]:
   # and also possible by integer position as if the
   # series was an array
   temp_diffs[2]

Out [13]:
   16

**The pandas DataFrame object**

A pandas `Series` represents a single array of values, with an index label for each value. If you want to have more than one `Series` of data that is aligned by a common index, then a pandas `DataFrame` is used.

In a way a `DataFrame` is analogous to a database table in that it contains one or more columns of data of heterogeneous type (but a single type for all items in each respective column).
A Tour of pandas

The following code creates a DataFrame object with two columns representing the temperatures from the Series objects used earlier.

In [14]:
    # create a DataFrame from the two series objects temp1 and temp2
    # and give them column names
    temps_df = DataFrame(
        {'Missoula': temps1,
         'Philadelphia': temps2})
    temps_df

Out [14]:
           Missoula  Philadelphia
    2014-07-01    80            70
    2014-07-02    82            75
    2014-07-03    85            69
    2014-07-04    90            83
    2014-07-05    83            79
    2014-07-06    87            77

This has created a DataFrame object with two columns, named Missoula and Philadelphia, and using the values from the respective Series objects for each. These are new Series objects contained within the DataFrame, with the values copied from the original Series objects.

Columns in a DataFrame object can be accessed using an array indexer [] with the name of the column or a list of column names. The following code retrieves the Missoula column of the DataFrame object:

In [15]
    # get the column with the name Missoula
    temps_df['Missoula']

Out [15]:
    2014-07-01    80
    2014-07-02    82
    2014-07-03    85
    2014-07-04    90
The following code retrieves the Philadelphia column:

In [16]:
    # likewise we can get just the Philadelphia column
    temps_df['Philadelphia']

Out [16]:
    2014-07-01    70
    2014-07-02    75
    2014-07-03    69
    2014-07-04    83
    2014-07-05    79
    2014-07-06    77
Freq: D, Name: Philadelphia, dtype: int64

The following code returns both the columns, but reversed.

In [17]:
    # return both columns in a different order
    temps_df[['Philadelphia', 'Missoula']]

Out [17]:

<table>
<thead>
<tr>
<th></th>
<th>Philadelphia</th>
<th>Missoula</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-07-01</td>
<td>70</td>
<td>80</td>
</tr>
<tr>
<td>2014-07-02</td>
<td>75</td>
<td>82</td>
</tr>
<tr>
<td>2014-07-03</td>
<td>69</td>
<td>85</td>
</tr>
<tr>
<td>2014-07-04</td>
<td>83</td>
<td>90</td>
</tr>
<tr>
<td>2014-07-05</td>
<td>79</td>
<td>83</td>
</tr>
<tr>
<td>2014-07-06</td>
<td>77</td>
<td>87</td>
</tr>
</tbody>
</table>

Notice that there is a subtle difference in a DataFrame object as compared to a Series object. Passing a list to the [] operator of DataFrame retrieves the specified columns, whereas Series uses it as index labels to retrieve rows.
Very conveniently, if the name of a column does not have spaces, you can use property-style names to access the columns in a DataFrame.

In [18]:
    # retrieve the Missoula column through property syntax
    temps_df.Missoula

Out [18]:
    2014-07-01    80
    2014-07-02    82
    2014-07-03    85
    2014-07-04    90
    2014-07-05    83
    2014-07-06    87
Freq: D, Name: Missoula, dtype: int64

Arithmetic operations between columns within a DataFrame are identical in operation to those on multiple Series as each column in a DataFrame is a Series. To demonstrate, the following code calculates the difference between temperatures using property notation.

In [19]:
    # calculate the temperature difference between the two cities
    temps_df.Missoula - temps_df.Philadelphia

Out [19]:
    2014-07-01    10
    2014-07-02     7
    2014-07-03    16
    2014-07-04     7
    2014-07-05     4
    2014-07-06    10
Freq: D, dtype: int64
A new column can be added to `DataFrame` simply by assigning another `Series` to a column using the array indexer `[]` notation. The following code adds a new column in the `DataFrame`, which contains the difference in temperature on the respective dates.

In [20]:
   # add a column to `temp_df` that contains the difference in temps
   temps_df['Difference'] = temp_diffs
   temps_df

Out [20]:

<table>
<thead>
<tr>
<th></th>
<th>Missoula</th>
<th>Philadelphia</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-07-01</td>
<td>80</td>
<td>70</td>
<td>10</td>
</tr>
<tr>
<td>2014-07-02</td>
<td>82</td>
<td>75</td>
<td>7</td>
</tr>
<tr>
<td>2014-07-03</td>
<td>85</td>
<td>69</td>
<td>16</td>
</tr>
<tr>
<td>2014-07-04</td>
<td>90</td>
<td>83</td>
<td>7</td>
</tr>
<tr>
<td>2014-07-05</td>
<td>83</td>
<td>79</td>
<td>4</td>
</tr>
<tr>
<td>2014-07-06</td>
<td>87</td>
<td>77</td>
<td>10</td>
</tr>
</tbody>
</table>

The names of the columns in a `DataFrame` are object accessible via the `DataFrame` object's `.columns` property, which itself is a pandas Index object.

In [21]:
   # get the columns, which is also an Index object
   temps_df.columns

Out [21]:
   Index([u'Missoula', u'Philadelphia', u'Difference'], dtype='object')

The `DataFrame` (and `Series`) objects can be sliced to retrieve specific rows. A simple example here shows how to select the second through fourth rows of temperature difference values.

In [22]:
   # slice the temp differences column for the rows at
   # location 1 through 4 (as though it is an array)
   temps_df.Difference[1:4]

Out [22]:
   2014-07-02    7
Entire rows from a DataFrame can be retrieved using its .loc and .iloc properties. The following code returns a Series object representing the second row of temps_df of the DataFrame object by zero-based position of the row using the .iloc property:

In [23]:
    # get the row at array position 1
    temps_df.iloc[1]

Out [23]:
    Missoula        82
    Philadelphia    75
    Difference       7
    Name: 2014-07-02 00:00:00, dtype: int64

This has converted the row into a Series, with the column names of the DataFrame pivoted into the index labels of the resulting Series.

In [24]:
    # the names of the columns have become the index
    # they have been 'pivoted'
    temps_df.ix[1].index

Out [24]:
    Index(['Missoula', 'Philadelphia', 'Difference'], dtype='object')

Rows can be explicitly accessed via index label using the .loc property. The following code retrieves a row by the index label:

In [25]:
    # retrieve row by index label using .loc
    temps_df.loc['2014-07-03']

Out [25]:
    Missoula        85
    Philadelphia    69
    Difference      16
    Name: 2014-07-03 00:00:00, dtype: int64
Specific rows in a DataFrame object can be selected using a list of integer positions. The following code selects the values from the `Difference` column in rows at locations 1, 3, and 5.

In [26]:
   # get the values in the Differences column in rows 1, 3, and 5
   # using 0-based location
   temps_df.iloc[[1, 3, 5]].Difference

Out [26]:
   2014-07-02     7
   2014-07-04     7
   2014-07-06    10
   Name: Difference, dtype: int64

Rows of a DataFrame can be selected based upon a logical expression applied to the data in each row. The following code returns the evaluation of the value in the `Missoula` temperature column being greater than 82 degrees:

In [27]:
   # which values in the Missoula column are > 82?
   temps_df.Missoula > 82

Out [27]:
   2014-07-01    False
   2014-07-02    False
   2014-07-03     True
   2014-07-04     True
   2014-07-05     True
   2014-07-06     True
   Freq: D, Name: Missoula, dtype: bool

When using the result of an expression as the parameter to the [] operator of a DataFrame, the rows where the expression evaluated to True will be returned.

In [28]:
   # return the rows where the temps for Missoula > 82
   temps_df[temps_df.Missoula > 82]

Out [28]:

---
This technique of selection in pandas terminology is referred to as a Boolean selection, and will form the basis of selecting data based upon its values.

## Loading data from files and the Web

The data used in analyses is typically provided from other systems via files that are created and updated at various intervals, dynamically via access over the Web, or from various types of databases. The pandas library provides powerful facilities for easy retrieval of data from a variety of data sources and converting it into pandas objects. Here, we will briefly demonstrate this ease of use by loading data from files and from financial web services.

### Loading CSV data from files

The pandas library provides built-in support for loading data in .csv format, a common means of storing structured data in text files. Provided with the code from this book is a file data/test1.csv in the CSV format, which represents some time series information. The specific content isn't important right now, as we just want to demonstrate the ease of loading data into a DataFrame.

The following statement in IPython uses the operating system to display the content of this file (the command to use is different based upon your operating system).

```
In [29]:
   #: display the contents of test1.csv
   #: which command to use depends on your OS
   !cat data/test1.csv # on non-windows systems
   #!type data\test1.csv # on windows systems

date,0,1,2
2000-01-01 00:00:00,1.10376250134,-1.90997889703,-0.808955536115
2000-01-02 00:00:00,1.18891664768,0.581119740849,0.86159734949
2000-01-03 00:00:00,-0.964200042412,0.779764393246,1.82906224532
2000-01-04 00:00:00,0.782130444001,-1.72066965573,-1.10824167327
```
This information can be easily imported into DataFrame using the `pd.read_csv()` function.

In [30]:
    # read the contents of the file into a DataFrame
    df = pd.read_csv('data/test1.csv')
    df

Out [30]:

```
     date         0         1         2
0  2000-01-01 00:00:00  1.103763 -1.909979 -0.808956
1  2000-01-02 00:00:00  1.188917  0.581120  0.861597
2  2000-01-03 00:00:00 -0.964200  0.779764  1.829062
3  2000-01-04 00:00:00  0.782130 -1.720670 -1.108242
4  2000-01-05 00:00:00 -1.867017 -0.528368 -2.488309
5  2000-01-06 00:00:00  2.569280 -0.471901 -0.835033
6  2000-01-07 00:00:00 -0.399323 -0.676426 -0.011256
7  2000-01-08 00:00:00  1.147307  2.137995  0.554172
8  2000-01-09 00:00:00  0.933766  1.387155 -0.560143
9  2000-01-10 00:00:00  0.933766  1.387155 -0.560143
```

pandas has no idea that the first column is a date and has treated the contents of the date field as a string. This can be verified using the following Python statements:

In [31]:
    # the contents of the date column
    df.date

Out [31]:

```
0  2000-01-01 00:00:00
1  2000-01-02 00:00:00
2  2000-01-03 00:00:00
```
A Tour of pandas

In [32]:
   # we can get the first value in the date column
   df.date[0]

Out [32]:
   '2000-01-01 00:00:00'

In [33]:
   # it is a string
   type(df.date[0])

Out [33]:
   str

To guide pandas on how to convert data directly into a Python/pandas date object, we can use the `parse_dates` parameter of the `pd.read_csv()` function. The following code informs pandas to convert the content of the ‘date’ column into actual `TimeStamp` objects.

In [34]:
   # read the data and tell pandas the date column should be
   # a date in the resulting DataFrame
   df = pd.read_csv('data/test1.csv', parse_dates=['date'])

Out [34]:

<table>
<thead>
<tr>
<th>date</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2000-01-01 1.103763 -1.909979 -0.808956</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2000-01-02 1.188917 0.581120 0.861597</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
On checking whether it worked, we see it is indeed a `Timestamp` object now.

In [35]:
   # verify the type now is date
   # in pandas, this is actually a Timestamp
   type(df.date[0])

Out [35]:
   pandas.tslib.Timestamp

Unfortunately, this has not used the date field as the index for the DataFrame, instead it uses the default zero-based integer index labels.

In [36]:
   # unfortunately the index is numeric, which makes
   # accessing data by date more complicated
   df.index

Out [36]:
   Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

This can be rectified using the `index_col` parameter of the `pd.read_csv()` method to specify which column in the file should be used as the index.

In [37]:
   # read in again, now specify the data column as being the
   # index of the resulting DataFrame
   df = pd.read_csv('data/test1.csv',
                   parse_dates=['date'],
                   index_col='date',
                   ...)
A Tour of pandas

    index_col='date')
    df

Out [37]:

          0        1       2
    date
    2000-01-01  1.103763  -1.909979  -0.808956
    2000-01-02  1.188917   0.581120   0.861597
    2000-01-03 -0.964200   0.779764   1.829062
    2000-01-04  0.782130  -1.720670  -1.108242
    2000-01-05 -1.867017  -0.528368  -2.488309
    2000-01-06  2.569280  -0.471901  -0.835033
    2000-01-07 -0.399323  -0.676427  -0.011256
    2000-01-08  1.642993   1.013420   1.435667
    2000-01-09  1.147308   2.138000   0.554171
    2000-01-10  0.933766   1.387155  -0.560143

In [38]:
    df.index

Out [38]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01, ..., 2000-01-10]
Length: 10, Freq: None, Timezone: None

**Loading data from the Web**

Data from the Web can also be easily read via pandas. To demonstrate this, we will perform a simple load of actual stock data. The example here uses the pandas.io.data.DataReader class, which is able to read data from various web sources, one of which is stock data from Yahoo! Finance.
The following reads the data of the previous three months for GOOG (based on the current date), and prints the five most recent days of stock data:

In [39]:

```python
# imports for reading data from Yahoo!
from pandas.io.data import DataReader
from datetime import date
from dateutil.relativedelta import relativedelta

# read the last three months of data for GOOG
goog = DataReader("GOOG", "yahoo",
                   date.today() +
                   relativedelta(months=-3))

# the result is a DataFrame
# and this gives us the 5 most recent prices
goog.tail()
```

Out [39]:

```
Open    High     Low   Close   Volume  Adj Close
Date
2015-02-02  531.73  533.00  518.55  528.48  2826300     528.48
2015-02-03  528.00  533.40  523.26  529.24  2029200     529.24
2015-02-04  529.24  532.67  521.27  522.76  1656800     522.76
2015-02-05  523.79  528.50  522.09  527.58  1840300     527.58
2015-02-06  527.64  537.20  526.41  531.00  1744600     531.00
```

This actually performs quite a bit of work on your behalf. It makes the web requests retrieving the CSV data and converting it into a DataFrame with the proper conversion of types for the various series of data.
Simplicity of visualization of pandas data

Visualizing pandas data is incredibly simple as pandas is built with tight integration with the matplotlib framework. To demonstrate how simple it is to visualize data with pandas, the following code plots the stock data we just retrieved from Yahoo! Finance:

```python
In [40]:
    # plot the Adj Close values we just read in
    goog.plot(y='Adj Close');
```

We will dive deeper and broader into pandas data visualization in a section dedicated to it later in this book.
Summary

In this chapter we have taken a quick tour of the capabilities of pandas, and how easily you can use it to create, load, manipulate, and visualize data. Through the remainder of this book, we will dive into everything covered in this chapter in significant detail, fully demonstrating how to utilize the facilities of pandas for powerful data manipulation.

In the next chapter, we will look at how to get and install both Python and pandas. Following the installation, in Chapter 3, NumPy for pandas, we will dive into the NumPy framework as it applies to pandas, demonstrating how NumPy provides the core functionality to slice and dice array-based data in array-like manner, as the pandas Series and DataFrame objects extensively leverage the capabilities of NumPy.
Where to buy this book

You can buy *Learning pandas* 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.